Software and Hardware Realization of Modular Arithmetic and Finite Fields for Public-Key Cryptography

Çetin Kaya Koç

koc@cs.ucsb.edu

Cryptographic Engineering
2012

Contents

- RSA, DSA, and elliptic curve basics

- Exponentiation algorithms

- Modular arithmetic operations

- Algorithms for hardware

- Finite fields

RSA Arithmetic

The RSA algorithm uses modular exponentiation for encryption

$$C = M^e \pmod{n}$$

and decryption

$$M = C^d \pmod{n}$$

The computation of $M^e \bmod n$ is performed using exponentiation heuristics

Modular exponentiation requires implementation of three basic modular arithmetic operations: addition, subtraction, and multiplication

Parameters

n, the k-bit public modulus, is a product of two secret primes

$$n = p \cdot q$$

p and q are approximately $k/2$ bits each

e is the h-bit public exponent

$$e = (e_{h-1} e_{h-2} \cdots e_0)$$

h can be small: 2-128 bits.

d is the k-bit private exponent

$$d = (d_{k-1} d_{k-2} \cdots d_0)$$

k is large: 1024 bits or more

RSA Decryption using CRT

The signature operation can be decomposed into two half-size modular exponentiations using the Chinese remainder theorem

The user knows the factors p and q of his own modulus $n = p \cdot q$

The decryption operation

$$M = C^d \pmod{n}$$

can be decomposed into two half-size modular exponentiations with the Chinese remainder theorem (*Quisquater & Couvreur*)

$$\begin{aligned} d_1 &= d \bmod (p-1) \\ d_2 &= d \bmod (q-1) \\ M_1 &= C^{d_1} \pmod{p} \\ M_2 &= C^{d_2} \pmod{q} \end{aligned}$$

$$M = M_2 + q \cdot [(M_1 - M_2) \cdot q^{-1} \pmod{p}]$$

DSS Algorithm

p is prime
q is prime (dividing $p-1$)
g is the qth root of 1 modulo p
x is private key (secret integer less than $q-1$)
$y = g^x \pmod{p}$ is the public key

The signature on M and k is the pair (r, s)

$$\begin{aligned} r &:= (g^k \bmod p) \bmod q \\ s &:= (M + xr)k^{-1} \bmod q \end{aligned}$$

The signature verification

$$\begin{aligned} w &:= s^{-1} \bmod q \\ u_1 &:= Mw \bmod q \\ u_2 &:= rw \bmod q \\ v &:= (g^{u_1} y^{u_2} \bmod p) \bmod q \end{aligned}$$

Check if $r = v$

Elliptic Curve Cryptosystems

Elliptic curves defined over $GF(p)$ or $GF(2^k)$ are used in cryptography

The arithmetic of $GF(p)$ is the usual mod p arithmetic

The arithmetic of $GF(2^k)$ is similar to that of $GF(p)$, however, there are some differences

Elliptic curves over $GF(2^k)$ are more popular due to the space and time-efficient algorithms for doing arithmetic in $GF(2^k)$

Elliptic curve cryptosystems based on discrete logarithms seem to provide similar amount of security to that of RSA, but with relatively shorter key sizes

Elliptic Curves over $GF(p)$

Let $p > 3$ be a prime number and $a, b \in GF(p)$ be such that $4a^3 + 27b^2 \neq 0$ in $GF(p)$. An elliptic curve E over $GF(p)$ is defined by the parameters a and b as the set of solutions (x, y) where $x, y \in GF(p)$ to the equation

$$y^2 = x^3 + ax + b$$

together with an extra point O. The set of points E form a group with respect to the addition rules:

- $O + O = O$
- $(x, y) + O = (x, y)$
- $(x, y) + (x, -y) = O$

Elliptic Curve Addition Formulae

Addition of two points (x_1, y_1) and (x_2, y_2) with $x_1 \neq x_2$

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

Doubling of a point (x_1, y_1) with $x_1 \neq 0$

$$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$$

$$\begin{aligned} \lambda &= \frac{3x_1^2 + a}{2y_1} \\ x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

Elliptic Curves over $GF(2^k)$

A non-supersingular elliptic curve E over the field $GF(2^k)$ is defined by parameters $a, b \in GF(2^k)$ with $b \neq 0$ is the set of solutions (x, y) where $x, y \in GF(2^k)$, to the equation

$$y^2 + xy = x^3 + ax^2 + b$$

together with an extra point O. The set of points E form a group with respect to the addition rules:

- $O + O = O$
- $(x, y) + O = (x, y)$
- $(x, y) + (x, x + y) = O$

Elliptic Curves over $GF(2^k)$

Addition of two points (x_1, y_1) and (x_2, y_2) with $x_1 \neq x_2$

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

$$\begin{aligned} \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \end{aligned}$$

Doubling of a point (x_1, y_1) with $x_1 \neq 0$

$$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$$

$$\begin{aligned} \lambda &= x_1 + \frac{y_1}{x_1} \\ x_3 &= \lambda^2 + \lambda + a \\ y_3 &= x_1^2 + (\lambda + 1)x_3 \end{aligned}$$

Elliptic Curve Cryptosystems

Based on the difficulty of computing e given eP where P is a point on the curve

Example: Elliptic Curve Diffie-Hellman

Alice and Bob agree on

- the elliptic curve E
- the underlying field $GF(2^k)$ or $GF(p)$
- the generating point P with order n

Elliptic Curve Diffie-Hellman

- Alice sends $Q_A = aP$ to Bob

- Bob sends $Q_B = bP$ to Alice

- Alice computes $k = a(Q_B) = abP$

- Bob computes $k = b(Q_A) = abP$

Adversary knows P, and sees Q_A and Q_B; computing k seems to require elliptic logarithms (*Miller*, *Koblitz*, *Menezes*)

Elliptic Curve Arithmetic

Computation of eP can be performed using exponentiation algorithms

In order to compute e multiple of P we perform elliptic curve additions

An elliptic curve addition is performed by using a few *finite field* operations

Implementation of elliptic curve addition operation requires implementation of four basic finite field operations: addition, subtraction, multiplication, and inversion

Parameters

$P = (x, y)$ is the generating point expressed in affine coordinate system, where $x, y \in GF(2^k)$

e is the h-bit integer

$$e = (e_{h-1}e_{h-2} \cdots e_0)$$

where $1 \leq e \leq n - 1$

n is the order of the generating point, i.e., the integer such that $nP = O$

The size of n is approximately k bits

Projective Coordinates

The addition formulae for a non-supersingular curve over $GF(2^k)$ requires inversion of elements in $GF(2^k)$

For example, addition of two distinct points requires two multiplications and one inversion

Doubling of a point (computation of $2P$) requires three multiplications and one inversion

Inversion in $GF(2^k)$ is a relatively expensive operation

Projective coordinates allow us to eliminate the need for performing inversion (*Menezes*)

Projective Coordinates

In projective coordinates, a point on E has three coordinate values

$$(x_1 \ : \ y_1 \ : \ z_1)$$

while a point in affine coordinates requires only two values:

$$(x_1, y_1)$$

Given the distinct points P and Q expressed in projective coordinates

$$\begin{aligned} P &= (x_1 \ : \ y_1 \ : \ z_1) \\ Q &= (x_2 \ : \ y_2 \ : \ z_2) \end{aligned}$$

We compute the projective coordinates of the elliptic curve sum

$$P + Q \ = \ (x_3 \ : \ y_3 \ : \ z_3)$$

Projective Addition Formulae

The addition formulae for computing $P_1 + P_2$ is given as

$$\begin{aligned}
A &= x_2 z_1 + x_1 \\
B &= y_2 z_1 + y_1 \\
C &= A + B \\
D &= A^2(A + a z_1) + z_1 B C \\
x_3 &= AD \\
y_3 &= CD + A^2(B x_1 + A y_1) \\
z_3 &= A^3 z_1
\end{aligned}$$

This computation requires 13 multiplications and no inversions (squarings are ignored)

Projective Addition Formulae

Similarly, the addition formulae for computing $2P$ is given as

$$
\begin{aligned}
A &= x_1 z_1 \\
B &= b z_1^4 + x_1^4 \\[1ex]
x_3 &= AB \\
y_3 &= x_1^4 A + B(x_1^2 + y_1 z_1 + A) \\
z_3 &= A^3
\end{aligned}
$$

This computation requires seven multiplications and no inversions

Thus, we have eliminated the inversions at the expense of storing three $GF(2^k)$ values to represent P and performing a few more multiplications

Exponentiation Heuristics

Given the integer e, the computation of M^e or eP is an exponentiation operation

The objective is to use as few multiplications (or elliptic curve additions) as possible for a given integer e

This problem is related to *addition chains*

An addition chain is a sequence of integers

$$a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_r$$

starting from $a_0 = 1$ and ending with $a_r = e$ such that any a_k is the sum of two earlier integers a_i and a_j in the chain:

$$a_k = a_i + a_j \quad \text{for } 0 < i, j < k$$

Addition Chains

Example: $e = 55$

1 2 3 6 12 13 26 27 54 55
1 2 3 6 12 13 26 52 55
1 2 4 5 10 20 40 50 55
1 2 3 5 10 11 22 44 55

An addition chain yields an algorithm for computing M^e or eP given the integer e

$$M^1 \; M^2 \; M^3 \; M^5 \; M^{10} \; M^{11} \; M^{22} \; M^{44} \; M^{55}$$

$$P \quad 2P \quad 3P \quad 5P \quad 10P \quad 11P \quad 22P \quad 44P \quad 55P$$

The length of the chain r gives the number of operations required to compute M^e or eP

Addition Chains

Finding the shortest addition chain is an NP-complete problem (*Downey et al.*)

Upper bound: $\lfloor \log_2 e \rfloor + H(e) - 1$, where $H(e)$ is the Hamming weight of e
(the binary method, *Knuth*)

Lower bound: $\log_2 e + \log_2 H(e) - 2.13$
(*Schönhage*)

Heuristics: binary, m-ary, adaptive m-ary, sliding windows, power tree methods

Statistical methods, such as simulated annealing, can be used to produce short addition chains for certain exponents

Binary Method

A simple approach: Scan the bits of e and perform squarings and multiplications to compute M^e

Similarly, elliptic curve doublings and additions are performed in order to compute eP

There are two versions: Left-to-Right (LR) and Right-to-Left (RL) depending on the order by which the bits of exponent are scanned

The binary method is generalized to the m-ary method by scanning several bits at a time

LR Binary Method

```
Input: $M, e, n$
Output: $C := M^e \bmod n$
1.    **if** $e_{h-1} = 1$ **then** $C := M$ **else** $C := 1$
2.    **for** $i = h-2$ **downto** 0
2a.       $C := C \cdot C \pmod{n}$
2b.       **if** $e_i = 1$ **then** $C := C \cdot M \pmod{n}$
3.    **return** $C$
```

```
Input: $P, e$
Output: $Q := eP$
1.    **if** $e_{h-1} = 1$ **then** $Q := P$ **else** $Q := O$
2.    **for** $i = h-2$ **downto** 0
2a.       $Q := Q + Q$
2b.       **if** $e_i = 1$ **then** $Q := Q + P$
3.    **return** $Q$
```

RL Binary Method

```
Input: $M, e, n$
Output: $C := M^e$ mod $n$
1.   $C := 1$ ; $D := M$
2.   for $i = 0$ to $h - 2$
2a.      if $e_i = 1$ then $C := C \cdot D \pmod{n}$
2b.      $D := D \cdot D \pmod{n}$
3.   if $e_{h-1} = 1$ then $C := C \cdot D \pmod{n}$
4.   return $C$
```

```
Input: $P, e$
Output: $Q := eP$
1.   $Q := O$ ; $R := P$
2.   for $i = 0$ to $h - 2$
2a.      if $e_i = 1$ then $Q := Q + R$
2b.      $R := R + R$
3.   if $e_{h-1} = 1$ then $Q := Q + R$
4.   return $Q$
```

Example: $e = 55 = (110111)$ and $h = 6$

LR Binary Method

Step 1: $e_5 = 1 \longrightarrow C := M$

i	e_i	Step 2a (C)	Step 2b (C)
4	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
3	0	$(M^3)^2 = M^6$	M^6
2	1	$(M^6)^2 = M^{12}$	$M^{12} \cdot M = M^{13}$
1	1	$(M^{13})^2 = M^{26}$	$M^{26} \cdot M = M^{27}$
0	1	$(M^{27})^2 = M^{54}$	$M^{54} \cdot M = M^{55}$

RL Binary Method

Step 1: $C := 1$ and $D := M$

i	e_i	Step 2a (C)	Step 2b (D)
0	1	$1 \cdot M = M$	$(M)^2 = M^2$
1	1	$M \cdot M^2 = M^3$	$(M^2)^2 = M^4$
2	1	$M^3 \cdot M^4 = M^7$	$(M^4)^2 = M^8$
3	0	M^7	$(M^8)^2 = M^{16}$
4	1	$M^7 \cdot M^{16} = M^{23}$	$(M^{16})^2 = M^{32}$

Step 3: $e_5 = 1 \longrightarrow C := M^{23} \cdot M^{32} = M^{55}$

Comparing LR and RL Methods

Both methods require $(h-1)$ squarings and an average of $0.5(h-1)$ multiplications

- LR requires two registers: M and C

- RL requires three registers: M, C, and D

  If the value of M is not needed thereafter, two registers suffice: M and C

Multiplication and squaring operations in RL method are independent and thus can be parallelized

The m-ary Method

By scanning the bits of e

2 at a time: The quaternary method

3 at a time: The octal method

etc.

d at a time: The m-ary method ($m = 2^d$)

Consider the quaternary method:

$$e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$$

- some pre-processing required

- at each step two squarings performed

Quaternary Method Example

bits	j	M^j
00	0	1
01	1	M
10	2	$M \cdot M = M^2$
11	3	$M^2 \cdot M = M^3$

$$e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$$

bits	Step 2a	Step 2b
11	M^3	M^3
11	$(M^3)^4 = M^{12}$	$M^{12} \cdot M^3 = M^{15}$
10	$(M^{15})^4 = M^{60}$	$M^{60} \cdot M^2 = M^{62}$
10	$(M^{62})^4 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The number of multiplications: $2+6+3 = 11$

The binary method requires $7+5 = 12$ multiplications

m-ary Method

The average number of multiplications plus squarings required by the m-ary method:

- pre-processing multiplications: $2^d - 2$
- squarings: $(\frac{k}{d} - 1) \cdot d = k - d$
- multiplications: $\frac{m-1}{m} \cdot (\frac{k}{d} - 1)$
- There is an optimal d for every k

k	BM	MM	d^*	Savings %
32	47	43	2,3	8.5
64	95	85	3	10.5
128	191	167	3,4	12.6
256	383	325	4	15.1
512	767	635	5	17.2
1024	1535	1246	5	18.8
2048	3071	2439	6	20.6

- Asymptotic savings: 33%

Sliding Window Techniques

Based on adaptive (data-dependent) m-ary partitioning of the exponent (*Bos & Coster*, *Yacobi*, *Koç*)

- Constant length nonzero windows

  Partition the exponent into zero words of any length and nonzero words of length d

- Variable length nonzero windows

  Partition the exponent into zero words of length at least q and nonzero words of length at most d

Constant Length Nonzero Windows

Example: For $d = 3$, we partition

$$e = 3665 = (111001010001)_2$$

as

$$\underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$$

First compute M^j for odd $j \in [1, m-1]$

bits	j	M^j
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M \cdot M^2 = M^3$
101	5	$M^3 \cdot M^2 = M^5$
111	7	$M^5 \cdot M^2 = M^7$

Constant Length Nonzero Windows

$$3665 = \underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$$

bits	Step 2a	Step 2b
111	M^7	M^7
00	$(M^7)^4 = M^{28}$	M^{28}
101	$(M^{28})^8 = M^{224}$	$M^{224} \cdot M^5 = M^{229}$
0	$(M^{229})^2 = M^{458}$	M^{458}
001	$(M^{458})^8 = M^{3664}$	$M^{3664} \cdot M^1 = M^{3665}$

Average number of multiplications

	m-ary		CLNW		for d^*
k	d^*	T/k	d^*	T_1/k	%
128	4	1.31	4	1.22	6.6
256	4	1.27	5	1.20	5.2
512	5	1.24	5	1.19	4.4
1024	5	1.22	6	1.17	4.1
2048	6	1.19	7	1.15	3.2

Variable Length Nonzero Windows

Example: $d = 5$ and $q = 2$

$\underline{101}$ 0 $\underline{11101}$ 00 $\underline{101}$

$\underline{10111}$ 000000 $\underline{1}$ 00 $\underline{111}$ 000 $\underline{1011}$

Example: $d = 10$ and $q = 4$

$\underline{1011011}$ 0000 $\underline{11}$ 0000

$\underline{11110111}$ 00 $\underline{1111110101}$ 0000 $\underline{11011}$

Average number of multiplications

	m-ary		VLNW		for q^*
k	d^*	T/k	d^*	T_2/k	%
128	4	1.31	4	1.20	7.8
256	4	1.27	4	1.18	6.8
512	5	1.24	5	1.16	6.4
1024	5	1.22	6	1.15	5.8
2048	6	1.19	6	1.13	5.0

Addition-Subtraction Chains

An addition-subtraction chain is a sequence of integers

$$a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_r$$

starting from $a_0 = \pm 1$ and ending with $a_r = e$ such that any a_k is the sum or the difference of two earlier integers a_i and a_j in the chain:

$$a_k = a_i \pm a_j \quad \text{for } 0 < i, j < k$$

Example: $e = 55$

$$\pm 1 \quad 2 \quad 4 \quad 8 \quad 7 \quad 14 \quad 28 \quad 56 \quad 55$$

An addition-subtraction chain yields an algorithm for computing M^e or eP given the integer e

However, it requires negative powers of M or negative multiples of P

Recoding Technique

We obtain a sparse signed-digit representation of the exponent with digits $\{0, 1, -1\}$, e.g.,

$$\begin{aligned} 30 = (011110) &= 2^4 + 2^3 + 2^2 + 2^1 \\ 30 = (1000\bar{1}0) &= 2^5 - 2^1 \end{aligned}$$

This method needs $M^{-1} \pmod{n}$ as input

Recoding Binary Method

Input: M, M^{-1}, e, n

Output: $C := M^e \bmod n$

```
0.    Obtain signed-digit recoding $f$ of $e$
1.    **if** $f_h = 1$ **then** $C := M$ **else** $C := 1$
2.    **for** $i = h - 1$ **downto** 0
2a.       $C := C \cdot C \pmod{n}$
2b.       **if** $f_i = 1$ **then** $C := C \cdot M \pmod{n}$
          **if** $f_i = \bar{1}$ **then** $C := C \cdot M^{-1} \pmod{n}$
3.    **return** $C$
```

Recoding Technique Example

Example: $e = 119 = (1110111)$

Binary method: $6 + 5 = 11$ multiplications

Exponent:	01110111
Recoded exponent:	$1000\bar{1}00\bar{1}$

f_i	Step 2a (C)	Step 2b (C)
1	M	M
0	$(M)^2 = M^2$	M^2
0	$(M^2)^2 = M^4$	M^4
0	$(M^4)^2 = M^8$	M^8
$\bar{1}$	$(M^8)^2 = M^{16}$	$M^{16} \cdot M^{-1} = M^{15}$
0	$(M^{15})^2 = M^{30}$	M^{30}
0	$(M^{30})^2 = M^{60}$	M^{60}
$\bar{1}$	$(M^{60})^2 = M^{120}$	$M^{120} \cdot M^{-1} = M^{119}$

The number of multiplications: $7 + 2 = 9$ Requires M^{-1} which is costly

Applications to Elliptic Curves

Addition-subtraction chains are suitable for elliptic curves since computing $-P$ is trivial

For non-supersingular elliptic curves over $GF(2^k)$, if $P = (x, y)$, then $-P = (x, x + y)$

For elliptic curves over $GF(p)$, if $P = (x, y)$, then $-P = (x, -y)$

```
Input: $P, -P, e$
Output: $Q := eP$
0.    Obtain signed-digit recoding $f$ of $e$
1.    if $f_h = 1$ then $C := M$ else $C := 1$
2.    for $i = h - 1$ downto 0
2a.       $Q := Q + Q$
2b.       if $f_i = 1$ then $Q := Q + P$
          if $f_i = \bar{1}$ then $Q := Q + (-P)$
3.    return $Q$
```

Efficiency of Recoding Techniques

The canonical recoding algorithm can be used to optimally encode the exponent with the digit set $\{0, 1, -1\}$

a_i	e_{i+1}	e_i	a_{i+1}	f_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	$\bar{1}$
1	0	0	0	1
1	0	1	1	0
1	1	0	1	$\bar{1}$
1	1	1	1	0

For example, $e = 3038$ is encoded as

$$\begin{array}{rcl} e & = & (0101111011110) \\ f & = & (10\bar{1}0000\bar{1}000\bar{1}0) \end{array}$$

requiring three multiplications instead of nine (in addition to the squarings)

If the digits of the canonically recoded exponent are scanned d at a time, we obtain the canonically recoding m-ary method (*Eğecioğlu & Koç*)

This method offers savings in terms of the required number of multiplications

For example, the standard binary method requires an average of 0.500$(h-1)$ multiplications while the canonically recoding binary method requires an average of 0.333$(h-1)$ multiplications, in addition to the squarings

d	standard	canonical
1	0.500	0.333
2	0.375	0.333
3	0.292	0.278
4	0.234	0.229
5	0.194	0.192
6	0.164	0.163
7	0.142	0.141
8	0.124	0.124

Arithmetic Operations

RSA requires mod n arithmetic: addition, subtraction, and multiplications

Elliptic curves require $GF(p)$ or $GF(2^k)$ arithmetic: addition, subtraction, multiplication, and inversion

The mod n arithmetic is the same as $GF(p)$ arithmetic (except inversion – which is not used in RSA)

We will study algorithms for

- addition and multiplication in $GF(p)$

- addition, multiplication, and inversion in $GF(2^k)$

Modular Addition

The computation of $S = A + B \bmod n$

Add and Reduce:

Given $A, B < n$
Compute $S' = A + B$
Compute $S'' = S' - n$
If $S'' \geq 0$, then $S = S'$ else $S = S''$

Requires fast sign detection: Is $S'' \geq 0$?

Omura's Method:

Given $A, B < 2^k$ $\qquad (A, B$ can be $\geq n)$
Compute $S' = A + B$
If there is a carry out,
        then $S = S' + m$
        else $S = S'$

Correction factor: $m = 2^k - n$ (precomputed)

Modular Addition

Carry out implies that

$$(S') = A + B \geq 2^k$$

Thus

$$S' = A + B - 2^k$$

is computed (by ignoring the carry-out bit)

The result is corrected by adding m to S'

$$\begin{aligned} S &= S' + m \\ &= A + B - 2^k + m \\ &= A + B - n \end{aligned}$$

A temporary value may be larger than n

However it is always less than 2^k

Whenever it exceeds 2^k, we ignore the carry out and perform a correction

Example

$n = 39$, thus $m = 64 - 39 = 25 = (011001)$

$$\begin{array}{lcrrl}
A & = & 40 = & (101000) & \\
B & = & 30 = & (011110) & \\
S' & = & A + B = & 1(000110) & \text{carry out} \\
m & = & & (011001) & \\
S & = & S' + m = & (011111) & \text{correction}
\end{array}$$

$$\begin{array}{lcrrl}
A & = & 23 = & (010111) & \\
B & = & 26 = & (011010) & \\
S' & = & A + B = & 0(110001) & \text{no carry out} \\
S & = & S' = & (110001) &
\end{array}$$

After all additions are completed, a final result that is out of range can be corrected by adding m:

$$\begin{array}{lcrr}
S & = & & (110001) \\
m & = & & (011001) \\
S & = & S + m = & 1(001010) \\
S & = & & (001010)
\end{array}$$

Modular Multiplication

Given $A, B < n$, compute $P = A \cdot B \bmod n$

Methods:

- Multiply and reduce:

  Multiply: $P' = A \cdot B$ ($2k$-bit number)

  Reduce: $P = P' \bmod n$ (k-bit number)

- Interleave multiply and reduce steps

- *Montgomery*'s method

- *Brickell*'s method

Interleaving Multiply and Reduce

The product P' can be written as

$$\begin{aligned} P' &= A \cdot B = A \cdot \sum_{i=0}^{k-1} B_i 2^i = \sum_{i=0}^{k-1} (A \cdot B_i) 2^i \\ &= 2(\cdots 2(2(0 + A \cdot B_{k-1}) + A \cdot B_{k-2}) + \cdots) + A \cdot B_0 \end{aligned}$$

This formulation yields the shift-add multiplication algorithm (*Blakley*)

We also reduce the partial product modulo n at each step:

```
1.     $P := 0$
2.     for $i = k - 1$ downto 0
2a.        $P := 2P + A \cdot B_i$
2b.        $P := P \bmod n$
3.     return $P$
```

Interleaving Multiply and Reduce

Assuming that $A, B, P < n$, we have

$$\begin{aligned} P &:= 2P + A \cdot B_j \\ &\leq 2(n-1) + (n-1) \;=\; 3n - 3 \end{aligned}$$

Thus, at most two subtractions are needed to reduce P to the range $0 \leq P < n$. We can use

$$P' := P - n \text{ ; } \textbf{if } P' \geq 0 \textbf{ then } P = P'$$
$$P' := P - n \text{ ; } \textbf{if } P' \geq 0 \textbf{ then } P = P'$$

Addition and subtraction steps need to be performed faster

Carry propagate adder gives $O(k)$ delay; Omura's method can be used to avoid unnecessary subtractions:

```
2a.  $P := 2P$
2b.  if carry-out then $P := P + m$
2c.  $P := P + A \cdot B_j$
2d.  if carry-out then $P := P + m$
```

Carry save adder gives $O(1)$ delay; fast sign detection is needed to decide if the partial product needs to be reduced modulo n

```
2a.  $(C, S) := 2C + 2S + A \cdot B_i$
2b.  $(C', S') := C + S - n$
2c.  if SIGN $\geq$ 0 then $(C, S) := (C', S')$
```

Function SIGN estimates the sign of $C' + S'$

Steps 2b & 2c may be repeated several times

Montgomery's Method

This method replaces division by n operation with division by $r = 2^k$

Assuming n is a k-bit odd integer, i.e.,

$$2^{k-1} < n < 2^k$$

we assign $r = 2^k$

Now, we map the integers $a \in [0, n-1]$ to the integers $\bar{a} \in [0, n-1]$ using the one-to-one mapping

$$\bar{a} = a \cdot r \pmod{n}$$

We call $\bar{a}$ the n-residue of a

Montgomery's Method

The Montgomery product of two n-residues is defined as

$$\text{MonPro}(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

where r^{-1} is the inverse of r modulo n

It has two important properties:

- If $c = a \cdot b \bmod n$, then $\bar{c} = \text{MonPro}(\bar{a}, \bar{b})$

$$\begin{aligned} \bar{c} &= a \cdot b \cdot r^{-1} \pmod{n} \\ &= (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \pmod{n} \\ &= \text{MonPro}(\bar{a}, \bar{b}) \end{aligned}$$

- For any $\bar{c}$, we have $c = \text{MonPro}(\bar{c}, 1)$

$$\begin{aligned} c &= (c \cdot r) \cdot 1 \cdot r^{-1} \pmod{n} \\ &= \text{MonPro}(\bar{c}, 1) \end{aligned}$$

Montgomery Product Computation

In order to compute

$$\text{MonPro}(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

we need an additional quantity n' such that

$$r \cdot r^{-1} - n \cdot n' = 1$$

The extended Euclidean algorithm computes r^{-1} and n'

```
function MonPro($\bar{a}, \bar{b}$)
1.    $t := \bar{a} \cdot \bar{b}$
2.    $m := t \cdot n' \bmod r$
3.    $u := (t + m \cdot n)/r$
4.    if $u \geq n$ then return $u - n$
                    else return $u$
```

This subroutine requires only modulo r arithmetic, which is easily accomplished

Montgomery Product Computation

The pre-processing operations

- computation of n'
- conversion from ordinary to n-residue
- conversion from n-residue to ordinary

are time-consuming

Montgomery's method is not suitable for a single modular multiplication

However, it is very suitable for modular exponentiation

Modular Exponentiation

```
function ModExp($M, e, n$) { $n$ is odd }
    Compute $n'$ using Euclid's algorithm
    $\bar{M} := M \cdot r \bmod n$
    $\bar{C} := 1 \cdot r \bmod n$
    for $i = h - 1$ down to 0 do
        $\bar{C} :=$ MonPro($\bar{C}, \bar{C}$)
        if $e_i = 1$ then $\bar{C} :=$ MonPro($\bar{C}, \bar{M}$)
    $C :=$ MonPro($\bar{C}, 1$)
    return $C$
```

The above function uses the binary method. However, any exponentiation algorithm can be used

Montgomery Example

Computation of $3^{50} \pmod{55}$

$r = 2^k = 64$. Also

$$64 \cdot 49 - 57 \cdot 55 = 1$$

which gives $r^{-1} = 49$ and $n' = 57$.

$M = 3$; $\bar{M} := M \cdot r \pmod{n}$

which gives $\bar{M} = 3 \cdot 64 \pmod{55} = 27$

$C = 1$; $\bar{C} := C \cdot r \pmod{n}$

which gives $\bar{C} = 1 \cdot 64 \pmod{55} = 9$

Thus, $\bar{M} = 27$ and $\bar{C} = 9$

e_i	Step 5	Step 6
1	MonPro$(9,9) = 9$	MonPro$(9,27) = 27$
1	MonPro$(27,27) = 26$	MonPro$(26,27) = 23$
0	MonPro$(23,23) = 16$	
0	MonPro$(16,16) = 4$	
1	MonPro$(4,4) = 14$	MonPro$(14,27) = 42$
0	MonPro$(42,42) = 31$	

Step 7: $C = \text{MonPro}(31,1) = 34$

- Computation of MonPro$(27,27)$:

$t = 27 \cdot 27 = 729$
$m = 729 \cdot 57 \pmod{64} = 17$
$u = (729 + 17 \cdot 55)/64 = 1664/64 = 26$

- Computation of MonPro$(31,1)$:

$t = 31 \cdot 1 = 31$
$m = 31 \cdot 57 \pmod{64} = 39$
$u = (31 + 39 \cdot 55)/64 = 2176/64 = 34$

Montgomery Algorithms

Separated Operand Scanning (SOS)

First computes $t = a \cdot b$ and then interleaves the computations of $m = t \cdot n' \bmod r$ and $u = (t + m \cdot n)/r$. Squaring can be optimized.

SOS requires $2s + 2$ words of space

Finely Integrated Product Scanning (FIPS)

Interleaves computation of $a \cdot b$ and $m \cdot n$ by scanning the words of m

It uses the same space to keep m and u, reducing the temporary space to $s + 3$ words

Finely Integrated Operand Scanning (FIOS)

The computation of $a \cdot b$ and $m \cdot n$ is performed in a single loop

FIOS also requires $s + 3$ words of space

Montgomery Algorithms

Coarsely Integrated Hybrid Scanning (CIHS)

The computation of $a \cdot b$ is split into 2 loops, and the second loop is interleaved with the computation of $m \cdot n$

CIHS also requires $s + 3$ words of space

Coarsely Integrated Operand Scanning(CIOS)

Improves the SOS method by integrating the multiplication and reduction steps. It alternates between iterations of the outer loops for multiplication and reduction

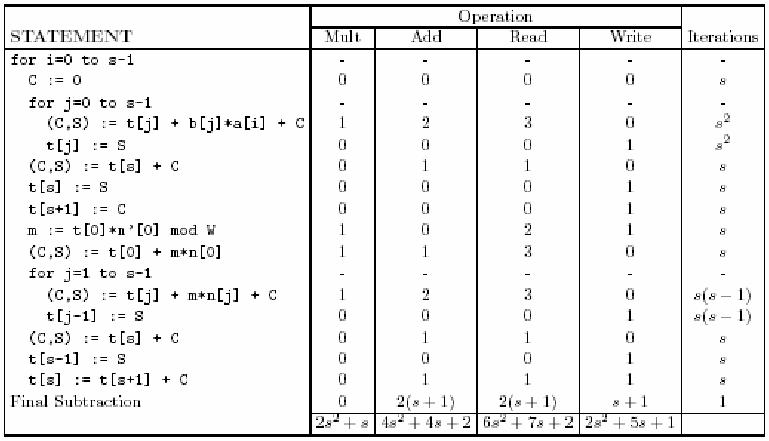
CIOS also requires $s + 3$ words of space

	Add	Read/Write	Space
SOS	$4s^2+4s+2$	$8s^2+13s+5$	$2s+2$
FIPS	$6s^2+2s+2$	$14s^2+16s+3$	$s+3$
FIOS	$5s^2+3s+2$	$10s^2+9s+3$	$s+3$
CIHS	$4s^2+4s+2$	$9.5s^2+11.5s+3$	$s+3$
CIOS	$4s^2+4s+2$	$8s^2+12s+3$	$s+3$

All methods require $2s^2 + s$ multiplications

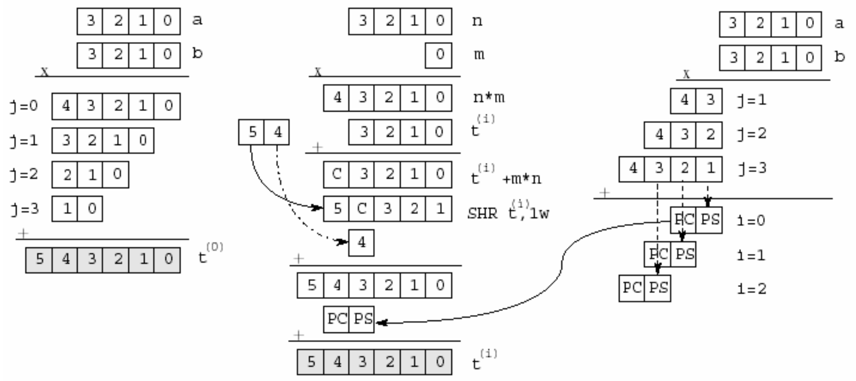
CIOS

Table 2. Calculating the operations of the CIOS method.

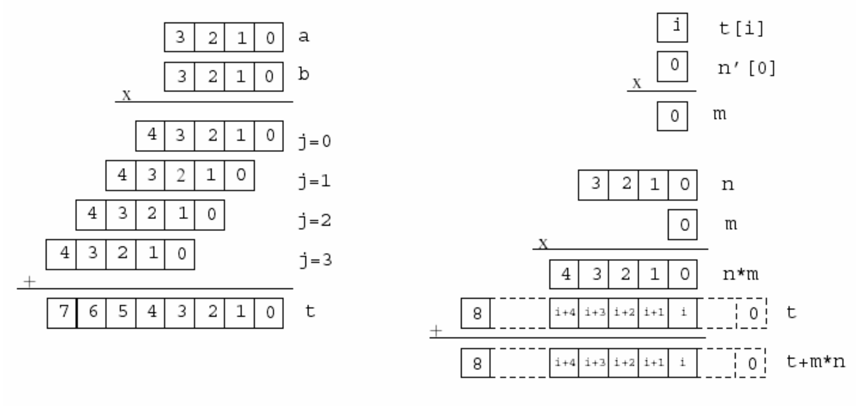


STATEMENT	Operation				Iterations
	Mult	Add	Read	Write	
for i=0 to s-1	-	-	-	-	-
C := 0	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := t[j] + b[j]*a[i] + C	1	2	3	0	s^2
t[j] := S	0	0	0	1	s^2
(C,S) := t[s] + C	0	1	1	0	s
t[s] := S	0	0	0	1	s
t[s+1] := C	0	0	0	1	s
m := t[0]*n'[0] mod W	1	0	2	1	s
(C,S) := t[0] + m*n[0]	1	1	3	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := t[j] + m*n[j] + C	1	2	3	0	$s(s-1)$
t[j-1] := S	0	0	0	1	$s(s-1)$
(C,S) := t[s] + C	0	1	1	0	s
t[s-1] := S	0	0	0	1	s
t[s] := t[s+1] + C	0	1	1	1	s
Final Subtraction	0	$2(s+1)$	$2(s+1)$	$s+1$	1
	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	

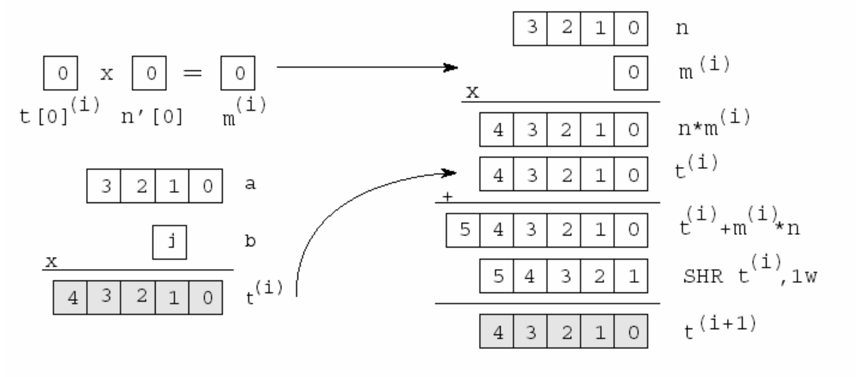
CIHS



SOS


3 2 1 0 a
3 2 1 0 b
x
4 3 2 1 0 j=0
4 3 2 1 0 j=1
4 3 2 1 0 j=2
4 3 2 1 0 j=3
+
7 6 5 4 3 2 1 0 t
i t[i]
x 0 n' [0]
0 m
3 2 1 0 n
0 m
x
4 3 2 1 0 n*m
8 i+4 i+3 i+2 i+1 i 0 t
+
8 i+4 i+3 i+2 i+1 i 0 t+m*n

FIOS


0 x 0 = 0
t [0](i) n' [0] m(i)
3 2 1 0 a
j b
x
4 3 2 1 0 t(i)
3 2 1 0 n
0 m(i)
x
4 3 2 1 0 n*m(i)
4 3 2 1 0 t(i)
+
5 4 3 2 1 0 t(i)+m(i)*n
5 4 3 2 1 SHR t(i),1w
4 3 2 1 0 t(i+1)

Derivation of CIOS

The binary Montgomery algorithm:

$$
\begin{aligned}
u &= \text{MP}(a, b) \\
&= a \cdot b \cdot r^{-1} \pmod{n} \\
&= a \cdot b \cdot 2^{-k} \pmod{n} \\
&= (\sum_{i=0}^{k-1} a_i 2^i) \cdot b \cdot 2^{-k} \pmod{n} \\
&= (\sum_{i=0}^{k-1} a_i 2^{i-k}) \cdot b \pmod{n}
\end{aligned}
$$

$$u = (a_0 2^{-k} + a_1 2^{-k+1} + \cdots + a_{k-1} 2^{-1}) \cdot b \bmod n$$

```
1.    $u := 0$
2.    for $i = 0$ to $k - 1$
2a.       $u := u + a_i b$
2b.       if $u$ is odd then $u := u + n$
3.        $u := u/2$
```

Derivation of CIOS

- Compute $u := u + a_i \cdot b$

- Add an integer (m) multiple of n to u so that the LSB of $u + m \cdot n$ is zero

We have

$$u + m \cdot n = u_0 + m \cdot n_0 = 0 \pmod 2$$

and thus,

$$m := u_0 \cdot (-n_0^{-1}) \pmod 2$$

n is odd, and therefore $(-n_0^{-1}) = 1 \pmod 2$

```
$u := 0$
for $i = 0$ to $k - 1$
    $u := u + a_i \cdot b$
    $m := u_0 \cdot (-n_0^{-1}) \pmod 2$
    $u := u + m \cdot n$
    $u := u/2$
```

Derivation of CIOS

$k = sw$, where w is the wordsize

```
1.    $u := 0$
2.    **for** $i = 0$ **to** $s - 1$
2a.       $u := u + a_i \cdot b$
2b        $m := u_0 \cdot (-n_0^{-1}) \pmod{2^w}$
2c.       $u := u + m \cdot n$
3.        $u := u / 2^w$
```

Since $r \cdot r^{-1} - n' \cdot n = 1$ and $r = 2^k$, we have

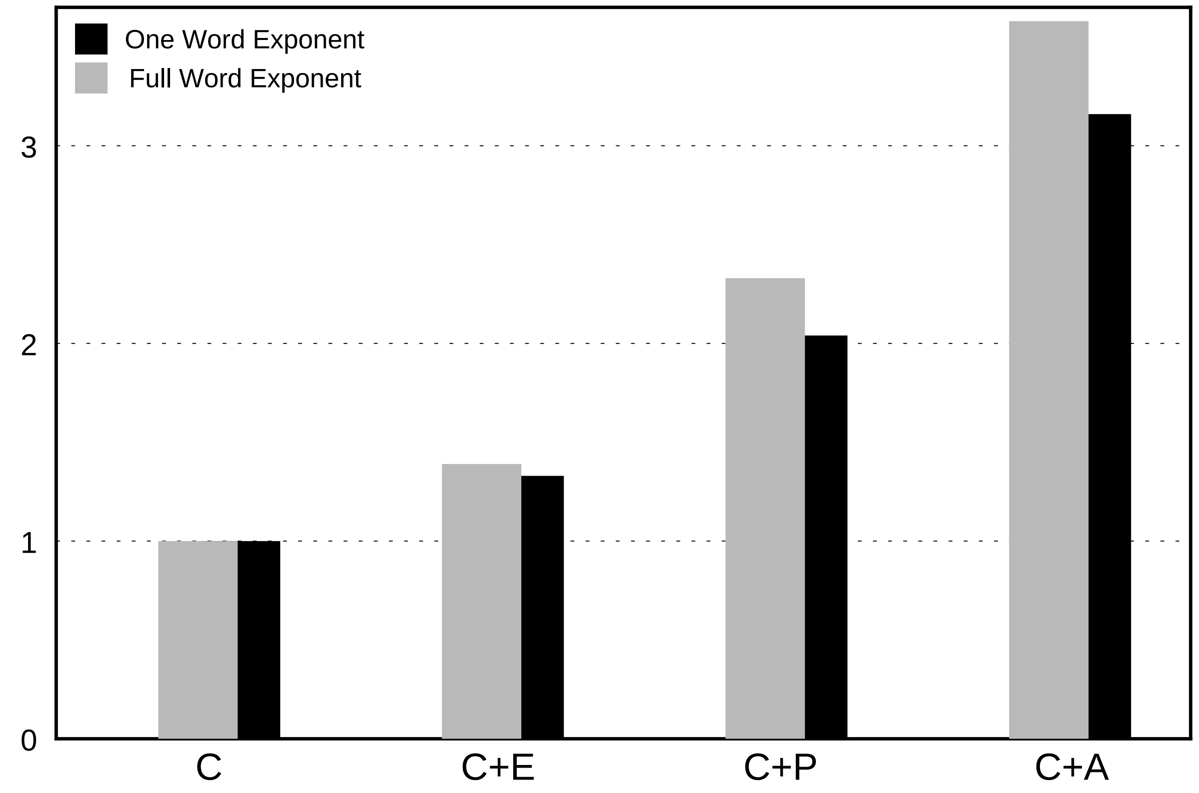
$$\begin{aligned} 2^{sw} \cdot 2^{sw} - n' \cdot n &= 1 \pmod{2^w} \\ -n_0' \cdot n_0 &= 1 \pmod{2^w} \end{aligned}$$

therefore $(-n_0^{-1}) = n_0'$

High-Speed Software for Crypto

Compiler vs Compiler
Compiler vs Assembler
Cost vs Speed

- ANSI C implementation
- C + Extended Types (__int64, long long)
- C + Primitive Operations in Assembler
- C + Basic Functions in Assembler



MMX Implementation

MMX Enhancements

- Register Usage
- Instruction Usage
- Selection of Algorithm

  SOS, FIPS, **FIOS**, CIHS, CIOS

```
for $j = 0$ to $s - 1$
    $(C, S) := t_j + a_j \cdot b_i + C$
    ADD $(t_{j+1}, C)$
    $(C, S) := S + m \cdot n_j$
    $t_{j-1} := S$
```

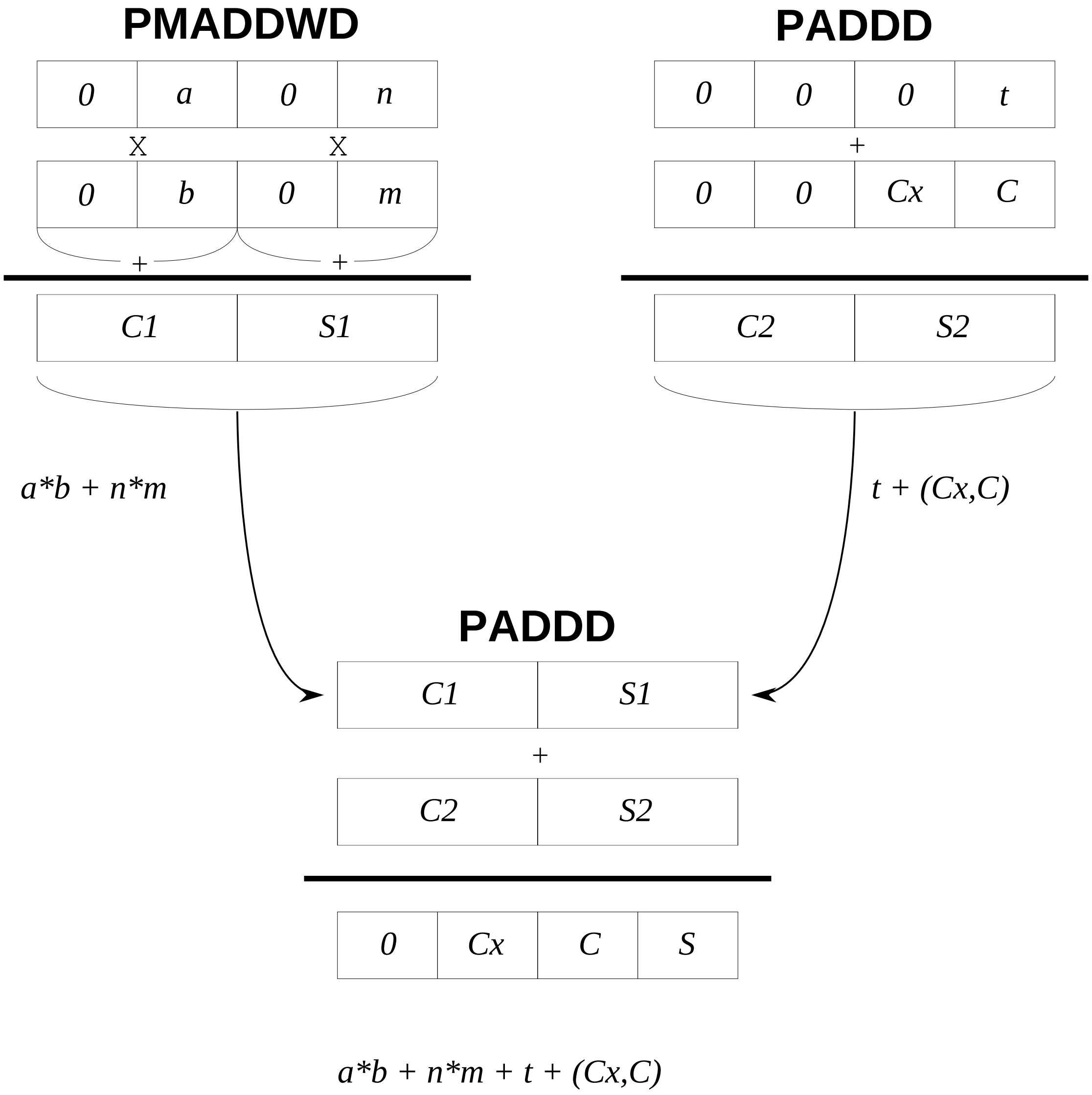
FIOS on MMX

FIOS Pseudocode for MMX:

```
for $j = 0$ to $s - 1$
    $(Cx, C, S) := t_j + a_j \cdot b_i + m \cdot n_j + C + Cx \cdot 2^w$
    $t_{j-1} := S$
```

Operation	**Instruction**
$(C1, S1) := a \cdot b + m \cdot n$	PMADDWD
$(C2, S2) := (Cx, C) + t$	PADDD
$(C, S) := (C1, S1) + (C2, S2)$	PADDD

MMX Instruction Usage



Algorithms for Hardware

- Integer addition

- Addition in $GF(2^k)$

- High-radix multiplication

- Montgomery example

- Squaring in $GF(2^k)$

Integer Addition

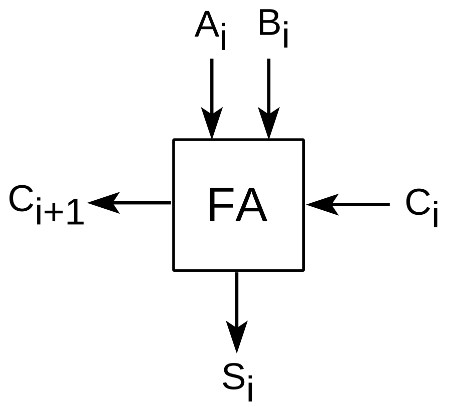
$$\begin{array}{cccccc} & A_{k-1} & A_{k-2} & \cdots & A_1 & A_0 \\ + & B_{k-1} & B_{k-2} & \cdots & B_1 & B_0 \\ \hline C_k & S_{k-1} & S_{k-2} & \cdots & S_1 & S_0 \end{array}$$

- Carry propagate adder
- Carry look-ahead adder
- Carry save adder; Brickell's algorithm
- Carry completion sensing adder

See the computer arithmetic books by *Waser & Flynn*, *Hwang*, *Swartzlander*, and *Koren* for more information

Carry Propagate Adder

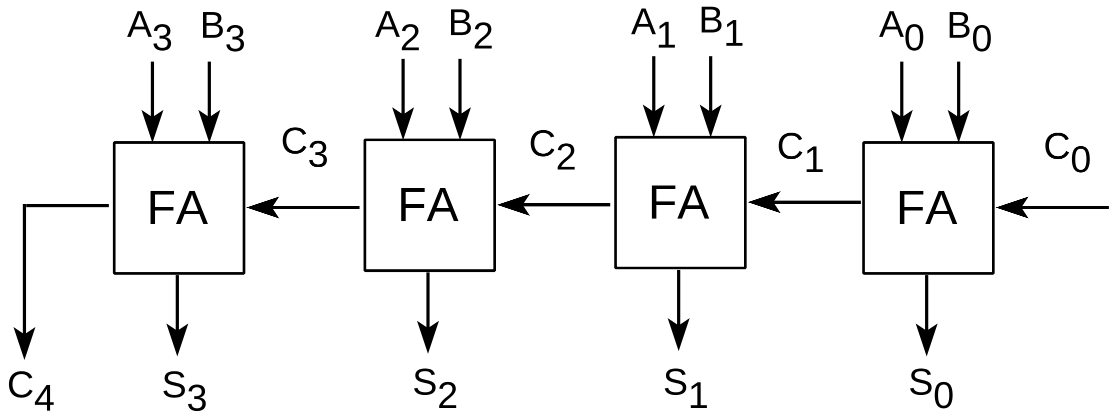
Design a full adder box: FA



$$C_{i+1} = A_iB_i+A_iC_i+B_iC_i$$

$$S_i = A_i \text{ xor } B_i \text{ xor } C_i$$

Topology:



Properties of CPA

- Total delay = k × FA delay

- Total area = k × FA area

- Scales up easily for large k

- Subtraction is easy: Use 2's complement arithmetic

- Sign detection is easy: MSB gives the sign

Carry Look-Ahead Adder

Compute C_is in advance using more logic

Then use C_is to compute S_is in parallel

Let $G_i = A_iB_i$ and $P_i = A_i + B_i$

$$\begin{aligned}
C_1 &= A_0B_0 + C_0(A_0 + B_0) \\
&= G_0 + C_0P_0 \\
C_2 &= G_1 + C_1P_1 \\
&= G_1 + G_0P_1 + C_0P_0P_1 \\
C_3 &= G_2 + C_2P_2 \\
&= G_2 + G_1P_2 + G_0P_1P_2 + C_0P_0P_1P_2 \\
C_4 &= G_3 + C_3P_3 \\
&= G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 \\
&\quad + C_0P_0P_1P_2P_3
\end{aligned}$$

C_i is a function of C_0 and $G_0, G_1, \ldots, G_i$ and $P_0, P_1, \ldots, P_i$

Properties of CLA

- The total delay is $O(\log k)$

- The total area is essentially $O(k)$ using parallel prefix circuits
  (*Ladner & Fischer*, *Brent & Kung*)

- A complete CLA is not cost-effective for large k (> 256)

- By grouping G and P functions, larger CLAs can be designed

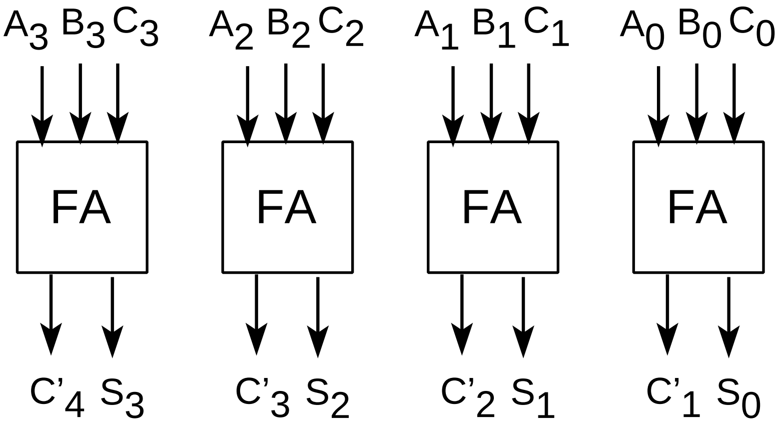
- Even with grouping, design of a 1024-bit adder may not be feasible or cost-effective

Carry Save Adder

Input: three k-bit numbers A, B, and C

Output: two k-bit numbers C' and S such that

$$C' + S = A + B + C$$



Example:

$$\begin{array}{rcrr}
A & = & 40 & 101000 \\
B & = & 25 & 011001 \\
C & = & 20 & 010100 \\
\hline
S & = & 37 & 100101 \\
C' & = & 48 & 011000
\end{array}$$

Properties of CSA

- The total delay is $O(1)$ (single FA delay)
- The total area is k × FA area
- Scales up easily for large k
- Subtraction is easy: Use 2's complement arithmetic
- Sign detection is HARD

Sign Detection Problem in CSA

Numbers are represented in pairs (C, S)

The value of the number is $C + S$

Unless the addition is performed in full length, the correct sign may never be determined

Example:

A	$=$	-18	101110	
B	$=$	19	010011	
C	$=$	6	000110	
S	$=$	-5	111011	
C'	$=$	12	000110	
			1	(1 MSB)
			11	(2 MSB)
			000	(3 MSB)
			0001	(4 MSB)
			00011	(5 MSB)
			000111	(6 MSB)

A Sign Estimation Algorithm

We add the most significant t bits of C and S to estimate the sign of $C+S$
(*Koç & Hung*)

Example:

$$\begin{aligned} C &= 011110 \\ S &= 001010 \\ \text{ESIGN(1)} &= \underline{0} \\ \text{ESIGN(2)} &= \underline{0}1 \\ \text{ESIGN(3)} &= \underline{1}00 \\ \text{ESIGN(4)} &= \underline{1}001 \\ \text{ESIGN(5)} &= \underline{1}0100 \\ \text{ESIGN(6)} &= \underline{1}01000 \quad \text{(exact)} \end{aligned}$$

If the most significant t bits used in estimation, the final result is bounded as

$$C+S < n + 2^{k-t}$$

Brickell's Algorithm

Iterates CSA one more level to reduce multiplication complexity

Two-level CSA: Carry delayed adder (CDA)

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \\ C_{i+1} &= A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i \\ T_i &= S_i \oplus C_i \\ D_{i+1} &= S_i \cdot C_i \end{aligned}$$

Note that $D_0 = 0$ and

$$\begin{aligned} T_i \cdot D_{i+1} &= (S_i \oplus C_i) \cdot S_i \cdot C_i \\ &= (\bar{S}_i \cdot C_i + S_i \cdot \bar{C}_i) \cdot S_i \cdot C_i \\ &= 0 \end{aligned}$$

The pair (T, D) is a carry delayed number

Example: Let $A = 40$, $B = 25$, and $C = 20$

These three numbers pass through a CSA, giving S and C as

$$\begin{array}{ccccc}
A & = & 40 & = & 0101000 \\
B & = & 25 & = & 0011001 \\
C & = & 20 & = & 0010100 \\
\hline
S & = & 37 & = & 0100101 \\
C & = & 48 & = & 0110000
\end{array}$$

From S and C, we compute T and D as

$$\begin{array}{ccccc}
S & = & 37 & = & 0100101 \\
C & = & 48 & = & 0110000 \\
\hline
T & = & 21 & = & 0010101 \\
D & = & 64 & = & 1000000 \\
\hline
\end{array}$$

As expected, $T_i \cdot D_{i+1} = 0$

Let A be a carry delayed integer:

$$A \;=\; \sum_{i=0}^{k-1} (T_i + D_i) \cdot 2^i$$

The product $P = A \cdot B$ can be computed by summing the terms:

$$\begin{array}{c}
(T_0 \cdot B + D_0 \cdot B) \cdot 2^0 \\
(T_1 \cdot B + D_1 \cdot B) \cdot 2^1 \\
(T_2 \cdot B + D_2 \cdot B) \cdot 2^2 \\
\vdots \\
(T_{k-1} \cdot B + D_{k-1} \cdot B) \cdot 2^{k-1}
\end{array}$$

Since $D_0 = 0$, we rearrange to obtain

$$\begin{array}{c}
2^0 \cdot T_0 \cdot B + 2^1 \cdot D_1 \cdot B \\
2^1 \cdot T_1 \cdot B + 2^2 \cdot D_2 \cdot B \\
2^2 \cdot T_2 \cdot B + 2^3 \cdot D_3 \cdot B \\
\vdots \\
2^{k-2} \cdot T_{k-2} \cdot B + 2^{k-1} \cdot D_{k-1} \cdot B \\
2^{k-1} \cdot T_{k-1} \cdot B
\end{array}$$

Also recall that either T_i or D_{i+1} is ZERO

Thus, each step requires a shift of B and addition of at most TWO carry delayed integers:

EITHER:
$(P_d, P_t) := (P_d, P_t) + 2^i \cdot T_i \cdot B$

OR:
$(P_d, P_t) := (P_d, P_t) + 2^{i+1} \cdot D_{i+1} \cdot B$

After k steps $P = (P_d, P_t)$ is obtained. In order to obtain $P \pmod{n}$, we perform reduction:

If $P \geq 2^{k-1} \cdot n$ then $P := P - 2^{k-1} \cdot n$
If $P \geq 2^{k-2} \cdot n$ then $P := P - 2^{k-2} \cdot n$
If $P \geq 2^{k-3} \cdot n$ then $P := P - 2^{k-3} \cdot n$
$\vdots$
If $P \geq n$ then $P := P - n$

We can also reverse the steps to obtain:

$$
\begin{aligned}
P &= T_{k-1} \cdot B \cdot 2^{k-1} \\
P &= P + T_{k-2} \cdot B \cdot 2^{k-2} + D_{k-1} \cdot B \cdot 2^{k-1} \\
P &= P + T_{k-3} \cdot B \cdot 2^{k-3} + D_{k-2} \cdot B \cdot 2^{k-2} \\
&\vdots \\
P &= P + T_1 \cdot B \cdot 2^1 + D_2 \cdot B \cdot 2^2 \\
P &= P + T_0 \cdot B \cdot 2^0 + D_1 \cdot B \cdot 2^1
\end{aligned}
$$

Also, multiplication steps can be interleaved with reduction steps

To perform reduction, the sign of $P - 2^i \cdot n$ needs to be determined (estimated)

Brickell's solution is essentially a combination of SIGN ESTIMATION and Omura's method of CORRECTION

Allow enough bits for P, and whenever P exceeds 2^k, add $m = 2^k - n$ to correct the result

Start subtracting multiples of n eleven steps after the multiplication procedure started

P is a delayed carry integer of $k+11$ bits

m is a binary integer of k bits

t_1 and t_2 control bits: Start $t_1 = t_2 = 0$

Add MS 4 bits of P and $m \cdot 2^{11}$
If OVF, then $t_2 = 1$ else $t_2 = 0$
Add MS 4 bits of P and MS 3 bits of $m \cdot 2^{10}$
If OVF and $t_2 = 0$, then $t_1 = 1$ else $t_1 = 0$

Multiplication and reduction step:

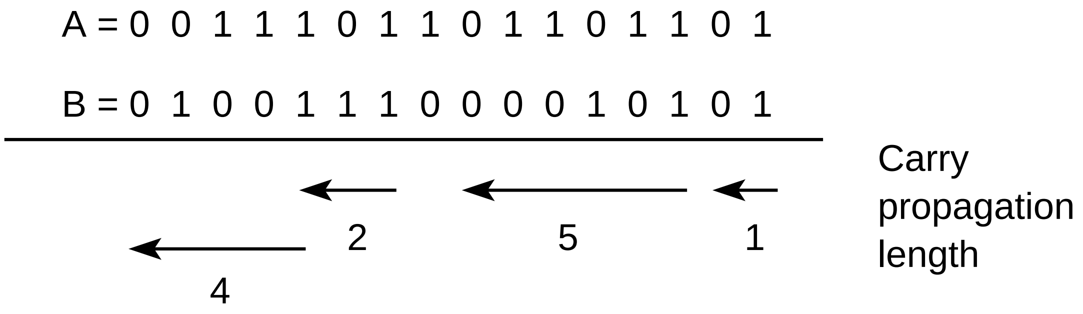
$$\begin{aligned} B' &:= T_i \cdot B + 2 \cdot D_{i+1} \cdot B \\ m' &:= t_2 \cdot m \cdot 2^{11} + t_1 \cdot m \cdot 2^{10} \\ P &:= 2(P + B' + m') \\ A &:= 2A \end{aligned}$$

Carry Completion Sensing Adder

An asynchronous adder which detects completion of the carry propagation process

Statistical analysis shows that average carry length is bounded by $\log_2(k)$

An example:



A = 0 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1

B = 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 1

Implementation

Carry completion signal is produced by a very wide AND gate which computes the product of all individual carry completion signals C+N

A	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1
B	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				1		1		0					1	0	1
N				0		0		1					0	1	0
C			1		1		0					1			
N			0		0		1					0			
C		1									1				
N		0									0				
C	1									1					
N	0									0					
C									1						
N									0						

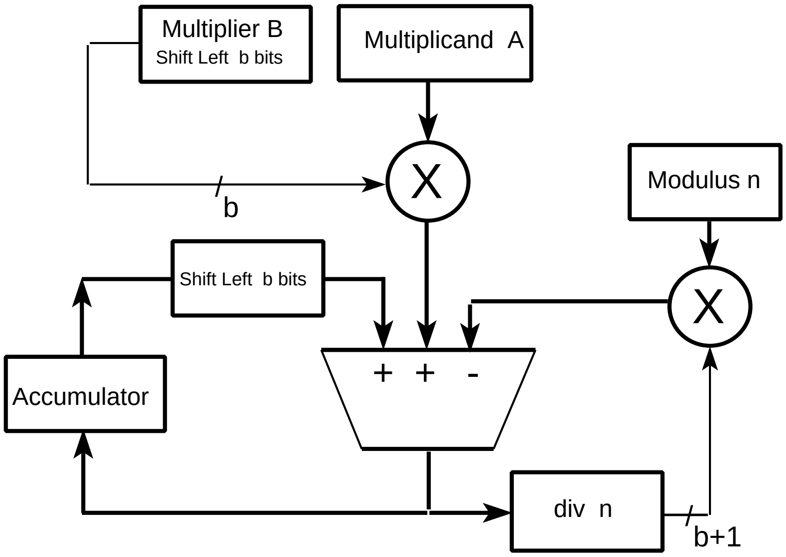
(A,B) = (0,0) ⟹ (C,N) = (0,1)
(A,B) = (1,1) ⟹ (C,N) = (1,0)
(A,B) = (0,1) ⟹ (C,N) = previous (C,N)
(A,B) = (1,0) ⟹ (C,N) = previous (C,N)

High-Radix Multiplication

The speed of radix 2 multipliers is approaching limits, and the use of higher radices has been investigated

High-radix operations require fewer clock cycles, however, the cycle time and the required area increases

Radix 2^b implementation of $A \cdot B \pmod{n}$



Issues in High-Radix Multiplication

- For radix 2, partial product generation is performed using an array of AND gates

- Partial product generation is more complex for higher radices, e.g., Wallace trees and generalized counters need to be used

- Generation of high-radix partial products does not greatly increase cycle time since it can be easily pipelined

- High-radix complicates the reduction step and necessitates more complex routing, increasing the chip area

Finite Fields

Abstractly, a finite field consists of a finite set of objects together with two binary operations: addition and multiplication.

- These binary operations must satisfy certain compatibility properties including associativity, commutativity and also distributivity.

- The finite field containing q elements is denoted by $GF(q)$.

- There exists precisely one finite field containing q field elements if and only if q is a power of a prime number.

Finite Fields

- For efficient implementation purposes, three types of finite fields are of interest; prime finite fields: $GF(p)$ with $q = p$, p an odd prime; extension fields: $GF(q)$ with $q = p^k$, p an odd special prime; and binary extension fields: $GF(2^k)$ thus with $q = 2^k$ for some $k \geq 1$.

- Observe that the arithmetic for prime fields $GF(p)$ is the usual modular arithmetic modulo p.

Properties of $GF(2^k)$ Arithmetic

$GF(2^k)$ is usually viewed as a k-dimensional vector space over the field $GF(2)$ hence, the elements of $GF(2^k)$ are represented by binary vectors $(a_{k-1}a_{k-2}\cdots a_1a_0)$ where $a_i \in GF(2)$.

- a set of k elements
  $$\alpha_0, \alpha_1, \ldots, \alpha_{k-1} \in GF(2^k)$$
  is called a basis for $GF(2^k)$ if every $a \in GF(2^k)$ can be written uniquely in the form
  $$a = a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{k-1}\alpha_{k-1}.$$

- Many different basis exist and the selection of these basis is quite important for computational purposes.

Properties of $GF(2^k)$ Arithmetic

- For example the set

  $$1, x^1, x^2 \ldots, x^{k-1} \in GF(2^k)$$

  is called the polynomial basis for $GF(2^k)$ over $GF(2)$. Therefore one might interpret the elements of $GF(2^k)$ simply by using the polynomials of degree $k-1$ and less

  $$a_{k-1}x^{k-1} + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$$

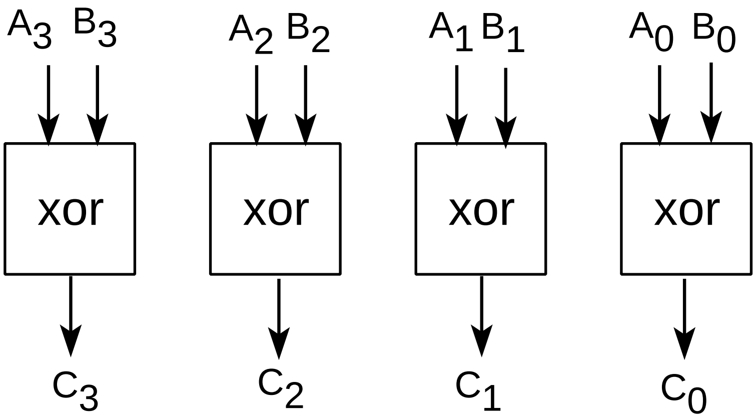
  instead of a vector notation.

- Binary operations (i.e. addition and multiplication) in $GF(2^k)$ are performed by polynomial addition and multiplication modulo $p(x)$, where $p(x)$ is an irreducible binary polynomial of degree k.

Addition in $GF(2^k)$

- $GF(2^k)$ is a vector space so elements (i.e. vectors) are added component-wisely.

- Since the base field is $GF(2)$, on the components addition corresponds to modulo 2 addition (the XOR operation) no matter what the basis is.



Multiplication in $GF(2^k)$

The multiplication consists of two phases:

- Polynomial Multiplication
- Reduction with the defining irreducible polynomial $p(x)$.

This is similar to the multiply and reduce method of modular multiplication. Similar algorithms (such as interleaving) can be used.

Especially, Squaring operation can be significantly simplified by judicious selection of the basis. For example, a normal basis can be used (*Agnew et al*)

By choosing $p(x)$ to have a minimal number of nonzero coefficients, in particular three or five (when $p(x)$ is called trinomial or pentanomial, respectively), the reduction step of polynomial multiplication can be simplified.

Squaring in a Normal Basis

A normal basis of $GF(2^k)$ is a basis of the form

$$\{\beta, \beta^2, \beta^4, \ldots, \beta^{2^{k-1}}\}$$

where β is an element of $GF(2^k)$. It is well known that such a basis always exists. Let A be expressed in a normal basis. We have

$$\begin{aligned} A &= (a_{k-1}a_{k-2}\cdots a_1 a_0) \\ &= a_0\beta + a_1\beta^2 + a_2\beta^4 + \cdots + a_{k-1}\beta^{2^{k-1}} \end{aligned}$$

We compute the square of A as

$$\begin{aligned} A^2 &= (\sum_{i=0}^{k-1} a_i\beta^{2^i}) \cdot (\sum_{i=0}^{k-1} a_i\beta^{2^i}) \\ &= \sum_{i=0}^{k-1} \left(a_i\beta^{2^i}\right)^2 = \sum_{i=0}^{k-1} a_i\beta^{2^{i+1}} \\ &= (a_{k-2}a_{k-3}\cdots a_1 a_0 a_{k-1}) \end{aligned}$$

which is a cyclic left shift of A

Squaring in $GF(2^k)$

$$(\sum_{i=0}^{k-1} a_i\beta^{2^i}) \cdot (\sum_{i=0}^{k-1} a_i\beta^{2^i}) \;=\; \sum_{i=0}^{k-1} a_i\beta^{2^{i+1}}$$

The above equality is due to the fact the cross terms cancel out, and $\beta^{2^k} = \beta$. For example, given $A = (a_2a_1a_0) \in GF(2^3)$, we have

$$\begin{aligned}
A &= (a_2a_1a_0) \\
A^2 &= (a_0\beta + a_1\beta^2 + a_2\beta^4)^2 \\
&= a_0\beta(a_0\beta + a_1\beta^2 + a_2\beta^4) + \\
&\ a_1\beta^2(a_0\beta + a_1\beta^2 + a_2\beta^4) + \\
&\ a_2\beta^4(a_0\beta + a_1\beta^2 + a_2\beta^4) \\
&= a_0a_0\beta^2 + a_0a_1\beta^3 + a_0a_2\beta^5 + \\
&\ a_1a_0\beta^3 + a_1a_1\beta^4 + a_1a_2\beta^6 + \\
&\ a_2a_0\beta^5 + a_2a_1\beta^6 + a_2a_2\beta^8 \\
&= a_0\beta^2 + a_1\beta^4 + a_2\beta^8 \\
&= a_0\beta^2 + a_1\beta^4 + a_2\beta \\
&= a_2\beta + a_0\beta^2 + a_1\beta^4 \\
&= (a_1a_0a_2)
\end{aligned}$$

Multiplication in a Normal Basis

The product $C = AB$ is given as

$$C = \sum_{i=0}^{k-1} C_i \beta^{2^i} = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} A_i B_j \beta^{2^i+2^j}$$

Since $\beta^{2^i+2^j}$ is also an element of $GF(2^k)$, it can be expressed as

$$\beta^{2^i+2^j} = \sum_{r=0}^{k-1} \lambda_{ij}^{(r)} \beta^{2^r}$$

where $\lambda_{ij}^{(r)} = 0$ or 1. This yields the formula

$$C_r = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} A_i B_j \lambda_{ij}^{(r)} \qquad \text{for } 0 \le r \le k-1$$

We also notice that

$$\beta^{2^{i-s}+2^{j-s}} = \sum_{r=0}^{k-1} \lambda_{i-s,j-s}^{(r)} \beta^{2^r} = \sum_{r=0}^{k-1} \lambda_{ij}^{(r)} \beta^{2^{r-s}}$$

which implies

$$\lambda_{ij}^{(s)} = \lambda_{i-s,j-s}^{(0)} \quad \text{for all } 0 \le i,j,s \le k-1$$

Multiplication Formulae

Thus, we have a formula for C_r as

$$C_r = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} A_{i+r} B_{j+r} \lambda_{ij}$$

Consider a circuit built for computing C_0 which receives the inputs as (in this order)

$$A_0, A_1, \ldots, A_{k-2}, A_{k-1}$$
$$B_0, B_1, \ldots, B_{k-2}, B_{k-1}$$

This circuit uses the formula

$$C_0 = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} A_i B_j \lambda_{ij}$$

to compute C_0

The same circuit can be used to compute C_1 with the inputs as

$$A_1, A_2, \ldots, A_{k-1}, A_0$$
$$B_1, B_2, \ldots, B_{k-1}, B_0$$

Multiplication Formulae

The number of nonzero λ_{ij}s determines the complexity of the multiplication circuit

- upper bound is k^2

- lower bound is shown to be $2k - 1$

A normal basis with $2k-1$ nonzero λs is called an *optimal normal basis*. Such a basis exists for certain fields

Thus, a circuit with area $O(k)$ can be built to multiply two elements of $GF(2^k)$ in k clock cycles

Values of $k < 265$ for which there exists an ONB

2	18	41	74	100	146	180	226
3	23	50	81	105	148	183	230
4	26	51	82	106	155	186	231
5	28	52	83	113	158	189	233
6	29	53	86	119	162	191	239
9	30	58	89	130	172	194	243
10	33	60	90	131	173	196	245
11	35	65	95	134	174	209	251
12	36	66	98	135	178	210	254
14	39	69	99	138	179	221	261

Inversion in $GF(2^k)$

An efficient algorithm for computing an inverse of an element of $GF(2^k)$ was proposed by Itoh, Teechai, and Tsujii (see *Agnew et al*)

If $a \in GF(2^k)$ and $a \neq 0$, then

$$a^{-1} = a^{2^k-2} = \left(a^{2^{k-1}-1}\right)^2$$

For k even or odd, we have

$$\begin{array}{lll} \text{odd:} & 2^{k-1}-1 & = (2^{(k-1)/2}-1)\cdot(2^{(k-1)/2}+1) \\ \text{even:} & 2^{k-1}-1 & = 2\cdot(2^{k-2}-1)+1 \\ & & = 2\cdot(2^{(k-2)/2}-1)\cdot(2^{(k-2)/2}+1) \\ & & \quad +1 \end{array}$$

These formulae yield an algorithm for computing the inverse by a series of squarings, exponentiations and multiplications

Example of Inverse Computation

Consider the field $GF(2^{155})$

$$\begin{aligned}
2^{155}-2 &= 2\cdot(2^{77}-1)\cdot(2^{77}+1)\\
2^{77}-1 &= 2\cdot(2^{38}-1)\cdot(2^{38}+1)+1\\
2^{38}-1 &= (2^{19}-1)\cdot(2^{19}+1)\\
2^{19}-1 &= 2\cdot(2^{9}-1)\cdot(2^{9}+1)+1\\
2^{9}-1 &= 2\cdot(2^{4}-1)\cdot(2^{4}+1)+1\\
2^{4}-1 &= (2^{2}-1)\cdot(2^{2}+1)\\
2^{2}-1 &= (2^{1}-1)\cdot(2^{1}+1)
\end{aligned}$$

It requires 10 multiplications to compute an inverse in $GF(2^{155})$ (and many squarings, which are essentially free)

In general, the method requires

$$\lfloor \log_2(k-1)\rfloor + H(k-1) - 1$$

multiplications

Properties of $GF(p^k)$ Arithmetic

$GF(p^k)$ is a k-dimensional vector space over the field $GF(p)$ hence, the elements of $GF(p^k)$ are represented by vectors $(a_{k-1}a_{k-2}\cdots a_1a_0)$ where $a_i \in GF(p)$.

- A carefully chosen p admits efficient reduction in the ground field operations.

- Bailey and Paar proposed the use of a pseudo-Mersenne prime of the form $p = 2^m \pm c$ with $\log_2 c < \lfloor \frac{m}{2} \rfloor$. This construction is also called Optimal extension field.

Properties of $GF(p^k)$ Arithmetic

- Addition is performed component-wise with modulo p arithmetic.

- The usual multiply-reduce methodology is used for multiplication.

- Observe that we are considering two types of reduction: the first one is modulo p reduction that we perform on the components where the second one is reducing the degree of the polynomials by using the defining polynomial.

References

G.B. Agnew, T. Beth, R.C. Mullin, and S.A. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6(1):3–13, 1993.

G.B. Agnew, R.C. Mullin, and S.A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

D.V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public–Key Algorithms, *in Advances in Cryptology CRYPTO 98 (H.Krawczyk,ed.)*, vol. LNCS 1462, (Berlin, Germany), pp. 472-485, Springer-Verlag, 1998.

G.R. Blakley. A computer algorithm for the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.

J. Bos and M. Coster. Addition chain heuristics. In G. Brassard, editor, *Advances in Cryptology — Crypto '89*, pages 400–407. Springer-Verlag, 1989.

R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, March 1982.

E.F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R.L. Rivest, and A.T. Sherman, editors, *Advances in Cryptology — Crypto '82*, pages 51–60. Plenum Press, 1982.

P. Downey, B. Leony, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 3:638–696, 1981.

S.R. Dussé and B.S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I.B. Damgard, editor, *Advances in Cryptology — Eurocrypt '90*, pages 230–244. Springer-Verlag, 1990.

Ö. Eğecioğlu and Ç.K. Koç. Exponentiation using canonical recoding. *Theoretical Computer Science*, 129(2):407–417, 1994.

K. Hwang. *Computer Arithmetic, Principles, Architecture, and Design*. John Wiley & Sons, 1979.

D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Second edition, 1981.

N. Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

Ç.K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

Ç.K. Koç. RSA Hardware Implementation. Technical Report, TR 801, RSA Laboratories. 30 pages, April 1996.

Ç.K. Koç and C.Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.

I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, 1993.

R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

D. Laurichesse and L. Blain. Optimized implementation of RSA cryptosystem. *Computers & Security*, 10(3):263–267, May 1991.

A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

V. Miller. Uses of elliptic curves in cryptography. In H.C. Williams, editor, *Advances in Cryptology —– Crypto '85*, pages 417–426. Springer-Verlag, 1985.

P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

J.K. Omura. A public key cell design for smart card chips. In *International Symposium on Information Theory and its Applications* (Hawaii, USA, November 27–30, 1990), pages 983–985.

J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18:905–907, October 1982.

A. Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1–12, 1975.

M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In E. Swartzlander Jr., M.J. Irwin, and G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259. IEEE Computer Society Press, 1993.

E.E. Swartzlander, editor. *Computer Arithmetic*, volume I and II. IEEE Computer Society Press, 1990.

S. Waser and M.J. Flynn. *Introduction to Arithmetic for Digital System Designers*. Holt, Rinehart and Winston, 1982.

Y. Yacobi. Exponentiating faster with addition chains. In I.B. Damgard, editor, *Advances in Cryptology —– Eurocrypt '90*, pages 222–229. Springer-Verlag, 1990.

Efficient Embedded Software Implementations of Public-Key Cryptography

Çetin Kaya Koç
koc@cs.ucsb.edu

Cryptographic Engineering 2012

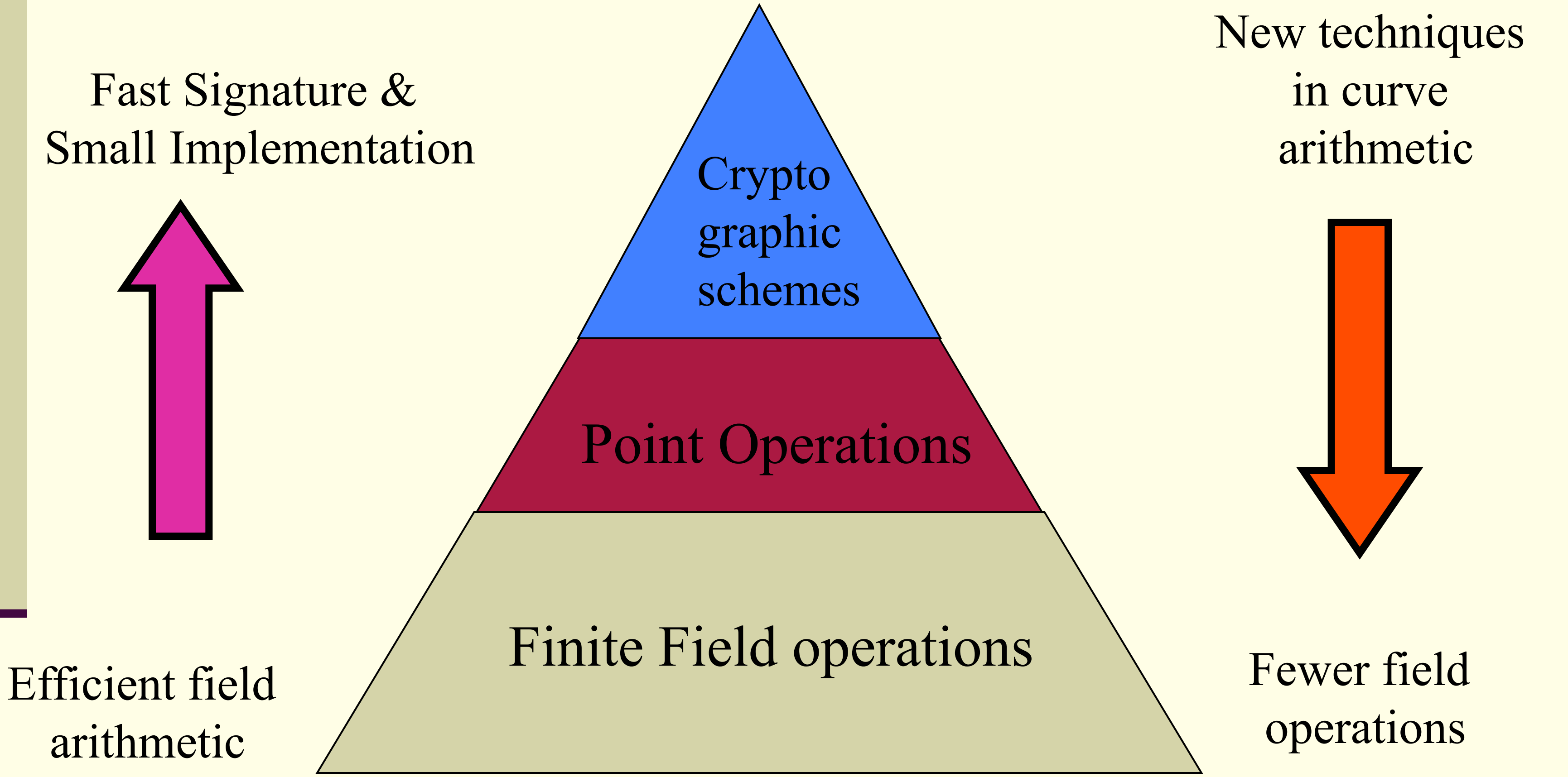
Characteristics of Embedded Systems

- Low Computing Power
- Low Dynamic Memory for Data
- Restricted Stack Memory Usage
- Diverse Memory Allocation Techniques
- Different Memory Management & Organization
- Limitations on Program Code Size
- Difficulty in handling exceptions on System Calls

Challenges of Public Key Cryptography in Embedded Systems

- High Latency of Public Key Operations
- Excessive Demand on Dynamic Memory
- High Stack Memory Requirement
- High footprint
- Exception handling
- Secure Execution
- Removal of Sensitive Information from Memory

Elliptic Curve Cryptography



Fast Signature & Small Implementation

Crypto graphic schemes

New techniques in curve arithmetic

Point Operations

Finite Field operations

Efficient field arithmetic

Fewer field operations

Finite Fields for ECC Usage

- Prime fields, GF(*p*)
- Binary extension fields, GF(2*k*).
- General Extension fields GF(*p*^*k*) - Not as common
- **The Focus:** GF(*p*) & GF(2^*k*)
- **The Goal:** Fast execution of finite filed operations in embedded environment requiring low memory and footprint

Finite Field Operations in ECC

- Addition in GF(p) and GF(2^k)
  Inexpensive in terms of time and area
- Multiplicative inversion in GF(p) and GF(2^k)
  Prohibitively expensive in terms of time
  Possible to avoid some of them
- Multiplication in GF(p) and GF(2^k)
  Expensive in terms of time and area
  Most important operation
  Focus of many design efforts

Incomplete Arithmetic in GF(p)

- An effective method to enhance the time performance of arithmetic operations in software particularly when the field degree is not on the word boundary of underlying processor
- For example, 2^176 > *p* □ 2^175 when the word size of the computer is 32 bits
- Based on a technique eliminating the need for reduction when the intermediary result does not exceed the word boundary.
- Up to 13% overall speedup

Addition & Subtraction in GF(p)

- Takes up 15% plus of total computation time in an EC point multiplication operation with projective coordinates
- Implemented in Assembly which improves both operations at least 100%
- Incomplete arithmetic provides up to 40% and 25% perfomance imrpovement for addition and subtraction, respectively

Multiplication in GF(p)

- Takes up 70% plus of total computation time in an EC point multiplication operation with projective coordinates
- Assembly implementation improves 150%
- Incomplete arithmetic provides up to 5%
- CIOS Montgomery algorithm seems to be the best performer when the prime p is arbitrarily chosen

Multiplicative Inversion in GF(p)

- Very important operation when affine EC coordinates are used
- The Montgomery inversion algorithm is used
- A new correction phase, which provides 1.5 overall speedup, is utilized
- No assembly is used

Arithmetic Operations in GF(2^k)

- No need to use Assembly language since there is no carry handling
- The subtraction operation is the same as the addition
- Addition is a trivial operation which does not need any special treatment in software
- All operations can be optimized for irreducible trinomials or pentanomials of any type

Multiplication in GF(2^k)

- Takes up about 70% of total computation time in an EC point multiplication operation with projective coordinates
- Montgomery’s algorithm is not used for trinomials or pentanomials
- Consists of two phases:
  - Polynomial Multiplication
  - Reduction with the irreducible polynomial

Multiplication Phase

- Non-recursive Karatsuba-Offman algorithm is quite effective since redundant operations are eliminated
- More improvement is possible with an increase in the code size and with acceleration tables
- Standard Karatsuba-Offman algorithm provides around 25% overall speedup

Reduction Phase

- Standard reduction method is the best when trinomials or pentanomials are used
- GF(2^k) version of Montgomery’s method performs better when randomly chosen irreducible polynomials are used
- Acceleration tables of different amounts can provide different levels of speedup

Elliptic Curve Point Operations

- Consists of a number of finite field operations
- For speed, mixed modified Jacobian coordinates offer the maximum performance
- For processors of low resources, affine coordinates provide a lean version of ECC
- Different acceleration techniques provide different levels of speedup at the expense of memory
- Simultaneous point multiplication with redundant representation improves signature verification

Special Curve Solutions

- All operations (field & point arithmetic) can be highly optimized if fixed special curves are chosen
- For GF(*p*), the speedup is up to 100% over the standard implementation
- For GF(2^*k*), the speedup is about 50%
- For GF(2^*k*), the speedup can increase as high as 400% if Koblitz curves are used

Desired Characteristics of the Embedded Software (1)

- Reentrant

  A computer program or routine is described as **reentrant** if it is designed such that a single copy of the program's instructions in memory can be shared by multiple users or separate processes. The key to the design of a reentrant program is to ensure that no portion of the program code is modified by the different users/processes, and that process-unique information (such as local variables) is kept in a separate area of memory that is distinct for each user or process

- Thread-safe

  A piece of code is **thread-safe** if it is reentrant or protected from multiple simultaneous execution by some form of mutual exclusion

Desired Characteristics of the Embedded Software (2)

- Modularity and OS independence
- No OS or C calls within the library
- Low stack usage (under 1K is typical)
- Streamlined dynamic memory usage
- All memory allocation should (can) be made at one point by user
- Allocation can be made automatic with either user-supplied or standard memory manager

Desired Characteristics of the Embedded Software (3)

- Memory requirement for any curve can be reported by a function before any cryptographic operation is performed
- Supports any curve of infinite precision with highly optimized manner
- Configurable RNG
- Safe execution
- Removes any sensitive information from the memory after the cryptographic computations

Desired Characteristics of the Embedded Software (4)

- A set of acceleration techniques provide different levels of speedup
- Dynamic memory is divided into two parts: curve-specific or temporary. This feature allows several possibilities:
  - Curve-specific memory can be shared between different threads using the same curve
  - Curve-specific memory can be allocated and initialized offline

Performance on 80-MHz ARM7TDMI

Bitsize	Signature (ms)			Memory (KB)		
	PC0	PC1	PC2	PC0	PC1	PC2
192 – GF(p)	70	28	13	2	12	69
256 – GF(p)	145	53	24	3	20	85
163 – GF(2^k)	105	40	23	2	10	59
257 – GF(2^k)	274	109	45	3	22	96

- Random curves are used. Further improvement is possible with fixed curves
- PC0: No precomputation
- PC1: Precomputation Level 1, etc.

Performance on 20-MHz PALM

Bitsize	Signature (s)			Memory (KB)		
	PC0	PC1	PC2	PC0	PC1	PC2
192 – GF(p)	5.1	2.1	1.4	2	12	26
256 – GF(p)	11.1	4.3	3.1	3	20	44
163 – $GF(2^k)$	4.9	2.0	1.5	2	10	22
257 – $GF(2^k)$	12.0	4.6	3.5	3	22	50

- Random curves are used. Further improvement is possible with fixed curves
- PC0: No precomputation
- PC1: Precomputation Level 1, etc.
- Precomputation is restricted due to limited memory

References

- C. K. Koc, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro,* 16(3):26-33, June 1996.
- C. K. Koc and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography,* 14(1):57-69, April 1998.
- E. Savas and C. K. Koc. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers,* 49(7):763-766, July 2000.
- M. Aydos, T. Yanik, and C. K. Koc. High-speed implementation of an ECC-based wireless authentication protocol on an ARM microprocessor. *IEE Proceedings - Communications,* 148(5):273-279, October 2001.
- T. Yanik, E. Savas, and C. K. Koc. Incomplete reduction in modular arithmetic. *IEE Proceedings - Computers and Digital Techniques,* 149(2):46-52, March 2002.
- T. Wollinger, J. Pelzl, V. Wittelsberger, C. Paar, G. Saldamli, and C. K. Koc. Elliptic and hyperelliptic curves on embedded µP. *ACM Transactions on Embedded Computing Systems,* 3(3):509-533, August 2004.

Spectral Modular Arithmetic

Çetin Kaya Koç

koc@cs.ucsb.edu

Cryptographic Engineering 2012

General Idea

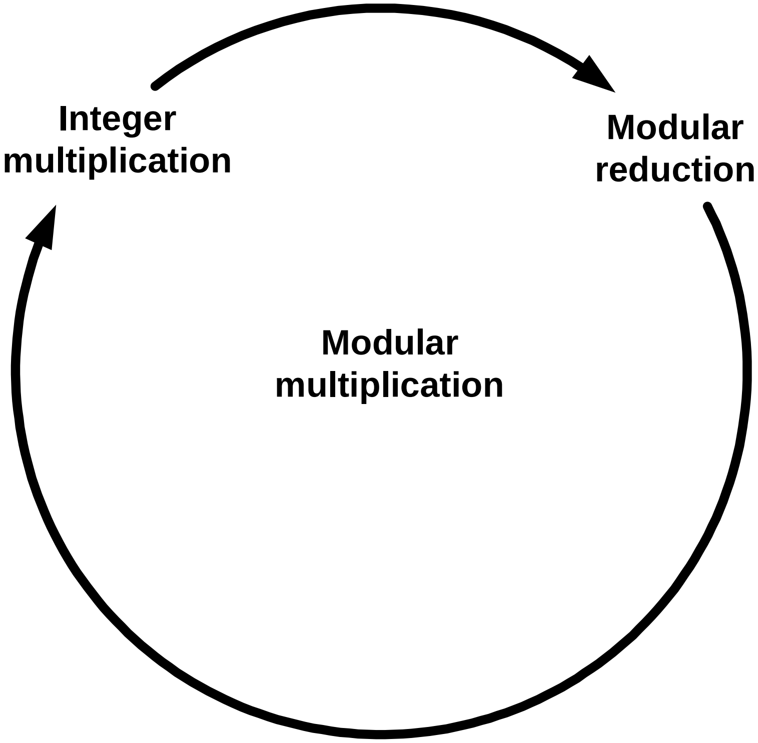
- Utilize finite ring and field arithmetic (avoid real or complex arithmetic; floating-point and fixed-point arithmetic: roundoff errors)

- Bring down the break-even point of efficiency for FFT-based multiplication

- Utilize the DFT and Montgomery properties to perform modular reduction

- It is related to: RNS, CRT, Polynomial arithmetic, etc.

Modular Multiplication

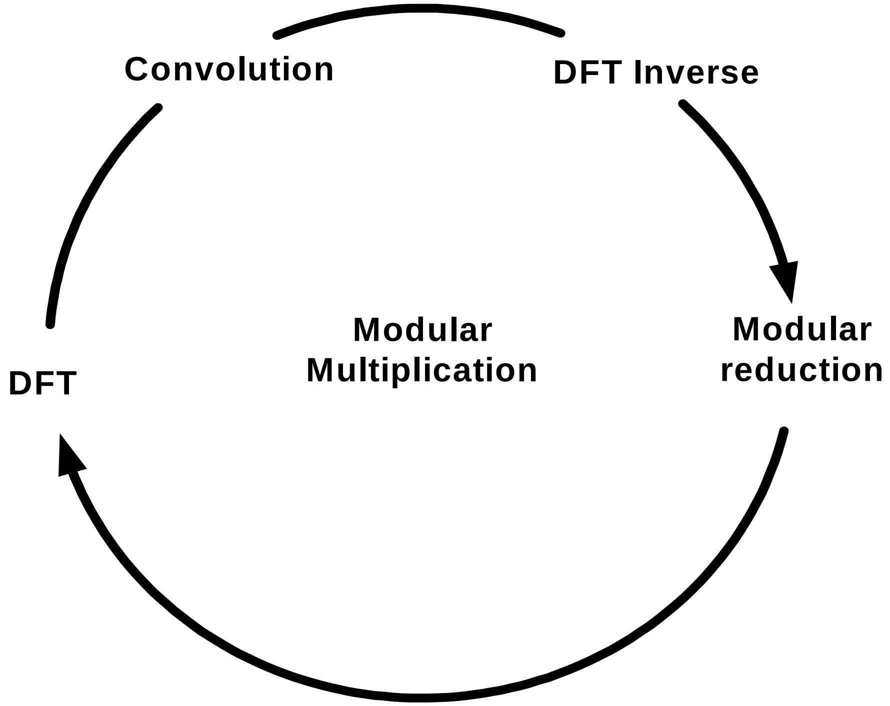


Integer multiplication

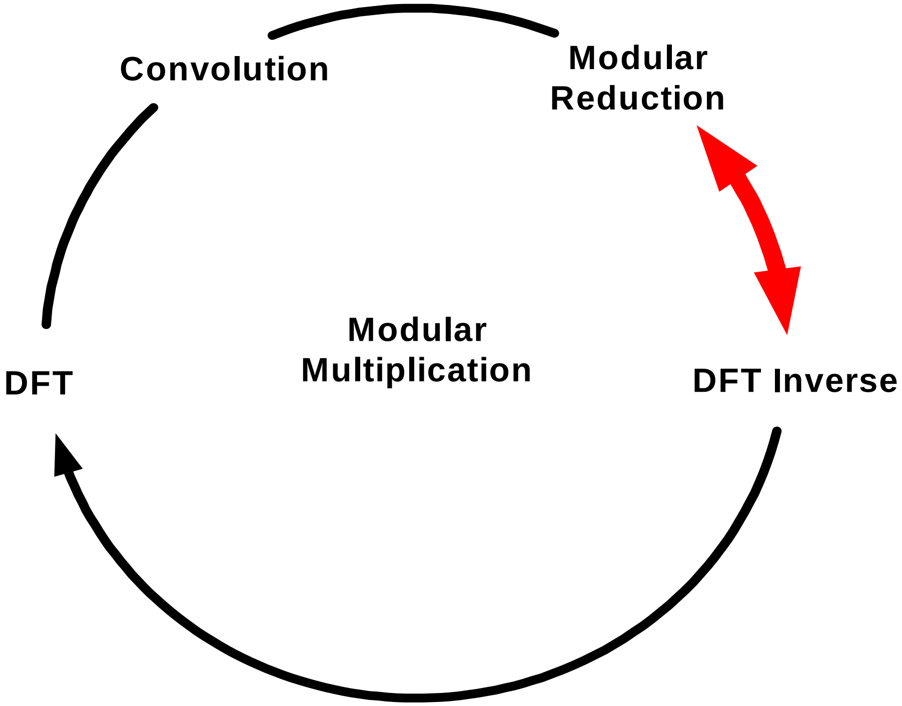
Modular reduction

Modular multiplication

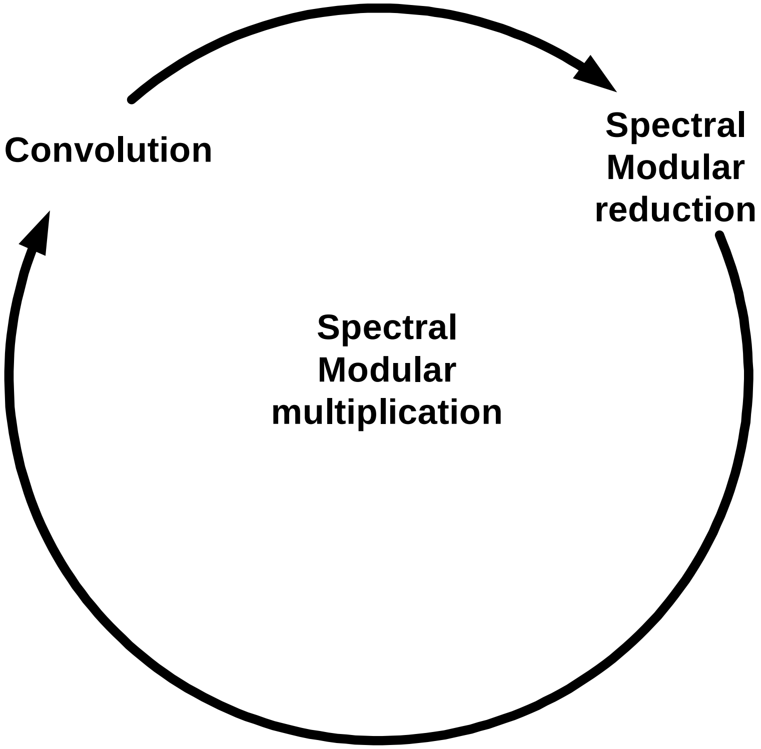
Multiplication using DFT



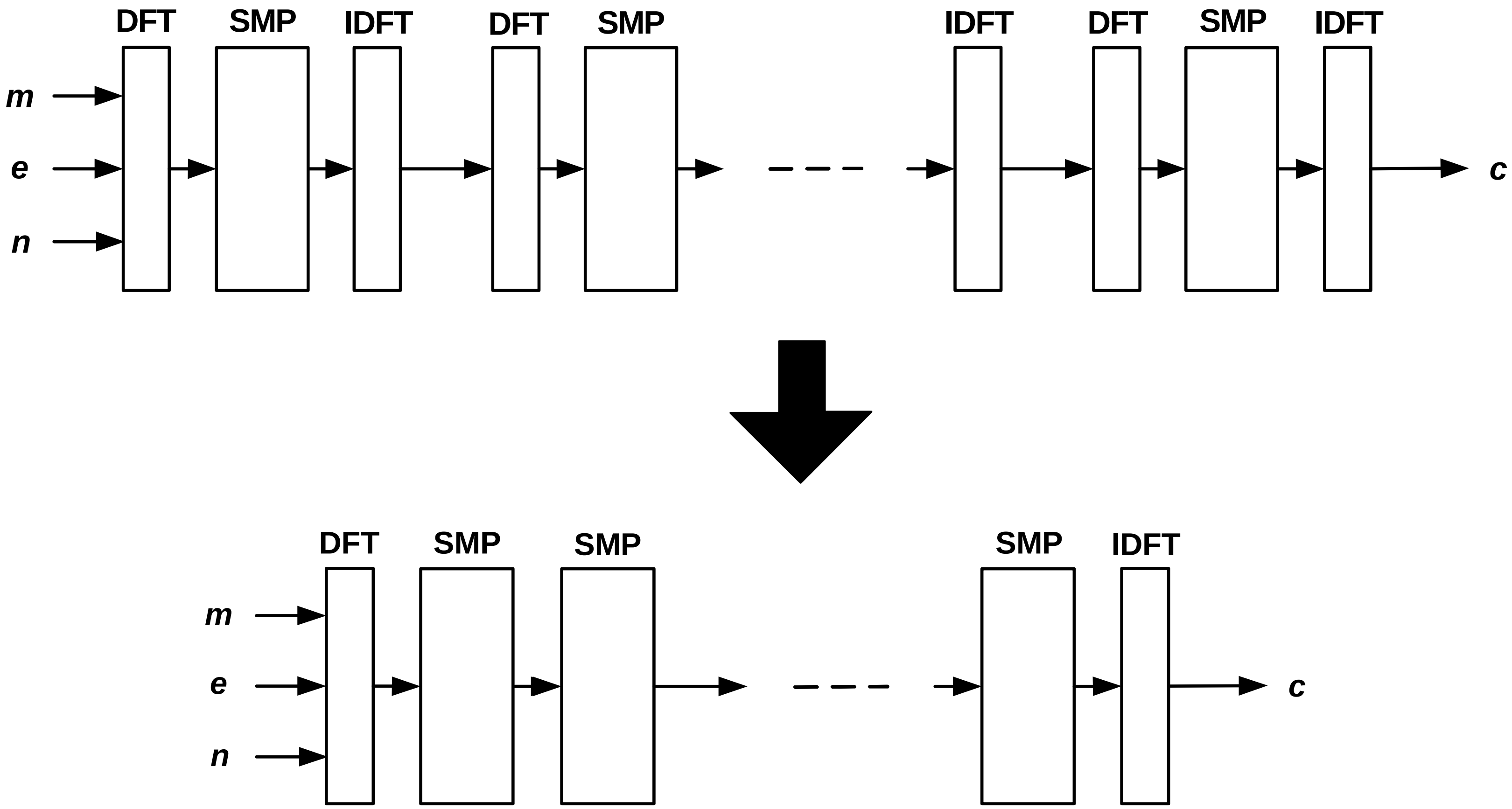
Reduction in Spectral Domain?



Spectral Modular Multiplication



Another View of Simplification



DFT over a Finite Ring: Definition

Let ω be a primitive d-th root of unity in $\mathbb{Z}_q$ and, let $x(t)$ and $X(t)$ be polynomials of degree $d-1$ having entries in $\mathbb{Z}_q$. The DFT map over $\mathbb{Z}_q$ is an invertible set map sending $x(t)$ to $X(t)$ given by

$$X_i = DFT_d^{\omega}(x(t)) := \sum_{j=0}^{d-1} x_j \omega^{ij} \bmod q,$$

with the inverse

$$x_i = IDFT_d^{\omega}(X(t)) := d^{-1} \cdot \sum_{j=0}^{d-1} X_j \omega^{-ij} \bmod q,$$

for $i, j = 0, 1, \ldots, d-1$.

DFT over a Finite Ring: Existence

We write

$$x(t) \quad \overset{\text{DFT}}{\longleftrightarrow} \quad X(t)$$

and say $x(t)$ and $X(t)$ are transform pairs; $x(t)$ is called a **time polynomial** and sometimes $X(t)$ is named as the **spectrum** of $x(t)$.

- (Convention) In the literature, DFT over a finite ring spectrum is also called as Number Theoretical Transform (NTT)

- (Existence) In order to have a DFT map over $\mathbb{Z}_q$:
  - The multiplicative inverse of DFT length d must exist in $\mathbb{Z}_q$ which requires that $\gcd(d, q) = 1$.
  - d has to divide $p - 1$ for every prime p divisor of q

DFT over a Finite Ring: Efficiency

In order to have simple arithmetic

- q should be chosen as
  a Mersenne number $q = 2^v - 1$, or
  a Fermat number $q = 2^v + 1$

- The principal root of unity ω should be selected as a power of 2 to simplify the multiplications with roots of unity

Properties of DFT

- Under certain conditions, the Fourier transform preserves some properties of the time sequences, e.g., linearity and convolution.

- The existence conditions of these properties differ when working in finite ring spectrums

- Let ϕ and Φ be operations on time and spectral domains respectively. We write

$$\phi \quad \overset{\text{DFT}}{\longleftrightarrow} \quad \Phi$$

  and say ϕ and Φ are transform pairs on $x(t)$ and sometimes declare that **the map DFT_d^ω respects the operation** ϕ on point $x(t)$ if following equation is satisfied

$$\phi(x(t)) = IDFT_d^\omega \circ \Phi \circ DFT_d^\omega(x(t))$$

Time-Frequency Dictionary

- **Time and frequency shifts** correspond to circular shifts Let

$$x(t) = x_0 + x_1 t + \ldots + x_{d-1} t^{d-1}$$

and

$$X(t) = X_0 + X_1 t + \ldots + X_{d-1} t^{d-1}$$

be a transform pair.

The *one-term right circular shift* is defined as $x(t) \circlearrowleft 1$

$$\begin{array}{c} x_1 + x_2 t + \ldots + x_{d-2} t^{d-1} + x_0 t^{d-1} \\ \updownarrow \text{DFT} \\ X(t) \odot \Gamma(t) \end{array}$$

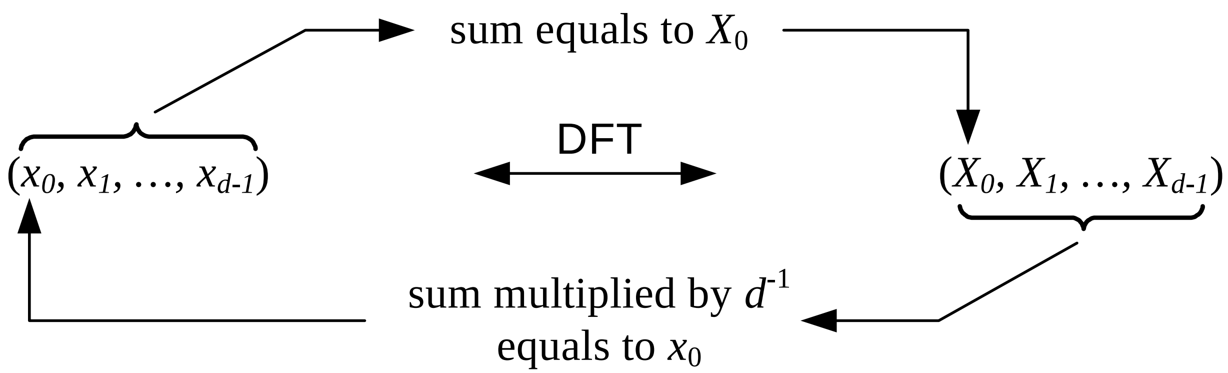
where $\odot$ stands for component-wise multiplication and

$$\Gamma(t) = 1 + \omega^{-1} t + \ldots + \omega^{-(d-1)} t^{d-1}$$

Time-Frequency Dictionary

- **Sum of sequence and first value:** The sum of the coefficients of a time polynomial equals to the zeroth coefficient of its spectral polynomial. Conversely the sum of the spectrum coefficients equals to d^{-1} times the zeroth coefficient of the time polynomial

$$x_0 = d^{-1} \cdot \sum_{i=0}^{d-1} X_i \omega^{-i} \quad \text{and} \quad X_0 = \sum_{i=0}^{d-1} x_i \omega^i$$



sum equals to X_0

DFT

$(x_0, x_1, \ldots, x_{d-1})$

$(X_0, X_1, \ldots, X_{d-1})$

sum multiplied by d^{-1} equals to x_0

Time-Frequency Dictionary

- **Left and right logical shifts:** By using the previous properties, it is possible to perform logical left and right digit shifts $x(t) \ll 1$ as follows:

$$\begin{tikzcd} (x(t) - x_0)/t = x_1 + \ldots + x_{d-1}t^{d-2} \arrow[d, leftrightarrow, "\text{DFT}"] \\ (X(t) - x_0(t)) \odot \Gamma(t) \end{tikzcd}$$

  where

$$x_0(t) = x_0 + x_0 t + x_0 t^2 + \ldots + x_0 t^{d-1}$$

- The right shifts are similar, where one then uses the

$$\Omega(t) = 1 + \omega^1 t + \ldots + \omega^{(d-1)} t^{d-1}$$

  polynomial instead of $\Gamma(t)$

Time Simulations and Spectral Algorithms

We define our algorithms as the images of “time algorithms”:

TIME DOMAIN	**FREQENCY DOMAIN**
Algorithm 1	**Algorithm 2**
$x(t), y(t) \in \mathbb{Z}[t]/(t^d - 1)$	$X(t), Y(t) \in \mathbb{Z}[t]/(t^d - 1)$
$z(t) = x(t)(5y(t) + x(t)) - 3x(t)$	$Z(t) = X(t) \odot (5Y(t) + X(t)) - 3X(t)$
$z(t) := x(t) + 5y(t)$	$Z(t) := X(t) + 5Y(t)$
$z(t) := x(t) \cdot z(t)$	$Z(t) := X(t) \odot Z(t)$
$z(t) := z(t) - 3x(t)$	$Z(t) := Z(t) - 3X(t)$
return $z(t)$	**return** $z(t)$

If these two algorithms produce agreeing results we write

$$\text{Algorithm 1} \quad \overset{\text{DFT}}{\longleftrightarrow} \quad \text{Algorithm 2}$$

On Translation to Spectrum

- Time simulations are translated into spectrum using properties of DFT

- In order to have valid a spectral algorithm, algorithms in two domains must agree: Time Simulation $\overset{\text{DFT}}{\longleftrightarrow}$ Spectral Algorithm.

- In other words
  - the inputs of time simulation and spectral algorithm must agree
  - intermediate results must agree
  - actual outputs must agree

- In a finite ring spectrum, if the magnitudes of the time coefficients exceed q, an overflow occurs, which prdoduces misleading outputs

- A boundary (overflow) analysis is needed for finite ring spectrum

Time Simulation of SMP

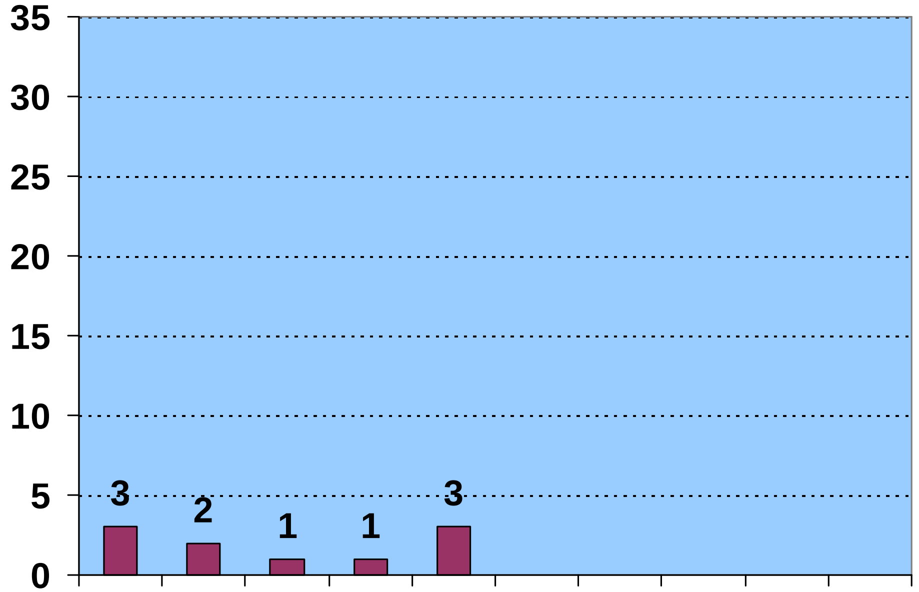
Algorithm 1. *Time Simulation of Spectral Modular Product: Suppose x, y, n are positive integers such that $x, y < n$. Let $x(t), y(t)$ and $n(t)$ be polynomials having the radix-b representation of integers x, y and a multiple of modulus n with $n_0 = 1$ respectively.*

Input: $x(t), y(t)$ *and* $n(t)$.
Output: $z(t) \equiv y(t) \cdot x(t) \cdot t^{-d} \bmod n(t)$.

```
$z(t) := x(t)y(t)$
$\alpha := 0$
for $i = 0$ to $d - 1$
    $\beta := -(z_0 + \alpha) \bmod b$
    $\alpha := (z_0 + \alpha + \beta)$ div $b$
    $z(t) := z(t) + \beta \cdot n(t) \bmod q$
    $z(t) := (z(t) - z_0)/t$
end for
$z(t) := z(t) + \alpha(t)$
return $z(t)$
```

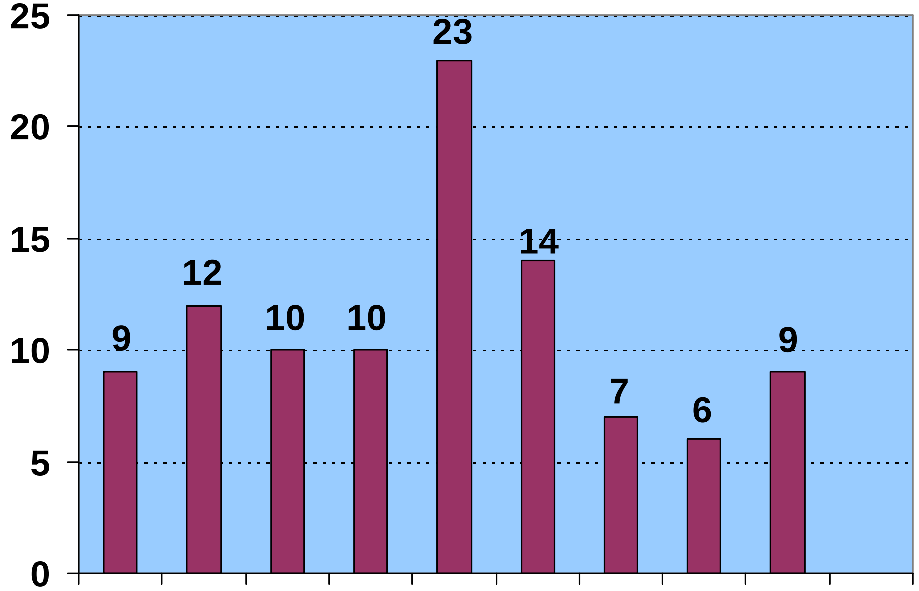
A Time Simulation for SMP

We would like to compute $859^2 \cdot 4^{-9} \pmod{1337}$.
Signal $x(t)$ representing $859 = x(4)$ in base 4.



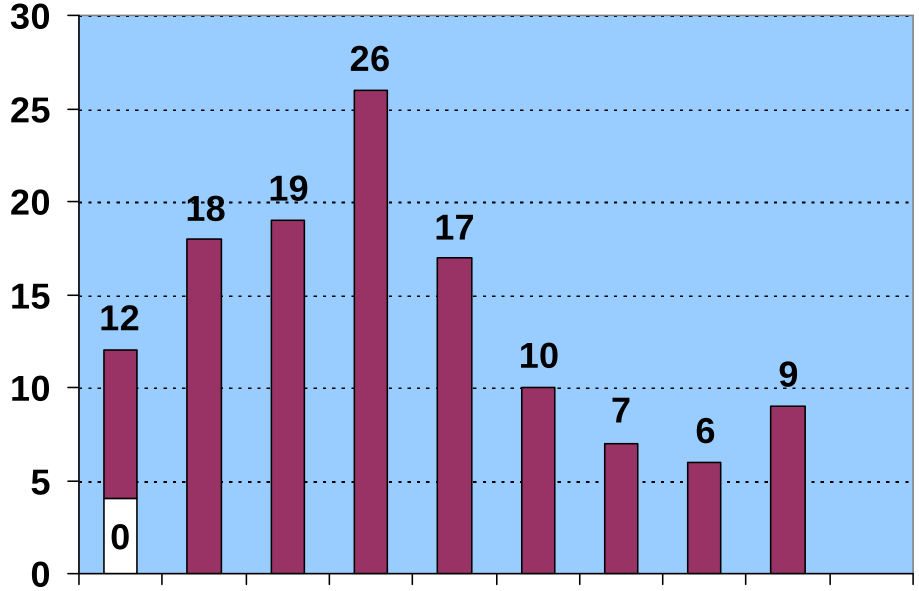
A Time Simulation for SMP

Convolving $x(t)$ with itself, we find $x^2(t) = 859^2 = 737881$.



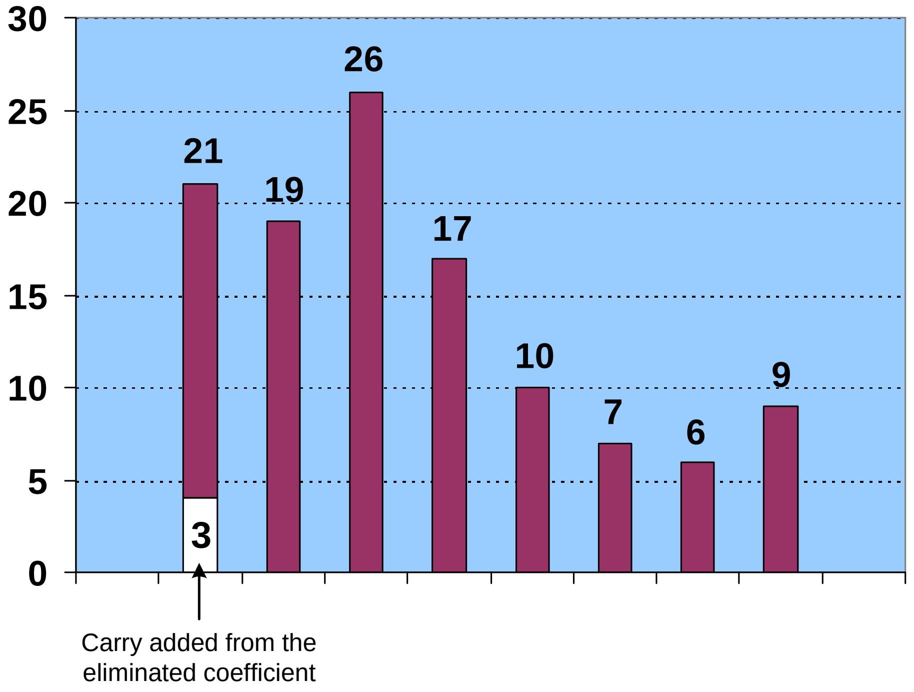
A Time Simulation for SMP

The modulus $m = 1337$ is represented as $m = 1 + 2t + 3t^2 + t^4 + t^5$. We add $3m$ to the sum to anhilate the least significant b bits of the least digit.



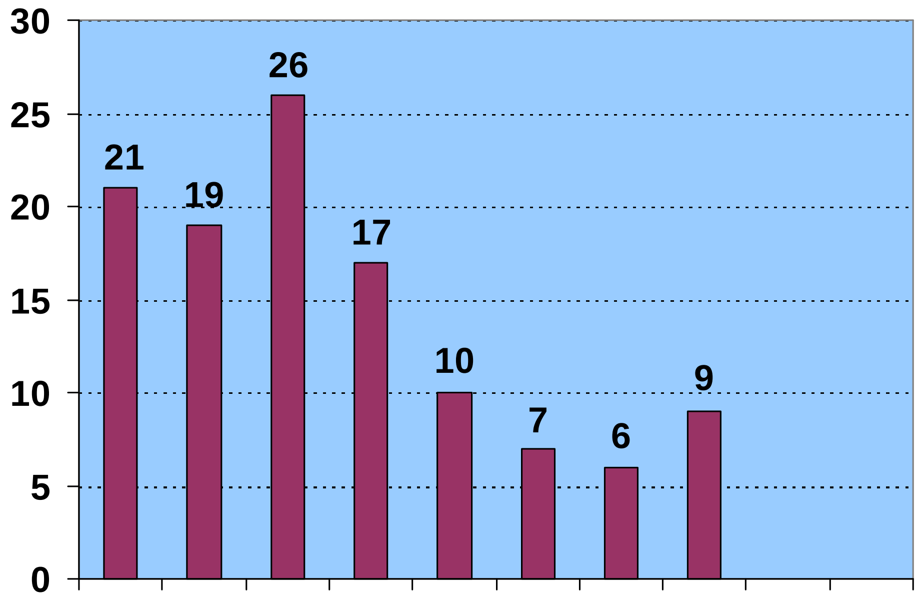
A Time Simulation for SMP

Carry goes to the next digit.



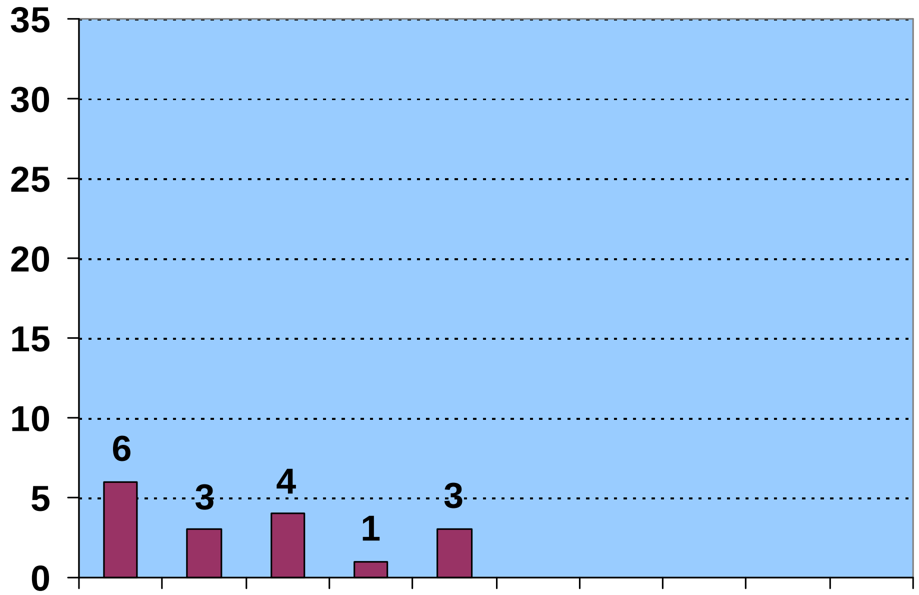
A Time Simulation for SMP

We then shift the digits.



A Time Simulation for SMP

After 9 iterations, we find the result: $914 \equiv 859^2 \cdot 4^{-9} \pmod{1337}$.



SMP Algorithm

Algorithm 2. *Spectral Modular Product: Suppose that there exist a d-point DFT map for some principal root of unity ω in $\mathbb{Z}_q$, and $X(t), Y(t)$ and $N(t)$ are transform pairs of $x(t), y(t)$ and $n(t)$ respectively.*

```
procedure SMP($X(t), Y(t), N(t)$)
    $Z(t) := X(t) \odot Y(t)$
    $\alpha := 0$
    for $i = 0$ to $d - 1$
        $z_0 := d^{-1} \cdot (Z_0 + Z_1 + \ldots + Z_d) \bmod q$
        $\beta := -(z_0 + \alpha)$ mod $b$
        $\alpha := (z_0 + \alpha + \beta)/b$
        $Z(t) := Z(t) + \beta \cdot N(t) \bmod q$
        $Z(t) := Z(t) - (z_0 + \beta)(t) \bmod q$
        $Z(t) := Z(t) \odot \Gamma(t) \bmod q$
    end for
    $Z(t) := Z(t) + \alpha(t)$
    return $Z(t)$
```

Spectral Modular Exponentiation

Algorithm 3. *Spectral Modular Exponentiation (SME): Suppose that there exist a d-point DFT map and $n(b)$ is a multiple of modulus n where $n_0 = 1$, deg$(n(t)) = s - 1$, $s = \lceil d/2 \rceil$, gcd$(b, n) = 1$ and $b > 0$.*

Input: *A modulus $n > 0$ and $m, e < n$*

Output: $c \equiv m^e$ *mod* n

```
Compute $\lambda(t)$ such that $\lambda = b^{2d} \bmod n$.
$N(t)$ := DFT($n(t)$)
$\Lambda(t)$ := DFT($\lambda(t)$)
$M(t)$ := DFT($m(t)$)
$M'(t)$ := SMP($M(t), \Lambda(t), N(t)$)
$C(t)$ := SMP($1(t), \Lambda(t), N(t)$)
for $i = j - 2$ downto 0
    $C(t)$ := SMP($C(t), C(t), N(t)$)
    if $e_i = 1$ then $C(t)$ := SMP($C(t), M'(t), N(t)$)
$C(t)$ := SMP($C(t), 1(t), N(t)$)
$c(t)$ := IDFT($C(t)$)
return $c(b)$
```

An Overflow Example

Consider the integer ring $\mathbb{Z}_{31}$; the element 2 is a principal 5th root of unity and 5^{-1} exists.

Observe also that there exists a 5-point NTT over ring $\mathbb{Z}_{31}$.

DFT can be computed by a matrix multiplication where the transformation matrix $\mathcal{T}$ is given by

$$\mathcal{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2^2 & 2^3 & 2^4 \\ 1 & 2^2 & 2^4 & 2^6 & 2^8 \\ 1 & 2^3 & 2^6 & 2^9 & 2^{12} \\ 1 & 2^4 & 2^8 & 2^{12} & 2^{16} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 4 & 16 & 2 & 8 \\ 1 & 8 & 2 & 16 & 4 \\ 1 & 16 & 8 & 4 & 2 \end{bmatrix}$$

Now, consider the sequences $x = (1, 2, 3, 0, 0)$ and $y = (2, 3, 3, 0, 0)$

An Overflow Example

The transforms of x and y can be computed as

$$\begin{aligned} X &= \mathcal{T}x^T = (6, 17, 26, 23, 26) \pmod{31} \\ Y &= \mathcal{T}y^T = (8, 20, 0, 1, 12) \pmod{31} \end{aligned}$$

At this point, we perform point-wise multiplication (using convolution property), and then take the inverse transform

$$\begin{aligned} Z &= X \odot Y = (17, 30, 0, 23, 2) \pmod{31} \\ z &= \mathcal{T}^{-1}(17, 30, 0, 23, 2)^T = (2, 7, 15, 15, 9) \end{aligned}$$

which is the convolution of the signals x and y

If $x = (10, 5, 1, 0, 0)$ is taken, by the same procedure one gets $z = (20, 9, 20, 24, 9)$ which is a wrong answer caused by an overflow

Boundary analysis for SME

Let there exist a d-point DFT over $\mathbb{Z}_q$. For $s = \lceil d/2 \rceil$ and b representing the radix, we have proven the following:

Theorem: *SME computes a modular reduction, $c \equiv m^e \bmod n$ if the parameters b, q and s satisfies the following inequality*

$$(b^2 + b)^2 B(s) + b^2 s < q$$

where

$$B(s) = \frac{-2s^3}{3} - s^2 + 2s^2 r_1 - \frac{s}{3} + \frac{r_1}{3} + 2sr_1$$

for

$$r_1 = -2 + \frac{1}{3}\sqrt{3 + 18s^2 + 18s}$$

Suitable Spectrum for SME

- Complex spectrum is not appropriate

- A finite ring spectrum for which q is of the form $2^v \pm 1$ is the most suitable choice

- The rings having $2^v - 1$ elements are called the *Mersenne rings*: Arithmetic corresponds to the one's complement arithmetic

- While the rings with $q = 2^v + 1$ are called the *Fermat rings*: Arithmetic is simple if a diminished-1 representation is used.

- Principle root of unity should be taken as a power of 2

- Short transform sizes are the biggest concern for both rings

Mersenne Number Transforms

The NTT over the rings with $q = 2^v - 1$ is called the Mersenne Number transform

Some nice MNT parameters for $2^{16} < q < 2^{80}$

ring $\mathbb{Z}_q$	prime factors	(ω, MNT length)	
$2^{17} - 1$	131071	$(2, 17)$	$(-2, 34)$
$2^{19} - 1$	524287	$(2, 19)$	$(-2, 38)$
$2^{23} - 1$	$47 \cdot 178481$	$(2, 23)$	$(-2, 46)$
$2^{29} - 1$	$233 \cdot 1103 \cdot 2089$	$(2, 29)$	$(-2, 58)$
$2^{31} - 1$	2147483647	$(2, 31)$	$(-2, 62)$
$2^{37} - 1$	$223 \cdot 616318177$	$(2, 37)$	$(-2, 74)$
$2^{41} - 1$	$13367 \cdot 164511353$	$(2, 41)$	$(-2, 82)$
$2^{43} - 1$	$431 \cdot 9719 \cdot 2099863$	$(2, 43)$	$(-2, 86)$
$2^{79} - 1$	$2687 \cdot 202029703 \cdot 1113491139767$	$(2, 79)$	$(-2, 158)$

Fermat Number Transforms

The NTT over the rings with $q = 2^v + 1$ is called the Fermat Number transform

Some nice FNT parameters for $2^{16} < q < 2^{81}$:

ring $\mathbb{Z}_q$	prime factors	(ω, MNT length)	
$2^{16}+1$	65537	$(4, 16)$	$(2, 32)$
$2^{20}+1$	$17 \cdot 61681$	$(32, 8)$	$(4100, 16)$
$2^{24}+1$	$97 \cdot 257 \cdot 673$	$(8, 16)$	$(\sqrt{8}, 32)$
$2^{32}+1$	$641 \cdot 6700417$	$(4, 32)$	$(2, 64)$
$2^{40}+1$	$257 \cdot 4278255361$	$(32, 16)$	$(\sqrt{32}, 32)$
$2^{64}+1$	$274177 \cdot 67280421310721$	$(4, 64)$	$(2, 128)$
$2^{80}+1$	$414721 \cdot 44479210368001$	$(32, 32)$	$(\sqrt{32}, 64)$

Pseudo (Mersenne or Fermat) Number Transforms

- If q is a Mersenne or a Fermat number having a small divisor p (not necessarily a prime), a transform having sufficient lenght may not exist in $\mathbb{Z}_q$, but the arithmetic in $\mathbb{Z}_{q/p}$ can be still efficient

- Since p divides q, the arithmetic modulo (q/p) can be carried in the ring $\mathbb{Z}_q$ with a final modulo (q/p) reduction

- Such transforms are called Pseudo (Mersenne or Fermat) Number Transforms (PNT): They tailor the Mersenne or Fermat rings in a way that larger length transforms are possible

- **Example:** In $\mathbb{Z}_{2^{15}-1}$, the maximum transform length is $\gcd(6, 30, 150) = 6$, but if the PNT is employed in the ring $\mathbb{Z}_{(2^{15}-1)/7}$, we get the transform lengths up to

$$\gcd(30, 150) = 30$$

Pseudo (Mersenne or Fermat) Number Transforms

Some nice PNT parameters for $2^{16} < q < 2^{32}$:

Ring $\mathbb{Z}_q$	Prime Factors	Modulus $\mathbb{Z}_{q/p}$	ω	d	ω	d
$2^{17}+1$	$3 \cdot 43691$	$(2^{17}+1)/3$	$-2, 4$	17	2	34
$2^{19}+1$	$3 \cdot 174763$	$(2^{19}+1)/3$	$-2, 4$	19	2	38
$2^{20}+1$	$17 \cdot 61681$	$(2^{20}+1)/17$	4	20	2	40
$2^{21}+1$	$3^2 \cdot 43 \cdot 5419$	$(2^{21}+1)/9$	$-2, 4$	21	2	42
$2^{22}+1$	$5 \cdot 397 \cdot 2113$	$(2^{22}+1)/5$	4	22	2	44
$2^{23}+1$	$3 \cdot 2796203$	$(2^{23}+1)/3$	$-2, 4$	23	2	46
$2^{25}-1$	$31 \cdot 601 \cdot 1801$	$(2^{25}-1)/31$	2	25	-2	50
$2^{26}-1$	$3 \cdot 2731 \cdot 8191$	$(2^{26}-1)/3$	2	26	-2	52
$2^{27}-1$	$7 \cdot 73 \cdot 262657$	$(2^{27}-1)/511$	2	27	-2	54
$2^{28}+1$	$17 \cdot 15790321$	$(2^{28}+1)/17$	4	28	2	56
$2^{29}+1$	$3 \cdot 59 \cdot 3033169$	$(2^{29}+1)/3$	$-2, 4$	29	2	58
$2^{31}+1$	$3 \cdot 715827883$	$(2^{31}+1)/3$	$-2, 4$	31	2	62

Parameter Selection for RSA

	Bits	Ring $\mathbb{Z}_q$	d	ω	$u,\ b = 2^u$
SMP	518	$2^{73} - 1$	73	2	14
	1,185	$2^{79} - 1$	158	-2	15
	2,060	$(2^{103} + 1)/3$	206	2	20
Modified SMP	540	$2^{59} - 1$	59	2	18
	1,080	$2^{79} - 1$	79	2	40
	2,054	$2^{79} - 1$	158	2	26
	4,251	$2^{109} - 1$	218	-2	39

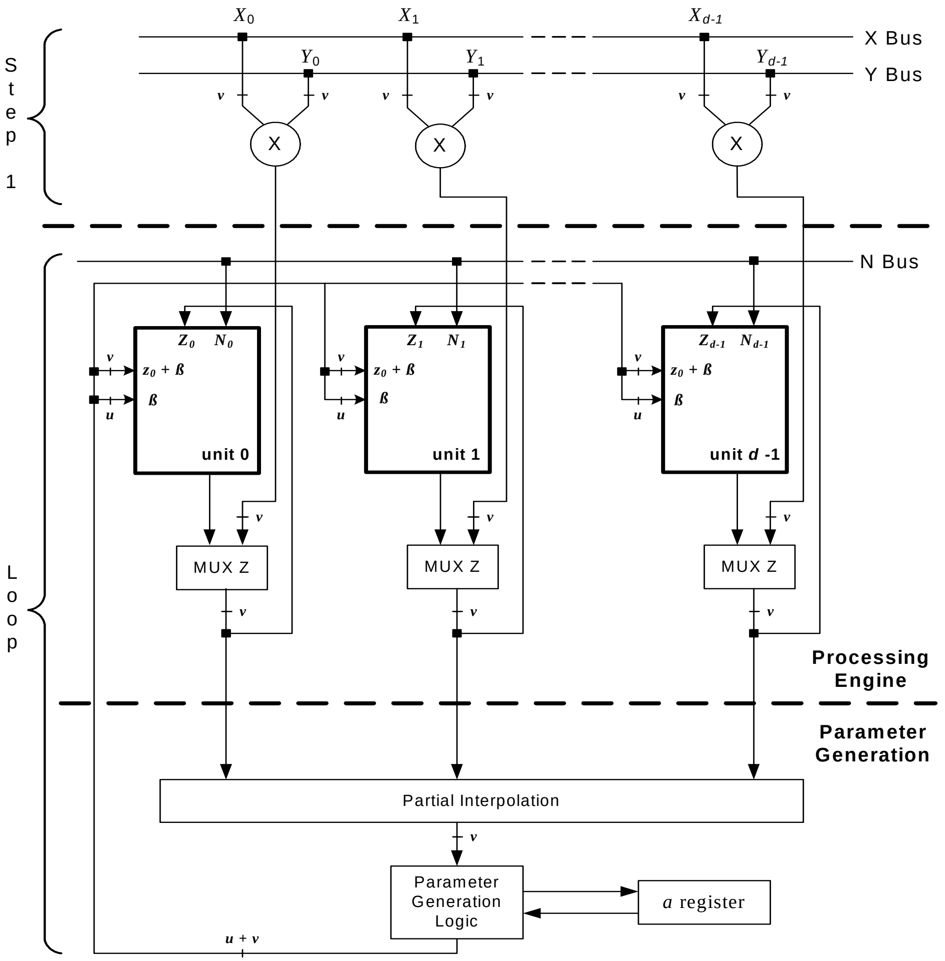
Hardware Architectures

- Spectral methods are powerful: they bring parallelism to modular exponentation computation by dividing the larger problem into smaller pieces

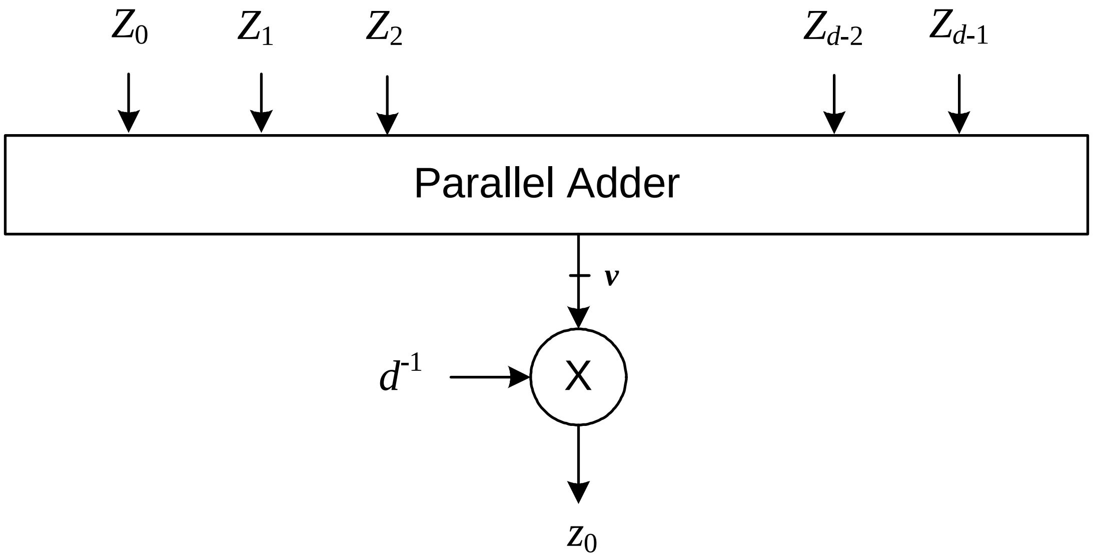
- Moreover, it is possible to employ some low level parallelism on the resulting partitions

- The main drawback is the area as in any other parallel systems

SMP Module

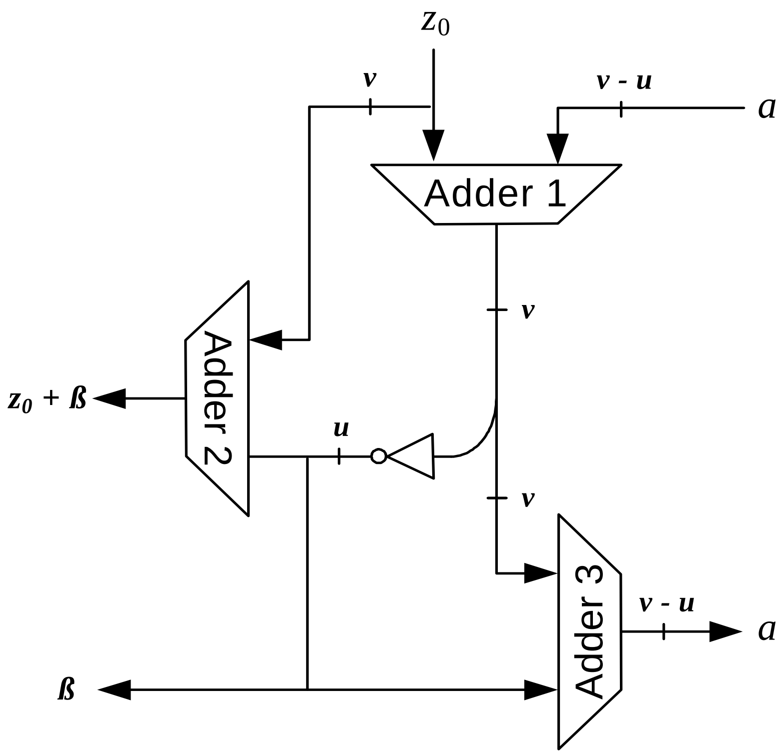

S t e p 1
X₀
X₁
X_{d-1}
X Bus
Y₀
Y₁
Y_{d-1}
Y Bus
v
X
N Bus
Z₀ N₀
Z₁ N₁
Z_{d-1} N_{d-1}
z₀ + ß
ß
u
unit 0
unit 1
unit d -1
MUX Z
L o o p
Processing Engine
Parameter Generation
Partial Interpolation
Parameter Generation Logic
a register
u + v

Partial Interpolation



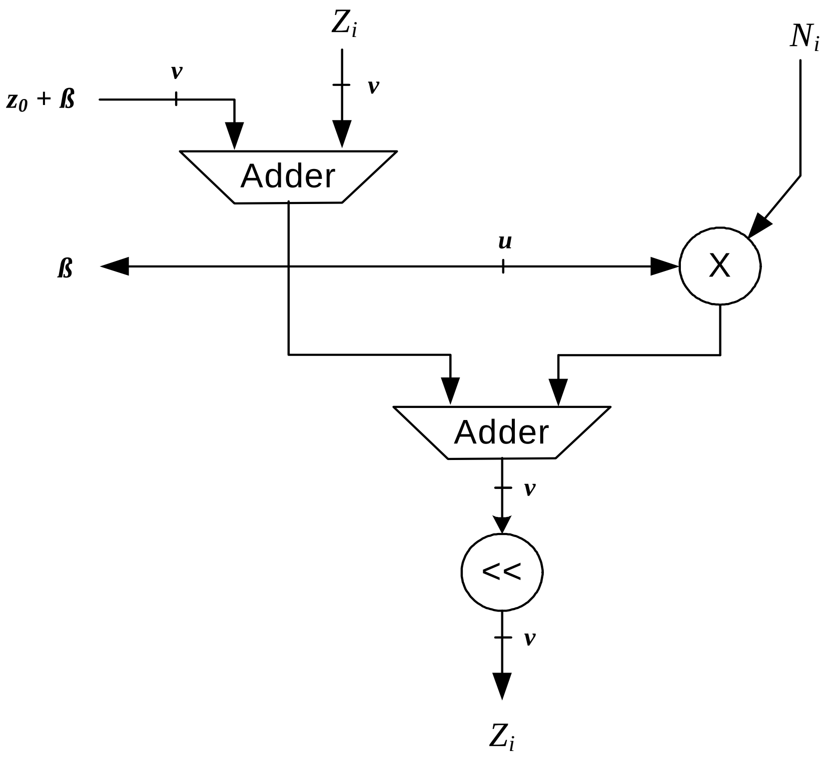
This circuit realizes step 4 of SMP, calculates the least time digit of the partial sum

Generation Logic



This circuit computes steps 5 and 6 of SMP, and generates the parameters α and β

Processing Unit



The main compuation unit corresponding to the Steps 7, 8 and 9

Performance Analysis

Under the following specific assumptions:

- Mersenne or Fermat arithmetic is used for every unit
- All adders except one are modular parallel CSA adders
- Parallel prefix adders are used for CPA
- For multi-operand addition, we use parallel Wallace tree networks

The Cost of SMP

ring	area	delay
M	$\frac{23}{2}v^2 + 7vd - 42v$ $+8uv + \frac{3}{2}u\log u + 7u$	$4\theta(v) + 1 + 2d(\lg(u) + 5)+$ $4d(\theta(d) + \theta(v/2) + \theta(u+1))$
F	$13v^2 + 8vd + 27v$ $+9uv + \frac{3}{2}u\log u + 7u$	$4\theta(v+1) + 4 + 2d(\lg(u) + 19/2)+$ $4d(\theta(d+1) + \theta(v/2+1) + \theta(u+2))$

d is the DFT length, $q = 2^v \pm 1$, $b = 2^u$ and $\theta(x)$ is as follows:

x	3	4	5–6	7–9	10–13	14–19	20–28	29–42	$\cdots$
$\theta(x)$	1	2	3	4	5	6	7	8	$\cdots$

A Sample Cost Computation

- Let's compute the complexity of SME with a modulus 704-bits by using the DFT in the ring $\mathbb{Z}_q$ with $q = 2^{64} + 1$, $\omega = 2$ and $u = 11$. Plugging these into the cost function gives

$$
\begin{aligned}
T_{SMP} &= 4\theta(v+1) + 4 + 2d(\lg(u) + 8) + 4d(\theta(d+1) + \theta(u+2)) \\
&= 4 \cdot 10 + 4 + 256(4+8) + 512(11+5) \\
&= 11308 \\
A_{SMP} &= 9v^2 + 8vd + 25v + 9uv + \frac{3}{2}u \log u + 7u \\
&= 177871
\end{aligned}
$$

- A realization of SMP with the above parameters requires 177871 gates and has 11308 gate delays for an output.
- If this SMP is used in a SME, with the same area complexity, one approximately has $11308K$ gate delays.

Conclusions

- This work is the first utilization of spectral techniques for modular arithmetic
- The asymptotic crossovers for modular exponentiation are brought to cryptographic sizes
- A boundary analysis for optimal domain and related parameter selection are given
- Highly parallel architectures for hardware realizations of the RSA, DH, DSA, and ECC over prime fields are proposed

Current and Future Work

- The use of complex spectrum can be investigated by performing a thorough roundoff error analysis

- The use of finite rings with modulus of the form $2^u \pm 2^v \pm 1$ gives longer DFT lengths with a slightly increased complexity

- Multidimensional transform methods can be investigated

- Extensions to point multiplication on elliptic curves over $GF(2^k)$ and $GF(p^k)$ can be be analyzed

- Actual hardware implementations need to be realized: VHDL and Verilog coding, FPGA and ASIC prototyping

Publications

1. G. Saldamlı and Ç. K. Koç. *Spectral Modular Arithmetic Method and Apparatus*, US Patent Application, April 2005.
2. G. Saldamlı and Ç. K. Koç. Spectral modular exponentiation. *Proceedings, 18th IEEE Symposium on Computer Arithmetic*, P. Kornerup and J.-M. Muller, editors, pages 123-130, IEEE Computer Society Press, Montpellier, France, June 25-27, 2007.
3. R. C. C. Cheung, Ç. K. Koç, and J. D. Villasenor. An efficient hardware architecture for Spectral Hash algorithm. *Proceedings, 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 215-218, IEEE Press, Boston, MA, July 7-9, 2009.
4. G. X. Yao, R. C. C. Cheung, Ç. K. Koç, and K. F. Man. Reconfigurable number theoretic transform architectures for cryptographic applications. *The 2010 International Conference on Field-Programmable Technology (FPT)*, pages 308-311, Beijing, China, December 8-10, 2010.
5. G. Saldamli, Y.-J. Baek, and Ç. K. Koç. Spectral modular arithmetic for binary extension fields. *The 2011 International Conference on Information and Computer Networks (ICICN)*, pages 323-328, Guiyang, China, January 26-28, 2011.

MPC180E Security Processor User's Manual

Rev. 2.1, 11/2000

PRELIMINARY—SUBJECT TO CHANGE WITHOUT NOTICE

Chapter 1 Overview

This chapter gives an overview of the MPC180E security processor, including the key features, typical system architecture, and MPC180E architecture.

1.1 Features

The MPC180E is designed to work with Motorola's PowerQUICC™ family of processors. The MPC180E interfaces gluelessly to both the PowerQUICC and PowerQUICC II™, accelerating the performance of computationally-intensive security functions, such as key generation and exchange, authentication, and bulk encryption. Support for 66MHz bus frequencies enables maximum utilization of the MPC8260 local bus as well as enhanced versions of the MPC8xx system bus.

The MPC180E is optimized to quickly process all the algorithms associated with IPSec, WTLS/WAP, SSL/TLS, and IKE, including RSA, RSA signature, Diffie-Hellman, Elliptic Curve Cryptography, DES, 3DES, SHA-1, MD4, MD5, and Arc Four.

Major features of MPC180E are as follows:

- Public key/ asymmetric key
  - RSA
    - Programmable field size of up to 2048 bits
- Elliptic curve cryptography
  - F_2m and F(p) modes
  - Programmable field size of up to 511 bits
- Symmetric key
  - DES
    - ECB (Electronic Code Book)
    - CBC (Cipher Block Chaining)
  - 3DES
    - Two-key (K1 = K3) or three-key (K1≠ K3) Triple-DES.
  - Arc Four Stream Cipher
    - key lengths of 40–128 bits

Figure 2-1 shows the MPC180E pinout.

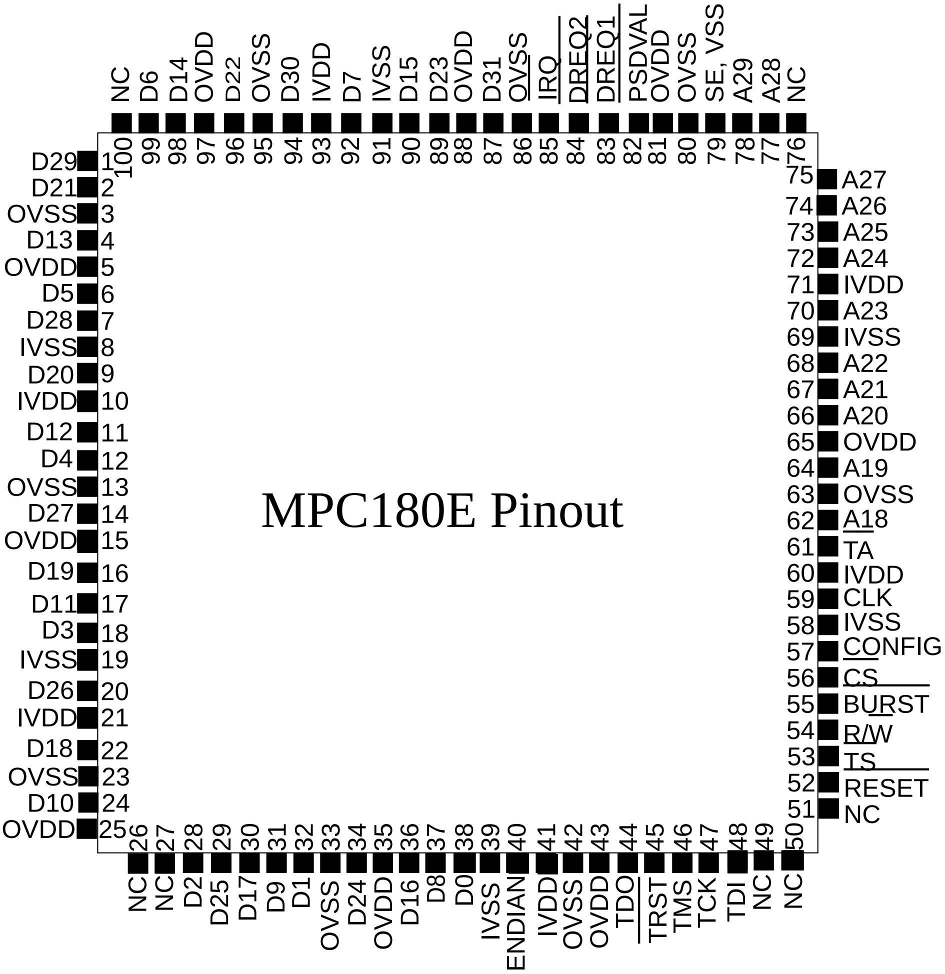


Figure 2-1. MPC180E Pin Diagram

7.3 ECC Routines

7.3.1 ECC F_p Point Multiply

The PKEU performs the Elliptic Curve point multiply function which is the highest level of ECC abstraction supported by the device. It is the intention that the host processor use the PKEU in such a way as to support ECC schemes defined in IEEE P1363 (and other ECC standards) where the point multiply is the critical and most computationally intensive, but not final, step in many of these schemes. The point multiply is performed in a near fully-automated fashion; however, there is some interaction required by the host processor (described below).

Point multiplies in F_p are carried out by the PKEU by performing repeated point add and point double operations using projective coordinates. As a result, the host processor is responsible for providing the point P represented as the point (X, Y, Z). For systems that do not operate in the projective coordinate scheme (i.e. point P is represented as the point (x,y)), X is simply x, Y is y, and Z is 1. The complete set of I/O conditions is shown below.

NOTE:

The scalar ‘k’ is assumed to be positive. If k = 0, the results of the point multiply are (1, 1, 0). If k < 0, then k ← (-k) and Y ← -Y (modP).

NOTE:

The input ‘Z’ is assumed to be non-zero. If zero, then the results of the point multiply are (1, 1, 0).

Table 7-5. ECC F_p Point Multiply

	F_p Point Multiply
Computation	Q = k*P, where Q ≡ (X_3,Y_3,Z_3), P ≡ (X_1,Y_1, Z_1)
Entry name	multkPtoQ
Entry address	0x001(FpmultkPtoQ)

Pre-conditions	A0 = x_1 (non-projective coordinate when XYZ=0) or X_1 (projective coordinate when XYZ=1)
	A1 = y_1 (non-projective coordinate when XYZ=0) or Y_1 (projective coordinate when XYZ=1)
	A2 = (z_1≡1) (non-proj. coordinate when XYZ=0) or Z_1 (projective coordinate when XYZ=1)
	A3 = a elliptic curve parameter
	B0 = b elliptic curve parameter
	B1 = R^2 mod N value
	N0 = prime p (modulus) of the ECC system

Run-time conditions	EXP(k) = ms 32-bits of k (provided in 32 bit words throughout the point multiply, msb to lsb); first word provides following routine invocation per ERDY assertion.

Table 7-5. ECC F_p Point Multiply (Continued)

	F_p Point Multiply

Post-conditions	B1 = X_2 / X'_2
	B2 = Y_2 / Y'_2
	B3 = Z_2 / Z'_2
	A2 = undefined (when XYZ = 1) or Z_2^2 (when XYZ = 0)
	A3 = undefined (when XYZ = 1) or Z_2^3 (when XYZ = 0)
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	—

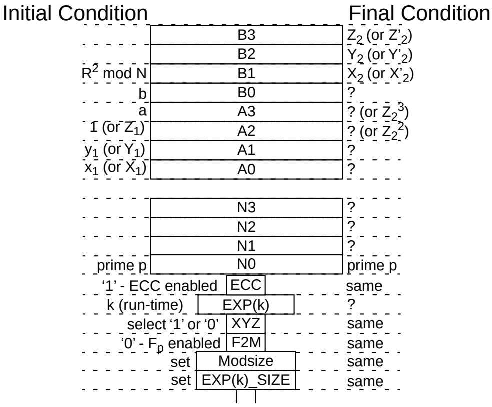


Figure 7-4. ECC F_p Point Multiply Register Usage

It is important to note that unlike the RSA exponentiation routine, the point to be multiplied is not expected to be in the Montgomery residue system when loaded into the PKEU. All of the other ECC parameters are also expected to be loaded in standard format. This includes the a and b parameters of the ECC system. In addition, the "R^2 mod N" term is also required. This term is used by the PKEU to put the operands in the Montgomery residue system. See the full description of this function/value below.

It is the responsibility of the host processor to provide multiplier data to the PKEU during the operation. That is, the 'k' from the point multiplication 'kP' must be provided dynamically by the host micro-processor in 32-bit words. Note that the host must supply the k data starting with the most significant 32-bit word and working down to the least significant word. Each individual word, however, is formatted msb to lsb (i.e. "k_word[msb:lsb]").

PKEU asserts the IRQ signal when it is ready to accept more data. This tells the host processor to read PKSR to see what was set. If the E_RDY bit is set, the host processor knows it must provide the next word of k - this data is written into the EXP(k) register one 8-bit word at a time. If this interrupt bit is masked, then it must poll the status register to determine when to provide the next word of k. The host should not look for the assertion of E_RDY until after the routine (i.e. PKCR[GO] bit). Any data written to EXP(K) prior to this will be ignored.

Pin IRDY_B also is used to signify when PKEU is ready for the next 32 bit word of EXP(k). IRDY_B is active (low) whenever E_RDY bit in the status register is active (high).

The point multiplication is optimized to efficiently produce results for systems that work in the projective coordinate scheme but can accelerate affine schemes as well. The host processor selects the scheme via the PKCR XYZ bit.

For affine coordinate systems (CR [XYZ]= 0):

The results of the calculation are returned to the A and B storage registers. Note that these values correspond to the projective coordinate values X, Y, Z, Z^2, and Z^3. X, Y, and Z are in the Montgomery residue system. In order to put the projective coordinates into their affine form, the following equations which define their relationships must be calculated:

$x = X/Z^2$;

$y = Y/Z^3$;

Because the PKEU does not support the inverse function, it is the responsibility of the host processor to find $(Z^2)^{-1}$ and $(Z^3)^{-1}$ by using any number of available modulo-n inversion techniques. Once this is accomplished, the host may then provide these values back to the PKEU to perform the final two *field* (modular) multiplications to find *x* and *y*. It is advisable that the user perform these multiplications in the PKEU to remove the values from the Montgomery residue system.

For projective coordinate systems (Control Register Bit XYZ = 1):

The results of the calculation are returned to the B memory. Note that these values correspond to the projective coordinate values X, Y, and Z and are *no longer* in the Montgomery residue system. The host may take these results as the complete point multiply (including the exit from the Montgomery residue system) (e.g. $(XR)(Z^2)^{-1}R^{-1}$ modN = X).

The following restrictions apply to the point multiply:

- The value of the k vector must be greater than one for this function to work properly.
- The point multiply operates with a minimum of five digits (Modsize = 4).

7.3.2 ECC F_p Point Add

This function is extensively utilized by the point multiply routine. However, its value as a stand-alone routine to the host processor is extremely limited. As a result, the information provided on the routine is primarily for testing and debug purposes.

Table 7-6. ECC F_p Point Add

	F_p Point Add
Computation	R = P + Q, where R ≡ (X_3,Y_3,Z_3), P ≡ (X_1,Y_1, Z_1), and Q ≡ (X_2,Y_2, Z_2)
Entry name	FpaddPtoQ
Entry address	0x002(FpaddPtoQ)

Pre-conditions	A0 = X'_1 (projective coordinate in Montgomery residue system)
	A1 = Y'_1 (projective coordinate in Montgomery residue system)
	A2 = Z'_1 (projective coordinate in Montgomery residue system)
	A3 = a’ (elliptic curve parameter in Montgomery residue system)
	B0 = b’ (elliptic curve parameter in Montgomery residue system)
	B1 = X'_2 (projective coordinate in Montgomery residue system)
	B2 = Y'_2 (projective coordinate in Montgomery residue system)
	B3 = Z'_2 (projective coordinate in Montgomery residue system)
	N0 = prime p (modulus) of the ECC system

Post-conditions	A0 = X'_1
	A1 = Y'_1
	A2 = Z'_1
	A3 = a’
	B0 = b’
	B1 = X'_3
	B2 = Y'_3
	B3 = Z'_3
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	All variables followed with the tick mark (‘) indicate it is in the Montgomery residue system.

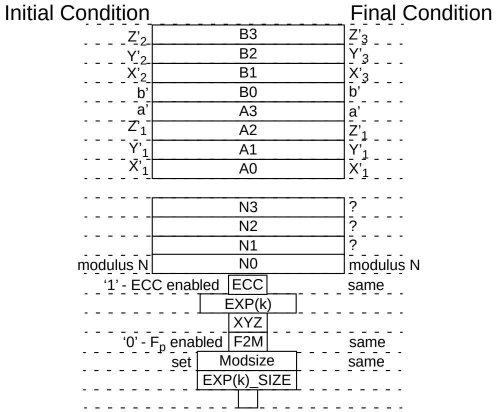


Figure 7-5. ECC F_p Point Add Register Usage

7.3.3 ECC F_p Point Double

This function is extensively utilized by the point multiply routine. However, its value as a stand-alone routine to the host processor is extremely limited. As a result, the information provided on the routine is primarily for testing and debug purposes.

Table 7-7. ECC F_p Point Double

	F_p Point Double
Computation	R = Q + Q = 2 * Q, where R ≡ (X_3,Y_3,Z_3), and Q ≡ (X_3,Y_3, Z_3)
Entry name	FpdoubleQ
Entry address	0x003(FpdoubleQ)

Pre-conditions	B1 = X'_1 (projective coordinate in Montgomery residue system)
	B2 = Y'_1 (projective coordinate in Montgomery residue system)
	B3 = Z'_1 (projective coordinate in Montgomery residue system)
	A3 = a' (elliptic curve parameter in Montgomery residue system)
	B0 = b' (elliptic curve parameter in Montgomery residue system)
	N0 = prime p (modulus) of the ECC system

Post-conditions	B1 = X'_3
	B2 = Y'_3
	B3 = Z'_3
	A3 = a'
	B0 = b'
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	All variables followed with the tick mark (‘) indicate it is in the Montgomery residue system. While not explicitly mentioned or necessary, the contents registers A0, A1, and A2 a left undisturbed in anticipation that these will store the generator point (P) during a point multiply.

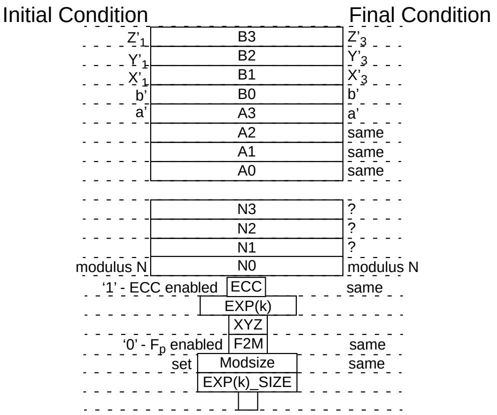


Figure 7-6. ECC F_p Point Double Register Usage

7.3.4 ECC F_p Modular Add

Modular addition may be performed on any two vectors loaded into A (A0-A3) and B (B0-B3), where both of these vectors are less than the value stored in the modulus register N (N0-N3). The results are stored in the respective B register. For ECC functionality, this function is used by the point add and point double routines but is available to the host interface - typically for higher-level ECC-related functions. This function operates with a minimum of four digits (Modsize = 3).

Prior to initiating this function, the A, B and N register pointers must be set in the control register which indicate which sub-registers (e.g A0, B0, A1, B1, etc.) are the targeted operands. See Table 7-2 for a detailed description. Once this is performed, the host processor may successfully initiate this function.

Table 7-8. Modular Add

	Modular Add
Computation	C = D + E mod N, where D, E, and C are integers and are less than N
Entry name	modularadd
Entry address	0x008(modularadd)

Pre-conditions	A0-3 = D (integer, exact A-location pre-selected in Control Register)
	B0-3 = E (integer, exact B-location pre-selected in Control Register)
	N0-3 = prime p (modulus) of the ECC system

Post-conditions	B0-3 = results of modular addition stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	The function operates the same regardless of whether or not the operands are in the Montgomery residue system.

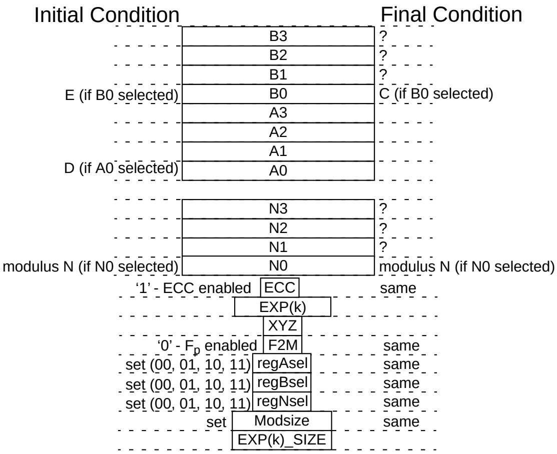


Figure 7-7. Modular Add Register Usage

7.3.5 ECC F_p Modular Subtract

Modular subtraction may be performed on any two vectors loaded into A (A0–A3) and B (B0–B3), where both of these vectors are less than the value stored in the modulus register N (N0–N3). This is accomplished by computing A-B if A > B or A-B+N if A < B. The results are stored in the respective B register. For ECC functionality, this function is used by the point add and point double routines but is available to the host interface. This function operates with a minimum of four digits (Modsize = 3).

Before this function is initialized, the A, B and N register pointers must be set in the control register which indicate which sub-registers (A0, B0, A1, B1, etc.) are the targeted operands. See Table 7-2 for a detailed description. Once this is performed, the host processor may successfully initiate this function.

Table 7-9. Modular Subtract

	Modular Subtract
Computation	C = D - E mod N, where D, E, and C are integers and are less than N
Entry name	modularsubtract;
Entry address	009h(modularsubtract)

Pre-conditions	A0-3 = D (integer, exact A-location pre-selected in Control Register)
	B0-3 = E (integer, exact B-location pre-selected in Control Register)
	N0-3 = prime p (modulus) of the ECC system

Post-conditions	B0-3 = results of modular subtraction stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	The function operates the same regardless of whether or not the operands are in the Montgomery residue system.

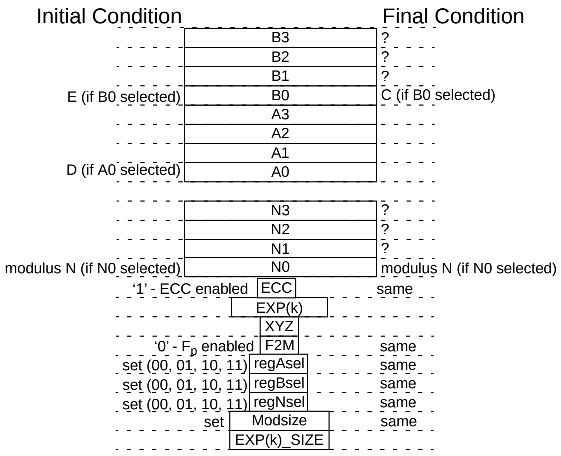


Figure 7-8. Modular Subtract Register Usage

7.3.6 ECC F_p Montgomery Modular Multiplication $((A \times B \times R^{-1}) \bmod N)$

The $(A \times B \times R^{-1}) \bmod N$ calculation is the core function of the PKEU. It is used to assist the point add and double routines in completing their functions. For ECC purposes, this function will rarely be used directly by the host processor. This function operates with a minimum of five digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-10. Modular Multiplication

	Modular Multiply
Computation	C = A * B * R^{-1} mod N, where A, B, and C are integers less than N and R = 2^{16D} where D is the number of digits of the modulus vector
Entry name	modularmultiply
Entry address	0x00a(modularmultiply)

Pre-conditions	A0-3 = A (integer, exact A-location pre-selected in Control Register)
	B0-3 = B (integer, exact B-location pre-selected in Control Register)
	N0-3 = prime p (modulus) of the ECC system

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	Typically, though it is not mandatory, the operands will be in the Montgomery residue system. The only time this would not be the case is when manually placing a value into the Montgomery residue system.

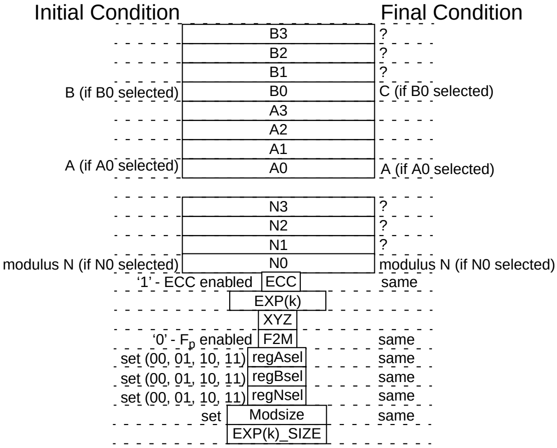


Figure 7-9. Modular Multiplication Register Usage

7.3.7 ECC F_p Montgomery Modular Multiplication ($(A \times B \times R^{-2}) \bmod N$)

The $(A \times B \times R^{-2}) \bmod N$ calculation is similar to the standard ‘R^{-1}’ Montgomery multiplication except an additional R is divided out. This function is ideal for those ECC applications which work in affine coordinates. In that case, the host may use this function to exit projective coordinates. For example, the host could find x, for $x = X/Z^2$, where X and $(Z^2)^{-1}$ are in the Montgomery residue system. Loading X and $(Z^2)^{-1}$ into the appropriate operand registers and initiating this function would yield x which is no longer in the Montgomery residue system. This function operates with a minimum of 5 digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-11. Modular Multiplication (with double reduction)

	Modular Multiply (with double reduction)
Computation	C = A * B * R^{-2} mod N, where A, B, and C are integers less than N and R = 2^{16D} where D is the number of digits of the modulus vector
Entry name	modularmultiply2
Entry address	0x00b (modularmultiply2)

Pre-conditions	A0-3 = A (integer, exact A-location pre-selected in Control Register)
	B0-3 = B (integer, exact B-location pre-selected in Control Register)
	N0-3 = prime p (modulus) of the ECC system

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	—

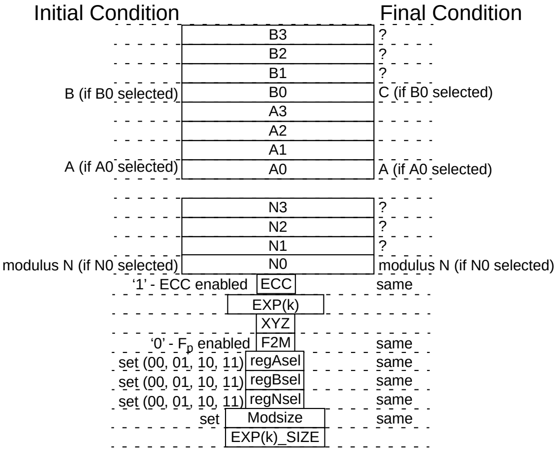


Figure 7-10. Modular Multiplication (with double reduction) Register Usage

7.3.8 ECC F$_2$m Polynomial-Basis Point Multiply

The PKEU performs the elliptic curve point multiply function which is the highest level of ECC abstraction supported by the device. It is the intention that the host processor use the PKEU in such a way as to support ECC schemes defined in IEEE P1363 (and other ECC standards) where the point multiply is the critical and most computationally intensive, but not final, step in many of these schemes. The point multiply is a nearly fully automated. However, some interaction is required by the host processor (described below).

Point multiplies in F$_2$m are carried out by the PKEU by performing repeated point add and point double operations using projective coordinates. As a result, the host processor is responsible for providing the point P represented as the point (X, Y, Z). For systems that do not operate in the projective coordinate scheme (that is, point P is represented as the point (x, y)), X is simply x, Y is y, and Z is 1. The complete set of I/O conditions is shown below:

Table 7-12. ECC F$_2$m Point Multiply

	F$_2$m Point Multiply
Computation	Q = k*P, where Q ≡ (X_3,Y_3,Z_3), P ≡ (X_1,Y_1, Z1)
Entry name	multkPtoQ(will probably be the same as F$_p$)
Entry address	0x001(multkPtoQ)

Pre-conditions	A0 = x_1 (when XYZ=0) or X_1 (when XYZ=1)
	A1 = y_1 (when XYZ=0) or Y_1 (when XYZ=1)
	A2 = (z_1≡1) (when XYZ=0) or Z_1 (when XYZ=1)
	A3 = a elliptic curve parameter
	B0 = c elliptic curve parameter
	B1 = R^2 mod N value
	N0 = prime p (modulus) of the ECC system

Run-time conditions	EXP(k) = ms 8-bits of k (provided in 8 bit words throughout the point multiply, msb to lsb); first word provides following routine invocation per ERDY assertion.

Post-conditions	B1 = X_2 / X'$_2$
	B2 = Y_2 / Y'$_2$
	B3 = Z_2 / Z'$_2$
	A2 = undefined (when XYZ = 1) or Z_2^2 (when XYZ = 0)
	A3 = undefined (when XYZ = 1) or Z_2^3 (when XYZ = 0)
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	The 'c' elliptic curve parameter is a function of the 'b' parameter and field size: $c = b^{2^{m-2}}$.

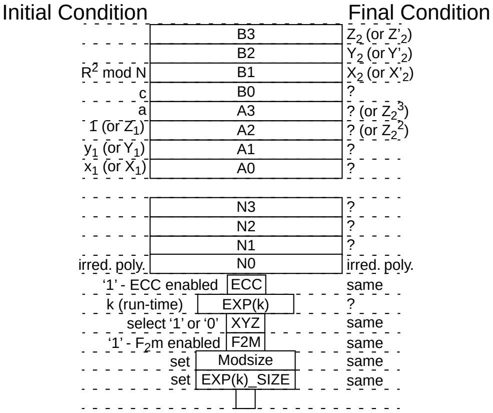


Figure 7-11. ECC F_2m Point Multiply I/O

It is important to note that unlike the RSA exponentiation routine, the point to be multiplied is not expected to be in the Montgomery residue system when loaded into the PKEU. All of the other ECC parameters are also expected to be loaded in standard format. This includes the a, c, and modulus parameters of the ECC system. In addition, the "R^2 mod N" term is also required. This term is used by the PKEU to put the operands in the Montgomery residue system. See the full description of this function below.

It is the responsibility of the host processor to provide multiplier data to the accelerator during the operation. That is, the 'k' from the point multiplication 'kP' must be provided dynamically by the host micro-processor in 32-bit words. Note that the host must supply the k data starting with the most significant 32-bit word and working down to the least significant word. Each individual word, however, is formatted msb to lsb (i.e. "k_word[msb:lsb]").

PKEU asserts the IRQ signal when it is ready to accept more data. This tells the host processor to read the status word to see what was set. If the E_RDY bit is set (or pin IRDY_B active low), the host processor knows it must provide the next word of k - this data is written into the EXP(k) register one 32-bit word at a time. If this interrupt is masked, then it must poll the status register to determine when to provide the next word of k. The host should not look for the assertion of E_RDY until after the routine (i.e. PKCR[GO] bit). Any data written to EXP(K) prior to this will be ignored.

The point multiplication is optimized to efficiently produce results for systems that work in the projective coordinate scheme but can accelerate affine schemes as well. The host processor selects the scheme via the CR XYZ-bit.

For affine coordinate systems (XYZ = 0):

The results of the calculation are returned to the A and B storage registers. Note that these values correspond to the projective coordinate values X, Y, Z, Z^2, and Z^3. X, Y, and Z are in the Montgomery residue system. In order to put the projective coordinates into their affine form, the following equations which define their relationships must be calculated:

$x = X/Z^2$;

$y = Y/Z^3$;

Since the PKEU does not support the inverse function, it is the responsibility of the host processor to find $(Z^2)^{-1}$ and $(Z^3)^{-1}$ by using any number of available modulo-irreducible-polynomial inversion techniques. Once this is accomplished, the host may then provide these values back to the PKEU to perform the final two *field* multiplications to find *x* and *y*. It is advisable that the user perform these multiplications in the PKEU to remove the values from the Montgomery residue system.

For projective coordinate systems (XYZ = 1):

The results of the calculation are returned to the B memory. Note that these values correspond to the projective coordinate values X, Y, and Z and are *no longer* in the Montgomery residue system. The host may take these results as the complete point multiply (including the exit from the Montgomery residue system) (e.g. $(XR)(Z^2)^{-1}R^{-1}$ modN = X).

The following restrictions apply to the point multiply:

- The value of the k vector must be greater than one for this function to work properly.
- The point multiply operates with a minimum of five digits (Modsize = 4).

7.3.9 ECC F_2m Point Add

This function is extensively utilized by the point multiply routine. However, its value as a stand-alone routine to the host processor is extremely limited. As a result, the information provided on the routine is primarily for testing and debug purposes.

Table 7-13. ECC F_2m Point Add

	F_2m Point Add
Computation	R = P + Q, where R ≡ (X_3,Y_3,Z_3), P ≡ (X_1,Y_1, Z_1), and Q ≡ (X_2,Y_2, Z_2)
Entry name	F_2maddPtoQ
Entry address	0x005(F_2maddPtoQ)

Pre-conditions	A0 = X'_1 (projective coordinate in Montgomery residue system)
	A1 = Y'_1 (projective coordinate in Montgomery residue system)
	A2 = Z'_1 (projective coordinate in Montgomery residue system)
	A3 = a' (elliptic curve parameter in Montgomery residue system)
	B0 = c' (elliptic curve parameter in Montgomery residue system)
	B1 = X'_2 (projective coordinate in Montgomery residue system)
	B2 = Y'_2 (projective coordinate in Montgomery residue system)
	B3 = Z'_2 (projective coordinate in Montgomery residue system)
	N0 = irreducible polynomial of the ECC system

Post-conditions	A0 = X'_1
	A1 = Y'_1
	A2 = Z'_1
	A3 = a'
	B0 = c'
	B1 = X'_3
	B2 = Y'_3
	B3 = Z'_3
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	The c elliptic curve parameter is a function of the 'b' parameter and field size: $c = b^{2^{m-2}}$. All variables followed with the tick mark (') indicate it is in the Montgomery residue system.

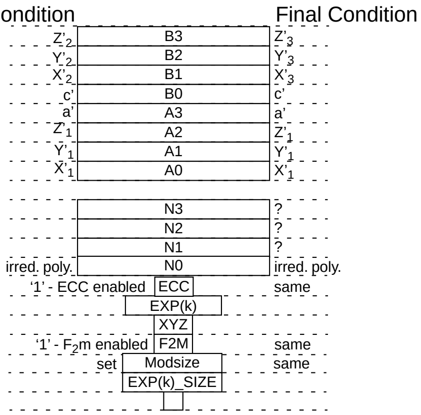


Figure 7-12. ECC F_2m Point Add Register Usage

7.3.10 ECC F_2m Point Double

This function is extensively utilized by the point multiply routine. However, its value as a stand-alone routine to the host processor is extremely limited. As a result, the information provided on the routine is primarily for testing and debug purposes.

Table 7-14. ECC F_2m Point Double

	F_2m Point Double
Computation	R = Q + Q = 2 * Q, where R ≡ (X_3,Y_3,Z_3), and Q ≡ (X_3,Y_3, Z_3)
Entry name	F_2mdoubleQ
Entry address	0x006(F_2mdoubleQ)

Pre-conditions	B1 = X'_1 (projective coordinate in Montgomery residue system)
	B2 = Y'_1 (projective coordinate in Montgomery residue system)
	B3 = Z'_1 (projective coordinate in Montgomery residue system)
	A3 = a' (elliptic curve parameter in Montgomery residue system)
	B0 = c' (elliptic curve parameter in Montgomery residue system)
	N0 = prime p (modulus) of the ECC system

Post-conditions	B1 = X'_3
	B2 = Y'_3
	B3 = Z'_3
	A3 = a'
	B0 = c'
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	The c elliptic curve parameter is a function of the 'b' parameter and field size: $c = b^{2^{m-2}}$. All variables followed with the tick mark (') indicate it is in the Montgomery residue system. While not explicitly mentioned or necessary, the contents registers A0, A1, and A2 a left undisturbed in anticipation that these will store the generator point (P) during a point multiply.

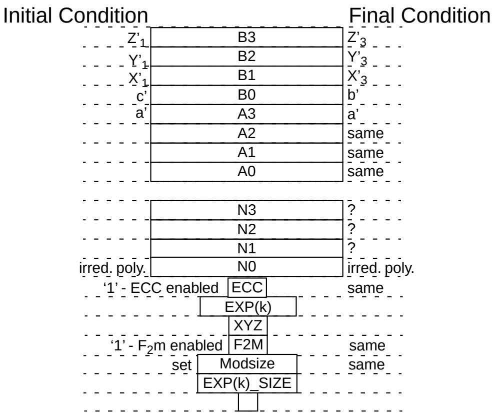


Figure 7-13. ECC F_2m Point Double Register Usage

7.3.11 ECC F_2m Add (Subtract)

Field addition in F_2m (polynomial-basis) may be performed on any two vectors loaded into A (A0-A3) and B (B0-B3), where both of these vectors are less than the value stored in the modulus (irreducible polynomial) register N (N0-N3). The results are stored in the respective B register. In F_2m, this function provides identical results for both addition as well as subtraction, therefore, it is sufficient to support both of these functions with this single routine. This function operates with a minimum of 4 digits (Modsize = 3).

Prior to initiating this function, the A, B, and N register pointers must be set in the Control Register which indicate which sub-registers (e.g A0, B0, A1, B1, etc.) are the targeted operands. See Control Register description for a detailed description. Once this is performed, the host processor may successfully initiate this function.

Table 7-15. F_2m Modular Add (Subtract)

	F_2m Modular Add (Subtract)
Computation	C = D + E mod N, where D, E, and C are integers and are less than N
Entry name	modularadd (same as with integer add)
Entry address	0x008(modularadd)
Pre-conditions	A0-3 = D (binary polynomial, exact A-location pre-selected in control register) B0-3 = E (binary polynomial, exact B-location pre-selected in control register) N0-3 = irreducible polynomial of the ECC system
Post-conditions	B0-3 = results of modular addition (subtraction) stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	The function operates the same regardless of whether or not the operands are in the Montgomery residue system.

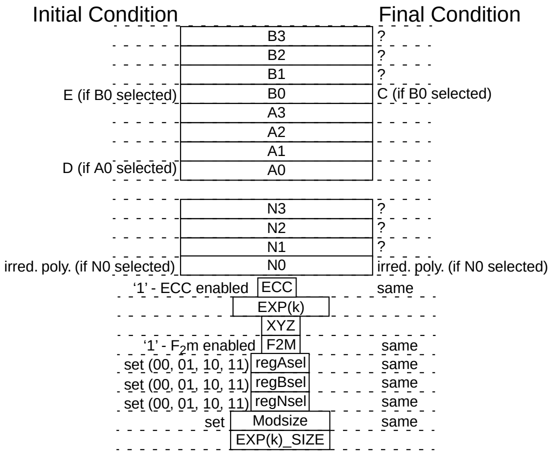


Figure 7-14. F_2m Modular Add (Subtract) Register Usage

7.3.12 ECC F_2m Montgomery Modular Multiplication ($(A \times B \times R^{-1}) \bmod N$)

The $(A \times B \times R^{-1})$ mod N calculation is the core function of the PKEU. This function is used to assist the point add and double routines in completing their functions. For ECC purposes, this function will rarely be used directly by the host processor. This function operates with a minimum of 5 digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-16. F_2m Modular Multiplication

	F_2m Modular Multiply
Computation	C = A * B * R^{-1} mod N, where A, B, and C are integers less than N and R = 2^{16D} where D is the number of digits of the modulus vector
Entry name	modularmultiply (same for F_p or F_2m)
Entry address	0x00a(modularmultiply)

Pre-conditions	A0-3 = A (binary polynomial, exact A-location pre-selected in Control Register)
	B0-3 = B (binary polynomial, exact B-location pre-selected in Control Register)
	N0-3 = irreducible polynomial of the ECC system

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	Typically, though it is not mandatory, the operands will be in the Montgomery residue system. The only time this would not be the case is when manually placing a value into the Montgomery residue system.

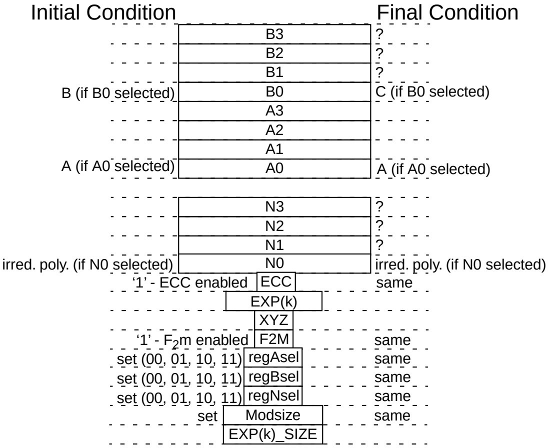


Figure 7-15. F_2m Modular Multiplication Register Usage

7.3.13 ECC F_2m Montgomery Modular Multiplication ($(A \times B \times R^{-2}) \bmod N$)

The $(A \times B \times R^{-2}) \bmod N$ calculation is similar to the standard 'R^{-1}' Montgomery multiplication except an additional R is divided out. This function is ideal for those ECC applications which work in affine coordinates. In that case, the host may use this function to exit projective coordinates. For example, the host could find x, for $x = X/Z^2$, where X and $(Z^2)^{-1}$ are in the Montgomery residue system. Loading X and $(Z^2)^{-1}$ into the appropriate operand registers and initiating this function would yield x which is no longer in the Montgomery residue system. This function operates with a minimum of 5 digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-17. F_2m Modular Multiplication (with double reduction)

	F_2m Modular Multiply (with double reduction)
Computation	$C = A * B * R^{-2}$ mod N, where A, B, and C are binary polynomials with order than N and $R = 2^{16D}$ where D is the number of digits of the irreducible polynomial
Entry name	modularmultiply2 (same as F_p)
Entry address	0x00b (modularmultiply2)

Pre-conditions	A0-3 = A (binary polynomial, exact A-location pre-selected in Control Register)
	B0-3 = B (binary polynomial, exact B-location pre-selected in Control Register)
	N0-3 = irreducible polynomial of the ECC system

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	—

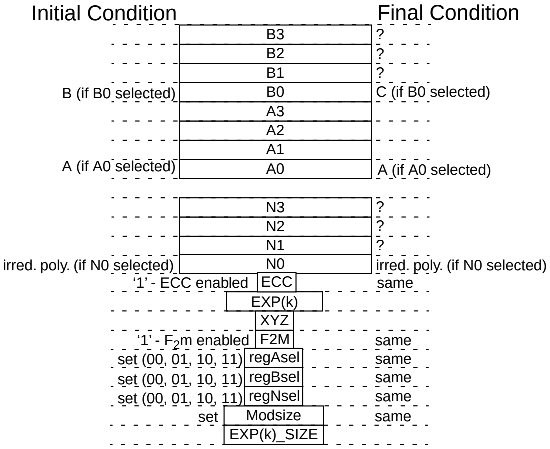


Figure 7-16. F_2m Modular Multiplication (with double reduction) Register Usage

7.4 RSA Routines

For the RSA-related descriptions which follow, it is generally recommended that all memory block pointers (regAsel, regBsel, etc.) are set to zero. For the modular exponentiation routine, the pointers are actually ignored. For the multiplies, add, subtract, and R^2 functions, it is possible to set these pointers and have the PKEU adhere to these settings.

While potentially dangerous due to the commonly large sizes of RSA operands, this flexibility is allowed to support Chinese Remainder Theorem (CRT). CRT often generates intermediate values which must be stored for later use. By using pointers, these values may be stored in the PKEU and efficiently used again without the host having to store/retrieve these values to/from general memory. It is left to the application developer to use these tools to support CRT.

7.4.1 $(A \times R^{-1})^{EXP}$ mod N

The PKEU carries out exponentiations by repeated multiply operations. The multiplies are controlled internally by the PKEU, however, it is the responsibility of the host processor to provide exponent data (32-bit words at a time) to the accelerator *during* the operation. Note that the host must supply the exponent data starting with the most significant 32-bit word and working down to the least significant word. Each individual word, however, is formatted msb to lsb (i.e. “exp_word[msb:lsb]”).

PKEU asserts the IRDY_B and IRQ signals when it is ready to accept more exponent data (IRQ only if E_RDY is not masked). This tells the host processor to read the SR to see what was set. If the E_RDY bit is set, the host processor knows it must provide the next word of the exponent - this data is written into the EXP(k) register one 32-bit word at a time. If this interrupt bit is masked, then it must poll the status register to determine when to provide the next word of the exponent. The host should not look for the assertion of E_RDY until after the routine (i.e. CR[GO] bit). Data previously written to EXP(K) is ignored.

The data to be exponentiated must be provided in the Montgomery format. Consider the vector A’, the data to be exponentiated where A’ = AR mod N. By providing A’, the results of $(A' \times R^{-1})^{EXP}$ mod N yields $(A \times R \times R^{-1})^{EXP}$ mod N, or equivalently, $(A)^{EXP}$ mod N.

The result of the calculation is returned to the B storage register. Note that this value has no remaining R terms and therefore is no longer in Montgomery format. The value of the exponent vector must be greater than one for this function to work properly. This function operates with a minimum of 5 digits (Modsize = 4). The exponent may be as small as one byte (EXP(k)_SIZE = 0).The complete set of I/O conditions is shown below:

Table 7-18. Integer Modular Exponentiation

	Integer Modular Exponentiation
Computation	$S = (A' * R^{-1})^{EXP} \bmod N$
Entry name	expA
Entry address	0x007(expA)

Pre-conditions	A0-3 = A’ (the value A in the Montgomery residue system)
	N0-3 = modulus

Run-time conditions	EXP(k) = msb exponent word (provided in 8-bit words throughout the exponentiation); first word provides following routine invocation per ERDY assertion.
Post-conditions	B0-3 = S Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	A, N, and B have the lsb digits in A0, N0, and B0, respectively. As required, data will occupy the more significant memory blocks.

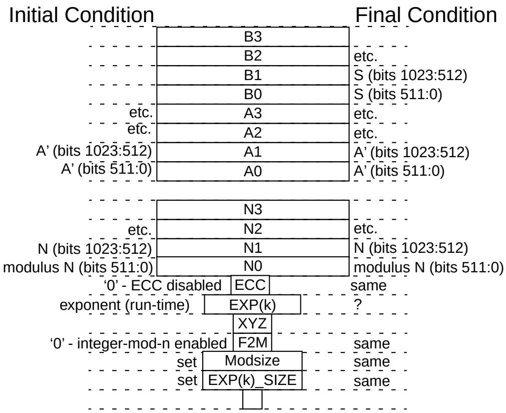


Figure 7-17. Integer Modular Exponentiation Register Usage

7.4.2 RSA Montgomery Modular Multiplication $((A \times B \times R^{-1}) \bmod N)$

The $(A \times B \times R^{-1}) \bmod N$ calculation is the core function of the PKEU. It is used to assist the exponentiation routine in completing its operation though it is also available to the host processor - typically to put messages into the Montgomery format. This function operates with a minimum of five digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-19. Modular Multiplication

	Modular Multiply
Computation	$C = A * B * R^{-1}$ mod N, where A, B, and C are integers less than N and $R = 2^{16D}$ where D is the number of digits of the modulus vector
Entry name	modularmultiply
Entry address	0x00a(modularmultiply)

Pre-conditions	A0-3 = A
	B0-3 = B
	N0-3 = modulus

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	Typically, though it is not mandatory, the operands will be in the Montgomery residue system. The only time this would not be the case is when manually placing a value into the Montgomery residue system.

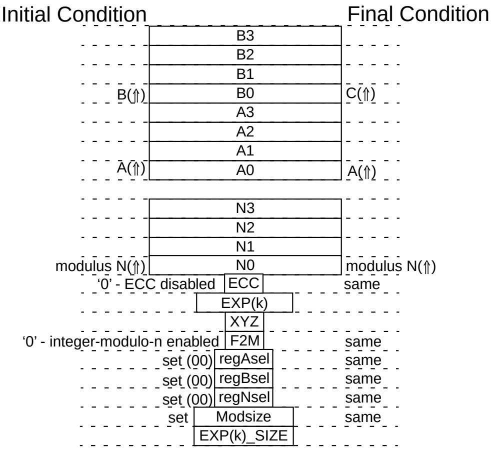


Figure 7-18. Modular Multiplication Register Usage

Prior to initiating this function, the A and B register pointers must be set in the control register which indicate which sub-registers (e.g A0, B0, A1, B1, etc.) are the targeted operands. See Table 7-2 for a detailed description. Once this is performed, the host processor may successfully initiate this function.

7.4.3 RSA Montgomery Modular Multiplication ((A × B × R^{-2}) mod N)

The (A × B × R^{-2}) mod N calculation is similar to the standard ‘R^{-1}’ Montgomery multiplication except an additional R is divided out. This function is particularly helpful when using the Chinese Remainder Theorem. This function operates with a minimum of five digits (Modsize = 4). The complete set of I/O conditions is shown below:

Table 7-20. Modular Multiplication (with double reduction)

	Modular Multiply (with double reduction)
Computation	C = A * B * R^{-2} mod N, where A, B, and C are integers less than N and R = 2^{16D} where D is the number of digits of the modulus vector
Entry name	modularmultiply2
Entry address	0x00b(modularmultiply2)

Pre-conditions	A0-3 = A
	B0-3 = B
	N0-3 = modulus

Post-conditions	A0-3 = A operand is preserved B0-3 = results of modular multiplication stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	—

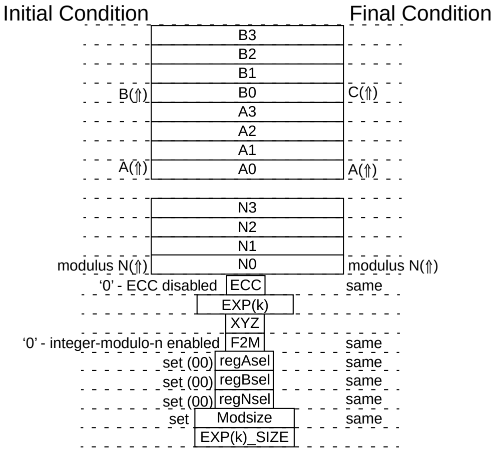


Figure 7-19. Modular Multiplication (with double reduction) Register Usage

7.4.4 RSA Modular Add

Modular addition may be performed on any two vectors loaded into A (A0-A3) and B (B0-B3), where both of these vectors are less than the value stored in the modulus register N (N0-N3). The results are stored in the respective B register. This function is particularly helpful when using the Chinese Remainder Theorem. This function operates with a minimum of 4 digits (Modsize = 3).

Prior to initiating this function, the A and B register pointers must be set in the control register which indicate which sub-registers (e.g A0, B0, A1, B1, etc.) are the targeted operands. See Table 7-2 for a detailed description. Once this is performed, the host processor may successfully initiate this function.

Table 7-21. Modular Add

	Modular Add
Computation	C = D + E mod N, where D, E, and C are integers and are less than N
Entry name	modularadd
Entry address	0x008(modularadd)

Pre-conditions	A0-3 = D
	B0-3 = E
	N0-3 = modulus

Post-conditions	B0-3 = results of modular addition stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	The function operates the same regardless of whether or not the operands are in the Montgomery residue system.

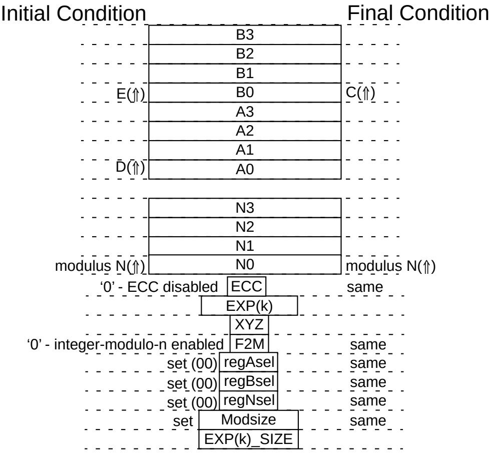


Figure 7-20. Modular Add Register Usage

7.4.5 RSA F_p Modular Subtract

Modular addition may be performed on any two vectors loaded into A (A0-A3) and B (B0-B3), where both of these vectors are less than the value stored in the modulus register N (N0-N3). This is accomplished by computing A-B if A > B or A-B+N if A < B. The results are stored in the respective B register. This function is particularly helpful when using the Chinese Remainder Theorem. This function operates with a minimum of 4 digits (Modsize = 3).

Prior to initiating this function, the A and B register pointers must be set in the control register which indicate which sub-registers (e.g A0, B0, A1, B1, etc.) are the targeted operands. See Table 7-2 for a detailed description. Once this is performed, the host processor may successfully initiate this function.

Table 7-22. Modular Subtract

	Modular Subtract
Computation	C = D - E mod N, where D, E, and C are integers and are less than N
Entry name	modularsubtract
Entry address	0x009(modularsubtract)

Pre-conditions	A0-3 = D
	B0-3 = E
	N0-3 = modulus

Post-conditions	B0-3 = results of modular subtraction stored where the B operand was located Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	The function operates the same regardless of whether or not the operands are in the Montgomery residue system.

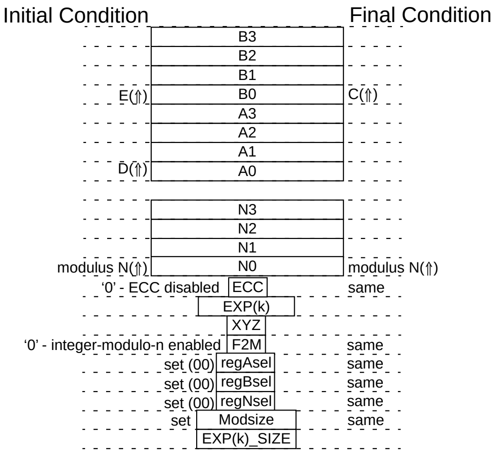


Figure 7-21. Modular Subtract Register Usage

7.5 Miscellaneous Routines

The remaining routines are general in nature and are not specific to any particular cryptographic algorithm.

7.5.1 Clear Memory

This routine clears all of the RAM memory locations in the PKEU. This includes the A, B, and N RAMs. All locations are set to zero. All other registers are cleared either via a reset (software or hardware) or by explicitly writing zeros to each register. Following a reset (software or hardware), this routine is automatically invoked. This accounts for the majority of time between reset and the assertion of the DONE bit in the status register.

Table 7-23. Clear Memory

	Clear Memory
Computation	A, B, N, and t memories are overwritten with zeros
Entry name	clearmemory
Entry address	0x00d(r2)
Pre-conditions	—
Post-conditions	A = B = N = 0 (all locations) Unless explicitly noted, all other registers are not guaranteed to be any particular value.
Special conditions	—

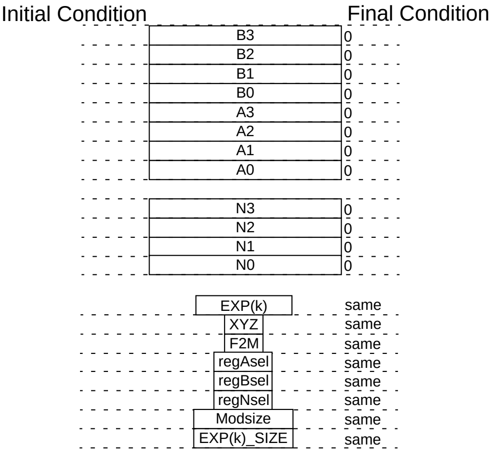


Figure 7-22. Clear Memory Register Usage

7.5.2 R^2 mod N Calculation

The PKEU has the capability to calculate R^2 mod N, where $R = 2^{16D}$ and D is the number of digits of the modulus vector (Modsize+1, where Modsize is specified independently). This function is used to assist in placing operands into the Montgomery residue system. When possible, this value should be pre-computed. If this value is not available, then the host processor may invoke this function to determine the value before the operation. This function takes a non-trivial amount of time (see Table 7-26) so if at all possible, this value should be stored for future use.

Note that this operation primarily exists to support RSA operations since R^2 mod N may not always be known prior to the execution of certain protocols. For ECC applications, the modulus is a system-wide parameter, which means that the R^2 mod N value may be pre-computed before any real-time operations by any other system entity and stored for future use. For this reason, R^2 mod N only supports integer-modulo-n computations (i.e. the control register bit F_2M must be 0).

This function operates with a minimum of 4 digits (Modsize = 3) and with the most significant digit (16-bits) of the modulus being non-zero. The complete set of I/O conditions is shown below:

Table 7-24. R^2 mod N

	R^2 mod N
Computation	R^2 mod N, where $R = 2^{16D}$ and D is the number of digits of the modulus vector
Entry name	r2
Entry address	0x00c(r2)
Pre-conditions	Modsize = number of digits of the modulus vector - 1 N0-3 = modulus

Post-conditions	B1 = R^2 mod N
	N0-3 = modulus
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	—

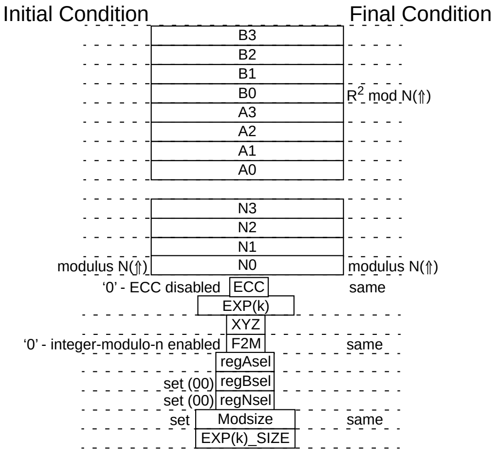


Figure 7-23. R^2 mod N Register Usage

7.5.3 R_pR_N mod P Calculation

The PKEU has the ability to calculate R_pR_N mod P, where $R_p = 2^{16D}$, and $R_N = 2^{16E}$; D is the number of digits of the modulus P, and E is the number of digits of the modulus N, and D + 4 < E. This constant is used in performing Chinese Remainder Theorem calculations given modulus N = P × Q, where P and Q are prime numbers. Although labelled R_PR_N mod P, this function can also compute R_QR_N mod Q. The requirement D + 4 < E is not a requirement of the command, but a system requirement, as for all subfunctions of Chinese Remainder Theorem to be executable on the PKEU, the number of digits of P and Q must each be at least five.

As with the standard R^2 mod N operation, this operation exists primarily to support RSA and only works with the Control Register F_2M bit set to zero.

To use this function, MOD_SIZE must be programmed with D-1, and EXP_SIZE must be programmed with E-1, and the prime modulus (either P or Q) is written into memory N. The complete set of I/O conditions is shown in Table 7-26.

Table 7-25. R_pR_N mod P

	R_pR_N mod P
Computation	R_pR_N mod P, where $R_p = 2^{16D}$, and $R_N = 2^{16E}$; D is the number of digits of the modulus P, and E is the number of digits of the modulus N, and D + 4 < E
Entry name	r2
Entry address	0x00c(r2)
Pre-conditions	Modsize = number of digits of the vector D - 1 EXP(k) SIZE = number of digits of the vector E-1

Post-conditions	B0-3 = R_pR_N mod P
	N0-3 = modulus
	Unless explicitly noted, all other registers are not guaranteed to be any particular value.

Special conditions	—

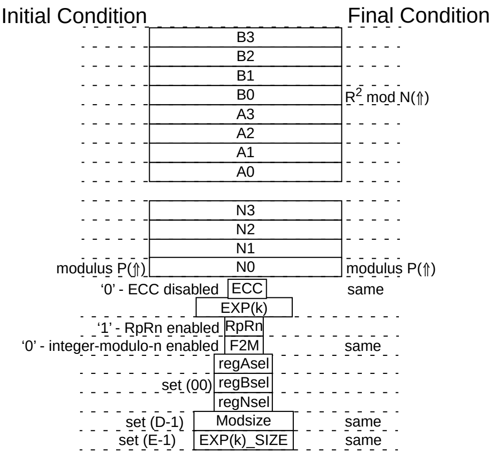


Figure 7-24. R_PR_N mod P Register Usage

ELLIPTIC CURVE CRYPTOSYSTEMS — READY FOR PRIME TIME

Alfred Menezes

University of Waterloo
ajmeneze@math.uwaterloo.ca

January 29, 1998

Outline

1. Introduction
2. The Digital Signature Algorithm (DSA)
3. Background on elliptic curves
4. Elliptic Curve Digital Signature Algorithm (ECDSA)
5. The RSA signature scheme
6. Evaluation criteria
7. Security
8. Comparison
9. Industry/Government standards
10. Conclusions

1. Introduction

- Discrete-log cryptographic protocols are usually described in the algebraic setting of the group $\mathbb{Z}_p^*$ (the multiplicative group of the integers modulo a prime p).
- These include Diffie-Hellman key agreement, ElGamal encryption, and the ElGamal signature scheme.
- They can also be described in the more abstract setting of a finite cyclic group G.

Diffie-Hellman key agreement

Objective: Alice and Bob establish a shared secret by communicating over an unsecured but authentic channel.

1. Public parameters: A prime p and a generator g of $\mathbb{Z}_p^*$.
2. Alice generates a random integer a, $1 \leq a \leq p-2$, and sends g^a to Bob.
3. Bob generates a random integer b, $1 \leq b \leq p-2$, and sends g^b to Alice.
4. Alice computes $K = (g^b)^a$.
5. Bob computes $K = (g^a)^b$.
6. The shared secret is $K = g^{ab}$.

Among the groups proposed:

- Multiplicative group of a finite field $\mathbb{F}_q$ (Diffie and Hellman, 1976).
- Group of points on an elliptic curve over a finite field (Koblitz, Miller, 1985).
- Class group of an imaginary quadratic number field (Buchmann, Williams, 1988).
- Subgroup of the multiplicative group of $\mathbb{Z}_p$ (Schnorr, 1989).
- Jacobian of a hyperelliptic curve defined over a finite field (Koblitz, 1989).

2. The Digital Signature Algorithm (DSA)

- Variant of ElGamal and Schnorr schemes.
- Proposed in 1991.
- US Federal Information Processing Standard (FIPS-180).
- Exploits small subgroups in $\mathbb{Z}_p^*$ in order to decrease the size of signatures.

DSA system parameter generation

1. Select a prime q such that $2^{159} < q < 2^{160}$.
2. Select a 1024-bit prime number p with the property that $q \mid p-1$.
3. (Select a generator g of the unique cyclic group of order q in $\mathbb{Z}_p^*$.)

   3.1. Select an element $h \in \mathbb{Z}_p^*$ and compute $g = h^{(p-1)/q} \bmod p$. (Repeat until $g \neq 1$.)
4. System parameters are p, q and g.

DSA key generation

Each entity A does the following:

1. Select a random integer x such that $1 \leq x \leq q-1$.
2. Compute $y = g^x \bmod p$.
3. A's public key is y; A's private key is x.

DSA signature generation

To sign a message m, A does the following:

1. Select a random integer k, $1 \leq k \leq q-1$.
2. Compute $r = (g^k \bmod p) \bmod q$.
3. Compute $k^{-1} \bmod q$.
4. Compute $s = k^{-1}\{h(m) + xr\} \bmod q$. If $s = 0$ then go to step 1.
5. The signature for the message m is (r, s).

- The signature is 320 bits in length.
- h is the hash function SHA-1.

DSA signature verification

To verify A's signature (r, s) on m, B should do the following:

1. Compute $w = s^{-1} \bmod q$ and $h(m)$.
2. Compute $u_1 = h(m)w \bmod q$ and $u_2 = rw \bmod q$.
3. Compute $v = (g^{u_1}y^{u_2} \bmod p) \bmod q$.
4. Accept the signature if and only if $v = r$.

Discrete logarithm problem

The security of DSA is based on the difficulty of the *discrete logarithm problem* (DLP):

Given a prime p, a generator g of $\mathbb{Z}_p^*$, and $y = g^x \bmod p$, find x.

3. Background on elliptic curves

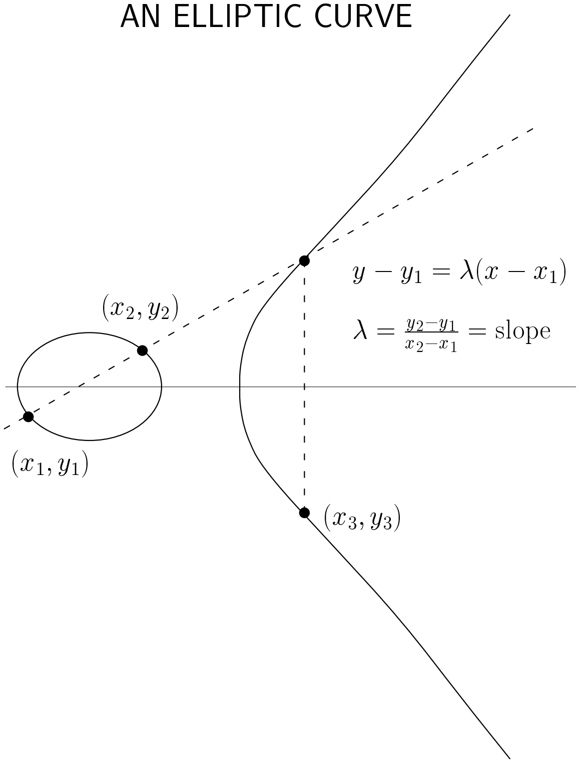
- Let $\mathbb{Z}_p$ be the set of integers modulo a prime p $(p > 3)$.
- An *elliptic curve* E over $\mathbb{Z}_p$ is defined by an equation of the form
$$y^2 = x^3 + ax + b,$$
where $a, b \in \mathbb{Z}_p$, $(4a^3 + 27b^2) \not\equiv 0 \pmod{p}$, together with the *point at infinity* $\mathcal{O}$.
- The set $E(\mathbb{Z}_p)$ consists of all points (x, y), $x \in \mathbb{Z}_p$, $y \in \mathbb{Z}_p$, which satisfy the defining equation, together with $\mathcal{O}$.

An example over $\mathbb{Z}_{23}$

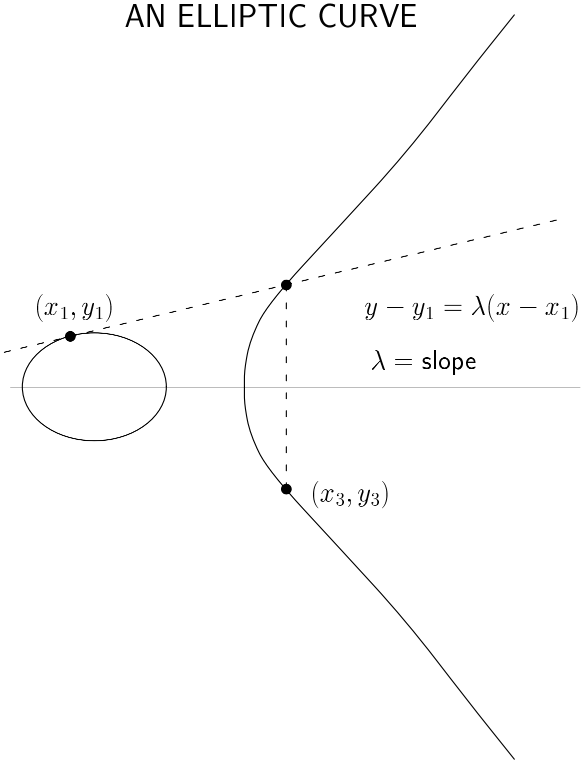
- Let $p = 23$.
- $y^2 = x^3 + x + 1$, (i.e. $a = 1$, $b = 1$).
- $E(\mathbb{Z}_{23}) = \{(x, y) : y^2 = x^3 + x + 1\} \cup \{\mathcal{O}\}$.
- Solutions to $y^2 = x^3 + x + 1$ over $\mathbb{Z}_{23}$:

(0,1)	(5,4)	(9,16)	(17,3)
(0,22)	(5,19)	(11,3)	(17,20
(1,7)	(6,4)	(11,20)	(18,3)
(1,16)	(6,19)	(12,4)	(18,20)
(3,10)	(7,11)	(12,19	(19,5)
(3,13)	(7,12)	(13,7)	(19,18)
(4,0)	(9,7)	(13,16)	

AN ELLIPTIC CURVE


(x_2, y_2)
$y - y_1 = \lambda(x - x_1)$
$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \text{slope}$
(x_1, y_1)
(x_3, y_3)

AN ELLIPTIC CURVE



Addition formula

$E : y^2 = x^3 + ax + b$.

- $\mathcal{O} + \mathcal{O} = \mathcal{O}$.
- $(x, y) + \mathcal{O} = (x, y)$ for all $(x, y) \in E$.
- $(x, y) + (x, -y) = \mathcal{O}$ for all $(x, y) \in E$. (i.e. $-(x, y) = (x, -y)$).
- Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$, where
  $x_3 = \lambda^2 - x_1 - x_2$
  $y_3 = \lambda(x_1 - x_3) - y_1$ and

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} & P = Q. \end{cases}$$

Examples of addition in $E(\mathbb{Z}_{23})$.

1. $P_1 = (3, 10)$, $P_2 = (9, 7)$,
   $P_1 + P_2 = (x_3, y_3)$.

$$\lambda = \frac{7-10}{9-3} = \frac{-3}{6} = \frac{-1}{2} = 11 \in \mathbb{Z}_{23}.$$

$x_3 = 11^2 - 3 - 9 = 6 - 3 - 9 = 17$,
$y_3 = 11(3 - (-6)) - 10 = 11(9) - 10$
$= 89 = 20 = 20$.
Therefore $P_1 + P_2 = (17, 20)$.

2. $P_1 = (3, 10)$, $2P_1 = (x_3, y_3)$,

$$\lambda = \frac{3(3^2)+1}{20} = \frac{5}{20} = \frac{1}{4} = 6.$$

$x_3 = 6^2 - 6 = 30 = 7$,
$y_3 = 6(3 - 7) - 10 = -24 - 10 = 12$.
Therefore $2P_1 = (7, 12)$.

Basic facts

- There are about $2p$ “different” elliptic curves over $\mathbb{Z}_p$.
- $E(\mathbb{Z}_p)$ is an abelian group with identity $\mathcal{O}$.
- The number of points on the elliptic curve is $\#E(\mathbb{Z}_p) = p + 1 - t$, where $|t| \le 2\sqrt{p}$. Hence, $\#E(\mathbb{Z}_p) \approx p$.
- $\#E(\mathbb{Z}_p)$ can be computed in polynomial time using Schoof’s algorithm.
- The above results are also true if $\mathbb{Z}_p$ is replaced by any finite field $\mathbb{F}_q$.

Example

- $E : y^2 = x^3 + x + 1$ over $\mathbb{Z}_{23}$.
- $\#E(\mathbb{Z}_{23}) = 28$.
- $E(\mathbb{Z}_{23})$ is a cyclic group, and $P = (0, 1)$ is a generator:

P=(0, 1)	$15P$=(9, 7)
$2P$=(6,19)	$16P$=(17,3)
$3P$=(3,13)	$17P$=(1,7)
$4P$=(13,16)	$18P$=(12,19)
$5P$=(18,3)	$19P$=(19,5)
$6P$=(7,11)	$20P$=(5,4)
$7P$=(11,3)	$21P$=(11,20)
$8P$=(5,19)	$22P$=(7 12)
$9P$=(19,18)	$23P$=(18,20)
$10P$ =(12,4)	$24P$=(13,7)
$11P$=(1,16)	$25P$=(3,10)
$12P$=(17,20)	$26P$=(6,4)
$13P$=(9,16)	$27P$=(0,22)
$14P$=(4,0)	$28P$=$\mathcal{O}$

$\mathbb{Z}_p^*$ and $E(\mathbb{F}_q)$ correspondence

Group	$\mathbb{Z}_p^*$	$E(\mathbb{F}_q)$
Group elements	Integers $\{1, 2, \ldots, p-1\}$	Points (x, y) on E plus $\mathcal{O}$
Group operation	multiplication modulo p	addition of points
Notation	Elements: g, h Multiplication: $g \cdot h$ Inverse: g^{-1} Division: g/h Exponentiation: g^a	Elements: P, Q Addition: $P + Q$ Negative: $-P$ Subtraction: $P - Q$ Multiple: aP
Discrete Logarithm Problem	Given $g \in \mathbb{Z}_p^*$ and $h = g^a \bmod p$, find a	Given $P \in E(\mathbb{F}_q)$ and $Q = aP$, find a.

4. ECDSA

- The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the DSA.
- Under consideration by IEEE, ANSI, FIPS, ISO as signature standards.

ECDSA system parameter generation

1. Select an elliptic curve E defined over $\mathbb{F}_q$.
2. $\#E(\mathbb{F}_q)$ should be divisible by a large prime n.
3. Select a point P of order n in $E(\mathbb{F}_q)$.
4. The system parameters are E, P and n.

ECDSA key generation

Each entity A does the following:

1. Select a random integer d in the interval $[1, n-1]$.
2. Compute $Q = dP$.
3. A public key is Q; A's private key is d.

DSA and ECDSA notation correspondence

DSA notation	ECDSA notation
q	n
g	P
x	d
y	Q

ECDSA signature generation

To sign a message m, A does the following:

1. Select a random integer k, $1 \leq k \leq n-1$.
2. Compute $kP = (x_1, y_1)$ and $r = x_1 \bmod n$. If $r = 0$ then go to step 1.
3. Compute $k^{-1} \bmod n$.
4. Compute $s = k^{-1}\{h(m) + dr\} \bmod n$. If $s = 0$ then go to step 1.
5. The signature for the message m is (r, s).

- If n is a 160-bit prime, then the signature is 320 bits in length.
- h is the hash function SHA-1

ECDSA signature verification

To verify A's signature (r, s) on m, B should do the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $w = s^{-1} \bmod n$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $u_1P + u_2Q = (x_1, y_1)$ and $v = x_1 \bmod n$.
5. Accept the signature if and only if $v = r$.

Elliptic Curve Discrete logarithm problem

The security of ECDSA is based on the difficulty of the *elliptic curved discrete logarithm problem* (ECDLP):

Given an elliptic curve E defined over $\mathbb{F}_{q,}$, a point P of order n, $Q = dP$, find d.

5. The RSA signature scheme

RSA key generation

Each entity A does the following:

1. Select large random primes p and q.
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3. Select an integer e, $1 \le e \le \phi - 1$, such that $\gcd(e, \phi) = 1$.
4. Compute the integer d, $1 \le d \le \phi - 1$, such that $ed \equiv 1 \pmod{\phi}$.
5. A's public key is (n, e); A's private key is d.

RSA signature generation

To sign a message M, A does the following:

1. Compute $m = H(M)$.
2. Compute $s = m^d \bmod n$.
3. The signature for M is s.

RSA signature verification

To verify A's signature s on M, B should do the following:

1. Compute $m = H(M)$.
2. Compute $m' = s^e \bmod n$.
3. Accept the signature if and only if $m = m'$.

Integer factorization problem

The security of RSA is based on the difficulty of the *integer factorization problem* (IFP):

Given an integer n that is a product of two distinct primes p and q, find p and q.

6. Evaluation criteria

- (Perceived) security.
- Key lengths.
- Signature size.
- Speed.
- Storage (precomputation?).
- Complexity of implementation (code size, gate count, power consumption, etc.).
- Platforms (hardware, software, firmware).
- Industry/government standards.
- Patent coverage.
- Licensing terms.

7. Security

History of math problems

	IFP	DLP and ECDLP
≈ 1920s	Random squares (Kraitchik)	Index-calculus (Kraitchik)
1975	Continued fraction	
1976		DLP proposed for use in cryptography
1977	RSA proposed	
1979		Index-calculus
1982	Quadratic sieve	

	IFP	DLP and ECDLP
1985		ECDLP proposed for use in cryptography
1990	Number field sieve	Number field sieve for DLP
1991		Reduction for supersingular curves (for ECDLP)
1994		Subexponential-time algorithm for high-genus hyperelliptic curves
1995		Trace 1 curves are weak (Semaev) (for ECDLP)
1998	?	?

Some questions to ponder:

1. Has the integer factorization problem indeed been *seriously* studied by thousands of mathematicians for hundreds of years?
2. Has the integer factorization problem been more carefully studied than the discrete logarithm problem?
3. Is the research on the discrete logarithm problem prior to 1985 of any relevance/significance to the elliptic curve discrete logarithm problem?

4. Have there been many more research papers on the security and/or implementation of RSA than that of elliptic curve cryptosystems?
5. Is it true that the elliptic curve discrete logarithm problem is not well understood due to the *abstruse nature* of elliptic curves? (As compared, say, to the number field sieve.)

My opinion:

While it is true that the integer factorization problem has been more heavily scrutinized than the elliptic curve discrete logarithm problem, the difference in the efforts these problems have received has been exaggerated.

Attacks on underlying math problems

1. Integer factorization problem (IFP):
   - Number field sieve $L_n[\frac{1}{3}, 1.923]$ (*subexponential-time* algorithm).
   - Easily parallelized in software.
   - Record: 130-decimal digit, 500 MIPS years.
   - Challenge: 512-bit RSA number.

2. Discrete logarithm problem (DLP):
   - Number field sieve (for $\mathbb{Z}_p$)
   - $L_p[\frac{1}{3}, 1.923]$ (*subexponential-time* algorithm).
   - Easily parallelized in software.
   - Record: 75-decimal digit (248 bits).
   - Challenge: $p = 2 \cdot 739 \cdot \frac{(7^{149}-1)}{6} + 1$ (427 bits).

3. Elliptic curve discrete logarithm problem (ECDLP): ($\#E(\mathbb{F}_q)$ divisible by large prime n)
   - Pollard-ρ
     Expected running time: $\sqrt{\frac{\pi n}{2}}$.
   - Distributed version (van Oorschot/Wiener)
     m processors: $\sqrt{\frac{\pi n}{2}}/m$
     (*fully exponential-time* algorithm).
   - Easily parallelized in hardware.
   - Certicom ECC Challenge – www.certicom.com.
     Challenges: 109, 131, 163, 191, 239, and 359-bits.

Factoring estimates (software)

(A. Odlyzko – CryptoBytes, Summer 1995)

Size of n (in bits)	MIPS years
512	3×10^4
768	2×10^8
1024	3×10^{11}
1280	1×10^{14}
1536	3×10^{16}
2048	3×10^{20}

Computing power available (MIPS years):

	covert attack	open project
2004	10^8	2×10^9
2014	$10^{10} - 10^{11}$	$10^{11} - 10^{13}$

(2014: 10^{10} people, 10 computers/person, typical computer rated at $10^4 - 10^5$ MIPS.)

EC discrete logarithm estimates

- **Software**
  Assumption: a 1 MIPS machine can perform 4×10^4 EC additions/sec, or 2^{40} EC additions/year.

Size of n ($\approx q$) (in bits)	MIPS years
160	9.6×10^{11}
186	7.9×10^{15}
234	1.3×10^{23}
354	1.5×10^{41}

- **Hardware** (van Oorschot/Wiener,1994)
  - \$10 million
  - 325,000 processors
  - $n \approx 2^{120}$
  - 1 logarithm in 35 days

8. Comparison

- Since no subexponential-time algorithm is known for the general ECDLP, a smaller underlying finite field $\mathbb{F}_q$ can be chosen (compared to traditional discrete log systems).
- A smaller field results in the following benefits of elliptic curve systems:
  - smaller key sizes (and certificates)
  - smaller signature sizes
  - bandwidth savings
  - smaller hardware processors
  - low power requirements
  - efficient implementations.

- RSA: 1024-bit modulus n.
- DSA: 1024-bit p, 160-bit q.
- ECDSA: 160-bit n (so q is $160+\epsilon$ bits).

Parameter sizes

	ECDSA (160-bit q)	RSA (1024-bit n, $e = 2^{16}+1$)	DSA 1024-bit p, 160-bit q)
System params	a, b, P, n 640 (bits)	— 0	p, q, g 2208
Public key	Q 161	n 1024	g^x 1024
Private key	d 160	d 1024	x 160

Software comparison (very rough)

Assumptions:

- 1 EC addition = 10 field multiplications.
- 40 160-bit field multiplications = 1 1024-bit modular multiplication.

	ECDSA	RSA $e = 2^{16}+1$, CRT	DSA
Signing time	(kP) 60	$(m^d \bmod n)$ 384	$(g^k \bmod p)$ 240
Verifying time	(2 exps) 120	$(s^e \bmod n)$ 17	(2 exps) 480

(# of 1024-bit modular multiplications)

Hardware comparison

- James Dworkin, Motorola, April 1997.
- 20 MHz.
- RSA: Montgomery multiplication, CRT, 64-bit e, 16×16 bit multiplier.
- No other assumptions were stated.

	160-bit EC	1024-bit RSA	210-bit EC	2048-bit RSA
Signature speed (ms)	5.3	85.7	7.1	657.3
Verification speed (ms)	10.5	24.1	14.2	94.4
Silicon area (mil/side)	72	73	86	83
Energy to sign (mW/s)	.095	2.228	.214	22.611
Energy to verify (mW/s)	.190	.626	.427	3.249

9. Industry/Government standards

Goals:

- Facilitate widespread use of cryptographically sound and well-accepted techniques.
- Promote interoperability.

Draft standards

1. ANSI X9.62 (The Elliptic Curve Digital Signature Algorithm (ECDSA))
    - Goals: high security and interoperability.
    - Elliptic curves over $\mathbb{Z}_p$.
    - Elliptic curves over $\mathbb{F}_{2^m}$ (polynomial bases, optimal normal bases).
    - Security constraint: $n > 2^{160}$.
    - (Optional) method for generating random curves *verifiably* at random.
    - Ballot date: December 1997.

2. ANSI X9.63 (Elliptic Curve Key Agreement and Transport Protocols)
    - Key agreement: two Diffie-Hellman variants (MQV, unified model).
    - Key transport.
    - (Hoped) ballot date: Fall 1998.

3. IEEE P1363 (Standard Specification for Public-key Cryptography)
    - RSA, discrete logs, elliptic curves.
    - Programmers reference guide, rather than an interoperability standard.
    - Lots of options. (e.g. arbitrary polynomial or normal bases for $\mathbb{F}_{2^m}$).
    - No minimal security requirements.
    - Elliptic curve protocols: ECDSA, Nyberg-Rueppel signature scheme, MQV and unified-model key agreement.
    - http://stdsbbs.ieee.org/
    - (Hoped) ballot date: Summer 1998.

4. Internet OAKLEY (variant of Diffie-Hellman)
    - http://www.ietf.cnri.reston.va.us/
5. ISO 14888 (digital signatures with appendix)
    - High-level description of elliptic curve signature algorithms.
6. ATM Forum
    - Elliptic curve signature schemes.
    - Elliptic curves over $\mathbb{Z}_p$ and $\mathbb{F}_{2^m}$.
7. Widespread support for inclusion of elliptic curve algorithms in SET 2.0.

8. US government FIPS
    - May 13 1997 Federal Registry announcement: NIST seeks comments on the possibility of allowing government agencies to use additional public-key based digital signature algorithms, such as the RSA and elliptic curve techniques.
    - Plans to develop a federal standard for public-key based cryptographic key agreement and exchange. The notice asked for comments on such techniques as RSA, Diffie-Hellman and elliptic curve.

10. Conclusions

- Elliptic curve cryptosystems are the next generation of public-key technology.
- They have been accepted by many as a mature technology.
- ECC will see widespread deployment in the coming years.

FIPS PUB 186-3

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

Digital Signature Standard (DSS)

CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

Issued June, 2009

U.S. Department of Commerce
Gary Locke, Secretary
National Institute of Standards and Technology
Patrick Gallagher, Deputy Director

FOREWORD

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST) is the official series of publications relating to standards and guidelines adopted and promulgated under the provisions of the Federal Information Security Management Act (FISMA) of 2002.

Comments concerning FIPS publications are welcomed and should be addressed to the Director, Information Technology Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

Cita Furlani, Director
Information Technology Laboratory

Abstract

This Standard specifies a suite of algorithms that can be used to generate a digital signature. Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of the signatory. In addition, the recipient of signed data can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation, since the signatory cannot easily repudiate the signature at a later time.

Key words: computer security, cryptography, digital signatures, Federal Information Processing Standards, public key cryptography.

Federal Information Processing Standards Publication 186-3

June 2009

Announcing the

DIGITAL SIGNATURE STANDARD (DSS)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106), and the Computer Security Act of 1987 (Public Law 100-235).

1. **Name of Standard**: Digital Signature Standard (DSS) (FIPS 186-3).

2. **Category of Standard**: Computer Security. **Subcategory.** Cryptography.

3. **Explanation**: This Standard specifies algorithms for applications requiring a digital signature, rather than a written signature. A digital signature is represented in a computer as a string of bits. A digital signature is computed using a set of rules and a set of parameters that allow the identity of the signatory and the integrity of the data to be verified. Digital signatures may be generated on both stored and transmitted data.

Signature generation uses a private key to generate a digital signature; signature verification uses a public key that corresponds to, but is not the same as, the private key. Each signatory possesses a private and public key pair. Public keys may be known by the public; private keys are kept secret. Anyone can verify the signature by employing the signatory's public key. Only the user that possesses the private key can perform signature generation.

A hash function is used in the signature generation process to obtain a condensed version of the data to be signed; the condensed version of the data is often called a message digest. The message digest is input to the digital signature algorithm to generate the digital signature. The hash functions to be used are specified in the Secure Hash Standard (SHS), FIPS 180-3. FIPS **approved** digital signature algorithms **shall** be used with an appropriate hash function that is specified in the SHS.

The digital signature is provided to the intended verifier along with the signed data. The verifying entity verifies the signature by using the claimed signatory's public key and the same hash function that was used to generate the signature. Similar procedures may be used to generate and verify signatures for both stored and transmitted data.

4. **Approving Authority:** Secretary of Commerce.

5. Maintenance Agency: Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division.

6. Applicability: This Standard is applicable to all Federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502 (2) of Title 44, United States Code. This Standard **shall** be used in designing and implementing public key-based signature systems that Federal departments and agencies operate or that are operated for them under contract. The adoption and use of this Standard is available to private and commercial organizations.

7. Applications: A digital signature algorithm allows an entity to authenticate the integrity of signed data and the identity of the signatory. The recipient of a signed message can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation, since the signatory cannot easily repudiate the signature at a later time. A digital signature algorithm is intended for use in electronic mail, electronic funds transfer, electronic data interchange, software distribution, data storage, and other applications that require data integrity assurance and data origin authentication.

8. Implementations: A digital signature algorithm may be implemented in software, firmware, hardware or any combination thereof. NIST has developed a validation program to test implementations for conformance to the algorithms in this Standard. Information about the validation program is available at http://csrc.nist.gov/cryptval. Examples for each digital signature algorithm are available at http://csrc.nist.gov/groups/ST/toolkit/examples.html.

Agencies are advised that digital signature key pairs **shall not** be used for other purposes.

9. Other Approved Security Functions: Digital signature implementations that comply with this Standard **shall** employ cryptographic algorithms, cryptographic key generation algorithms, and key establishment techniques that have been approved for protecting Federal government sensitive information. Approved cryptographic algorithms and techniques include those that are either:

a. specified in a Federal Information Processing Standard (FIPS),

b. adopted in a FIPS or a NIST Recommendation, or

c. specified in the list of approved security functions for FIPS 140-2.

10. Export Control: Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Exports of cryptographic modules implementing this Standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Industry and Security of the U.S. Department of Commerce. Information about export regulations is available at: http://www.bis.doc.gov.

11. Patents: The algorithms in this Standard may be covered by U.S. or foreign patents.

12. Implementation Schedule: This Standard becomes effective immediately upon approval by the Secretary of Commerce. A transition strategy for validating algorithms and cryptographic modules will be posted on NIST's Web page at http://csrc.nist.gov/groups/STM/cmvp/index.html under Notices. The transition plan addresses the transition by Federal agencies from modules tested and validated for compliance to FIPS 186-2 to modules tested and validated for compliance to FIPS 186-3 under the Cryptographic Module Validation Program. The transition plan allows Federal agencies and vendors to make a smooth transition to FIPS 186-3.

13. Specifications: Federal Information Processing Standard (FIPS) 186-3 Digital Signature Standard (affixed).

14. Cross Index: The following documents are referenced in this Standard.

a. FIPS PUB 140-2, Security Requirements for Cryptographic Modules.

b. FIPS PUB 180-3, Secure Hash Standard.

c. ANS X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA).

d. ANS X9.62-2005, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

e. ANS X9.80, Prime Number Generation, Primality Testing and Primality Certificates.

f. Public Key Cryptography Standard (PKCS) #1, RSA Encryption Standard.

g. Special Publication (SP) 800-57, Recommendation for Key Management.

h. Special Publication (SP) 800-89, Recommendation for Obtaining Assurances for Digital Signature Applications.

i. Special Publication (SP) 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators.

j. Special Publication (SP) 800-102, Recommendation for Digital Signature Timeliness

k. IEEE Std. 1363-2000, Standard Specifications for Public Key Cryptography.

15. Qualifications: The security of a digital signature system is dependent on maintaining the secrecy of the signatory's private keys. Signatories **shall**, therefore, guard against the disclosure of their private keys. While it is the intent of this Standard to specify general security requirements for generating digital signatures, conformance to this Standard does not assure that a particular implementation is secure. It is the responsibility of an implementer to ensure that any module that implements a digital signature capability is designed and built in a secure manner.

Similarly, the use of a product containing an implementation that conforms to this Standard does not guarantee the security of the overall system in which the product is used. The responsible authority in each agency or department **shall** assure that an overall implementation provides an acceptable level of security.

Since a standard of this nature must be flexible enough to adapt to advancements and innovations in science and technology, this Standard will be reviewed every five years in order to assess its adequacy.

16. Waiver Procedure: The Federal Information Security Management Act (FISMA) does not allow for waivers to Federal Information Processing Standards (FIPS) that are made mandatory by the Secretary of Commerce.

17. Where to Obtain Copies of the Standard: This publication is available by accessing http://csrc.nist.gov/publications/. Other computer security publications are available at the same web site.

Table of Contents

Federal Information Processing Standards Publication 186-3

June 2009

Specifications for the

DIGITAL SIGNATURE STANDARD (DSS)

1. Introduction

This Standard defines methods for digital signature generation that can be used for the protection of binary data (commonly called a message), and for the verification and validation of those digital signatures. Three techniques are approved.

(1) The Digital Signature Algorithm (DSA) is specified in this Standard. The specification includes criteria for the generation of domain parameters, for the generation of public and private key pairs, and for the generation and verification of digital signatures.

(2) The RSA digital signature algorithm is specified in American National Standard (ANS) X9.31 and Public Key Cryptography Standard (PKCS) #1. FIPS 186-3 approves the use of implementations of either or both of these standards, but specifies additional requirements.

(3) The Elliptic Curve Digital Signature Algorithm (ECDSA) is specified in ANS X9.62. FIPS 186-3 approves the use of ECDSA, but specifies additional requirements. Recommended elliptic curves for Federal Government use are provided herein.

This Standard includes requirements for obtaining the assurances necessary for valid digital signatures. Methods for obtaining these assurances are provided in NIST Special Publication (SP) 800-89, *Recommendation for Obtaining Assurances for Digital Signature Applications*.

2. Glossary of Terms, Acronyms and Mathematical Symbols

2.1 Terms and Definitions

Approved
FIPS-approved and/or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST Recommendation, or 2) adopted in a FIPS or NIST Recommendation or 3) specified in a list of NIST approved security functions.

Assurance of domain parameter validity
Confidence that the domain parameters are arithmetically correct.

Assurance of possession
Confidence that an entity possesses a private key and any associated keying material.

Assurance of public key validity
Confidence that the public key is arithmetically correct.

Bit string
An ordered sequence of 0's and 1's. The leftmost bit is the most significant bit of the string. The rightmost bit is the least significant bit of the string.

Certificate
A set of data that uniquely identifies a key pair and an owner that is authorized to use the key pair. The certificate contains the owner's public key and possibly other information, and is digitally signed by a Certification Authority (i.e., a trusted party), thereby binding the public key to the owner.

Certification Authority (CA)
The entity in a Public Key Infrastructure (PKI) that is responsible for issuing certificates and exacting compliance with a PKI policy.

Claimed signatory
From the verifier's perspective, the claimed signatory is the entity that purportedly generated a digital signature.

Digital signature
The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity and signatory non-repudiation.

Domain parameter seed
A string of bits that is used as input for a domain parameter generation or validation process.

Domain parameters
Parameters used with cryptographic algorithms that are usually common to a domain of users. A DSA or ECDSA cryptographic key pair is associated with a specifc set of domain parameters.

Entity

An individual (person), organization, device or process. Used interchangeably with “party”.

Equivalent process

Two processes are equivalent if, when the same values are input to each process (either as input parameters or as values made available during the process or both), the same output is produced.

Hash function

A function that maps a bit string of arbitrary length to a fixed length bit string. Approved hash functions are specified in FIPS 180-3 and are designed to satisfy the following properties:

1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output, and
2. (Collision resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.

Hash value

See “message digest”.

Intended signatory

An entity that intends to generate digital signatures in the future.

Key

A parameter used in conjunction with a cryptographic algorithm that determines its operation. Examples applicable to this Standard include:

1. The computation of a digital signature from data, and
2. The verification of a digital signature.

Key pair

A public key and its corresponding private key.

Message

The data that is signed. Also known as “signed data” during the signature verification and validation process.

Message digest

The result of applying a hash function to a message. Also known as “hash value”.

Non-repudiation

A service that is used to provide assurance of the integrity and origin of data in such a way that the integrity and origin can be verified and validated by a third party as having originated from a specific entity in possession of the private key (i.e., the signatory).

Owner

A key pair owner is the entity that is authorized to use the private key of a key pair.

Party

An individual (person), organization, device or process. Used interchangeably with “entity”.

Per-message secret number

A secret random number that is generated prior to the generation of each digital signature.

Public Key Infrastructure (PKI)

A framework that is established to issue, maintain and revoke public key certificates.

Prime number generation seed

A string of random bits that is used to determine a prime number with the required characteristics.

Private key

A cryptographic key that is used with an asymmetric (public key) cryptographic algorithm. For digital signatures, the private key is uniquely associated with the owner and is not made public. The private key is used to compute a digital signature that may be verified using the corresponding public key.

Probable prime

An integer that is believed to be prime, based on a probabilistic primality test. There should be no more than a negligible probability that the so-called probable prime is actually composite.

Provable prime

An integer that is either constructed to be prime or is calculated to be prime using a primality-proving algorithm.

Pseudorandom

A process or data produced by a process is said to be pseudorandom when the outcome is deterministic, yet also effectively random as long as the internal action of the process is hidden from observation. For cryptographic purposes, "effectively" means "within the limits of the intended security strength."

Public key

A cryptographic key that is used with an asymmetric (public key) cryptographic algorithm and is associated with a private key. The public key is associated with an owner and may be made public. In the case of digital signatures, the public key is used to verify a digital signature that was signed using the corresponding private key.

Random number generator

A device or algorithm that can produce a sequence of random numbers that appears to be statistically independent and unbiased.

Security strength

A number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system. Sometimes referred to as a security level.

Shall

Used to indicate a requirement of this Standard.

Should

Used to indicate a strong recommendation, but not a requirement of this Standard.

Signatory

The entity that generates a digital signature on data using a private key.

Signature generation

The process of using a digital signature algorithm and a private key to generate a digital signature on data.

Signature validation — The (mathematical) verification of the digital signature and obtaining the appropriate assurances (e.g., public key validity, private key possession, etc.).

Signature verification — The process of using a digital signature algorithm and a public key to verify a digital signature on data.

Signed data — The data or message upon which a digital signature has been computed. Also, see “message”.

Subscriber — An entity that has applied for and received a certificate from a Certificate Authority.

Trusted third party (TTP) — An entity other than the owner and verifier that is trusted by the owner or the verifier or both. Sometimes shortened to “trusted party”.

Verifier — The entity that verifies the authenticity of a digital signature using the public key.

2.2 Acronyms

ANS	American National Standard.
CA	Certification Authority.
DSA	Digital Signature Algorithm; specified in this Standard.
ECDSA	Elliptic Curve Digital Signature Algorithm; specified in ANS X9.62.
FIPS	Federal Information Processing Standard.
NIST	National Institute of Standards and Technology.
PKCS	Public Key Cryptography Standard.
PKI	Public Key Infrastructure.
RBG	Random Bit Generator; specified in SP 800-90.
RSA	Algorithm developed by Rivest, Shamir and Adelman; specified in ANS X9.31 and PKCS #1.
SHA	Secure Hash Algorithm; specified in FIPS 180-3.
SP	NIST Special Publication
TTP	Trusted Third Party.

2.3 Mathematical Symbols

$a \bmod n$ — The unique remainder r, $0 \le r \le (n - 1)$, when integer a is divided by the positive integer n. For example, 23 mod 7 = 2.

$b \equiv a \bmod n$ — There exists an integer k such that $b - a = kn$; equivalently, $a \bmod n = b \bmod n$.

counter — The counter value that results from the domain parameter generation process when the domain parameter seed is used to generate DSA domain parameters.

d — 1. For RSA, the private signature exponent of a private key.

2. For ECDSA, the private key.

domain_parameter_seed — A seed used for the generation of domain parameters.

e — The public verification exponent of an RSA public key.

g — One of the DSA domain parameters; g is a generator of the q-order cyclic group of GF(p)*; that is, an element of order q in the multiplicative group of GF(p).

GCD (a, b) — Greatest common divisor of the integers a and b.

Hash (M) — The result of a hash computation (message digest or hash value) on message M using an approved hash function.

index — A value used in the generation of g to indicate its intended use (e.g., for digital signatures).

k — For DSA and ECDSA, a per-message secret number.

L — For DSA, the length of the parameter p in bits.

(L, N) — The associated pair of length parameters for a DSA key pair, where L is the length of p, and N is the length of q.

LCM (a, b) — The least common multiple of the integers a and b.

len (a) — The length of a in bits.

M — The message that is signed using the digital signature algorithm.

m — For ECDSA, the degree of the finite field GF_{2^m}.

N — For DSA, the length of the parameter q in bits.

n

1. For RSA, the modulus; the bit length of n is considered to be the key size.
2. For ECDSA, the order of the base point of the elliptic curve; the bit length of n is considered to be the key size.

(n, d) An RSA private key, where n is the modulus, and d is the private signature exponent.

(n, e) An RSA public key, where n is the modulus, and e is the public verification exponent.

nlen The length of the RSA modulus n in bits.

p

1. For DSA, one of the DSA domain parameters; a prime number that defines the Galois Field GF(p) and is used as a modulus in the operations of GF(p).
2. For RSA, a prime factor of the modulus n.

q

1. For DSA, one of the DSA domain parameters; a prime factor of $p - 1$.
2. For RSA, a prime factor of the modulus n.

Q An ECDSA public key.

r One component of a DSA or ECDSA digital signature. See the definition of (r, s).

(r, s) A DSA or ECDSA digital signature, where r and s are the digital signature components.

s One component of a DSA or ECDSA digital signature. See the definition of (r, s).

seedlen The length of the *domain_parameter_seed* in bits.

SHAx(M) The result when M is the input to the SHA-x hash function, where SHA-x is specified in FIPS 180-3.

x The DSA private key.

y The DSA public key.

$\oplus$ Bitwise logical “exclusive-or” on bit strings of the same length; for corresponding bits of each bit string, the result is determined as follows: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, or $1 \oplus 1 = 0$.

Example: $01101 \oplus 11010 = 10111$

$+$ Addition.

$*$ Multiplication.

$/$ Division.

$a \| b$ The concatenation of two strings a and b. Either a and b are both bit strings, or both are byte strings.

$\lceil a \rceil$ The ceiling of a: the smallest integer that is greater than or equal to a. For example, $\lceil 5 \rceil = 5$, $\lceil 5.3 \rceil = 6$, and $\lceil -2.1 \rceil = -2$.

$\lfloor a \rfloor$ The floor of a; the largest integer that is less than or equal to a. For example, $\lfloor 5 \rfloor = 5$, $\lfloor 5.3 \rfloor = 5$, and $\lfloor -2.1 \rfloor. = -3$.

$|a|$ The absolute value of a; $|a|$ is $- a$ if $a < 0$; otherwise, it is simply a. For example, $|2| = 2$, and $|-2| = 2$.

$[a, b]$ The interval of integers between and including a and b. For example, [1, 4] consists of the integers 1, 2, 3 and 4.

$\{, a, b, \ldots\}$ Used to indicate optional information.

0x The prefix to a bit string that is represented in hexadecimal characters.

3. General Discussion

A digital signature is an electronic analogue of a written signature; the digital signature can be used to provide assurance that the claimed signatory signed the information. In addition, a digital signature may be used to detect whether or not the information was modified after it was signed (i.e., to detect the integrity of the signed data). These assurances may be obtained whether the data was received in a transmission or retrieved from storage. A properly implemented digital signature algorithm that meets the requirements of this Standard can provide these services.

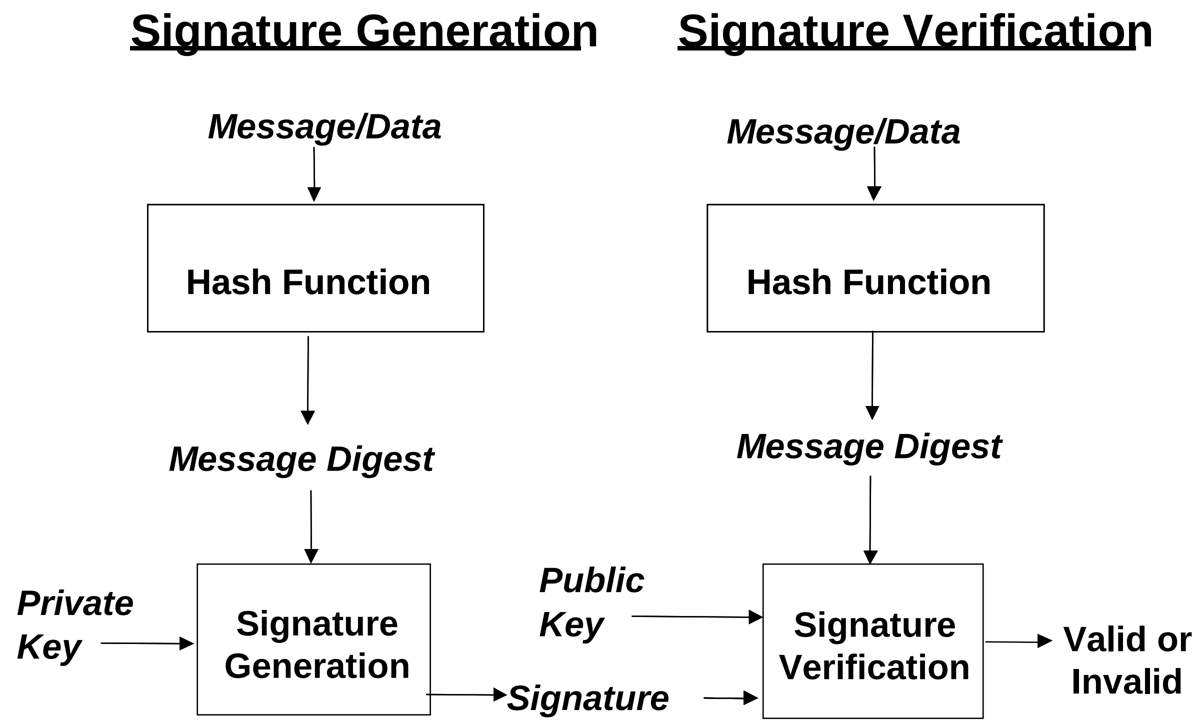


Figure 1: Digital Signature Processes

A digital signature algorithm includes a signature generation process and a signature verification process. A signatory uses the generation process to generate a digital signature on data; a verifier uses the verification process to verify the authenticity of the signature. Each signatory has a public and private key and is the owner of that key pair. As shown in Figure 1, the private key is used in the signature generation process. The key pair owner is the only entity that is authorized to use the private key to generate digital signatures. In order to prevent other entities from claiming to be the key pair owner and using the private key to generate fraudulent signatures, the private key must remain secret. The approved digital signature algorithms are designed to prevent an adversary who does not know the signatory's private key from generating the same signature as the signatory on a different message. In other words, signatures are designed so that they cannot be forged. A number of alternative terms are used in this Standard to refer to the signatory or key pair owner. An entity that intends to generate digital signatures in the future may be referred to as the *intended signatory*. Prior to the verification of a signed message, the signatory is referred to as the *claimed signatory* until such time as adequate assurance can be obtained of the actual identity of the signatory.

The public key is used in the signature verification process (see Figure 1). The public key need not be kept secret, but its integrity must be maintained. Anyone can verify a correctly signed message using the public key.

For both the signature generation and verification processes, the message (i.e., the signed data) is converted to a fixed-length representation of the message by means of an approved hash function. Both the original message and the digital signature are made available to a verifier.

A verifier requires assurance that the public key to be used to verify a signature belongs to the entity that claims to have generated a digital signature (i.e., the claimed signatory). That is, a verifier requires assurance that the signatory is the actual owner of the public/private key pair used to generate and verify a digital signature. A binding of an owner's identity and the owner's public key **shall** be effected in order to provide this assurance.

A verifier also requires assurance that the key pair owner actually possesses the private key associated with the public key, and that the public key is a mathematically correct key.

By obtaining these assurances, the verifier has assurance that if the digital signature can be correctly verified using the public key, the digital signature is valid (i.e., the key pair owner really signed the message). Digital signature validation includes both the (mathematical) verification of the digital signature and obtaining the appropriate assurances. The following are reasons why such assurances are required.

1. If a verifier does not obtain assurance that a signatory is the actual owner of the key pair whose public component is used to verify a signature, the problem of forging a signature is reduced to the problem of falsely claiming an identity. For example, anyone in possession of a mathematically consistent key pair can sign a message and claim that the signatory was the President of the United States. If a verifier does not require assurance that the President is actually the owner of the public key that is used to mathematically verify the message's signature, then successful signature verification provides assurance that the message has not been altered since it was signed, but does not provide assurance that the message came from the President (i.e., the verifier has assurance of the data's integrity, but source authentication is lacking).
2. If the public key used to verify a signature is not mathematically valid, the arguments used to establish the cryptographic strength of the signature algorithm may not apply. The owner may not be the only party who can generate signatures that can be verified

with that public key.

3. If a public key infrastructure cannot provide assurance to a verifier that the owner of a key pair has demonstrated knowledge of a private key that corresponds to the owner's public key, then it may be possible for an unscrupulous entity to have their identity (or an assumed identity) bound to a public key that is (or has been) used by another party. The unscrupulous entity may then claim to be the source of certain messages signed by that other party. Or, it may be possible that an unscrupulous entity has managed to obtain ownership of a public key that was chosen with the sole purpose of allowing for the verification of a signature on a specific message.

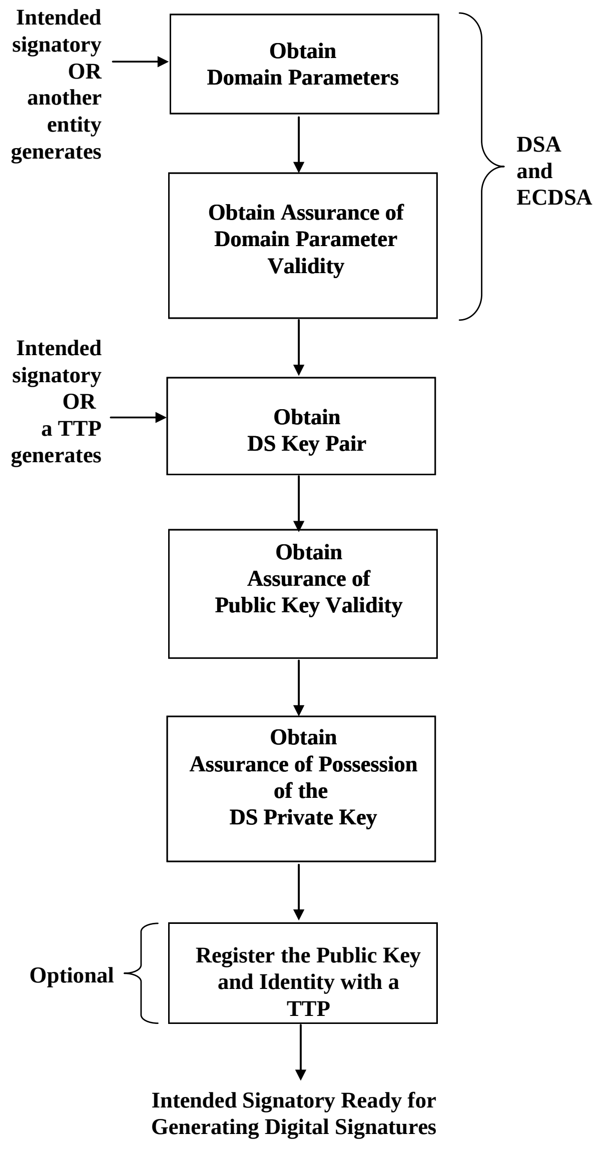

Intended signatory OR another entity generates
Obtain Domain Parameters
Obtain Assurance of Domain Parameter Validity
DSA and ECDSA
Intended signatory OR a TTP generates
Obtain DS Key Pair
Obtain Assurance of Public Key Validity
Obtain Assurance of Possession of the DS Private Key
Optional
Register the Public Key and Identity with a TTP
Intended Signatory Ready for Generating Digital Signatures

Figure 2: Initial Setup by an Intended Signatory

Technically, a key pair used by a digital signature algorithm could also be used for purposes other than digital signatures (e.g., for key establishment). However, a key pair used for digital signature generation and verification as specified in this Standard **shall not** be used for any other purpose. See SP 800-57 on Key Usage for further information.

A number of steps are required to enable a digital signature generation or verification capability in accordance with this Standard. All parties that generate digital signatures **shall** perform the initial setup process as discussed in Section 3.1. Digital signature generation **shall** be performed as discussed in Section 3.2. Digital signature verification and validation **shall** be performed as discussed in Section 3.3.

3.1 Initial Setup

Figure 2 depicts the steps that are performed prior to generating a digital signature by an entity intending to act as a signatory.

For the DSA and ECDSA algorithms, the intended signatory **shall** first obtain appropriate domain parameters, either by generating the domain parameters itself, or by obtaining domain parameters that another entity has generated. Having obtained the set of domain parameters, the intended signatory **shall** obtain assurance of the validity of those domain parameters; approved methods for obtaining this assurance are provided in SP 800-89. Note that the RSA algorithm does not use domain parameters.

Each intended signatory **shall** obtain a digital signature key pair that is generated as specified for the appropriate digital signature algorithm, either by generating the key pair itself or by obtaining the key pair from a trusted party. The intended signatory is authorized to use the key pair and is the owner of that key pair. Note that if a trusted party generates the key pair, that party needs to be trusted not to masquerade as the owner, even though the trusted party knows the private key.

After obtaining the key pair, the intended signatory (now the key pair owner) **shall** obtain (1) assurance of the validity of the public key and (2) assurance that he/she actually possesses the associated private key. Approved methods for obtaining these assurances are provided in SP 800-89.

A digital signature verifier requires assurance of the identity of the signatory. Depending on the environment in which digital signatures are generated and verified, the key pair owner (i.e., the intended signatory) may register the public key and establish proof of identity with a mutually trusted party. For example, a certification authority (CA) could sign credentials containing an owner's public key and identity to form a certificate after being provided with proof of the owner's identity. Systems for certifying credentials and distributing certificates are beyond the scope of this Standard. Other means of establishing proof of identity (e.g., by providing identity credentials along with the public key directly to a prospective verifier) are allowed.

3.2 Digital Signature Generation

Figure 3 depicts the steps that are performed by an intended signatory (i.e., the entity that generates a digital signature).

Prior to the generation of a digital signature, a message digest **shall** be generated on the information to be signed using an appropriate approved hash function.

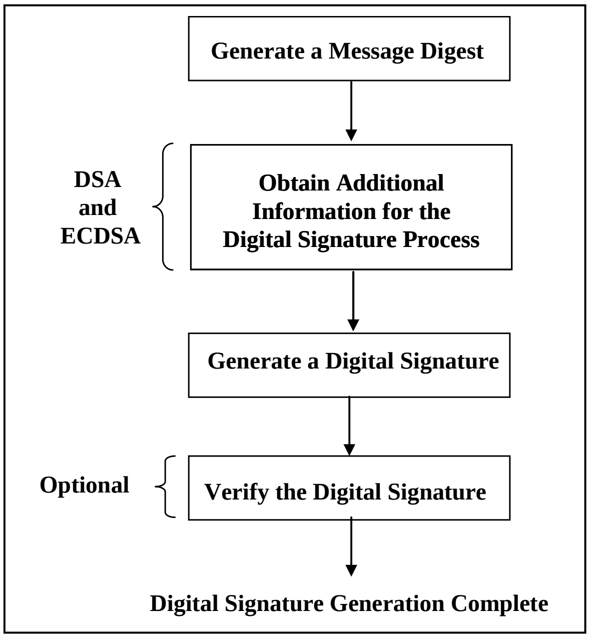

Generate a Message Digest
DSA and ECDSA
Obtain Additional Information for the Digital Signature Process
Generate a Digital Signature
Optional
Verify the Digital Signature
Digital Signature Generation Complete

Figure 3: Digital Signature Generation

Depending on the digital signature algorithm to be used, additional information **shall** be obtained. For example, a random per-message secret number **shall** be obtained for DSA and ECDSA.

Using the selected digital signature algorithm, the signature private key, the message digest, and any other information required by the digital signature process, a digital signature **shall** be generated in accordance with this Standard.

The signatory may optionally verify the digital signature using the signature verification process and the associated public key. This optional verification serves as a final check to detect otherwise undetected signature generation computation errors; this verification may be prudent when signing a high-value message, when multiple users are expected to verify the signature, or if the verifier will be verifying the signature at a much later time.

3.3 Digital Signature Verification and Validation

Figure 4 depicts the digital signature verification and validation process that are performed by a verifier (e.g., the intended recipient of the signed data and associated digital signature). Note that the figure depicts a successful verification and validation process (i.e., no errors are detected).

In order to verify a digital signature, the verifier **shall** obtain the public key of the claimed signatory, (usually) based on the claimed identity. If DSA or ECDSA has been used to generate the digital signature, the verifier **shall** also obtain the domain parameters. The public key and domain parameters may be obtained, for example, from a certificate created by a trusted party (e.g., a CA) or directly from the claimed signatory. A message digest **shall** be generated on the data whose signature is to be verified (i.e., not on the received digital signature) using the same hash function that was used during the digital signature generation process. Using the appropriate digital signature algorithm, the domain parameters (if appropriate), the public key and the newly computed message digest, the received digital signature is verified in accordance with this Standard. If the verification process fails, no inference can be made as to whether the data is correct, only that in using the specified public key and the specified signature format, the digital signature cannot be verified for that data.

Before accepting the verified digital signature as valid, the verifier **shall** have (1) assurance of the signatory's claimed identity, (2) assurance of the validity of the domain parameters (for DSA and ECDSA), (3) assurance of the validity of the public key, and (4) assurance that the claimed signatory actually possessed the private key that was used to generate the digital signature at the time that the signature was generated. Methods for the verifier to obtain these assurances are provided in SP 800-89. Note that assurance of domain parameter validity may have been obtained during initial setup (see Section 3.1).

If the verification and assurance processes are successful, the digital signature and signed data **shall** be considered valid. However, if a verification or assurance process fails, the digital signature **should** be considered invalid. An organization's policy **shall** govern the action to be taken for an invalid digital signature. Such policy is outside the scope of this Standard.

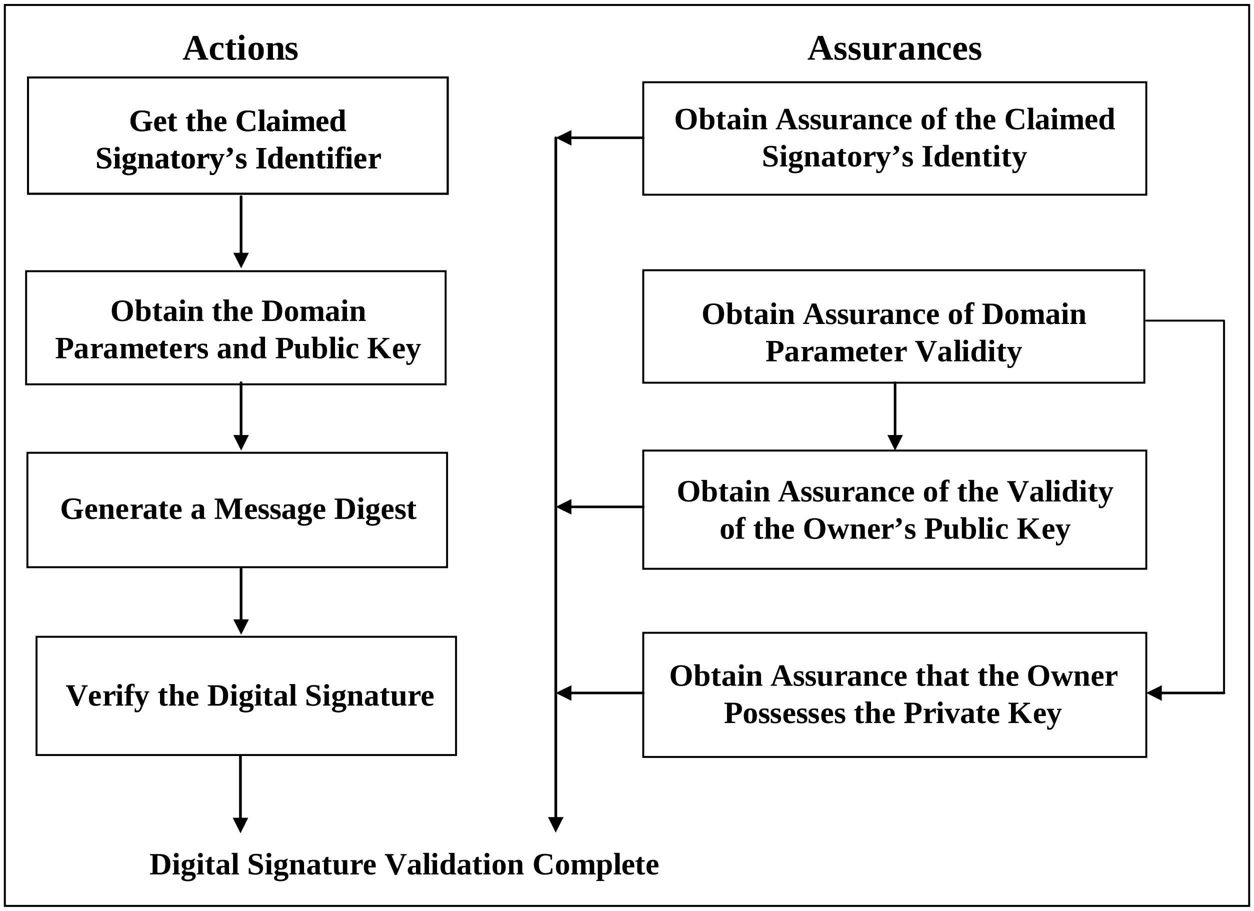

Actions
Assurances
Get the Claimed Signatory's Identifier
Obtain Assurance of the Claimed Signatory's Identity
Obtain the Domain Parameters and Public Key
Obtain Assurance of Domain Parameter Validity
Generate a Message Digest
Obtain Assurance of the Validity of the Owner's Public Key
Verify the Digital Signature
Obtain Assurance that the Owner Possesses the Private Key
Digital Signature Validation Complete

Figure 4: Digital Signature Verification and Validation

4 The Digital Signature Algorithm (DSA)

4.1 DSA Parameters

A DSA digital signature is computed using a set of domain parameters, a private key x, a per-message secret number k, data to be signed, and a hash function. A digital signature is verified using the same domain parameters, a public key y that is mathematically associated with the private key x used to generate the digital signature, data to be verified, and the same hash function that was used during signature generation. These parameters are defined as follows:

p a prime modulus, where $2^{L-1} < p < 2^L$, and L is the bit length of p. Values for L are provided in Section 4.2.

q a prime divisor of $(p - 1)$, where $2^{N-1} < q < 2^N$, and N is the bit length of q. Values for N are provided in Section 4.2.

g a generator of the subgroup of order q mod p, such that $1 < g < p$.

x the private key that must remain secret; x is a randomly or pseudorandomly generated integer, such that $0 < x < q$, i.e., x is in the range $[1, q–1]$.

y the public key, where $y = g^x \bmod p$.

k a secret number that is unique to each message; k is a randomly or pseudorandomly generated integer, such that $0 < k < q$, i.e., k is in the range $[1, q–1]$.

4.2 Selection of Parameter Sizes and Hash Functions for DSA

This Standard specifies the following choices for the pair L and N (the bit lengths of p and q, respectively):

$L = 1024, N = 160$

$L = 2048, N = 224$

$L = 2048, N = 256$

$L = 3072, N = 256$

Federal Government entities **shall** generate digital signatures using use one or more of these choices.

An approved hash function, as specified in FIPS 180-3, **shall** be used during the generation of digital signatures. The security strength associated with the DSA digital signature process is no greater than the minimum of the security strength of the (L, N) pair and the security strength of the hash function that is employed. Both the security strength of the hash function used and the security strength of the (L, N) pair **shall** meet or exceed the security strength required for the digital signature process. The security strength for each (L, N) pair and hash function is provided in SP 800-57.

SP 800-57 provides information about the selection of the appropriate (*L*, *N*) pair in accordance with a desired security strength for a given time period for the generation of digital signatures. An (*L*, *N*) pair **shall** be chosen that protects the signed information during the entire expected lifetime of that information. For example, if a digital signature is generated in 2009 for information that needs to be protected for five years, and a particular (*L*, *N*) pair is invalid after 2010, then a larger (*L*, *N*) pair **shall** be used that remains valid for the entire period of time that the information needs to be protected.

It is recommended that the security strength of the (*L*, *N*) pair and the security strength of the hash function used for the generation of digital signatures be the same unless an agreement has been made between participating entities to use a stronger hash function. When the length of the output of the hash function is greater than *N* (i.e., the bit length of *q*), then the leftmost *N* bits of the hash function output block **shall** be used in any calculation using the hash function output during the generation or verification of a digital signature. A hash function that provides a lower security strength than the (*L*, *N*) pair ordinarily **should not** be used, since this would reduce the security strength of the digital signature process to a level no greater than that provided by the hash function.

A Federal Government entity other than a Certification Authority (CA) **should** use only the first three (*L*, *N*) pairs (i.e., the (1024, 160), (2048, 224) and (2048, 256) pairs). A CA **shall** use an (*L*, *N*) pair that is equal to or greater than the (*L*, *N*) pairs used by its subscribers. For example, if subscribers are using the (2048, 224) pair, then the CA **shall** use either the (2048, 224), (2048, 256) or (3072, 256) pair. Possible exceptions to this rule include cross certification between CAs, certifying keys for purposes other than digital signatures and transitioning from one key size or algorithm to another. See SP 800-57 for further guidance.

4.3 DSA Domain Parameters

DSA requires that the private/public key pairs used for digital signature generation and verification be generated with respect to a particular set of domain parameters. These domain parameters may be common to a group of users and may be public. A user of a set of domain parameters (i.e., both the signatory and the verifier) **shall** have assurance of their validity prior to using them (see Section 3). Although domain parameters may be public information, they **shall** be managed so that the correct correspondence between a given key pair and its set of domain parameters is maintained for all parties that use the key pair. A set of domain parameters may remain fixed for an extended time period.

The domain parameters for DSA are the integers *p*, *q*, and *g*, and optionally, the *domain_parameter_seed* and *counter* that were used to generate *p* and *q* (i.e., the full set of domain parameters is (*p*, *q*, *g* {, *domain_parameter_seed*, *counter*})).

4.3.1 Domain Parameter Generation

Domain parameters may be generated by a trusted third party (a TTP, such as a CA) or by an entity other than a TTP. Assurance of domain parameter validity **shall** be obtained prior to key pair generation, digital signature generation or digital signature verification (see Section 3).

The integers *p* and *q* **shall** be generated as specified in Appendix A.1. The input to the generation process is the selected values of *L* and *N* (see Section 4.2); the output of the generation process is the values for *p* and *q*, and optionally, the values of the *domain_parameter_seed* and *counter*.

The generator *g* **shall** be generated as specified in Appendix A.2.

The security strength of a hash function used during the generation of the domain parameters **shall** meet or exceed the security strength associated with the (*L*, *N*) pair. Note that this is more restrictive than the hash function that can be used for the digital signature process (see Section 4.2).

4.3.2 Domain Parameter Management

Each digital signature key pair **shall** be correctly associated with one specific set of domain parameters (e.g., by a public key certificate that identifies the domain parameters associated with the public key). The domain parameters **shall** be protected from unauthorized modification until the set is deactivated (if and when the set is no longer needed). The same domain parameters may be used for more than one purpose (e.g., the same domain parameters may be used for both digital signatures and key establishment). However, using different values for the generator *g* reduces the risk that key pairs generated for one purpose could be accidentally used (successfully) for another purpose.

4.4 Key Pairs

Each signatory has a key pair: a private key *x* and a public key *y* that are mathematically related to each other. The private key **shall** be used for only a fixed period of time (i.e., the private key cryptoperiod) in which digital signatures may be generated; the public key may continue to be used as long as digital signatures that were generated using the associated private key need to be verified (i.e., the public key may continue to be used beyond the cryptoperiod of the associated private key). See SP 800-57 for further guidance.

4.4.1 DSA Key Pair Generation

A digital signature key pair *x* and *y* is generated for a set of domain parameters (*p*, *q*, *g* {, *domain_parameter_seed*, *counter*}). Methods for the generation of *x* and *y* are provided in Appendix B.1.

4.4.2 Key Pair Management

Guidance on the protection of key pairs is provided in SP 800-57. The secure use of digital signatures depends on the management of an entity's digital signature key pair as follows:

1. The validity of the domain parameters **shall** be assured prior to the generation of the key pair, or the verification and validation of a digital signature (see Section 3).
2. Each key pair **shall** be associated with the domain parameters under which the key pair was generated.
3. A key pair **shall** only be used to generate and verify signatures using the domain parameters associated with that key pair.
4. The private key **shall** be used only for signature generation as specified in this Standard and **shall** be kept secret; the public key **shall** be used only for signature verification and may be made public.
5. An intended signatory **shall** have assurance of possession of the private key prior to or concurrently with using it to generate a digital signature (see Section 3.1).
6. A private key **shall** be protected from unauthorized access, disclosure and modification.
7. A public key **shall** be protected from unauthorized modification (including substitution). For example, public key certificates that are signed by a CA may provide such protection.
8. A verifier **shall** be assured of a binding between the public key, its associated domain parameters and the key pair owner (see Section 3).
9. A verifier **shall** obtain public keys in a trusted manner (e.g., from a certificate signed by a CA that the entity trusts, or directly from the intended or claimed signatory, provided that the entity is trusted by the verifier and can be authenticated as the source of the signed information that is to be verified).
10. Verifiers **shall** be assured that the claimed signatory is the key pair owner, and that the owner possessed the private key that was used to generate the digital signature at the time that the signature was generated (i.e., the private key that is associated with the public key that will be used to verify the digital signature) (see Section 3.3).
11. A signatory and a verifier **shall** have assurance of the validity of the public key (see Sections 3.1 and 3.3).

4.5 DSA Per-Message Secret Number

A new secret random number *k* **shall** be generated prior to the generation of each digital signature for use during the signature generation process. This secret number **shall** be protected from unauthorized disclosure and modification.

k^{-1} is the multiplicative inverse of *k* with respect to multiplication modulo *q*; i.e., $0 < k^{-1} < q$

and 1 = (k^{-1} k) mod q. This inverse is required for the signature generation process (see Section 4.6). A technique is provided in Appendix C.1 for deriving k^{-1} from k.

k and k^{-1} may be pre-computed, since knowledge of the message to be signed is not required for the computations. When k and k^{-1} are pre-computed, their confidentiality and integrity **shall** be protected.

Methods for the generation of the per-message secret number are provided in Appendix B.2.

4.6 DSA Signature Generation

The intended signatory **shall** have assurances as specified in Section 3.1.

Let N be the bit length of q. Let **min**(N, *outlen*) denote the minimum of the positive integers N and *outlen*, where *outlen* is the bit length of the hash function output block.

The signature of a message M consists of the pair of numbers r and s that is computed according to the following equations:

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

z = the leftmost **min**(N, *outlen*) bits of **Hash**(M).

$$s = (k^{-1}(z + xr)) \bmod q.$$

When computing s, the string z obtained from **Hash**(M) **shall** be converted to an integer. The conversion rule is provided in Appendix C.2.

Note that r may be computed whenever k, p, q and g are available, e.g., whenever the domain parameters p, q and g are known, and k has been pre-computed (see Section 4.5), r may also be pre-computed, since knowledge of the message to be signed is not required for the computation of r. Pre-computed k, k^{-1} and r values **shall** be protected in the same manner as the the private key x until s has been computed (see SP 800-57).

The values of r and s **shall** be checked to determine if $r = 0$ or $s = 0$. If either $r = 0$ or $s = 0$, a new value of k **shall** be generated, and the signature **shall** be recalculated. It is extremely unlikely that $r = 0$ or $s = 0$ if signatures are generated properly.

The signature (r, s) may be transmitted along with the message to the verifier.

4.7 DSA Signature Verification and Validation

Signature verification may be performed by any party (i.e., the signatory, the intended recipient or any other party) using the signatory's public key. A signatory may wish to verify that the computed signature is correct, perhaps before sending the signed message to the intended recipient. The intended recipient (or any other party) verifies the signature to determine its authenticity.

Prior to verifying the signature of a signed message, the domain parameters, and the claimed signatory's public key and identity **shall** be made available to the verifier in an authenticated manner. The public key may, for example, be obtained in the form of a certificate signed by a trusted entity (e.g., a CA) or in a face-to-face meeting with the public key owner.

Let M', r', and s' be the received versions of M, r, and s, respectively; let y be the public key of the claimed signatory; and let N be the bit length of q. Also, let **min**(N, *outlen*) denote the minimum of the positive integers N and *outlen*, where *outlen* is the bit length of the hash function output block.

The signature verification process is as follows:

1. The verifier **shall** check that $0 < r' < q$ and $0 < s' < q$; if either condition is violated, the signature **shall** be rejected as invalid.
2. If the two conditions in step 1 are satisfied, the verifier computes the following:

   $w = (s')^{-1} \bmod q$.

   z = the leftmost **min**(N, *outlen*) bits of **Hash**(M').

   $u1 = (zw) \bmod q$.

   $u2 = ((r')w) \bmod q$.

   $v = (((g)^{u1} (y)^{u2}) \bmod p) \bmod q$.

   A technique is provided in Appendix C.1 for deriving $(s')^{-1}$ (i.e., the multiplicative inverse of s' mod q).

   The string z obtained from **Hash**(M') **shall** be converted to an integer. The conversion rule is provided in Appendix C.2.
3. If $v = r'$, then the signature is verified. For a proof that $v = r'$ when $M' = M$, $r' = r$, and $s' = s$, see Appendix E.
4. If v does not equal r', then the message or the signature may have been modified, there may have been an error in the signatory's generation process, or an imposter (who did not know the private key associated with the public key of the claimed signatory) may have attempted to forge the signature. The signature **shall** be considered invalid. No inference can be made as to whether the data is valid, only that when using the public key to verify the signature, the signature is incorrect for that data.
5. Prior to accepting the signature as valid, the verifier **shall** have assurances as specified in Section 3.3.

An organization's policy may govern the action to be taken for invalid digital signatures. Such policy is outside the scope of this Standard. Guidance about determining the timeliness of digitally signed messages is addressed in SP 800-102, Recommendation for Digital Signature Timeliness.

5. The RSA Digital Signature Algorithm

The use of the RSA algorithm for digital signature generation and verification is specified in American National Standard (ANS) X9.31 and Public Key Cryptography Standard (PKCS) #1. While each of these standards uses the RSA algorithm, the format of the ANS X9.31 and PKCS #1 data on which the digital signature is generated differs in details that make the algorithms non-interchangeable.

5.1 RSA Key Pair Generation

An RSA digital signature key pair consists of an RSA private key, which is used to compute a digital signature, and an RSA public key, which is used to verify a digital signature. An RSA key pair used for digital signatures **shall** only be used for one digital signature scheme (e.g., ANS X9.31, RSASSA-PKCS1 v1.5 or RSASSA-PSS; see Sections 5.4 and 5.5). In addition, an RSA digital signature key pair **shall not** be used for other purposes (e.g., key establishment).

An RSA public key consists of a modulus *n*, which is the product of two positive prime integers *p* and *q* (i.e., $n = pq$), and a public key exponent *e*. Thus, the RSA public key is the pair of values (*n*, *e*) and is used to verify digital signatures. The size of an RSA key pair is commonly considered to be the length of the modulus *n* in bits (*nlen*).

The corresponding RSA private key consists of the same modulus *n* and a private key exponent *d* that depends on *n* and the public key exponent *e*. Thus, the RSA private key is the pair of values (*n*, *d*) and is used to generate digital signatures. Note that an alternative method for representing (*n*, *d*) using the Chinese Remainder Theorem (CRT) is allowed.

In order to provide security for the digital signature process, the two integers *p* and *q*, and the private key exponent *d* **shall** be kept secret. The modulus *n* and the public key exponent *e* may be made known to anyone. Guidance on the protection of these values is provided in SP 800-57.

This Standard specifies three choices for the length of the modulus (i.e., *nlen*): 1024, 2048 and 3072 bits. Federal Government entities **shall** generate digital signatures using one or more of these choices.

An approved hash function, as specified in FIPS 180-3, **shall** be used during the generation of key pairs and digital signatures. When used during the generation of an RSA key pair (as specified in this Standard), the length in bits of the hash function output block **shall** meet or exceed the security strength associated with the bit length of the modulus *n* (see SP 800-57).

The security strength associated with the RSA digital signature process is no greater than the minimum of the security strength associated with the bit length of the modulus and the security strength of the hash function that is employed. The security strength for each modulus length and hash function used during the digital signature process is provided in SP 800-57. Both the security strength of the hash function used and the security strength associated with the bit length of the modulus *n* **shall** meet or exceed the security strength required for the digital signature process.

It is recommended that the security strength of the modulus and the security strength of the hash function be the same unless an agreement has been made between participating entities to use a stronger hash function. A hash function that provides a lower security strength than the security strength associated with the bit length of the modulus ordinarily **should not** be used, since this would reduce the security strength of the digital signature process to a level no greater than that provided by the hash function.

Federal Government entities other than CAs **should** use only the first two choices (i.e., *nlen* = 1024 or 2048) during the timeframes indicated in SP 800-57. A CA **should** use a modulus whose length *nlen* is equal to or greater than the moduli used by its subscribers. For example, if the subscribers are using *nlen* = 2048, then the CA **should** use *nlen* ≥ 2048. SP 800-57 provides further information about the selection of the bit length of *n*. Possible exceptions to this rule include cross certification between CAs, certifying keys for purposes other than digital signatures and transitioning from one key size or algorithm to another.

Criteria for the generation of RSA key pairs are provided in Appendix B.3.1.

When RSA parameters are randomly generated (i.e., the primes *p* and *q*, and optionally, the public key exponent *e*), they **shall** be generated using an approved random or pseudorandom number generator (see SP 800-90). The resulting (pseudo) random numbers **shall** be used as seeds for generating RSA parameters (e.g., the (pseudo) random number is used as a prime number generation seed). Prime number generation seeds **shall** be kept secret or destroyed when the modulus *n* is computed. If the prime number generation seeds are retained, they **shall** only be used as evidence that the generated values (i.e., *p* and *q*) were determined in an arbitrary manner, and the seeds **shall** be protected in a manner that is (at least) equivalent to the protection required for the private key.

5.2 Key Pair Management

The secure use of digital signatures depends on the management of an entity's digital signature key pair. Key pair management requirements for RSA are provided in Section 4.4.2, requirements 4 – 11. Note that the first three requirements in Section 4.4.2, which address the relationship between domain parameters and key pairs, are not applicable to RSA.

5.3 Assurances

The intended signatory **shall** have assurances as specified in Section 3.1. Prior to accepting a digital signature as valid, the verifier **shall** have assurances as specified in Section 3.3.

5.4 ANS X9.31

ANS X9.31, *Digital Signatures Using Reversible Public Key Cryptography for the Financial*

Services Industry (*rDSA*), was developed for the American National Standards Institute by the Accredited Standards Committee on Financial Services, X9. See http://www.x9.org for information about obtaining copies of ANS X9.31 and any associated errata. The following discussions are based on the version of ANS X9.31 that was approved in 1998.

Methods for the generation of the private prime factors *p* and *q* are provided in Appendix B.3.

In ANS X9.31, the length of the modulus *n* is allowed in increments of 256 bits beyond a minimum of 1024 bits. Implementations claiming conformance to FIPS 186-3 **shall** include one or more of the modulus sizes specified in Section 5.1.

Two methods for the generation of digital signatures are included in ANS X9.31. When the public signature verification exponent *e* is odd, the digital signature algorithm is commonly known as RSA; when the public signature verification exponent *e* is even, the digital signature algorithm is commonly known as Rabin-Williams. This Standard (i.e., FIPS 186-3) adopts the use of RSA, but does not adopt the use of Rabin-Williams.

During signature verification, the extraction of the hash value H(*M*)′ from the data structure *IR*′ **shall** be accomplished by either:

- Selecting the *hashlen* bytes of the data structure *IR*′ that immediately precedes the two bytes of trailer information, where *hashlen* is the length in bytes of the hash function used, regardless of the length of the padding, or
- If the hash value H(*M*)′ is selected by its location with respect to the last byte of padding (i.e., 0xBA), including a check that the hash value is followed by only two bytes containing the expected trailer value.

ANS X9.31 contains an annex on random number generation. However, implementations of ANS X9.31 **shall** use the approved random number generation methods specified in SP 800-90.

Annexes in ANS X9.31 provide informative discussions of security and implementation considerations.

5.5 PKCS #1

Public-Key Cryptography Standard (PKCS) #1, *RSA Cryptography Standard*, defines mechanisms for encrypting and signing data using the RSA algorithm. PKCS #1 v2.1 specifies two digital signature processes and corresponding formats: RSASSA-PKCS1-v1.5 and RSASSA-PSS. Both signature schemes are approved for use, but additional constraints are imposed beyond those specified in PKCS #1 v2.1.

(a) Implementations that generate RSA key pairs **shall** use the criteria and methods in Appendix B.3 to generate those key pairs,

(b) Only approved hash functions **shall** be used.

(c) Only two prime factors *p* and *q* **shall** be used to form the modulus *n*.

(d) Random numbers **shall** be generated in accordance with SP 800-90.

(e) For RSASSA-PSS, the length of the salt (*sLen*) **shall** be: $0 \leq sLen \leq hlen$, where *hlen* is the length of the hash function output block.

(f) For RSASSA-PKCS-v1.5, when the hash value is recovered from the encoded message *EM* during the verification of the digital signature[1], the extraction of the hash value **shall** be accomplished by either:

- Selecting the rightmost (least significant) bits of *EM*, based on the size of the hash function used, regardless of the length of the padding, or
- If the hash value is selected by its location with respect to the last byte of padding, including a check that the hash value is located in the rightmost (least significant) bytes of *EM* (i.e., no other information follows the hash value in the encoded message).

Note: PKCS #1 was initially developed by RSA Laboratories in 1991 and has been revised as multiple versions. At the time of the approval of FIPS 186-3, three versions of PKSC #1 were available: version 1.5, version 2.0 and version 2.1. This Standard references only version 2.1.

[1] PKCS #1, v2.1 provides two methods for comparing the hash values: by comparing the encoded messages *EM* and *EM*′, and by extracting the hash value from the decoding of the encoded message (see the Note in PKCS #1, v2.1). Step (f) above applies to the latter case.

6. The Elliptic Curve Digital Signature Algorithm (ECDSA)

ANS X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Standard* (*ECDSA*), was developed for the American National Standards Institute by the Accredited Standards Committee on Financial Services, X9. Information about obtaining copies of ANS X9.62 is available at http://www.x9.org. The following discussions are based on the version of ANS X9.62 that was approved in 2005. This version of ANS X9.62 **shall** be used, subject to the transition period referenced in the implementation schedule of this Standard.

ANS X9.62 defines methods for digital signature generation and verification using the Elliptic Curve Digital Signature Algorithm (ECDSA). Specifications for the generation of the domain parameters used during the generation and verification of digital signatures are also included in ANS X9.62. ECDSA is the elliptic curve analog of DSA. ECDSA keys **shall not** be used for any other purpose (e.g., key establishment).

6.1 ECDSA Domain Parameters

ECDSA requires that the private/public key pairs used for digital signature generation and verification be generated with respect to a particular set of domain parameters. These domain parameters may be common to a group of users and may be public. Domain parameters may remain fixed for an extended time period.

Domain parameters for ECDSA are of the form (*q, FR, a, b* {, *domain_parameter_seed*}*, G, n, h*), where *q* is the field size; *FR* is an indication of the basis used; *a* and *b* are two field elements that define the equation of the curve; *domain_parameter_seed* is the domain parameter seed and is an optional bit string that is present if the elliptic curve was randomly generated in a verifiable fashion, *G* is a base point of prime order on the curve (i.e., $G = (x_{G,} y_G)$), *n* is the order of the point *G*, and *h* is the cofactor (which is equal to the order of the curve divided by *n*).

6.1.1 Domain Parameter Generation

This Standard specifies five ranges for *n* (see Table 1). For each range, a maximum cofactor size is also specified. Note that the specification of a cofactor *h* in a set of domain parameters is optional in ANS X9.62, whereas implementations conforming to this Standard (i.e., FIPS 186-3) **shall** specify the cofactor *h* in the set of domain parameters. Table 1 provides the maximum sizes for the cofactor *h*.

Table 1: ECDSA Security Parameters

Bit length of *n* $\lceil \log_2 n \rceil$	**Maximum Cofactor (*h*)**
160 - 223	2^{10}
224 - 255	2^{14}
256 - 383	2^{16}
384 - 511	2^{24}
≥ 512	2^{32}

ECDSA is defined for two arithmetic fields: the finite field GF_p and the finite field GF_{2^m}. For the field GF_p, p is required to be an odd prime.

NIST Recommended curves are provided in Appendix D of this Standard (i.e., FIPS 186-3). Three types of curves are provided:

1. Curves over prime fields, which are identified as P-xxx,
2. Curves over Binary fields, which are identified as B-xxx, and
3. Koblitz curves, which are identified as K-xxx,

where xxx indicates the bit length of the field size.

Alternatively, ECDSA domain parameters may be generated as specified in ANS X9.62; when ECDSA domain parameters are generated (i.e., the NIST Recommended curves are not used), the value of *G* **should** be generated canonically (verifiably random). An approved hash function, as specified in FIPS 180-3, **shall** be used during the generation of ECDSA domain parameters. When generating these domain parameters, the security strength of a hash function used **shall** meet or exceed the security strength associated with the bit length of *n* (see footnote 2 below).

An approved hash function, as specified in FIPS 180-3, is required during the generation of domain parameters. The security strength of the hash function used **shall** meet or exceed the security strength associated with the bit length of *n*. The security strengths for the ranges of *n* and the hash functions are provided in SP 800-57. It is recommended that the security strength associated with the bit length of *n* and the security strength of the hash function be the same

[2] The NIST Recommended curves were generated prior to the formulation of this guidance and using SHA-1, which was the only approved hash function available at that time. Since SHA-1 was considered secure at the time of generation, the curves were made public, and SHA-1 will only be used to validate those curves, the NIST Recommended curves are still considered secure and appropriate for Federal government use.

unless an agreement has been made between participating entities to use a stronger hash function; a hash function that provides a lower security strength than is associated with the bit length of n **shall not** be used. If the length of the output of the hash function is greater than the bit length of n, then the leftmost n bits of the hash function output block **shall** be used in any calculation using the hash function output during the generation or verification of a digital signature.

Normally, a CA **should** use a bit length of n whose assessed security strength is equal to or greater than the assessed security strength associated with the bit length of n used by its subscribers. For example, if subscribers are using a bit length of n with an assessed security strength of 112 bits, then CAs **should** use a bit length of n whose assessed security strength is equal to or greater than 112 bits. SP 800-57 provides further information about the selection of a bit length of n. Possible exceptions to this rule include cross certification between CAs, certifying keys for purposes other than digital signatures and transitioning from one key size or algorithm to another. However, these exceptions require further analysis.

6.1.2 Domain Parameter Management

Each ECDSA key pair **shall** be correctly associated with one specific set of domain parameters (e.g., by a public key certificate that identifies the domain parameters associated with the public key). The domain parameters **shall** be protected from unauthorized modification until the set is deactivated (if and when the set is no longer needed). The same domain parameters may be used for more than one purpose (e.g., the same domain parameters may be used for both digital signatures and key establishment). However, using different domain parameters reduces the risk that key pairs generated for one purpose could be accidentally used (successfully) for another purpose.

6.2 Private/Public Keys

An ECDSA key pair consists of a private key d and a public key Q that is associated with a specific set of ECDSA domain parameters; d, Q and the domain parameters are mathematically related to each other. The private key is normally used for a period of time (i.e., the cryptoperiod); the public key may continue to be used as long as digital signatures that have been generated using the associated private key need to be verified (i.e., the public key may continue to be used beyond the cryptoperiod of the associated private key). See SP 800-57 for further guidance.

ECDSA keys **shall** only be used for the generation and verification of ECDSA digital signatures.

6.2.1 Key Pair Generation

A digital signature key pair d and Q is generated for a set of domain parameters (q, FR, a, b {, *domain_parameter_seed*}, G, n, h). Methods for the generation of d and Q are provided in

Appendix B.4.

6.2.2 Key Pair Management

The secure use of digital signatures depends on the management of an entity's digital signature key pair as specified in Section 4.4.2.

6.3 Secret Number Generation

A new secret random number *k* **shall** be generated prior to the generation of each digital signature for use during the signature generation process. This secret number **shall** be protected from unauthorized disclosure and modification. Methods for the generation of the per-message secret number are provided in Appendix B.5.

k^{-1} is the multiplicative inverse of *k* with respect to multiplication modulo *n*; i.e., $0 < k^{-1} < n$ and $1 = (k^{-1}\ k) \bmod n$. This inverse is required for the signature generation process. A technique is provided in Appendix C.1 for deriving k^{-1} from *k*.

k and k^{-1} may be pre-computed, since knowledge of the message to be signed is not required for the computations. When *k* and k^{-1} are pre-computed, their confidentiality and integrity **shall** be protected.

6.4 ECDSA Digital Signature Generation and Verification

An ECDSA digital signature (*r*, *s*) **shall** be generated as specified in ANS X9.62, using:

1. Domain parameters that are generated in accordance with Section 6.1.1,
2. A private key that is generated as specified in Section 6.2.1,
3. A per-message secret number that is generated as specified in Section 6.3,
4. An approved hash function as discussed below, and
5. An approved random number generator as specified in SP 800-90.

An ECDSA digital signature **shall** be verified as specified in ANS X9.62, using the same domain parameters and hash function that were used during signature generation.

An approved hash function, as specified in FIPS 180-3, **shall** be used during the generation of digital signatures. The security strength associated with the ECDSA digital signature process is no greater than the minimum of the security strength associated with the bit length of *n* and the security strength of the hash function that is employed. Both the security strength of the hash function used and the security strength associated with the bit length of *n* **shall** meet or exceed the security strength required for the digital signature process. The security strengths for the ranges of the bit lengths of *n* and for each hash function is provided in SP 800-57.

It is recommended that the security strength associated with the bit length of *n* and the security strength of the hash function be the same unless an agreement has been made between participating entities to use a stronger hash function. When the length of the output of the hash function is greater than the bit length of *n*, then the leftmost *n* bits of the hash function output block **shall** be used in any calculation using the hash function output during the generation or verification of a digital signature. A hash function that provides a lower security strength than the security strength associated with the bit length of *n* ordinarily **should not** be used, since this would reduce the security strength of the digital signature process to a level no greater than that provided by the hash function.

6.5 Assurances

The intended signatory **shall** have assurances as specified in Section 3.1. Prior to accepting a signature as valid, the verifier **shall** have assurances as specified in Section 3.3.

APPENDIX A: Generation and Validation of FFC Domain Parameters

Finite field cryptography (FFC) is a method of implementing discrete logarithm cryptography using finite field mathematics. DSA, as specified in this Standard, is an example of FFC. The Diffie-Hellman and MQV key establishment algorithms specified in SP 800-56A can also be implemented as FFC.

The domain parameters for FFC consist of the set of values (*p*, *q*, *g* {, *domain_parameter_seed*, *counter*}). This appendix specifies techniques for the generation of the FFC domain parameters *p*, *q* and *g* and performing an explicit domain parameter validation. During the generation process, the values for *domain_parameter_seed* and *counter* are obtained.

A.1 Generation of the FFC Primes *p* and *q*

This section provides methods for generating the primes *p* and *q* that fulfill the criteria specified in Sections 4.1 and 4.2. One of these methods **shall** be used when generating these primes. A method is provided in Appendix A.1.1 to generate random candidate integers and then test them for primality using a probabilistic algorithm. A second method is provided in Appendix A.1.2 that constructs integers from smaller integers so that the constructed integer is guaranteed to be prime.

During the generation, validation and testing processes, conversions between bit strings and integers are required. Appendix C.2 provides methods for these conversions.

A.1.1 Generation and Validation of Probable Primes

Previous versions of this Standard contained a method for the generation of the domain parameters *p* and *q* using SHA-1 and probabilistic methods. This method is no longer approved for domain parameter generation; however, the validation process for this method is provided in Appendix A.1.1.1 to validate previously generated domain parameters.

A method for the generation and validation of the primes *p* and *q* using probabilistic methods is provided in Appendix A.1.1.2 and is based on the use of an approved hash function; this method **shall** be used for generating probable primes. The validation process for this method is provided in Appendix A.1.1.3.

The probabilistic methods use a hash function and an arbitrary seed (*domain_parameter_seed*). Arbitrary seeds could be anything, e.g., a user’s favorite number or a random or pseudorandom number output by a random number generator (see SP 800-90). The *domain_parameter_seed* determines a sequence of candidates for *p* and *q* in the required intervals that are then tested for primality using a probabilistic primality test (see Appendix C.3). The test determines that the candidate is either not a prime (i.e., it is a composite integer) or is “probably a prime” (i.e., there is a very small probability that a composite integer will be declared to be a prime). *p* and *q* **shall** be the first candidate set that passes the primality tests. Note that the *domain_parameter_seed*

shall be unique for every unique set of domain parameters that are generated using the same method.

A.1.1.1 Validation of the Probable Primes *p* and *q* that were Generated Using SHA-1 as Specified in Prior Versions of this Standard

This prime validation algorithm is used to validate that the primes *p* and *q* that were generated by the prime generation algorithm specified in previous versions of this Standard. The algorithm requires the values of *p*, *q*, *domain_parameter_seed* and *counter*, which were output from the prime generation algorithm.

Let **SHA1( )** be the SHA-1 hash function specified in FIPS 180-3. The following process or its equivalent **shall** be used to validate *p* and *q* for this method.

Input:

1. *p*, *q*	The generated primes *p* and *q*.
2. *domain_parameter_seed*	A seed that was used to generate *p* and *q*.
3. *counter*	A count value that was determined during generation.

Output:

1. *status*	The status returned from the validation procedure, where status is either **VALID** or **INVALID**.

Process:

```
1. If (len (p) ≠ 1024) or (len (q) ≠ 160), then return INVALID.
2. If (counter > 4095), then return INVALID.
3. seedlen = len (domain_parameter_seed).
4. If (seedlen < 160), then return INVALID.
5. computed_q = SHA1(domain_parameter_seed) ⊕ SHA1((domain_parameter_seed +
   1) mod $2^{seedlen}$).
6. Set the first and last bits of computed_q equal to 1 (i.e., the 159th and 0th bits).
7. Test whether or not computed_q is prime as specified in Appendix C.3. If
   (computed_q ≠ q) or (computed_q is not prime), then return INVALID.
8. offset = 2.
9. For i = 0 to counter do
   9.1   For j = 0 to 6 do
             $V_j$ = SHA1((domain_parameter_seed + offset + j) mod $2^{seedlen}$).
   9.2   $W = V_0 + (V_1 * 2^{160}) + (V_2 * 2^{320}) + (V_3 * 2^{480}) + (V_4 * 2^{640}) + (V_5 * 2^{800}) +$
```

$((V_6 \bmod 2^{63}) * 2^{960})$.

9.3 $X = W + 2^{1023}$. Comment: $0 \le W < 2^{L-1}$.

9.4 $c = X \bmod 2q$.

9.5 *computed_p* $= X - (c - 1)$. Comment: *computed_p* $\equiv 1 \pmod{2q}$.

9.6 If (*computed_p* $< 2^{1023}$), then go to step 9.8.

9.7 Test whether or not *computed_p* is prime as specified in Appendix C.3. If *computed_p* is determined to be prime, then go to step 10.

9.8 *offset* = *offset* + 7.

10. If (($i \ne$ *counter*) or (*computed_p* $\ne p$) or (*computed_p* is not prime)), then return **INVALID.**

11. Return **VALID**.

A.1.1.2 Generation of the Probable Primes *p* and *q* Using an Approved Hash Function

This method uses an approved hash function and may be used for the generation of the primes *p* and *q* for any application (e.g., digital signatures or key establishment). The security strength of the hash function **shall** be equal to or greater than the security strength associated with the (*L*, *N*) pair.

An arbitrary *domain_parameter_seed* of *seedlen* bits is also used, where *seedlen* **shall** be equal to or greater than *N*.

The generation process returns a set of integers *p* and *q* that have a very high probability of being prime. For another entity to validate that the primes were generated correctly using the validation process in Appendix A.1.1.3, the value of the *domain_parameter_seed* and the *counter* used to generate the primes must also be returned and made available to the validating entity; the *domain_parameter_seed* and *counter* need not be kept secret. Let **Hash( )** be the selected hash function, and let *outlen* be the bit length of the output block, where *outlen* **shall** be equal to or greater than *N*.

The following process or its equivalent **shall** be used to generate *p* and *q* for this method.

Input:

1. *L* The desired length of the prime *p* (in bits).
2. *N* The desired length of the prime *q* (in bits).
3. *seedlen* The desired length of the domain parameter seed; *seedlen* **shall** be equal to or greater than *N*.

Output:

1. *status* The status returned from the generation procedure, where status is

either **VALID** or **INVALID**. If **INVALID** is returned as the *status*, either no values for the other output parameters **shall** be returned, or invalid values **shall** be returned (e.g., zeros or Null strings).

2. *p*, *q* The generated primes *p* and *q*.

3. *domain_parameter_seed*

   (Optional) A seed that was used to generate *p* and *q*.

4. *counter* (Optional) A count value that was determined during generation.

Process:

```
1. Check that the $(L, N)$ pair is in the list of acceptable ($L$, $N$ pairs) (see Section 4.2). If
   the pair is not in the list, then return INVALID.
2. If ($seedlen < N$), then return INVALID.
3. $n = \lceil L / outlen \rceil - 1$.
4. $b = L - 1 - (n * outlen)$.
5. Get an arbitrary sequence of seedlen bits as the domain_parameter_seed.
6. $U = \textbf{Hash}\ (domain\_parameter\_seed) \bmod 2^{N-1}$.
7. $q = 2^{N-1} + U + 1 - (\ U \bmod 2)$.
8. Test whether or not $q$ is prime as specified in Appendix C.3.
9. If $q$ is not a prime, then go to step 5.
10. $offset = 1$.
11. For $counter = 0$ to $(4L - 1)$ do
    11.1 For $j = 0$ to $n$ do
         $V_j = \textbf{Hash}\ ((domain\_parameter\_seed + offset + j) \bmod 2^{seedlen})$.
    11.2 $W = V_0 + (V_1 * 2^{outlen}) + \ldots + (V_{n-1} * 2^{(n-1) * outlen}) + ((V_n \bmod 2^b) * 2^{n * outlen})$.
    11.3 $X = W + 2^{L-1}$.            Comment: $0 \le W < 2^{L-1}$; hence, $2^{L-1} \le X < 2^L$.
    11.4 $c = X \bmod 2q$.
    11.5 $p = X - (c - 1)$.            Comment: $p \equiv 1 \pmod{2q}$.
    11.6 If ($p < 2^{L-1}$), then go to step 11.9.
    11.7 Test whether or not $p$ is prime as specified in Appendix C.3.
    11.8 If $p$ is determined to be prime, then return VALID and the values of $p$, $q$ and
         (optionally) the values of domain_parameter_seed and counter.
```

11.9 *offset* = *offset* + *n* + 1. Comment: Increment *offset*; then, as part of the loop in step 11, increment *counter*; if *counter* < 4*L*, repeat steps 11.1 through 11.8.

12. Go to step 5.

A.1.1.3 Validation of the Probable Primes *p* and *q* that were Generated Using an Approved Hash Function

This prime validation algorithm is used to validate that the integers *p* and *q* were generated by the prime generation algorithm given in Appendix A.1.1.2. The validation algorithm requires the values of *p*, *q*, *domain_parameter_seed* and *counter*, which were output from the prime generation algorithm. Let **Hash( )** be the hash function used to generate *p* and *q*, and let *outlen* be its output block length.

The following process or its equivalent **shall** be used to validate *p* and *q* for this method.

Input:

1. *p*, *q* The generated primes *p* and *q*.
3. *domain_parameter_seed* The domain parameter seed that was used to generate *p* and *q*.
4. *counter* A count value that was determined during generation.

Output:

1. *status* The status returned from the validation procedure, where status is either **VALID** or **INVALID**.

Process:

1. L = **len** (p).
2. N = **len** (q).
3. Check that the (L, N) pair is in the list of acceptable (L, N) pairs (see Section 4.2). If the pair is not in the list, return **INVALID**.
4. If $(counter > (4L - 1))$, then return **INVALID**.
5. *seedlen* = **len** (*domain_parameter_seed*).
6. If $(seedlen < N)$, then return **INVALID.**
7. U = **Hash**(*domain_parameter_seed*) mod 2^{N-1}.
8. $computed\_q = 2^{N-1} + U + 1 - (U \bmod 2)$.
9. Test whether or not *computed_q* is prime as specified in Appendix C.3. If ($computed\_q \neq q$) or (*computed_q* is not prime), then return **INVALID.**

```
10. $n = \lceil L / outlen \rceil - 1$.
11. $b = L - 1 - (n * outlen)$.
12. $offset = 1$.
13. For $i = 0$ to $counter$ do
    13.1 For $j = 0$ to $n$ do
         $V_j = \mathbf{Hash}((domain\_parameter\_seed + offset + j) \bmod 2^{seedlen})$.
    13.2 $W = V_0 + (V_1 * 2^{outlen}) + \ldots + (V_{n-1} * 2^{(n-1) * outlen}) + ((V_n \bmod 2^b) * 2^{n * outlen})$.
    13.3 $X = W + 2^{L-1}$.
    13.4 $c = X \bmod 2q$.
    13.5 $computed\_p = X - (c - 1)$.
    13.6 If ($computed\_p < 2^{L-1}$), then go to step 13.9
    13.7 Test whether or not $computed\_p$ is prime as specified in Appendix C.3.
    13.8 If $computed\_p$ is determined to be a prime, then go to step 14.
    13.9 $offset = offset + n + 1$.
14. If (($i \neq counter$) or ($computed\_p \neq p$) or ($computed\_p$ is not a prime)), then return
    INVALID.
15. Return VALID.
```

A.1.2 Construction and Validation of the Provable Primes *p* and *q*

Primes can be generated so that they are guaranteed to be prime. The following algorithm for generating *p* and *q* uses an approved hash function and an arbitrary seed (*firstseed*) to construct *p* and *q* in the required intervals. The security strength of the hash function **shall** be equal to or greater than the security strength associated with the (*L*, *N*) pair.

Arbitrary seeds can be anything, e.g., a user's favorite number or a random or pseudorandom number that is output from a random number generator. Note that the *firstseed* must be unique to produce a unique set of domain parameters. Candidate primes are tested for primality using a deterministic primality test that proves whether or not the candidate is prime. The resulting *p* and *q* are guaranteed to be primes.

A.1.2.1 Construction of the Primes *p* and *q* Using the Shawe-Taylor Algorithm

For each set of domain parameters generated, an arbitrary initial seed (*firstseed*) of at least *seedlen* bits **shall** be determined, where *seedlen* **shall** be $\geq N$.

The generation process returns a set of integers *p* and *q* that are guaranteed to be prime. For another entity to validate that the primes were generated correctly (using the validation process in Appendix A.1.2.2), the value of the *firstseed*, the two computed seeds (*pseed* and *qseed*) and the counters used to generate the primes (*pgen_counter* and *qgen_counter*) must be made available to the validating entity; the seeds and the counters need not be kept secret. The domain parameters for DSA are identified in Section 4.3 as (p, q, g {, *domain_parameter_seed*, *counter*}). When using the Shawe-Taylor algorithm for generating p and q, the *domain_parameter_seed* consists of three seed values (*firstseed, pseed,* and *qseed*), and the *counter* consists of the pair of counter values (*pgen_counter* and *qgen_counter*).

Let **Hash( )** be the selected hash function (see Appendix A.1.2), and let *outlen* be the bit length of the output block of that hash function.

A.1.2.1.1 Get the First Seed

The following process or its equivalent **shall** be used to generate *firstseed* for this constructive method.

Input:

1. N The length of q in bits.
2. *seedlen* The length of firstseed, where $seedlen \geq N$.

Output:

1. *status* The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE.** If **FAILURE** is returned, then either no *firstseed* value **shall** be provided or an invalid value **shall** be returned.
2. *firstseed* The first seed generated.

Process:

1. *firstseed* = 0.
2. Check that N is in the list of acceptable (L, N) pairs (see Section 4.2). If not, then return **FAILURE**.
3. If ($seedlen < N$), then return **FAILURE**.
4. While $firstseed < 2^{N-1}$.

   Get an arbitrary sequence of *seedlen* bits as *firstseed.*
5. Return **SUCCESS** and the value of *firstseed*.

Note: This routine could be incorporated into the beginning of the constructive prime generation procedure in Appendix A.1.2.1.2. However, this was not done in this specification so that the validation process in Appendix A.1.2.2 could also call the constructive prime generation procedure and provide the value of *firstseed* as input.

A.1.2.1.2 Constructive Prime Generation

The following process or its equivalent **shall** be used to generate *p* and *q* for this constructive method.

Input:

1. *L* The requested length for *p* (in bits).
2. *N* The requested length for *q* (in bits).
3. *firstseed* The first seed to be used. This was obtained as specified in Appendix A.1.2.1.1.

Output:

1. *status* The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE.** If **FAILURE** is returned, then either no other values **shall** be returned, or invalid values **shall** be returned.
2. *p*, *q* The requested primes.
3. *pseed*, *qseed* (Optional) Computed seed values that were used to generate *p* and *q*. The entire *seed* for the generation of *p* and *q* consists of *firstseed*, *pseed* and *qseed*.
4. *pgen_counter*, *qgen_counter*

   (Optional) The count values that were determined during generation.

Process:

1. Check that the (*L*, *N*) pair is in the list of acceptable (*L*, *N*) pairs (see Section 4.2). If the pair is not in the list, return **FAILURE**.

   Comment: Use the Shawe-Taylor random prime routine in Appendix C.6 to generate random primes.

2. Using *N* as the length and *firstseed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain *q*, *qseed* and *qgen_counter*. If **FAILURE** is returned, then return **FAILURE**.
3. Using $\lceil L / 2 + 1 \rceil$ as the *length* and *qseed* as the *input_seed*, use the random prime routine in Appendix C.6 to obtain p_0*, pseed,* and *pgen_counter*. If **FAILURE** is returned, then return **FAILURE**.
4. $iterations = \lceil L / outlen \rceil - 1$.

```
5. $old\_counter = pgen\_counter$.
                                Comment: Generate a (pseudo) random $x$ in
                                the interval $[2^{L-1}, 2^{L}]$.
6. $x = 0$.
7. For $i = 0$ to $iterations$ do
   $x = x + (\textbf{Hash}(pseed + i) * 2^{i * outlen})$.
8. $pseed = pseed + iterations + 1$.
9. $x = 2^{L-1} + (x \bmod 2^{L-1})$.
                                Comment: Generate $p$, a candidate for the
                                prime, in the interval $[2^{L-1}, 2^{L}]$.
10. $t = \lceil x / (2qp_0) \rceil$.
11. If $(2tqp_0 + 1) > 2^L$, then $t = \lceil 2^{L-1} / (2qp_0) \rceil$.
12. $p = 2tqp_0 + 1$.
13. $pgen\_counter = pgen\_counter + 1$.
                                Comment: Test $p$ for primality; choose an
                                integer $a$ in the interval $[2, p–2]$.
14. $a = 0$
15. For $i = 0$ to $iterations$ do
    $a = a + (\textbf{Hash}(pseed + i) * 2^{i * outlen})$.
16. $pseed = pseed + iterations + 1$.
17. $a = 2 + (a \bmod (p–3))$.
18. $z = a^{2tq} \bmod p$.
19. If $((1 = \textbf{GCD}(z–1, p))$ and $(1 = z^{p_0} \bmod p))$, then return SUCCESS and the values
    of $p$, $q$ and (optionally) $pseed$, $qseed$, $pgen\_counter$, and $qgen\_counter$.
20. If $(pgen\_counter > (4L + old\_counter))$, then return FAILURE.
21. $t = t + 1$.
22. Go to step 11.
```

A.1.2.2 Validation of the DSA Primes *p* and *q* that were Constructed Using the Shawe-Taylor Algorithm

The validation of the primes p and q that were generated by the method described in Appendix A.1.2.1.2 may be performed if the values of *firstseed*, *pseed*, *qseed*, *pgen_counter* and *qgen_counter* were saved and are provided for use in the following algorithm.

The following process or its equivalent **shall** be used to validate *p* and *q* for this constructive method.

Input:

1. *p*, *q* — The primes to be validated.
2. *firstseed, pseed*, *qseed* — Seed values that were used to generate *p* and *q*.
3. *pgen_counter*, *qgen_counter*

   The count values that were determined during generation.

Output:

1. *status* — The status returned from the validation procedure, where *status* is either **SUCCESS** or **FAILURE.**

Process:

1. L = **len** (p).
2. N = **len** (q).
3. Check that the (L, N) pair is in the list of acceptable (L, N) pairs (see Section 4.2). If the pair is not in the list, then return **FAILURE.**
4. If ($firstseed < 2^{N-1}$), then return **FAILURE**.
5. If ($2^N \leq q$), then return **FAILURE**).
6. If ($2^L \leq p$), then return **FAILURE**.
7. If (($p - 1$) mod $q \neq 0$), then return **FAILURE**.
8. Using L, N and *firstseed*, perform the constructive prime generation procedure in Appendix A.1.2.1.2 to obtain *p_val*, *q_val*, *pseed_val*, *qseed_val*, *pgen_counter_val*, and *qgen_counter_val*. If **FAILURE** is returned, or if ($q\_val \neq q$) or ($qseed\_val \neq qseed$) or ($qgen\_counter\_val \neq qgen\_counter$) or ($p\_val \neq p$) or ($pseed\_val \neq pseed$) or ($pgen\_counter\_val \neq pgen\_counter$), then return **FAILURE.**
9. Return **SUCCESS.**

A.2 Generation of the Generator *g*

The generator g depends on the values of p and q. Two methods for determining the generator g are provided; one of these methods **shall** be used. The first method, discussed in Appendix A.2.1, may be used when complete validation of the generator g is not required; it is recommended that this method be used only when the party generating g is trusted to not deliberately generate a g that has a potentially exploitable relationship to another generator g'. For example, it must be hard to determine an exponential relationship between the generators such that $g = (g')^x \bmod p$ for a known value of x. (Note: Read $(g')^x$ as g prime to the x.)

Appendix A.2.2 provides a method for partial validation when the method of generation in Appendix A.2.1 is used. The second method for generating g, discussed in Appendix A.2.3, **shall** be used when validation of the generator g is required; the method for the validation of a generator determined using the method in Appendix A.2.3 is provided in Appendix A.2.4.

A.2.1 Unverifiable Generation of the Generator *g*

This method is used to determine a value for g, based on the values of p and q. It may be used when validation of the generator g is not required. The correct generation of g cannot be completely validated (see Appendix A.2.2). Note that this generation method for g was also specified in previous versions of this Standard.

The following process or its equivalent **shall** be used to generate the generator g for this method.

Input:

1. p, q The generated primes.

Output:

1. g The requested value of g.

Process:

1. $e = (p - 1)/q$.
2. Set h = any integer satisfying $1 < h < (p - 1)$, such that h differs from any value previously tried. Note that h could be obtained from a random number generator or from a counter that changes after each use.
3. $g = h^e \bmod p$.
4. If ($g = 1$), then go to step 2.
5. Return g.

A.2.2 Assurance of the Validity of the Generator *g*

The order of the generator g that was generated using Appendix A.2.1 can be partially validated by checking the range and order, thereby performing a partial validation of *g*.

The following process or its equivalent **shall** be used when partial validation of the generator *g* is required:

Input:

1. *p*, *q, g* The domain parameters.

Output:

1. *status* The status returned from the generation routine, where *status* is either **PARTIALLY VALID** or **INVALID.**

Process:

1. Verify that $2 \le g \le (p–1)$. If not true, return **INVALID.**
2. If ($g^q = 1 \bmod p$), then return **PARTIALLY VALID.**
3. Return **INVALID.**

The non-existence of a potentially exploitable relationship of *g* to another generator *g*′ (that is known to the entity that generated *g*, but may not be known by other entities) cannot be checked. In this sense, the correct generation of *g* cannot be completely validated.

A.2.3 Verifiable Canonical Generation of the Generator *g*

The generation of *g* is based on the values of *p*, *q* and *domain_parameter_seed* (which are outputs of the generation processes in Appendix A.1). When *p* and *q* were generated using the method in Appendix A.1.1.2, the *domain_parameter_seed* value must have been returned from the generation routine. When *p* and *q* were generated using the method in Appendix A.1.2.1, the *firstseed*, *pseed*, and *qseed* values must have been returned from the generation routine; in this case, *domain_parameter_seed* = *firstseed* || *pseed* || *qseed* **shall** be used in the following process.

This method of generating a generator *g* can be validated (see Appendix A.2.4).

This generation method supports the generation of multiple values of *g* for specific values of *p* and *q*. The use of different values of *g* for the same *p* and *q* may be used to support key separation; for example, using the *g* that is generated with *index* = 1 for digital signatures and with *index* = 2 for key establishment.

Let **Hash( )** be the hash function used to generate *p* and *q* (see Appendix A.1). The following process or its equivalent **shall** be used to generate the generator *g*.

Input:

1. *p*, *q* The primes.
2. *domain_parameter_seed* The seed used during the generation of *p* and *q*.

3. *index* The index to be used for generating *g*. *index* is a bit string of length 8 that represents an unsigned integer.

Output:

1. *status* The status returned from the generation routine, where *status* is either **VALID** or **INVALID.**

2. *g* The value of *g* that was generated.

Process:

```
                                  Note: count is an unsigned 16-bit integer.

                                  Comment: Check that a valid value of the index has
                                  been provided (see above).
1. If (index is incorrect), then return INVALID.
2. $N = \mathbf{len}(q)$.
3. $e = (p - 1)/q$.
4. $count = 0$.
5. $count = count + 1$.
                                  Comment: Check that count does not wrap around
                                  to 0.
6. If ($count = 0$), then return INVALID.
                                  Comment: the length of the
                                  domain_parameter_seed has already been checked.
                                  "ggen" is the bit string 0x6767656E.
7. $U = domain\_parameter\_seed$ || "ggen" || $index$ || $count$.
8. $W = \mathbf{Hash}(U)$.
9. $g = W^e \bmod p$.
10. If ($g < 2$), then go to step 5.  Comment: If a generator has not been found.
11. Return VALID and the value of g.
```

A.2.4 Validation Routine when the Canonical Generation of the Generator *g* Routine Was Used

This algorithm **shall** be used to validate the value of *g* that was generated using the process in Appendix A.2.3, based on the values of *p*, *q, domain_parameter_seed,* and the appropriate value of *index*. It is assumed that the values of *p* and *q* have been previously validated according to Appendix A.1. Note that the method specified in Appendix A.2.3 for the generation of *g* was not included in previous versions of this Standard; therefore, this validation method is not appropriate for that case.

The *domain_parameter_seed* is an output from the generation of *p* and *q*. When *p* and *q* were generated using the method in Appendix A.1.1.2, the *domain_parameter_seed* must have been returned from the generation routine and made available to the validating party. When *p* and *q* were generated using the method in Appendix A.1.2.1, the *firstseed*, *pseed*, and *qseed* values must have been returned from the generation routine and made available; *firstseed*, *pseed*, and *qseed* **shall** be concatenated to form the *domain_parameter_seed* used in the following process. Let **Hash( )** be the hash function used to generate *g* (i.e., the hash function also used to generate *p* and *q*).

The input *index* is the index number for the generator *g.* See Appendix A.2.3 for more details.

The following process or its equivalent **shall** be used to validate the generator *g* for this method.

Input:

1. *p*, *q* The primes.
2. *domain_parameter_seed* The seed used to generate *p* and q.
3. *index* The index used in Appendix A.2.3 to generate *x*. *index* is a bit string of length 8 that represents an unsigned integer.
4. *g* The value of *g* to be validated.

Output:

1. *status* The status returned from the generation routine, where *status* is either **VALID** or **INVALID.**

Process: Note**:** *count* is an unsigned 16-bit integer.

Comment: Check that a valid value of the *index* has been provided (see above).

1. If (*index* is incorrect), then return **INVALID.**
2. Verify that $2 \le g \le (p–1)$. If not true, return **INVALID**.
3. If ($g^q \ne 1 \bmod p$), then return **INVALID**.
4. N = **len**(q).
5. $e = (p - 1)/q$.
6. *count* = 0.
7. *count* = *count* + 1.

Comment: Check that *count* does not wrap around to 0.

```
8.  If ($count = 0$), then return INVALID.
                                                  Comment: “ggen” is the bit string 0x6767656E.
9.  $U = domain\_parameter\_seed$ || “ggen” || $index$ || $count$.
10. $W$ = Hash($U$).
11. $computed\_g = W^e \bmod p$.
12. If ($computed\_g < 2$), then go to step 7.    Comment: If a generator has not been found.
13. If ($computed\_g = g$), then return VALID, else return INVALID.
```

APPENDIX B: Key Pair Generation

Discrete logarithm cryptography (DLC) is divided into finite field cryptography (FFC) and elliptic curve cryptography (ECC); the difference between the two is the type of math that is used. DSA is an example of FFC; ECDSA is an example of ECC. Other examples of DLC are the Diffie-Hellman and MQV key agreement algorithms, which have both FFC and ECC forms.

The most common example of integer factorization cryptography (IFC) is RSA.

This appendix specifies methods for the generation of FFC and ECC key pairs and secret numbers, and the generation of IFC key pairs. All generation methods require the use of an approved, properly instantiated random bit generator (RBG) as specified in SP 800-90; the RBG **shall** have a security strength equal to or greater than the security strength associated with the key pairs and secret numbers to be generated. See SP 800-57 for guidance on security strengths and key sizes.

This appendix does not indicate the required conversions between bit strings and integers. When required by a process in this appendix, the conversion **shall** be accomplished as specified in Appendix C.2.

B.1 FFC Key Pair Generation

An FFC key pair (x, y) is generated for a set of domain parameters (p, q, g {, *domain_parameter_seed*, *counter*}). Two methods are provided for the generation of the FFC private key x and public key y; one of these two methods **shall** be used. Prior to generating DSA key pairs, assurance of the validity of the domain parameters (p, q and g) **shall** have been obtained as specified in Section 3.1.

For DSA, the valid values of L and N are provided in Section 4.2.

B.1.1 Key Pair Generation Using Extra Random Bits

In this method, 64 more bits are requested from the RBG than are needed for x so that bias produced by the mod function in step 6 is negligible.

The following process or its equivalent may be used to generate an FFC key pair.

Input:

(p, q, g) The subset of the domain parameters that are used for this process. p, q and g **shall** either be provided as integers during input, or **shall** be converted to integers prior to use.

Output:

1. *status* The status returned from the key pair generation process. The status will indicate **SUCCESS** or an **ERROR**.
2. (x, y) The generated private and public keys. If an error is encountered during the generation process, invalid values for x and y **should** be returned, as represented by *Invalid_x* and *Invalid_y* in the following specification. x and y are returned as integers. The generated private key x is in the range $[1, q–1]$, and the public key is in the range $[1, p–1]$.

Process:

```
1. $N = \mathbf{len}(q)$; $L = \mathbf{len}(p)$.
                                  Comment: Check that the $(L, N)$ pair is specified in
                                  Section 4.2.
2. If the $(L, N)$ pair is invalid, then return an ERROR indicator, Invalid_x, and
   Invalid_y.
3. requested_security_strength = the security strength associated with the $(L, N)$ pair;
   see SP 800-57.
4. Obtain a string of $N$+64 returned_bits from an RBG with a security strength of
   requested_security_strength or more. If an ERROR indication is returned, then
   return an ERROR indication, Invalid_x, and Invalid_y.
5. Convert returned_bits to the (non-negative) integer $c$ (see Appendix C.2.1).
6. $x = (c \bmod (q–1)) + 1$.    Comment: $0 \le c \bmod (q–1) \le q–2$ and implies that
                                  $1 \le x \le q–1$.
7. $y = g^x \bmod p$.
8. Return SUCCESS, $x$, and $y$.
```

B.1.2 Key Pair Generation by Testing Candidates

In this method, a random number is obtained and tested to determine that it will produce a value of x in the correct range. If x is out-of-range, another random number is obtained (i.e., the process is iterated until an acceptable value of x is obtained.

The following process or its equivalent may be used to generate an FFC key pair.

Input:

(p, q, g) The subset of the domain parameters that are used for this process. p, q and g **shall** either be provided as integers during input, or **shall** be converted to integers prior to use.

Output:

1. *status* The status returned from the key pair generation process. The status will indicate **SUCCESS** or an **ERROR**.
2. (x, y) The generated private and public keys. If an error is encountered during the generation process, invalid values for x and y **should** be returned, as represented by *Invalid_x* and *Invalid_y* in the following specification. x and y are returned as integers. The generated private key x is in the range $[1, q–1]$, and the public key is in the range $[1, p–1]$.

Process:

1. N = **len**(q); L = **len**(p).

   Comment: Check that the (L, N) pair is specified in Section 4.2.

2. If the (L, N) pair is invalid, then return an **ERROR** indication, *Invalid_x*, and *Invalid_y*.
3. *requested_security_strength* = the security strength associated with the (L, N) pair; see SP 800-57.
4. Obtain a string of N *returned_bits* from an **RBG** with a security strength of *requested_security_strength* or more. If an **ERROR** indication is returned, then return an **ERROR** indication, *Invalid_x*, and *Invalid_y*.
5. Convert *returned_bits* to the (non-negative) integer c (see Appendix C.2.1).
6. If ($c > q–2$), then go to step 4.
7. $x = c + 1$.
8. $y = g^x \bmod p$.
9. Return **SUCCESS**, x, and y.

B.2 FFC Per-Message Secret Number Generation

DSA requires the generation of a new random number k for each message to be signed. Two methods are provided for the generation of k; one of these two methods **shall** be used.

The valid values of N are provided in Section 4.2. Let **inverse**(k, q) be a function that computes the inverse of a (non-negative) integer k with respect to multiplication modulo the prime number q. A technique for computing the inverse is provided in Appendix C.1.

B.2.1 Per-Message Secret Number Generation Using Extra Random Bits

In this method, 64 more bits are requested from the RBG than are needed for k so that bias produced by the mod function in step 6 is not readily apparent.

The following process or its equivalent may be used to generate a per-message secret number.

Input:

(p, q, g) DSA domain parameters that are generated as specified in Section 4.3.1.

Output:

1. *status* The status returned from the secret number generation process. The status will indicate **SUCCESS** or an **ERROR**.
2. (k, k^{-1}) The per-message secret number k and its mod q inverse, k^{-1}. If an error is encountered during the generation process, invalid values for k and k^{-1} **should** be returned, as represented by *Invalid_k* and *Invalid_k_inverse* in the following specification. k and k^{-1} are in the range [1, q–1].

Process:

1. N = **len**(q); L = **len**(p).

   Comment: Check that the (L, N) pair is specified in Section 4.2.

2. If the (L, N) pair is invalid, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.
3. *requested_security_strength* = the security strength associated with the (L, N) pair; see SP 800-57.
4. Obtain a string of N+64 *returned_bits* from an **RBG** with a security strength of *requested_security_strength* or more. If an **ERROR** indication is returned, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.
5. Convert *returned_bits* to the (non-negative) integer c (see Appendix C.2.1).
6. $k = (c \bmod (q–1)) + 1$.
7. (*status*, k^{-1}) = **inverse** (k, q).
8. Return *status*, k, and k^{-1}.

B.2.2 Per-Message Secret Number Generation by Testing Candidates

In this method, a random number is obtained and tested to determine that it will produce a value of k in the correct range. If k is out-of-range, another random number is obtained (i.e., the process is iterated until an acceptable value of k is obtained.

The following process or its equivalent may be used to generate a per-message secret number.

Input:

(p, q, g)	DSA domain parameters that are generated as specified in Section 4.3.1.

Output:

1. *status* The status returned from the secret number generation process. The status will indicate **SUCCESS** or an **ERROR**.

2. (k, k^{-1}) The per-message secret number k and its inverse, k^{-1}. If an error is encountered during the generation process, invalid values for k and k^{-1} **should** be returned, as represented by *Invalid_k* and *Invalid_k_inverse* in the following specification. k and k^{-1} are in the range $[1, q–1]$.

Process:

```
1. $N$ = len($q$); $L$ = len($p$).
                         Comment: Check that the ($L$, $N$) pair is specified in
                         Section 4.2).
2. If the ($L$, $N$) pair is invalid, then return an ERROR indication, Invalid_k, and
   Invalid_k_inverse.
3. requested_security_strength = the security strength associated with the ($L$, $N$) pair;
   see SP 800-57.
4. Obtain a string of $N$ returned_bits from an RBG with a security strength of
   requested_security_strength or more. If an ERROR indication is returned, then
   return an ERROR indication, Invalid_k, and Invalid_k_inverse.
5. Convert returned_bits to the (non-negative) integer $c$ (see Appendix C.2.1).
6. If ($c > q–2$), then go to step 4.
7. $k = c + 1$.
8. ($status, k^{-1}$) = inverse($k$, $q$).
9. Return status, $k$, and $k^{-1}$.
```

B.3 IFC Key Pair Generation

B.3.1 Criteria for IFC Key Pairs

Key pairs for IFC consist of a public key (n, e), and a private key (n, d), where n is the modulus and is the product of two prime numbers p and q. The security of IFC depends on the quality and secrecy of these primes and the private exponent d. The primes p and q **shall** be generated using one of the following methods:

A. Both p and q are randomly generated prime numbers (Random Primes), where p and q **shall** both be either :

   1. Provable primes (see Appendix B.3.2), or
   2. Probable primes (see Appendix B.3.3).

   Using these methods, primes of 2048 or 3072 bits may be generated; primes of 1024 bits **shall not** be generated using these methods. Primes of 1024 bits **shall** be generated using conditions based on auxiliary primes (see Appendices B.3.4, B.3.5, or B.3.6).

B. Both p and q are randomly generated prime numbers that satisfy the following additional conditions (Primes with Conditions):

   - $(p–1)$ has a prime factor p_1
   - $(p+1)$ has a prime factor p_2
   - $(q–1)$ has a prime factor q_1
   - $(q+1)$ has a prime factor q_2

   where p_1, p_2, q_1 and q_2 are called auxiliary primes of p and q.

   Using this method, one of the following cases **shall** apply:

   1. The primes p_1, p_2, q_1, q_2, p and q **shall** all be provable primes (see Appendix B.3.4),
   2. The primes p_1, p_2, q_1 and q_2 **shall** be provable primes, and the primes p and q **shall** be probable primes (see Appendix B.3.5), or
   3 The primes p_1, p_2, q_1, q_2, p and q **shall** all be probable primes (see Appendix B.3.6).

   The minimum lengths for each of the auxiliary primes p_1, p_2, q_1 and q_2 are dependent on *nlen*, where *nlen* is the length of the modulus n in bits. Note that *nlen* is also called the key size. The lengths of the auxiliary primes may be fixed or randomly chosen, subject to the restrictions in Table B.1. The maximum length is determined by *nlen* (the sum of the length of each auxiliary prime pair) and whether the primes p and q are probable primes or provable primes (e.g., for the auxiliary prime pair p_1 and p_2, **len**(p_1) + **len**(p_2) **shall** be less than a value determined by *nlen*, whether p_1 and p_2 are generated to be probable or provable primes)[3].

[3] For the probable primes p and q: **len**(p_1) + **len**(p_2) < **len**(p) – $\log_2$(**len**(p)) – 6; similarly for **len**(q_1) + **len**(q_2) and **len**(q). For the provable primes p and q: **len**(p_1) + **len**(p_2) < **len**(p)/2 – $\log_2$(**len**(p)) – 7; similarly for **len**(q_1) + **len**(q_2) and **len**(q). In each case, **len**(p) = **len**(q) = *nlen*/2.

Table B.1. Minimum and maximum lengths of p_1, p_2, q_1 and q_2

nlen	**Min. length of auxiliary primes p_1, p_2, q_1 and q_2**	**Max. length of len(p_1) + len(p_2) and len(q_1) + len(q_2)**	
		p, q **Probable primes**	***p, q*** **Provable primes**
1024	> 100 bits	< 496 bits	< 239 bits
2048	> 140 bits	< 1007 bits	< 494 bits
3072	> 170 bits	< 1518 bits	< 750 bits

For different values of *nlen* (i.e., different key sizes), the methods allowed for the generation of *p* and *q* are specified in Table B.2.

Table B.2. Allowable Prime Generation Methods

nlen	**Random Primes**	**Primes with Conditions**
1024	No	Yes
2048	Yes	Yes
3072	Yes	Yes

In addition, all IFC keys **shall** meet the following criteria in order to conform to FIPS 186-3:

1. The public exponent *e* **shall** be selected with the following constraints:

   (a) The public verification exponent *e* **shall** be selected prior to generating the primes *p* and *q*, and the private signature exponent *d*.

   (b) The exponent *e* **shall** be an odd positive integer such that:

   $$2^{16} < e < 2^{256}.$$

   Note that the value of *e* may be any value that meets constraint 1(b), i.e., *e* may be either a fixed value or a random value.

2. The primes *p* and *q* **shall** be selected with the following constraints:

   (a) (p–1) and (q–1) **shall** be relatively prime to the public exponent *e*.

   (b) The private prime factor *p* **shall** be selected randomly and **shall** satisfy $(\sqrt{2})(2^{(nlen/2)-1}) \leq p \leq (2^{nlen/2} - 1)$, where *nlen* is the appropriate length for the desired *security_strength*.

   (c) The private prime factor *q* **shall** be selected randomly and **shall** satisfy

$(\sqrt{2})(2^{(nlen/2)-1}) \le q \le (2^{nlen/2} - 1)$, where *nlen* is the appropriate length for the desired *security_strength*.

(d) $|p - q| > 2^{(nlen/2)-100}$.

3. The private signature exponent *d* **shall** be selected with the following constraints after the generation of *p* and *q*:

   (a) The exponent *d* **shall** be a positive integer value such that $2^{nlen/2} < d < \mathrm{LCM}(p–1, q–1)$, and

   (b) $d = e^{-1} \bmod (\mathrm{LCM}(p–1, q–1))$.

   That is, the inequality in (a) holds, and $1 \equiv (ed) \pmod{\mathrm{LCM}(p–1, q–1)}$.

   In the extremely rare event that $d \le 2^{nlen/2}$, then new values for *p*, *q* and *d* **shall** be determined. A different value of *e* may be used, although this is not required.

Any hash function used during the generation of the key pair **shall** be approved (i.e., specified in FIPS 180-3).

B.3.2 Generation of Random Primes that are Provably Prime

An approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC random primes *p* and *q* that are provably prime (see case A.1). One such method is provided in Appendix B.3.2.1 and B.3.2.2. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus *n*. After the seed is obtained, the primes can be generated (see Appendix B.3.2.2).

B.3.2.1 Get the Seed

The following process or its equivalent **shall** be used to generate the seed for this method.

Input:

nlen The intended bit length of the modulus *n*.

Output:

status The status to be returned, where *status* is either **SUCCESS** or **FAILURE**.

seed The seed. If *status* = **FAILURE**, a value of zero is returned as the *seed*.

Process:

1. If *nlen* is not valid (see Section 5.1), then Return (**FAILURE**, 0).
2. Let *security_strength* be the security strength associated with *nlen*, as specified in SP 800-57, Part 1.
3. Obtain a string *seed* of (2 * *security_strength*) bits from an **RBG** that supports the

security_strength.

4. Return (**SUCCESS**, *seed*).

B.3.2.2 Construction of the Provable Primes *p* and *q*

The following process or its equivalent **shall** be used to construct the random primes *p* and *q* (to be used as factors of the RSA modulus *n*) that are provably prime:

Input:

nlen	The intended bit length of the modulus *n*.
e	The public verification exponent.
seed	The seed obtained using the method in Appendix B.3.2.1.

Output:

status	The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**. When **FAILURE** is returned, zero values **shall** be returned as the other parameters.
p and *q*	The private prime factors of *n*.

Process:

```
1. If $nlen$ is neither 2048 nor 3072, then return (FAILURE, 0, 0).
2. If (($e \le 2^{16}$) OR ($e \ge 2^{256}$) OR ($e$ is not odd)), then return (FAILURE, 0, 0).
3. Set the value of security_strength in accordance with the value of nlen, as specified in SP 800-57, Part 1.
4. If (len(seed) $\ne$ 2 * security_strength), then return (FAILURE, 0, 0).
5. working_seed = seed.
6. Generate $p$:
    6.1 Using $L = nlen/2$, $N_1 = 1$, $N_2 = 1$, first_seed = working_seed and $e$, use the provable prime construction method in Appendix C.10 to obtain $p$ and pseed. If FAILURE is returned, then return (FAILURE, 0, 0).
    6.2 working_seed = pseed.
7. Generate $q$:
    7.1 Using $L = nlen/2$, $N_1 = 1$, $N_2 = 1$, first_seed = working_seed and $e$, use the provable prime construction method in Appendix C.10 to obtain $q$ and qseed. If FAILURE is returned, then return (FAILURE, 0, 0).
    7.2 working_seed = qseed.
```

```
8. If ( $|p - q|$ $\leq 2^{nlen/2 - 100}$), then go to step 7.
9. Zeroize the internally generated seeds:
   9.1 $pseed = 0$;
   9.2 $qseed = 0$;
   9.3 $working\_seed = 0$.
10. Return (SUCCESS, $p$, $q$).
```

B.3.3 Generation of Random Primes that are Probably Prime

An approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC random primes *p* and *q* that are probably prime (see case A.2).

The following process or its equivalent **shall** be used to construct the random probable primes *p* and *q* (to be used as factors of the RSA modulus *n*):

Input:

nlen The intended bit length of the modulus *n*.

e The public verification exponent.

Output:

status The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**.

p and *q* The private prime factors of *n*. When **FAILURE** is returned, zero values **shall** be returned as *p* and *q*.

Process:

```
1. If $nlen$ is neither 2048 nor 3072, return (FAILURE, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$) OR ($e$ is not odd)), then return (FAILURE, 0, 0).
3. Set the value of $security\_strength$ in accordance with the value of $nlen$, as specified in SP 800-57, Part 1.
4. Generate $p$:
   4.1 $i = 0$.
   4.2 Obtain a string $p$ of ($nlen/2$) bits from an RBG that supports the $security\_strength$.
   4.3 If ($p$ is not odd), then $p = p + 1$.
   4.4 If (($p < (\sqrt{2}\,)(2^{(nlen\,/\,2) - 1}$)), then go to step 4.2.
   4.5 If (GCD($p-1$, $e$) = 1), then
```

```
            4.5.1 Test $p$ for primality as specified in Appendix C.3, using an appropriate
                  value from Table C-2 or C-3 in Appendix C.3 as the number of iterations.
            4.5.2 If $p$ is **PROBABLY PRIME**, then go to step 5.
      4.6   $i= i + 1$.
      4.7   If ($i \geq 5(nlen/2)$), then return (**FAILURE**, 0, 0)
            Else go to step 4.2.
   5. Generate $q$:
      5.1   $i = 0$.
      5.2   Obtain a string $q$ of ($nlen/2$) bits from an **RBG** that supports the $security\_strength$
      5.3   If ($q$ is not odd), then $q = q + 1$.
      5.4   If ($|p - q| \leq 2^{nlen/2 - 100}$), then go to step 5.2.
      5.5   If (($q < (\sqrt{2})(2^{(nlen / 2) - 1})$)), then go to step 5.2.
      5.6   If (**GCD**($q-1, e$) = 1) then
            5.6.1 Test $q$ for primality as specified in Appendix C.3, using an appropriate
                  value from Table C-2 or C-3 in Appendix C.3 as the number of iterations.
            5.6.2 If $q$ is **PROBABLY PRIME**, then return (**SUCCESS**, $p$, $q$).
      5.7   $i = i + 1$.
      5.8   If ($i \geq 5(nlen/2)$), then return (**FAILURE**, 0, 0)
            Else go to step 5.2.
```

B.3.4 Generation of Provable Primes with Conditions Based on Auxiliary Provable Primes

This section specifies an approved method for the generation of the IFC primes p and q with the additional conditions specified in Appendix B.3.1, case B.1, where p, p_1, p_2, q, q_1 and q_2 are all provable primes. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus n. After the first seed is obtained, the primes can be generated.

Let $bitlen_1$, $bitlen_2$, $bitlen_3$, and $bitlen_4$ be the bit lengths for p_1, p_2, q_1 and q_2, respectively, in accordance with Table B.1. The following process or its equivalent **shall** be used to generate the provable primes:

Input:

nlen	The intended bit length of the modulus n.

e The public verification exponent.

seed The seed obtained using the method in Appendix B.3.2.1.

Output:

status The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**. If **FAILURE** is returned then zeros **shall** be returned as the values for *p* and *q*.

p and *q* The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$) OR (*e* is not odd)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. If (**len**(*seed*) ≠ 2 * *security_strength*), then return (**FAILURE**, 0, 0).
5. *working_seed* = *seed*.
6. Generate *p*:

   6.1 Using $L = nlen/2$, $N_1 = bitlen_1$, $N_2 = bitlen_2$, *firstseed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *p*, p_1, p_2 and *pseed*. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

   6.2 *working_seed* = *pseed.*
7. Generate *q*:

   7.1 Using $L = nlen/2$, $N_1 = bitlen_3$, $N_2 = bitlen_4$ and *firstseed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *q*, q_1, q_2 and *qseed*. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

   7.2 *working_seed* = *qseed*.
8. If ($|p - q| \leq 2^{nlen/2 - 100}$), then go to step 7.
9. Zeroize the internally generated seeds:

   9.1 *pseed* = 0.

   9.2 *qseed* = 0.

   9.3 *working_seed* = 0.
10. Return (**SUCCESS**, *p*, *q*).

B.3.5 Generation of Probable Primes with Conditions Based on Auxiliary Provable Primes

This section specifies an approved method for the generation of the IFC primes *p* and *q* with the additional conditions specified in Appendix B.3.1, case B.2, where p_1, p_2, q_1 and q_2 are provably prime, and *p* and *q* are probably prime. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus *n*. After the first seed is obtained, the primes can be generated.

Let $bitlen_1$, $bitlen_2$, $bitlen_3$, and $bitlen_4$ be the bit lengths for p_1, p_2, q_1 and q_2, respectively in accordance with Table B.1. The following process or its equivalent **shall** be used to construct *p* and *q*.

Input:

nlen The intended bit length of the modulus *n*.

e The public verification exponent.

seed The seed obtained using the method in Appendix B.3.2.1.

Output:

status The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**. If **FAILURE** is returned then zeros **shall** be returned as the values for *p* and *q*.

p and *q* The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$) OR (*e* is not odd)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. If (**len**(*seed*) ≠ 2 * *security_strength*), then return (**FAILURE**, 0, 0).

Comment: Generate four primes p_1, p_2, q_1 and q_2 that are provably prime.

5. Generate *p*:

   5.1 Using $bitlen_1$ as the length, and *seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain p_1 and *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

   5.2 Using $bitlen_2$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain p_2 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

5.3 Generate a prime p using the routine in Appendix C.9 with inputs of p_1, p_2, *nlen,* *e* and *security_strength*, also obtaining X_p. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

6. Generate q:

6.1. Using $bitlen_3$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain q_1 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

6.2 Using $bitlen_4$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain q_2 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

6.3 Generate a prime q using the routine in Appendix C.9 with inputs of q_1, q_2, *nlen,* *e* and *security_strength*, also obtaining X_q. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

7. If $((|p - q| \leq 2^{nlen/2 - 100})$ OR $(|X_p - X_q| \leq 2^{nlen/2 - 100}))$, then go to step 6.

8. Zeroize the internally generated that are not returned:

8.1 $X_p = 0$.

8.2 $X_q = 0$.

8.3 *prime_seed* = 0.

8.4 $p_1 = 0$.

8.5 $p_2 = 0$.

8.6 $q_1 = 0$.

8.7 $q_2 = 0$.

9. Return (**SUCCESS**, p, q).

B.3.6 Generation of Probable Primes with Conditions Based on Auxiliary Probable Primes

An approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC primes p and q that are probably prime and meet the additional constraints of Appendix B.3.1 (see case B.3). For this case, the prime factors p_1, p_2, q_1 and q_2 are also probably prime.

Four random numbers X_{p1}, X_{p2}, X_{q1} and X_{q2} are generated, from which the prime factors p_1, p_2, q_1 and q_2 are determined. p_1 and p_2, and an additional random number X_p are then used to determine p, and q_1 and q_2 and a random number X_q are used to obtain q. Let $bitlen_1$, $bitlen_2$, $bitlen_3$, and $bitlen_4$ be the bit lengths for p_1, p_2, q_1 and q_2, respectively chosen in accordance with Table B.1.

The following process or its equivalent **shall** be used to generate *p* and *q*:

Input:

nlen The intended bit length of the modulus *n*.

e The public verification exponent.

Output:

status The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**. If **FAILURE** is returned then zeros **shall** be returned as the values for *p* and *q*.

p and *q* The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$) OR (*e* is not odd)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. Generate *p*:

   4.1 Generate an odd integer X_{p1} of length $bitlen_1$ bits, and a second odd integer X_{p2} of length $bitlen_2$ bits, using an approved random number generator that supports the *security_strength*.

   4.2 Sequentially search successive odd integers, starting at X_{p1} until the first probable prime p_1 is found. Candidate integers **shall** be tested for primality as specified in Appendix C.3. Repeat the process to find p_2, starting at X_{p2}. The probable primes p_1 and p_2 **shall** be the first integers that pass the primality test.

   4.3 Generate a prime *p* using the routine in Appendix C.9 with inputs of p_1, p_2, *nlen*, *e* and *security_ strength*, also obtaining X_p. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

5. Generate *q*:

   5.1 Generate an odd integer X_{q1} of length $bitlen_3$ bits, and a second odd integer X_{q2} of length $bitlen_4$ bits, using an approved random number generator that supports the *security_strength*.

   5.2 Sequentially search successive odd integers, starting at X_{q1} until the first probable prime q_1 is found. Candidate integers **shall** be tested for primality as specified in Appendix C.3. Repeat the process to find q_2, starting at X_{q2}. The probable primes q_1 and q_2 **shall** be the first integers that pass the primality test.

   5.3 Generate a prime *q* using the routine in Appendix C.9 with inputs of q_1, q_2, *nlen,*

e and *security_ strength*, also obtaining X_q. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

```
6.  If (($|X_p - X_q| \le 2^{nlen/2 - 100}$) OR ($|p - q| \le 2^{nlen/2 - 100}$)), then go to step 5.
7.  Zeroize the internally generated values that are not returned:
    7.1   $X_p = 0$.
    7.2   $X_q = 0$.
    7.3   $X_{p1} = 0$.
    7.4   $X_{p2} = 0$.
    7.5   $X_{q1} = 0$.
    7.6   $X_{q2} = 0$.
    7.7   $p_1 = 0$.
    7.8   $p_2 = 0$.
    7.9   $q_1 = 0$.
    7.10  $q_2 = 0$.
8.  Return (SUCCESS, $p$, $q$).
```

B.4 ECC Key Pair Generation

An ECC key pair d and Q is generated for a set of domain parameters (q, FR, a, b {, *domain_parameter_seed*}, G, n, h). Two methods are provided for the generation of the ECC private key d and public key Q; one of these two methods **shall** be used to generate d and Q. Prior to generating ECDSA key pairs, assurance of the validity of the domain parameters (q, FR, a, b {, *domain_parameter_seed*}, G, n, h) **shall** have been obtained as specified in Section 3.1.

For ECDSA, the valid bit-lengths of n are provided in Section 6.1.1. See ANS X9.62 for definitions of the elliptic curve math and the conversion routines.

B.4.1 Key Pair Generation Using Extra Random Bits

In this method, 64 more bits are requested from the RBG than are needed for d so that bias produced by the mod function in step 6 is negligible.

The following process or its equivalent may be used to generate an ECC key pair.

Input:

1. (q, FR, a, b {, *domain_parameter_seed*}, G, n, h)

    The domain parameters that are used for this process. n is a prime number, and *G* is a point on the elliptic curve.

Output:

1. *status* The status returned from the key pair generation procedure. The status will indicate **SUCCESS** or an **ERROR**.

2. (*d*, *Q*) The generated private and public keys. If an error is encountered during the generation process, invalid values for *d* and *Q* **should** be returned, as represented by *Invalid_d* and *Invalid_Q* in the following specification. *d* is an integer, and *Q* is an elliptic curve point. The generated private key *d* is in the range [1, *n*–1].

Process:

1. *N* = **len**(*n*).

   Comment: Check that *N* is included in Table 1 of Section 6.1.1.

2. If *N* is invalid, then return an **ERROR** indication, *Invalid_d*, and *Invalid_Q*.

3. *requested_security_strength* = the security strength associated with *N*; see SP 800-57, Part 1.

4. Obtain a string of *N*+64 *returned_bits* from an **RBG** with a security strength of *requested_security_strength* or more. If an **ERROR** indication is returned, then return an **ERROR** indication, *Invalid_d*, and *Invalid_Q*.

5. Convert *returned_bits* to the (non-negative) integer *c* (see Appendix C.2.1).

6. *d* = (*c* mod (*n*–1)) + 1.

7. *Q* = *dG.*

8. Return **SUCCESS**, *d*, and *Q*.

B.4.2 Key Pair Generation by Testing Candidates

In this method, a random number is obtained and tested to determine that it will produce a value of *d* in the correct range. If *d* is out-of-range, another random number is obtained (i.e., the process is iterated until an acceptable value of *d* is obtained.

The following process or its equivalent may be used to generate an ECC key pair.

Input:

1. (*q, FR, a, b* {, *domain_parameter_seed*}*, G, n, h*)

   The domain parameters that are used for this process. *n* is a prime number, and *G* is a point on the elliptic curve.

Output:

1. *status* The status returned from the key pair generation procedure. The status will indicate **SUCCESS** or an **ERROR**.
2. (*d*, *Q*) The generated private and public keys. If an error is encountered during the generation process, invalid values for *d* and *Q* **should** be returned, as represented by *Invalid_d* and *Invalid_Q* in the following specification. *d* is an integer, and *Q* is an elliptic curve point. The generated private key *d* is in the range [1, *n*–1].

Process:

```
1. N = len(n).
                                    Comment: Check that N is included in Table 1 of
                                    Section 6.1.1.
2. If N is invalid, then return an ERROR indication, Invalid_d, and Invalid_Q.
3. requested_security_strength = the security strength associated with N; see SP 800-57,
   Part 1.
4. Obtain a string of N returned_bits from an RBG with a security strength of
   requested_security_strength or more. If an ERROR indication is returned, then
   return an ERROR indication, Invalid_d, and Invalid_Q.
5. Convert returned_bits to the (non-negative) integer c (see Appendix C.2.1).
6. If (c > n–2), then go to step 4.
7. d = c + 1.
8. Q = dG.
9. Return SUCCESS, d, and Q.
```

B.5 ECC Per-Message Secret Number Generation

ECDSA requires the generation of a new random number *k* for each message to be signed. Two methods are provided for the generation of *k*; one of these two methods **shall** be used.

The valid values of *n* are provided in Section 6.1.1. See ANS X9.62 for definitions of the elliptic curve math and the conversion routines.

Let **inverse**(*k*, *n*) be a function that computes the inverse of a (non-negative) integer *k* with respect to multiplication modulo the prime number *n*. A technique for computing the inverse is provided in Appendix C.1.

B.5.1 Per-Message Secret Number Generation Using Extra Random Bits

In this method, 64 more bits are requested from the RBG than are needed for *k* so that bias produced by the mod function in step 6 is not readily apparent.

The following process or its equivalent may be used to generate a per-message secret number.

Input:

1. (*q, FR, a, b* {, *domain_parameter_seed*}, *G, n, h*)

   The domain parameters that are used for this process. *n* is a prime number, and *G* is a point on the elliptic curve.

Output:

1. *status* The status returned from the key pair generation procedure. The status will indicate **SUCCESS** or an **ERROR**.
2. $(k,\ k^{-1})$ The generated secret number *k* and its inverse k^{-1}. If an error is encountered during the generation process, invalid values for *k* and k^{-1} **should** be returned, as represented by *Invalid_k* and *Invalid_k_inverse* in the following specification. *k* and k^{-1} are integers in the range [1, *n*–1].

Process:

1. N = **len**(*q*).

   Comment: Check that *N* is included in Table 1 of Section 6.1.1.

2. If *N* is invalid, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.
3. *requested_security_strength* = the security strength associated with *N*; see SP 800-57, Part 1.
4. Obtain a string of *N*+64 *returned_bits* from an **RBG** with a security strength of *requested_security_strength* or more. If an **ERROR** indication is returned, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.
5. Convert *returned_bits* to the non-negative integer *c* (see Appendix C.2.1).
6. $k = (c \bmod (n–1)) + 1$.
7. $(status,\ k^{-1})$ = **inverse**(*k*, *n*).
8. Return *status*, *k*, and k^{-1}.

B.5.2 Per-Message Secret Number Generation by Testing Candidates

In this method, a random number is obtained and tested to determine that it will produce a value of *k* in the correct range. If *k* is out-of-range, another random number is obtained (i.e., the process is iterated until an acceptable value of *k* is obtained.

The following process or its equivalent may b used to generate a per-message secret number.

Input:

1. (*q, FR, a, b* {, *domain_parameter_seed*}, *G, n, h*)

   The domain parameters that are used for this process. *n* is a prime number*,* and *G* is a point on the elliptic curve.

Output:

1. *status* The status returned from the key pair generation procedure. The status will indicate **SUCCESS** or an **ERROR**.

2. (k, k^{-1}) The generated secret number *k* and its inverse k^{-1}. If an error is encountered during the generation process, invalid values for *k* and k^{-1} **should** be returned, as represented by *Invalid_k* and *Invalid_k_inverse* in the following specification. *k* and k^{-1} are integers in the range [1, *n*–1].

Process:

1. *N* = **len**(*q*).

   Comment: Check that *N* is included in Table 1 of Section 6.1.1.

2. If *N* is invalid, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.

3. *requested_security_strength* = the security strength associated with *N*; see SP 800-57, Part 1.

4. Obtain a string of *N returned_bits* from an **RBG** with a security strength of *requested_security_strength* or more. If an **ERROR** indication is returned, then return an **ERROR** indication, *Invalid_k*, and *Invalid_k_inverse*.

5. Convert *returned_bits* to the (non-negative) integer *c* (see Appendix C.2.1).

6. If ($c > n$–2), then go to step 4.

7. $k = c + 1$.

8. (*status*, k^{-1}) = **inverse**(*k*, *n*).

9. Return *status*, *k*, and k^{-1}.

Appendix C: Generation of Other Quantities

This appendix contains routines for supplementary processes required for the implementation of this Standard. Appendix C.1 is needed to produce the inverse of the per-message secret k (see Section 4.5, and Appendices B.2.1, B.2.2, B.5.1 and B.5.2) and the inverse of the signature portion s that is used during signature verification (see Section 4.7). The routines in Appendix C.2 are required to convert between bit strings and integers where required in implementing this Standard. Appendix C.3 contains probabilistic primality tests to be used during the generation of DSA domain parameters and RSA key pairs. Appendices C.4 and C.5 contain algorithms required during the Lucas probabilistic primality test of Appendix C.3.3 to check for a perfect square and to compute the Jacobi symbol. Appendix C.6 contains the Shawe-Taylor algorithm for the construction of primes. Appendix C.7 provides a process to perform trial division, as required by the random prime generation routine in Appendix C.6. The sieve procedure in Appendix C.8 is needed by the trial division routine in Appendix C.7. The trial division process in Appendix C.7 and the sieve procedure in Appendix C.8 have been extracted from ANS X9.80, *Prime Number Generation, Primality Testing, and Primality Certificates*. Appendix C.9 is required during the generation of RSA key pairs. Appendix C.10 provides a method for constructing provable primes for RSA (see Appendix B.3.2.2 and B.3.4).

C.1 Computation of the Inverse Value

This algorithm or an algorithm that produces an equivalent result **shall** be used to compute the multiplicative inverse z^{-1} mod a, where $0 < z < a$, $0 < z^{-1} < a$, and a is a prime number. In this Standard, z is either k or s, and a is either q or n.

Input:

1. z The value to be inverted mod a (i.e., either k or s).
2. a The domain parameter and (prime) modulus (i.e., either q or n).

Output:

1. *status* The status returned from this function, where the *status* is either **SUCCESS** or **ERROR**.
2. z^{-1} The multiplicative inverse of z mod a, if it exists.

Process:

1. Verify that a and z are positive integers such that $z < a$; if not, return an **ERROR** indication.
2. Set $i = a$, $j = z$, $y_2 = 0$, and $y_1 = 1$.
3. $quotient = \lfloor i/j \rfloor$.

4. *remainder* = *i* –(*j* * *quotient*).
5. $y = y_2 - (y_1 * quotient)$.
6. Set $i = j$, $j = remainder$, $y_2 = y_1$, and $y_1 = y$.
7. If (j > 0), then go to step 3.
8. If ($i \neq 1$), then return an **ERROR** indication.
9. Return **SUCCESS** and y_2 mod a.

C.2 Conversion Between Bit Strings and Integers

C.2.1 Conversion of a Bit String to an Integer

An n-long sequence of bits $\{ x_1, \ldots, x_n \}$ is converted to an integer by the rule

$$\{ x_1, \ldots, x_n \} \rightarrow (x_1 * 2^{n-1}) + (x_2 * 2^{n-2}) + \ldots + (n_1 * 2) + x_n .$$

Note that the first bit of a sequence corresponds to the most significant bit of the corresponding integer, and the last bit corresponds to the least significant bit.

Input:

1. $b_1, b_2, \ldots, b_n$ The bit string to be converted.

Output:

1. C The requested integer representation of the bit string.

Process:

1. Let $(b_1, b_2, \ldots, b_n)$ be the bits of b from leftmost to rightmost.
2. $C = \sum_{i=1}^{n} 2^{(n-i)} b_i$
3. Return C.

In this Standard, the binary length of an integer C is defined as the smallest integer n satisfying $C < 2^n$.

C.2.2 Conversion of an Integer to a Bit String

An integer x in the range $0 \le x < 2^n$ may be converted to an n-long sequence of bits by using its binary expansion as shown below:

$$x = (x_1 * 2^{n-1}) + (x_2 * 2^{n-2}) + \ldots + (x_{n-1} * 2) + x_n \rightarrow \{x_1, \ldots, x_n\}$$

Note that the first bit of a sequence corresponds to the most significant bit of the corresponding integer, and the last bit corresponds to the least significant bit.

Input:

1. C The non-negative integer to be converted.

Output:

1. $b_1, b_2, \ldots, b_n$ The bit string representation of the integer C.

Process:

1. Let $(b_1, b_2, \ldots, b_n)$ represent the bit string, where $b_i = 0$ or 1, and b_1 is the most significant bit, while b_n is the least significant bit.
2. For any integer n that satisfies $C < 2^n$, the bits b_i **shall** satisfy:

$$C = \sum_{i=1}^{n} 2^{(n-i)} b_i$$

3. Return $b_1, b_2, \ldots, b_n$.

In this Standard, the binary length of the integer C is defined as the smallest integer n that satisfies $C < 2^n$.

C.3 Probabilistic Primality Tests

A probabilistic primality test may be required during the generation and validation of prime numbers. An approved robust probabilistic primality test **shall** be selected and used.

There are several probabilistic algorithms available. The Miller-Rabin probabilistic primality tests described in Appendices C.3.1 and C.3.2 are versions of a procedure due to M.O. Rabin, based in part on ideas of Gary L. Miller; one of these versions **shall** be used as the Miller-Rabin test discussed below. For more information, see [4]. For these tests, let **RBG** be an approved random bit generator (see SP 800-90).

There are several Lucas probabilistic primality tests available; the version provided in [5] is specified in Appendix C.3.3.

This Standard allows two alternatives for testing primality: either using several iterations of only the Miller-Rabin test, or using the iterated Miller-Rabin test, followed by a single Lucas test. The value of *iterations* (as used in Appendices C.3.1 and C.3.2) depends on the algorithm being used, the security strength, the error probability used, the length (in bits) of the candidate prime and the type of tests to be performed. Tables C.1, C.2 and C.3 list the minimum number of *iterations* of the Miller-Rabin tests that **shall** be performed.

As stated in Appendix F, if the definition of the error probability that led to the values of the number of Miller-Rabin tests for p and q in Tables C.1, C.2 and C.3 is not conservative enough, the prescribed number of Miller-Rabin tests can be followed by a single Lucas test. Since there are no known non-prime values that pass the two test combination (i.e., the indicated number of rounds of the Miller-Rabin test with randomly selected bases, followed by one round of the Lucas test), the two test combination may provide additional assurance of primality over the use of only the Miller-Rabin test. For DSA, the two-test combination may provide better performance. However, the Lucas test is not required when testing the p_1, p_2, q_1 and q_2 values for primality when generating RSA primes. See Appendix F for further information.

Table C.1. Minimum number of Miller-Rabin iterations for DSA

Parameters	M-R Tests Only	M-R Tests when followed by One Lucas test
p: 1024 bits q: 160 bits Error probability = 2^{-80}	For p and q: 40	For p: 3 For q: 19
p: 2048 bits q: 224 bits Error probability = 2^{-112}	For p and q: 56	For p: 3 For q: 24
p: 2048 bits q: 256 bits Error probability = 2^{-112}	For p and q: 56	For p: 3 For q: 27
p: 3072 bits q: 256 bits Error probability = 2^{-128}	For p and q: 64	For p: 2 For q: 27

Table C.2. Minimum number of rounds of M-R testing when generating primes for use in RSA Digital Signatures

Parameters	M-R Tests Only
p_1, p_2, q_1 and q_2 > 100 bits p and q: 512 bits Error probability = 2^{-80}	For p_1, p_2, q_1 and q_2: 28 For p and q: 5

p_1, p_2, q_1 and q_2 > 140 bits p and q: 1024 bits Error probability = 2^{-112}	For p_1, p_2, q_1 and q_2: 38 For p and q: 5
p_1, p_2, q_1 and q_2 > 170 bits p and q: 1536 bits Error probability = 2^{-128}	For p_1, p_2, q_1 and, q_2: 41 For p and q: 4

Table C.3. Minimum number of rounds of M-R testing when generating primes for use in RSA Digital Signatures using an error probability of 2^{-100}

Parameters	M-R Tests Only
p_1, p_2, q_1 and q_2 > 100 bits p and q: 512	For p_1, p_2, q_1 and q_2: 38 For p and q: 7
p_1, p_2, q_1 and q_2 > 140 bits p and q: 1024 bits	For p_1, p_2, q_1 and q_2: 32 For p and q: 4
p_1, p_2, q_1 and q_2 > 170 bits p and q: 1536 bits	For p_1, p_2, q_1 and q_2: 27 For p and q: 3

C.3.1 Miller-Rabin Probabilistic Primality Test

Let **RBG** be an approved random bit generator (see SP 800-90).

Input:

1. *w* The odd integer to be tested for primality. This will be either *p* or *q*, or one of the auxiliary primes p_1, p_2, q_1 or q_2.
2. *iterations* The number of iterations of the test to be performed; the value **shall** be consistent with Table C.1, C.2 or C.3.

Output:

1. *status* The status returned from the validation procedure, where *status* is either **PROBABLY PRIME** or **COMPOSITE**.

Process:

1. Let *a* be the largest integer such that 2^a divides $w-1$.

```
2. $m = (w-1) / 2^{a.}$
3. $wlen$ = **len** ($w$).
4. For $i = 1$ to $iterations$ do
    4.1 Obtain a string $b$ of $wlen$ bits from an RBG.
                                        Comment: Ensure that $1 < b < w-1$.
    4.2 If (($b \leq 1$) or ($b \geq w-1$)), then go to step 4.1.
    4.3 $z = b^m \bmod w$.
    4.4 If (($z = 1$) or ($z = w - 1$)), then go to step 4.7.
    4.5 For $j = 1$ to $a - 1$ do.
        4.5.1 $z = z^2 \bmod w$.
        4.5.2 If ($z = w-1$), then go to step 4.7.
        4.5.3 If ($z = 1$), then go to step 4.6.
    4.6 Return COMPOSITE.
    4.7 Continue.                       Comment: Increment $i$ for the do-loop in
                                        step 4.
5. Return PROBABLY PRIME.
```

C.3.2 Enhanced Miller-Rabin Probabilistic Primality Test

This method provides additional information when an error is encountered that may be useful when generating or validating RSA moduli. Let **RBG** be an approved random bit generator (see SP 800-90).

Input:

1. *w* — The odd integer to be tested for primality. This will be either p or q, or one of the auxiliary primes p_1, p_2, q_1 or q_2.
2. *iterations* — The number of iterations of the test to be performed; the value **shall** be consistent with Table C.1, C.2 or C.3.

Output:

1. *status* — The status returned from the validation procedure, where *status* is either **PROBABLY PRIME**, **PROVABLY COMPOSITE WITH FACTOR** (returned with the factor), and **PROVABLY COMPOSITE AND NOT A POWER OF A PRIME**.

Process:

```
1. Let $a$ be the largest integer such that $2^a$ divides $w–1$.
2. $m = (w–1) / 2^{a.}$
3. $wlen$ = **len** ($w$).
4. For $i = 1$ to *iterations* do
    4.1 Obtain a string $b$ of $wlen$ bits from an RBG.
                                        Comment: Ensure that $1 < b < w–1$.
    4.2 If (($b \le 1$) or ($b \ge w–1$)), then go to step 4.1.
    4.3 $g$ = **GCD**($b$, $w$).
    4.4 If ($g > 1$), then return **PROVABLY COMPOSITE WITH FACTOR** and the
        value of $g$.
    4.5 $z = b^m \bmod w$.
    4.6 If (($z = 1$) or ($z = w – 1$)), then go to step 4.15.
    4.7 For $j = 1$ to $a – 1$ do.
        4.7.1 $x = z$.                  Comment: $x \ne 1$ and $x \ne w–1$.
        4.7.2 $z = x^2 \bmod w$.
        4.7.3 If ($z = w–1$), then go to step 4.15.
        4.7.4 If ($z = 1$), then go to step 4.12.
    4.8 $x = z$.                        Comment: $x = b^{(w–1)/2} \bmod w$ and $x \ne w–1$.
    4.9 $z = x^2 \bmod w$.
    4.10 If ($z = 1$), then go to step 4.12.
    4.11 $x = z$.                       Comment: $x = b^{(w–1)} \bmod w$ and $x \ne 1$.
    4.12 $g$ = **GCD**($x–1$, $w$).
    4.13 If (g > 1), then return **PROVABLY COMPOSITE WITH FACTOR** and the
         value of $g$.
    4.14 Return **PROVABLY COMPOSITE AND NOT A POWER OF A PRIME.**
    4.15 Continue.                           Comment: Increment $i$ for the do-loop in
                                             step 4.
5. Return **PROBABLY PRIME.**
```

C.3.3 (General) Lucas Probabilistic Primality Test

The following process or its equivalent **shall** be used as the Lucas test.

Input:

C The candidate odd integer to be tested for primality.

Output:

status Where *status* is either **PROBABLY PRIME** or **COMPOSITE**.

Process:

1. Test whether C is a perfect square (see Appendix C.4). If so, return (**COMPOSITE**).
2. Find the first D in the sequence $\{5, -7, 9, -11, 13, -15, 17, \ldots\}$ for which the Jacobi symbol $\left(\frac{D}{C}\right) = -1$. See Appendix C.5 for an approved method to compute the Jacobi Symbol. If $\left(\frac{D}{C}\right) = 0$ for any D in the sequence, return (**COMPOSITE**).
3. $K = C+1$.
4. Let $K_r K_{r-1} \ldots K_0$ be the binary expansion of K, with $K_r = 1$.
5. Set $U_r = 1$ and $V_r = 1$.
6. For $i = r-1$ to 0, do

    6.1 $U_{temp} = U_{i+1} V_{i+1} \bmod C$.

    6.2 $V_{temp} = \dfrac{V_{i+1}{}^2 + DU_{i+1}{}^2}{2} \bmod C$.

    6.3 If ($K_i = 1$), then — Comment: If $K_i = 1$, then do steps 6.3.1 and 6.3.2; otherwise, do steps 6.3.3 and 6.3.4.

    6.3.1 $U_i = \dfrac{U_{temp} + V_{temp}}{2} \bmod C$.

    6.3.2 $V_i = \dfrac{V_{temp} + DU_{temp}}{2} \bmod C$.

    Else

    6.3.3 $U_i = U_{temp}$.

    6.3.4 $V_i = V_{temp}$.

7. If ($U_0 = 0$), then return (**PROBABLY PRIME**). Otherwise, return (**COMPOSITE**).

Steps 6.2, 6.3.1 and 6.3.2 contain expressions of the form $A/2 \bmod C$, where A is an integer, and

C is an odd integer. If $A/2$ is not an integer (i.e., A is odd), then $A/2$ mod C may be calculated as $(A+C)/2$ mod C. Alternatively, $A/2$ mod $C = A\cdot(C+1)/2$ mod C, for any integer A, without regard to A being odd or even.

C.4 Checking for a Perfect Square

The following algorithm may be used to determine whether an n-bit positive integer C is a perfect square:

Input:

C The integer to be checked.

Output:

status Where *status* is either **PERFECT SQUARE** or **NOT A PERFECT SQUARE**.

Process:

```
1. Set $n$, such that $2^n > C \geq 2^{(n-1)}$.
2. $m = \lceil n/2 \rceil$.
3. $i = 0$.
4. Select $X_{0,}$ such that $2^m > X_0 \geq 2^{(m-1)}$.
5. Repeat
       5.1   $i = i + 1$.
       5.2   $X_i = ((X_{i-1})^2 + C)/(2X_{i-1})$.
   Until $(X_i)^2 < 2^m + C$.
6. If $C = \lfloor X_i \rfloor^2$, then
       status = PERFECT SQUARE.
   Else
       status = NOT A PERFECT SQUARE.
7. Return status.
```

Notes:

1. By starting with $X_0 > (1/2)$ **Sqrt**(C), $|X_0 - \textbf{Sqrt}(C)|$ is guaranteed to be less than X_0 . This inequality is maintained in step 5; i.e., $|X_i - \textbf{Sqrt}(C)| < X_i$ for all i.
2. For $i \geq 1$, $0 \leq X_i - \textbf{Sqrt}(C) = (X_{i-1} - \textbf{Sqrt}(C))^2 / (2\,X_{i-1}) < X_0/2^i$.

   In particular, $0 \leq X_m - \textbf{Sqrt}(C) < 1$. If **Sqrt**($C$) were an integer, then it would be equal to the floor of X_m .

3. In general, the inequality X_i – **Sqrt**(C) < 1 will occur for values of i that are much less than m. To detect this, the fact that $2^{(m-1)} \leq$ **Sqrt**(C) < X_i for all $i \geq 1$ can be used,

$$X_i - \mathbf{Sqrt}(C) = ((X_i)^2 - C)/(\,X_i + \mathbf{Sqrt}(C))$$

$$\leq ((X_i)^2 - C)/(\,2\,\mathbf{Sqrt}(C))$$

$$\leq\ ((X_i)^2 - C)/(2^m)$$

Thus, the condition $(X_i)^2 < 2^m + C$ implies that X_i – **Sqrt**(C) < 1.

C.5 Jacobi Symbol Algorithm

This routine computes the Jacobi symbol $\left(\frac{a}{n}\right)$.

Jacobi( ):

Input:

a Any integer. For this Standard, the initial value is in the sequence {5, –7, 9, –11, 13, –15, 17, **. . .**}, as determined by Appendix C.3.3.

n Any integer. For this Standard, the initial value is the candidate being tested, as determined by Appendix C.3.3.

Output:

result The calculated Jacobi symbol.

Process:

```
1. $a = a \bmod n$.                      Comment: $a$ will be in the range $0 \leq a <$ n.
2. If $a = 1$, or $n = 1$, then return (1).
3. If $a = 0$, then return (0).
4. Define $e$ and $a_1$ such that $a = 2^e a_1$, where $a_1$ is odd.
5. If $e$ is even, then $s = 1$.
   Else if (($n \equiv 1 \pmod 8$) or ($n \equiv 7 \pmod 8$))), then $s = 1$.
   Else if (($n \equiv 3 \pmod 8$) or ($n \equiv 5 \pmod 8$)), then $s = -1$.
6. If (($n \equiv 3 \pmod 4$) and ($a_1 \equiv 3 \pmod 4$))), then $s = -s$.
7. $n_1 = n \bmod a_1$.
8. Return ($s$ * Jacobi ($n_1$, $a_1$)).     Comment: Call this routine recursively.
```

Example: Compute the Jacobi symbol for $a = 5$ and $n = 3439601197$:

1. n is not 1, and a is not 1, so proceed to Step 2.
2. a is not 0, so proceed to Step 3.
3. $5 = 2^0 * 5$, so $e = 0$, and $a_1 = 5$.
4. e is even, so $s = 1$.
5. a_1 is not congruent to 3 mod 4, so do not change s.
6. $n_1 = 2 = n \bmod 5$.
7. Compute and return (1 * Jacobi(2, 5)). This calls Jacobi recursively. Compute the Jacobi symbol for $a = 2$ and $n = 5$:

   7.1 n is not 1, and a is not 1, so proceed to Step 7.2.

   7.2 a is not 0, so proceed to Step 7.3.

   7.3 $2 = 2^1 * 1$, so $e = 1$, and $a_1 = 1$.

   7.4 e is odd, and $n \equiv 5 \pmod 8$, so set $s = –1$.

   7.5 n is not 3 mod 4, and a_1 is not 3 mod 4, so proceed to step 7.6.

   7.6 $n_1 = 0 = n \bmod 1$.

   7.7 Return (–1 * Jacobi(0, 1) = –1). This calls Jacobi recursively. Compute the Jacobi symbol for $a = 0$ and $n = 1$:

      7.7.1 $n = 1$, so return 1.

Thus, Jacobi (0,1) = 1, so Jacobi (2,5) = –1*(1) = –1, and Jacobi (5, 3439601197) = 1* (–1) = –1.

C.6 Shawe-Taylor Random_Prime Routine

This routine is recursive and may be used to construct a provable prime number using a hash function.

Let **Hash( )** be the selected hash function, and let *outlen* be the bit length of the hash function output block. The following process or its equivalent **shall** be used to generate a prime number for this constructive method.

ST_Random_Prime ( ):

Input:

1. *length* The length of the prime to be generated.
2. *input_seed* The seed to be used for the generation of the requested prime.

Output:

1. *status* The status returned from the generation routine, where *status* is either **SUCCESS** or **FAILURE.** If **FAILURE** is returned, then zeros are returned as the other output values.

2. *prime* The requested prime.

3 *prime_seed* A seed determined during generation.

4. *prime_gen_counter* (Optional) A counter determined during the generation of the prime.

Process:

```
1. If (length < 2), then return (FAILURE, 0, 0 {, 0}).
2. If (length ≥ 33), then go to step 14.
3. prime_seed = input_seed.
4. prime_gen_counter = 0.
                                   Comment: Generate a pseudorandom integer
                                   c of length bits.
5. $c$ = Hash(prime_seed) ⊕ Hash(prime_seed + 1).
6. $c = 2^{length - 1} + (c \bmod 2^{length - 1})$.
7. $c = (2 * \lfloor c / 2 \rfloor) + 1$.
                                       Comment: Set prime to the least odd
                                       integer greater than or equal to c.
8. prime_gen_counter = prime_gen_counter + 1.
9. prime_seed = prime_seed + 2.
10. Perform a deterministic primality test on c. For example, since c is small, its primality
    can be tested by trial division. See Appendix C.7.
11. If (c is a prime number), then
    11.1   prime = c.
    11.2   Return (SUCCESS, prime, prime_seed {, prime_gen_counter}).
12. If (prime_gen_counter > (4 * length)), then return (FAILURE, 0, 0 {, 0}).
13. Go to step 5.
14. (status, $c_0$, prime_seed, prime_gen_counter) = (ST_Random_Prime (($\lceil length / 2 \rceil$ +
    1), input_seed).
```

```
15. If **FAILURE** is returned, return (**FAILURE**, 0, 0 {, 0}).
16. $iterations = \lceil length\ /\ outlen \rceil - 1$.
17. $old\_counter = prime\_gen\_counter$.
                                        Comment: Generate a pseudorandom integer
                                        $x$ in the interval [$2^{length-1}$, $2^{length}$].
18. $x = 0$.
19. For $i = 0$ to $iterations$ do
        $x = x + (\mathbf{Hash}(prime\_seed + i) * 2^{i \times outlen})$.
20. $prime\_seed = prime\_seed + iterations + 1$.
21. $x = 2^{length-1} + (x \bmod 2^{length-1})$.
                                        Comment: Generate a candidate prime $c$ in
                                        the interval [$2^{length-1}$, $2^{length}$].
22. $t = \lceil x\ /\ (2c_0) \rceil$.
23. If $(2tc_0 + 1 > 2^{length})$, then $t = \lceil 2^{length-1}\ /\ (2c_0) \rceil$.
24. $c = 2tc_0 + 1$.
25. $prime\_gen\_counter = prime\_gen\_counter + 1$.
                                        Comment: Test the candidate prime $c$ for
                                        primality; first pick an integer $a$ between 2
                                        and $c - 2$.
26. $a = 0$.
27. For $i = 0$ to $iterations$ do
        $a = a + (\mathbf{Hash}(prime\_seed + i) * 2^{i * outlen})$.
28. $prime\_seed = prime\_seed + iterations + 1$.
29. $a = 2 + (a \bmod (c - 3))$.
30. $z = a^{2t} \bmod c$.
31. If $((1 = \mathbf{GCD}(z - 1, c))$ and $(1 = z^{c_0} \bmod c))$, then
    31.1    $prime = c$.
    31.2    Return (**SUCCESS**, $prime$, $prime\_seed$ {, $prime\_gen\_counter$}).
32. If $(prime\_gen\_counter \geq ((4 * length) + old\_counter))$, then return (**FAILURE**, 0, 0
    {, 0}).
```

33. $t = t + 1$.
34. Go to step 23.

C.7 Trial Division

An integer is proven to be prime by showing that it has no prime factors less than or equal to its square root. This procedure is not recommended for testing any integers longer than 10 digits.

To prove that c is prime:

1. Prepare a table of primes less than $\sqrt{c}$. This can be done by applying the sieve procedure in Appendix C.8.
2. Divide c by every prime in the table. If c is divisible by one of the primes, then declare that c is composite and exit. If convenient, c may be divided by composite numbers. For example, rather than preparing a table of primes, it might be more convenient to divide by all integers except those divisible by 3 or 5.
3. Otherwise, declare that c is prime and exit.

C.8 Sieve Procedure

A *sieve procedure* is described as follows: Given a sequence of integers Y_0, $Y_0 + 1$, . . . , $Y_0 + J$, a sieve will identify the integers in the sequence that are divisible by primes up to some selected limit.

Note that the definitions of the mathematical symbols in this process (e.g., h, L, M, p) are internal to this process only, and should not be confused with their use elsewhere in this Standard.

Start by selecting a *factor base* of all the primes p_j, from 2 up to some selected limit L. The value of L is arbitrary and may be determined by computer limitations. A good, typical value of L would be anywhere from 10^3 to 10^5.

1. Compute $S_j = Y_0 \bmod p_j$ for all p_j in the factor base.
2. Initialize an array of length $J + 1$ to zero.
3. Starting at $Y_0 - S_j + p_j$, let every p_j^{th} element of the array be set to 1. Do this for the entire length of the array and for every j.
4. When finished, every location in the array that has the value 1 is divisible by some small prime, and is therefore a composite.

The array can be either a bit array for compactness when memory is small, or a byte array for speed when memory is readily available. There is no need to sieve the entire sieve interval at once. The array can be partitioned into suitably small pieces, sieving each piece before going on to the next piece. When finished, every location with the value 0 is a candidate for prime testing.

The amount of work for this procedure is approximately $M \log \log L$, where M is the length of the sieve interval; this is a very efficient procedure for removing composite candidates for primality testing. If $L = 10^5$, the sieve will remove about 96% of all composites.

In some cases, rather than having a set of consecutive integers to sieve, the set of integers to be tested consists of integers lying in an arithmetic progression $Y_0, Y_0 + h, Y_0 + 2h, \ldots, Y_0 + Jh,$ where h is large and not divisible by any primes in the factor base.

1. Select a factor base and initialize an array of length $J + 1$ to 0.
2. Compute $S_j = Y_0 \bmod p_j$ for all p_j in the factor base.
3. Compute $T_j = h \bmod p_j$ and $r = - S_j T_j^{-1} \bmod p_j$.
4. Starting at $Y_0 + r$, let every p_j^{th} element of the array be set to 1. Do this for the entire length of the array and for every j. Note that the position $Y_0 + r$ in the array actually denotes the number $Y_0 + rh$.
5. When finished, every location in the array that has the value 1 is divisible by some small prime and is therefore composite.

Note: The prime "2" takes the longest amount of time ($M/2$) to sieve, since it touches the most locations in the sieve array. An easy optimization is to combine the initialization of the sieve array with the sieving of the prime "2". It is also possible to sieve the prime "3" during initialization. These optimizations can save about 1/3 of the total sieve time.

C.9 Compute a Probable Prime Factor Based on Auxiliary Primes

This routine constructs a probable prime (a candidate for p or q) using two auxiliary prime numbers and the Chinese Remainder Theorem (CRT).

Input:

r_1 and r_2 Two odd prime numbers satisfying $\log_2(r_1 r_2) \le (nlen/2) - \log_2(nlen/2) - 6$.

nlen The desired length of n, the RSA modulus.

e The public verification exponent.

security_strength The minimum security strength required for random number generation.

Output:

status The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE**. If **FAILURE** is returned, then zeros are returned as the other output values.

private_prime_factor The prime factor of n.

X The random number used during the generation of the *private_prime_factor*.

Process:

```
1. If (GCD($2r_1$, $r_2$) $\neq$ 1), then return (FAILURE, 0, 0).
2. $R = ((r_2^{-1} \bmod 2r_1) * r_2) - (((2r_1)^{-1} \bmod r_2) * 2r_1)$.
       Comment: Apply the CRT, so that $R \equiv 1 \pmod{2r_1}$ and $R \equiv -1 \pmod{r_2}$.
3. Generate a random number $X$ using an approved random number generator that
   supports the security_ strength, such that $(\sqrt{2})(2^{nlen/2-1}) \le X \le (2^{nlen/2} - 1)$.
4. $Y = X + ((R - X) \bmod 2r_1r_2)$.
       Comment: $Y$ is the first odd integer $\geq X$, such that $r_1$ is a prime factor of $Y$–1, and $r_2$ is a prime factor of $Y$+1.
       Comment: Determine the requested prime number by constructing candidates from a sequence and performing primality tests.
5. $i = 0$.
6. If ($Y \geq 2^{nlen/2}$), then go to step 3.
7. If (GCD($Y$–1, $e$) = 1), then
   7.1 Check the primality of $Y$ as specified in Appendix C.3. If PROBABLY PRIME
       is not returned, go to step 8.
   7.2 private_prime_factor = $Y$.
   7.3 Return (SUCCESS, private_prime_factor, $X$).
8. $i = i + 1$.
9. If ($i \geq 5(nlen/2)$), then return (FAILURE, 0, 0).
10. $Y = Y + (2r_1r_2)$.
11. Go to step 6.
```

C.10 Construct a Provable Prime (possibly with Conditions), Based on Contemporaneously Constructed Auxiliary Provable Primes

The following process (or its equivalent) **shall** be used to generate an L-bit provable prime p (a candidate for one of the prime factors of an RSA modulus). Note that the use of p in this specification is used generically; both RSA prime factors p and q may be generated using this method.

If a so-called "strong prime" is required, this process can generate primes p_1 and p_2 (of specified bit-lengths N_1 and N_2) that divide $p-1$ and $p+1$, respectively. The resulting prime p will satisfy the conditions traditionally required of a strong prime, provided that the requested bit-lengths for p_1 and p_2 have appropriate sizes.

Regardless of the bit-lengths selected for p_1 and p_2, the quantity $p-1$ will have a prime divisor p_0 whose bit-length is slightly more than half that of p. In addition, the quantity p_0-1 will have a prime divisor whose bit-length is slightly more than half that of p_0.

This algorithm requires that $N_1 + N_2 \leq L - \lceil L/2 \rceil - 4$. Values for N_1 and N_2 **should** be chosen such that $N_1 + N_2 \leq (L/2) - \log_2(L) - 7$, to ensure that the algorithm can generate as many as $5L$ distinct candidates for p.

Let **Hash** be the selected hash function to be used, and let *outlen* be the bit length of the hash function output block.

Provable_Prime_Construction():

Input:

1. L A positive integer equal to the requested bit-length for p. Note that acceptable values for $L = nlen/2$ are computed as specified in Appendix B.3.1, criteria 2(b) and (c), with *nlen* assuming a value specified in Table B.1.
2. N_1 A positive integer equal to the requested bit-length for p_1. If $N_1 \geq 2$, then p_1 is an odd prime of N_1 bits; otherwise, $p_1 = 1$. Acceptable values for $N_1 \geq 2$ are provided in Table B.1
3. N_2 A positive integer equal to the requested bit-length for p_2. If $N_2 \geq 2$, then p_2 is an odd prime of N_2 bits; otherwise, $p_2 = 1$. Acceptable values for $N_2 \geq 2$ are provided in Table B.1
4. *firstseed* A bit string equal to the first seed to be used.
5. e The public verification exponent.

Output:

1. *status* The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE.** If **FAILURE** is returned, then zeros are returned as the other output values.
2. p, p_1, p_2 The required prime p, along with p_1 and p_2 having the property that p_1 divides $p-1$ and p_2 divides $p+1$.
3. *pseed* A seed determined during generation.

Process:

```
1. If $L$, $N_1$, and $N_2$ are not acceptable, then, return (**FAILURE**, 0, 0, 0, 0).
                                        Comment: Generate $p_1$ and $p_2$, as well as the prime $p_0$.
2. If $N_1 = 1$, then
   2.1 $p_1 = 1$.
   2.2 $p_2seed = firstseed$.
3. If $N_1 \geq 2$, then
   3.1 Using $N_1$ as the length and *firstseed* as the *input_seed*, use the random prime
       generation routine in Appendix C.6 to obtain $p_1$ and $p_2seed$.
   3.2 If **FAILURE** is returned, then return (**FAILURE**, 0, 0, 0, 0).
4. If $N_2 = 1$, then
   4.1 $p_2 = 1$.
   4.2 $p_0seed = p_2seed$.
5. If $N_2 \geq 2$, then
   5.1 Using $N_2$ as the length and $p_2seed$ as the *input_seed*, use the random prime
       generation routine in Appendix C.6 to obtain $p_2$ and $p_0seed$.
   5.2 If **FAILURE** is returned, then return (**FAILURE**, 0, 0, 0, 0).
6. Using $\lceil L / 2 \rceil + 1$ as the length and $p_0seed$ as the *input_seed*, use the random prime
   generation routine in Appendix C.6 to obtain $p_0$ and $pseed$. If **FAILURE** is returned,
   then return (**FAILURE**, 0, 0, 0, 0).
                                        Comment: Generate a (strong) prime $p$ in the
                                        interval $[(\sqrt{2})(2^{L-1}), 2^L - 1]$.
7. $iterations = \lceil L / outlen \rceil - 1$.
8. $pgen\_counter = 0$.
                                        Comment: Generate pseudo-random $x$ in the
                                        interval $[(\sqrt{2})(2^{L-1}) - 1, 2^L - 1]$.
9. $x = 0$.
10. For $i = 0$ to $iterations$ do
        $x = x + (\mathbf{Hash}(pseed + i)) * 2^{i * outlen}$.
11. $pseed = pseed + iterations + 1$.
```

```
12. $x = \lfloor(\sqrt{2})(2^{L-1})\rfloor + (x \bmod (2^L - \lfloor(\sqrt{2})(2^{L-1})\rfloor))$.
                                        Comment: Generate a candidate for the prime $p$.
13. If ($\mathbf{GCD}(p_0 p_1, p_2) \neq 1$), then return (FAILURE, 0, 0, 0, 0).
14. Compute $y$ in the interval $[1, p_2]$ such that $0 = (y\, p_0\, p_1 - 1) \bmod p_2$.
15. $t = \lceil((2y\, p_0\, p_1) + x)/(2 p_0\, p_1\, p_2)\rceil$.
16. If $((2(t\, p_2 - y)\, p_0\, p_1 + 1) > 2^L)$, then
        $t = \lceil((2y\, p_0\, p_1) + \lfloor(\sqrt{2})(2^{L-1})\rfloor) / (2 p_0\, p_1\, p_2)\rceil$.
                                        Comment: $p$ satisfies
                                            $0 = (p-1) \bmod (2p_0\, p_1)$ and
                                            $0 = (p+1) \bmod p_2$.
17. $p = 2(t\, p_2 - y)\, p_0\, p_1 + 1$.
18. $pgen\_counter = pgen\_counter + 1$.
19. If ($\mathbf{GCD}(p-1, e) = 1$), then
                                        Comment: Choose an integer $a$ in the interval $[2, p-2]$.
    19.1  $a = 0$
    19.2  For $i = 0$ to $iterations$ do
              $a = a + (\mathbf{Hash}(pseed + i)) * 2^{i * outlen}$.
    19.3  $pseed = pseed + iterations + 1$.
    19.4  $a = 2 + (a \bmod (p-3))$.
                                        Comment: Test $p$ for primality:
    19.5  $z = a^{2(t\, p_2 - y)\, p_1} \bmod p$.
    19.6  If ($(1 = \mathbf{GCD}(z-1, p))$ and $(1 = (z^{p_0} \bmod p))$), then return (SUCCESS, $p$, $p_1$, $p_2$, $pseed$).
20. If ($pgen\_counter \geq 5L$), then return (FAILURE, 0, 0, 0, 0).
21. $t = t + 1$.
22. Go to step 16.
```

Appendix D: Recommended Elliptic Curves for Federal Government Use

This collection of elliptic curves is recommended for Federal government use and contains choices for the private key length and underlying fields. These curves were generated using SHA-1 and the method given in the ANS X9.62 and IEEE Standard 1363-2000 standards. This appendix describes the process that was used. Note that these curves are the same as those included in the previous version of this Standard.

D.1 NIST Recommended Elliptic Curves

D.1.1 Choices

D.1.1.1 Choice of Key Lengths

The principal parameters for elliptic curve cryptography are the elliptic curve E and a designated point G on E called the *base point*. The base point has order n, which is a large prime. The number of points on the curve is hn for some integer h (the *cofactor*), which is not divisible by n. For efficiency reasons, it is desirable to have the cofactor be as small as possible.

All of the curves given below have cofactors 1, 2, or 4. As a result, the private and public keys for a curve are approximately the same length.

D.1.1.2 Choice of Underlying Fields

For each key length, two kinds of fields are provided.

- A *prime field* is the field $GF(p)$, which contains a prime number p of elements. The elements of this field are the integers modulo p, and the field arithmetic is implemented in terms of the arithmetic of integers modulo p.
- A *binary field* is the field $GF(2^m)$, which contains 2^m elements for some m (called the *degree* of the field). The elements of this field are the bit strings of length m, and the field arithmetic is implemented in terms of operations on the bits.

The security strengths for five ranges of the bit length of n is provided in SP 800-57. For the field $GF(p)$, the security strength is dependent on the length of the binary expansion of p. For the field $GF(2^m)$, the security strength is dependent on the value of m. Table E-1 provides the bit lengths of the various underlying fields of the curves provided in this appendix. Column 1 lists the ranges for the bit length of n (also see Table 1 in Section 6.1.1). Column 2 identifies the value of p used for the curves over prime fields, where **len**(p) is the length of the binary expansion of the integer p. Column 3 provides the value of m for the curves over binary fields.

Table D-1: Bit Lengths of the Underlying Fields of the Recommended Curves

Bit Length of *n*	Prime Field	Binary Field
161 – 223	**len**(p) = 192	m = 163
224 – 255	**len**(p) = 224	m = 233
256 – 383	**len**(p) = 256	m = 283
384 – 511	**len**(p) = 384	m = 409
≥ 512	**len**(p) = 521	m = 571

D.1.1.3 Choice of Basis for Binary Fields

To describe the arithmetic of a binary field, it is first necessary to specify how a bit string is to be interpreted. This is referred to as choosing a *basis* for the field. There are two common types of bases: a *polynomial basis* and a *normal basis*.

- A polynomial basis is specified by an irreducible polynomial modulo 2, called the *field polynomial*. The bit string (a_{m-1} ... a_2 a_1 a_0) is taken to represent the polynomial

  $$a_{m-1}\, t^{\,m-1} + \ldots + a_2\, t^2 + a_1\, t + a_0$$

  over $GF(2)$. The field arithmetic is implemented as polynomial arithmetic modulo $p(t)$, where $p(t)$ is the field polynomial.

- A normal basis is specified by an element θ of a particular kind. The bit string (a_0 a_1 a_2 ... a_{m-1}) is taken to represent the element

  $$a_0\theta + a_1\theta^2 + a_2\theta^{2^2} + \ldots + a_{m-1}\theta^{2^{m-1}}.$$

  Normal basis field arithmetic is not easy to describe or efficient to implement in general, except for a special class called *Type T low-complexity* normal bases. For a given field degree *m*, the choice of *T* specifies the basis and the field arithmetic (see Appendix D.3).

There are many polynomial bases and normal bases from which to choose. The following procedures are commonly used to select a basis representation.

- *Polynomial Basis*: If an irreducible *trinomial* $t^m + t^k + 1$ exists over GF (2)*,* then the field polynomial $p(t)$ is chosen to be the irreducible trinomial with the lowest-degree middle term t^k . If no irreducible trinomial exists, then a *pentanomial* $t^{\,m} + t^{\,a} + t^{\,b} + t^{\,c} + 1$ is selected. The particular pentanomial chosen has the following properties: the second term t^a has the lowest degree *m*; the third term t^b has the lowest degree among all irreducible pentanomials of degree *m* and second term t^a; and the fourth term t^c has the lowest degree among all irreducible pentanomials of degree *m*, second term t^a, and third term t^b.

- *Normal Basis*: Choose the Type T low-complexity normal basis with the smallest *T*.

For each binary field, the parameters are given for the above basis representations.

D.1.1.4 Choice of Curves

Two kinds of curves are given:

- *Pseudo-random* curves are those whose coefficients are generated from the output of a seeded cryptographic hash function. If the domain parameter seed value is given along with the coefficients, it can be easily verified that the coefficients were generated by that method.
- *Special curves* are those whose coefficients and underlying field have been selected to optimize the efficiency of the elliptic curve operations.

For each curve size range, the following curves are given:

→ A pseudo-random curve over $GF(p)$.

→ A pseudo-random curve over $GF(2^m)$.

→ A special curve over $GF(2^m)$ called a *Koblitz curve* or *anomalous binary curve*.

The pseudo-random curves were generated as specified in ANS X9.62 using SHA-1.

D.1.1.5 Choice of Base Points

Any point of order n can serve as the base point. Each curve is supplied with a sample base point $G = (G_x , G_y)$. Users may want to generate their own base points to ensure cryptographic separation of networks. See ANS X9.62 or IEEE Standard 1363-2000.

D.1.2 Curves over Prime Fields

For each prime p, a pseudo-random curve

$$E : \; y^2 \equiv x^3 - 3x + b \pmod{p}$$

of prime order n is listed[4]. (Thus, for these curves, the cofactor is always $h = 1$.) The following parameters are given:

- The prime modulus p
- The order n
- The 160-bit input seed *SEED* to the SHA-1 based algorithm (i.e., the domain parameter seed)
- The output c of the SHA-1 based algorithm

[4] The selection $a \equiv -3$ for the coefficient of x was made for reasons of efficiency; see IEEE Std 1363-2000.

- The coefficient b (satisfying $b^2 c \equiv -27 \pmod{p}$)
- The base point x coordinate G_x
- The base point y coordinate G_y

The integers p and n are given in decimal form; bit strings and field elements are given in hexadecimal.

D.1.2.1 Curve P-192

$p =$ 6277101735386680763835789423207666416083908700390324961279

$n =$ 6277101735386680763835789423176059013767194773182842284081

$SEED =$ 3045ae6f c8422f64 ed579528 d38120ea e12196d5

$c =$ 3099d2bb bfcb2538 542dcd5f b078b6ef 5f3d6fe2 c745de65

$b =$ 64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1

$G_x =$ 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012

$G_y =$ 07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811

D.1.2.2 Curve P-224

$p =$ 2695994666715063979466701508701963067355791626002630814351
0066298881

$n =$ 2695994666715063979466701508701962594045780771442439172168
2722368061

$SEED =$ bd713447 99d5c7fc dc45b59f a3b9ab8f 6a948bc5

$c =$ 5b056c7e 11dd68f4 0469ee7f 3c7a7d74 f7d12111 6506d031
218291fb

$b =$ b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943
2355ffb4

$G_x =$ b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6
115c1d21

$G_y =$ bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199
85007e34

D.1.2.3 Curve P-256

$p =$ 1157920892103562487626974469494075735300861434152903141955
33631308867097853951

$n =$ 1157920892103562487626974469494075735299969552241357603424
22259061068512044369

$SEED =$ c49d3608 86e70493 6a6678e1 139d26b7 819f7e90

$c =$ 7efba166 2985be94 03cb055c 75d4f7e0 ce8d84a9 c5114abc
af317768 0104fa0d

$b =$ 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6
3bce3c3e 27d2604b

$G_x =$ 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0
f4a13945 d898c296

$G_y =$ 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece
cbb64068 37bf51f5

D.1.2.4 Curve P-384

$p =$ 3940200619639447921227904010014361380507973927046544666794
8293404245721771496870329047266088258938001861606973112319

$n =$ 3940200619639447921227904010014361380507973927046544666794
6905279627659399113263569398956308152294913554433653942643

$SEED =$ a335926a a319a27a 1d00896a 6773a482 7acdac73

$c =$ 79d1e655 f868f02f ff48dcde e14151dd b80643c1 406d0ca1
0dfe6fc5 2009540a 495e8042 ea5f744f 6e184667 cc722483

$b =$ b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112
0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

$G_x =$ aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98
59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7

$G_y =$ 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c
e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

D.1.2.5 Curve P-521

p = 68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545549772963113914808580371219879997166438125740282911150571 51

n = 686479766013060971498190079908139321726943530014330540939
446345918554318339765539424505774633321719753296399637136
3321113864768612440380340372808892707005449

$SEED$ = d09e8800 291cb853 96cc6717 393284aa a0da64ba

c = 0b4 8bfa5f42 0a349495 39d2bdfc 264eeeeb 077688e4
4fbf0ad8 f6d0edb3 7bd6b533 28100051 8e19f1b9 ffbe0fe9
ed8a3c22 00b8f875 e523868c 70c1e5bf 55bad637

b = 051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b
99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd
3bb1bf07 3573df88 3d2c34f1 ef451fd4 6b503f00

G_x = c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139
053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127
a2ffa8de 3348b3c1 856a429b f97e7e31 c2e5bd66

G_y = 118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449
579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901
3fad0761 353c7086 a272c240 88be9476 9fd16650

D.1.3 Curves over Binary Fields

For each field degree m, a pseudo-random curve is given, along with a Koblitz curve. The pseudo-random curve has the form

$$E: y^2 + x\,y = x^3 + x^2 + b,$$

and the Koblitz curve has the form

$$E_a: y^2 + x\,y = x^3 + ax^2 + 1,$$

where $a = 0$ or 1.

For each pseudorandom curve, the cofactor is $h = 2$. The cofactor of each Koblitz curve is $h = 2$ if $a = 1$, and $h = 4$ if $a = 0$.

The coefficients of the pseudo-random curves, and the coordinates of the base points of both kinds of curves, are given in terms of both the polynomial and normal basis representations discussed in Appendix D.1.1.3.

For each *m*, the following parameters are given:

Field Representation:

- The normal basis type *T*
- The field polynomial (a trinomial or pentanomial)

Koblitz Curve:

- The coefficient *a*
- The base point order *n*
- The base point x coordinate G_x
- The base point y coordinate G_y

Pseudo-random curve:

- The base point order *n*

Pseudo-random curve (Polynomial Basis representation):

- The coefficient *b*
- The base point x coordinate G_x
- The base point y coordinate G_y

Pseudo-random curve (Normal Basis representation):

- The 160-bit input seed *SEED* to the SHA-1 based algorithm (i.e., the domain parameter seed)
- The coefficient *b* (i.e., the output of the SHA-1 based algorithm)
- The base point *x* coordinate G_x
- The base point *y* coordinate G_y

Integers (such as *T*, *m*, and *n*) are given in decimal form; bit strings and field elements are given in hexadecimal.

D.1.3.1 Degree 163 Binary Field

$T =$ 4

$p(t) = t^{163} + t^7 + t^6 + t^3 + 1$

D.1.3.1.1 Curve K-163

$a =$ 1

$n =$ 5846006549323611672814741753598448348329118574063

Polynomial Basis:

$G_x =$ 2 fe13c053 7bbc11ac aa07d793 de4e6d5e 5c94eee8

$G_y =$ 2 89070fb0 5d38ff58 321f2e80 0536d538 ccdaa3d9

Normal Basis:

$G_x =$ 0 5679b353 caa46825 fea2d371 3ba450da 0c2a4541

$G_y =$ 2 35b7c671 00506899 06bac3d9 dec76a83 5591edb2

D.1.3.1.2 Curve B-163

$n =$ 5846006549323611672814742442876390689256843201587

Polynomial Basis:

$b =$ 2 0a601907 b8c953ca 1481eb10 512f7874 4a3205fd

$G_x =$ 3 f0eba162 86a2d57e a0991168 d4994637 e8343e36

$G_y =$ 0 d51fbc6c 71a0094f a2cdd545 b11c5c0c 797324f1

Normal Basis:

$SEED =$ 85e25bfe 5c86226c db12016f 7553f9d0 e693a268

$b =$ 6 645f3cac f1638e13 9c6cd13e f61734fb c9e3d9fb

$G_x =$ 0 311103c1 7167564a ce77ccb0 9c681f88 6ba54ee8

$G_y =$ 3 33ac13c6 447f2e67 613bf700 9daf98c8 7bb50c7f

D.1.3.2 Degree 233 Binary Field

$T =$ 2

$p(t) = t^{233} + t^{74} + 1$

D.1.3.2.1 Curve K-233

$a =$ 0

$n =$ 34508731733952818937173779311385127605709409888622521\
6328087024741343

Polynomial Basis:

$G_x =$ 172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5
0a4c9d6e efad6126

$G_y =$ 1db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b
56e0c110 56fae6a3

Normal Basis:

$G_x =$ 0fd e76d9dcd 26e643ac 26f1aa90 1aa12978 4b71fc07
22b2d056 14d650b3

$G_y =$ 064 3e317633 155c9e04 47ba8020 a3c43177 450ee036
d6335014 34cac978

D.1.3.2.2 Curve B-233

$n =$ 69017463467905637874347558622770255558398127373450135\
5379383634485463

Polynomial Basis:

$b =$ 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42
81fe115f 7d8f90ad

$G_x =$ 0fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36
f8f8eb73 71fd558b

$G_y =$ 100 6a08a419 03350678 e58528be bf8a0bef f867a7ca
36716f7e 01f81052

Normal Basis:

SEED = 74d59ff0 7f6b413d 0ea14b34 4b20a2db 049b50c3

b = 1a0 03e0962d 4f9a8e40 7c904a95 38163adb 82521260
0c7752ad 52233279

G_x = 18b 863524b3 cdfefb94 f2784e0b 116faac5 4404bc91
62a363ba b84a14c5

G_y = 049 25df77bd 8b8ff1a5 ff519417 822bfedf 2bbd7526
44292c98 c7af6e02

D.1.3.3 Degree 283 Binary Field

T = 6

$p(t) = t^{283} + t^{12} + t^{7} + t^{5} + 1$

D.1.3.3.1 Curve K-283

a = 0

n = 3885337784451458141838923813647037813284811733793061324

295874997529815829704422603873

Polynomial Basis:

G_x = 503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a

16876913 b0c2ac24 58492836

G_y = 1ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698

e4596236 4e341161 77dd2259

Normal Basis:

G_x = 3ab9593 f8db09fc 188f1d7c 4ac9fcc3 e57fcd3b db15024b

212c7022 9de5fcd9 2eb0ea60

G_y = 2118c47 55e7345c d8f603ef 93b98b10 6fe8854f feb9a3b3

04634cc8 3a0e759f 0c2686b1

D.1.3.3.2 Curve B-283

n = 7770675568902916283677847627294075626569625924376904889

109196526770044277787378692871

Polynomial Basis:

b = 27b680a c8b8596d a5a4af8a 19a0303f ca97fd76 45309fa2

a581485a f6263e31 3b79a2f5

G_x = 5f93925 8db7dd90 e1934f8c 70b0dfec 2eed25b8 557eac9c

80e2e198 f8cdbecd 86b12053

G_y = 3676854 fe24141c b98fe6d4 b20d02b4 516ff702 350eddb0

826779c8 13f0df45 be8112f4

Normal Basis:

SEED = 77e2b073 70eb0f83 2a6dd5b6 2dfc88cd 06bb84be

b = 157261b 894739fb 5a13503f 55f0b3f1 0c560116 66331022

01138cc1 80c0206b dafbc951

G_x = 749468e 464ee468 634b21f7 f61cb700 701817e6 bc36a236

4cb8906e 940948ea a463c35d

G_y = 62968bd 3b489ac5 c9b859da 68475c31 5bafcdc4 ccd0dc90

5b70f624 46f49c05 2f49c08c

D.1.3.4 Degree 409 Binary Field

T = 4

$p(t) = t^{409} + t^{87} + 1$

D.1.3.4.1 Curve K-409

a = 0

n = 33052798439512429947595765401638551991420234148214060964\

232439502288071128924919105067325845777745801409636659061

7731358671

Polynomial Basis:

G_x = 060f05f 658f49c1 ad3ab189 0f718421 0efd0987 e307c84c
27accfb8 f9f67cc2 c460189e b5aaaa62 ee222eb1 b35540cf
e9023746

G_y = 1e36905 0b7c4e42 acba1dac bf04299c 3460782f 918ea427
e6325165 e9ea10e3 da5f6c42 e9c55215 aa9ca27a 5863ec48
d8e0286b

Normal Basis:

G_x = 1b559c7 cba2422e 3affe133 43e808b5 5e012d72 6ca0b7e6
a63aeafb c1e3a98e 10ca0fcf 98350c3b 7f89a975 4a8e1dc0
713cec4a

G_y = 16d8c42 052f07e7 713e7490 eff318ba 1abd6fef 8a5433c8
94b24f5c 817aeb79 852496fb ee803a47 bc8a2038 78ebf1c4
99afd7d6

D.1.3.4.2 Curve B-409

n = 66105596879024859895191530803277103982840468296428121928464879830415777482737480520814372376217911096597986728 8366567526771

Polynomial Basis:

b = 021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67
4761fa99 d6ac27c8 a9a197b2 72822f6c d57a55aa 4f50ae31
7b13545f

G_x = 15d4860 d088ddb3 496b0c60 64756260 441cde4a f1771d4d
b01ffe5b 34e59703 dc255a86 8a118051 5603aeab 60794e54
bb7996a7

$G_y =$ 061b1cf ab6be5f3 2bbfa783 24ed106a 7636b9c5 a7bd198d
0158aa4f 5488d08f 38514f1f df4b4f40 d2181b36 81c364ba
0273c706

Normal Basis:

$SEED =$ 4099b5a4 57f9d69f 79213d09 4c4bcd4d 4262210b

$b =$ 124d065 1c3d3772 f7f5a1fe 6e715559 e2129bdf a04d52f7
b6ac7c53 2cf0ed06 f610072d 88ad2fdc c50c6fde 72843670
f8b3742a

$G_x =$ 0ceacbc 9f475767 d8e69f3b 5dfab398 13685262 bcacf22b
84c7b6dd 981899e7 318c96f0 761f77c6 02c016ce d7c548de
830d708f

$G_y =$ 199d64b a8f089c6 db0e0b61 e80bb959 34afd0ca f2e8be76
d1c5e9af fc7476df 49142691 ad303902 88aa09bc c59c1573
aa3c009a

D.1.3.5 Degree 571 Binary Field

$T =$ 10

$p(t) = t^{571} + t^{10} + t^{5} + t^{2} + 1$

D.1.3.5.1 Curve K-571

$a =$ 0

$n =$ 19322687615086291723476759454659936721494636648532174993286176257257595711447802122681339785227067118347067128008253514612736749740666173119296824216170925035557336852
76673

Polynomial Basis:

$G_x =$ 26eb7a8 59923fbc 82189631 f8103fe4 ac9ca297 0012d5d4
60248048 01841ca4 43709584 93b205e6 47da304d b4ceb08c

bbd1ba39 494776fb 988b4717 4dca88c7 e2945283 a01c8972

$G_y =$ 349dc80 7f4fbf37 4f4aeade 3bca9531 4dd58cec 9f307a54
ffc61efc 006d8a2c 9d4979c0 ac44aea7 4fbebbb9 f772aedc
b620b01a 7ba7af1b 320430c8 591984f6 01cd4c14 3ef1c7a3

Normal Basis:

$G_x =$ 04bb2db a418d0db 107adae0 03427e5d 7cc139ac b465e593
4f0bea2a b2f3622b c29b3d5b 9aa7a1fd fd5d8be6 6057c100
8e71e484 bcd98f22 bf847642 37673674 29ef2ec5 bc3ebcf7

$G_y =$ 44cbb57 de20788d 2c952d7b 56cf39bd 3e89b189 84bd124e
751ceff4 369dd8da c6a59e6e 745df44d 8220ce22 aa2c852c
fcbbef49 ebaa98bd 2483e331 80e04286 feaa2530 50caff60

D.1.3.5.2 Curve B-571

$n =$ 3864537523017258344695351890931987344298927329706434998657235251451519142289560424536143999389415773083133881121926944486246872462816813070234528288303332411393191105285703

Polynomial Basis:

$b =$ 2f40e7e 2221f295 de297117 b7f3d62f 5c6a97ff cb8ceff1
cd6ba8ce 4a9a18ad 84ffabbd 8efa5933 2be7ad67 56a66e29
4afd185a 78ff12aa 520e4de7 39baca0c 7ffeff7f 2955727a

$G_x =$ 303001d 34b85629 6c16c0d4 0d3cd775 0a93d1d2 955fa80a
a5f40fc8 db7b2abd bde53950 f4c0d293 cdd711a3 5b67fb14
99ae6003 8614f139 4abfa3b4 c850d927 e1e7769c 8eec2d19

$G_y =$ 37bf273 42da639b 6dccfffe b73d69d7 8c6c27a6 009cbbca
1980f853 3921e8a6 84423e43 bab08a57 6291af8f 461bb2a8
b3531d2f 0485c19b 16e2f151 6e23dd3c 1a4827af 1b8ac15b

Normal Basis:

SEED =
```
2aa058f7 3a0e33ab 486b0f61 0410c53a 7f132310
```

b =
```
 3762d0d 47116006 179da356 88eeaccf 591a5cde a7500011
8d9608c5 9132d434 26101a1d fb377411 5f586623 f75f0000
1ce61198 3c1275fa 31f5bc9f 4be1a0f4 67f01ca8 85c74777
```

G_x =
```
 0735e03 5def5925 cc33173e b2a8ce77 67522b46 6d278b65
0a291612 7dfea9d2 d361089f 0a7a0247 a184e1c7 0d417866
e0fe0feb 0ff8f2f3 f9176418 f97d117e 624e2015 df1662a8
```

G_y =
```
 04a3642 0572616c df7e606f ccadaecf c3b76dab 0eb1248d
d03fbdfc 9cd3242c 4726be57 9855e812 de7ec5c5 00b4576a
24628048 b6a72d88 0062eed0 dd34b109 6d3acbb6 b01a4a97
```

D.2 Implementation of Modular Arithmetic

The prime moduli in the above examples are of a special type (called *generalized Mersenne numbers*) for which modular multiplication can be carried out more efficiently than in general. This section provides the rules for implementing this faster arithmetic for each of the prime moduli appearing in the examples.

The usual way to multiply two integers (mod m) is to take the integer product and reduce it (mod m). One therefore has the following problem: given an integer A less than m^2, compute

$$B = A \bmod m.$$

In general, one must obtain B as the remainder of an integer division. If m is a generalized Mersenne number, however, then B can be expressed as a sum or difference (mod m) of a small number of terms. To compute this expression, the integer sum or difference can be evaluated and the result reduced modulo m. The latter reduction can be accomplished by adding or subtracting a few copies of m.

The prime modulus p for each of the five example curves is a generalized Mersenne number.

D.2.1 Curve P-192

The modulus for this curve is $p = 2^{192} - 2^{64} - 1$. Every integer A less than p^2 can be written as

$$A = A_5 \cdot 2^{320} + A_4 \cdot 2^{256} + A_3 \cdot 2^{192} + A_2 \cdot 2^{128} + A_1 \cdot 2^{64} + A_0,$$

where each A_i is a 64-bit integer. As a concatenation of 64-bit words, this can be denoted by

$$A = (A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_0).$$

The expression for B is

$$B = T + S_1 + S_2 + S_3 \bmod p,$$

where the 192-bit terms are given by

$T =$ $(A_2 \parallel A_1 \parallel A_0)$

$S_1 =$ $(A_3 \parallel A_3)$

$S_2 =$ $(A_4 \parallel A_4 \parallel 0)$

$S_3 =$ $(A_5 \parallel A_5 \parallel A_5)$.

D.2.2 Curve P-224

The modulus for this curve is $p = 2^{224} - 2^{96} + 1$. Every integer A less than p^2 can be written as:

$$A = A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} + A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by:

$$A = (A_{13} \parallel A_{12} \parallel \ldots \parallel A_0).$$

The expression for B is:

$$B = T + S_1 + S_2 - D_1 - D_2 \bmod p,$$

where the 224-bit terms are given by:

$T =$ $(A_6 \parallel A_5 \parallel A_4 \parallel A_3 \parallel A_2 \parallel A_1 \parallel A_0)$

$S_1 =$ $(A_{10} \parallel A_9 \parallel A_8 \parallel A_7 \parallel 0 \parallel 0 \parallel 0)$

$S_2 =$ $(0 \parallel A_{13} \parallel A_{12} \parallel A_{11} \parallel 0 \parallel 0 \parallel 0)$

$D_1 =$ $(A_{13} \parallel A_{12} \parallel A_{11} \parallel A_{10} \parallel A_9 \parallel A_8 \parallel A_7)$

$D_2 =$ $(0 \parallel 0 \parallel 0 \parallel 0 \parallel A_{13} \parallel A_{12} \parallel A_{11})$.

D.2.3 Curve P-256

The modulus for this curve is $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. Every integer A less than p^2 can be written as:

$$A = A_{15} \cdot 2^{480} + A_{14} \cdot 2^{448} + A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} + A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by

$$A = (A_{15} \parallel A_{14} \parallel \cdots \parallel A_0).$$

The expression for B is:

$$B = T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4 \bmod p,$$

where the 256-bit terms are given by:

$T = (A_7 \,||\, A_6 \,||\, A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_1 \,||\, A_0)$

$S_1 = (A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{11} \,||\, 0 \,||\, 0 \,||\, 0)$

$S_2 = (0 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, 0 \,||\, 0 \,||\, 0)$

$S_3 = (A_{15} \,||\, A_{14} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{10} \,||\, A_9 \,||\, A_8)$

$S_4 = (A_8 \,||\, A_{13} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{11} \,||\, A_{10} \,||\, A_9)$

$D_1 = (A_{10} \,||\, A_8 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{13} \,||\, A_{12} \,||\, A_{11})$

$D_2 = (A_{11} \,||\, A_9 \,||\, 0 \,||\, 0 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12})$

$D_3 = (A_{12} \,||\, 0 \,||\, A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13})$

$D_4 = (A_{13} \,||\, 0 \,||\, A_{11} \,||\, A_{10} \,||\, A_9 \,||\, 0 \,||\, A_{15} \,||\, A_{14})$

D.2.4 Curve P-384

The modulus for this curve is $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$. Every integer A less than p^2 can be written as:

$$A = A_{23} \cdot 2^{736} + A_{22} \cdot 2^{704} + A_{21} \cdot 2^{672} + A_{20} \cdot 2^{640} + A_{19} \cdot 2^{608} + A_{18} \cdot 2^{576} + A_{17} \cdot 2^{544} + A_{16} \cdot 2^{512} +$$
$$A_{15} \cdot 2^{480} + A_{14} \cdot 2^{448} + A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} + A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} +$$
$$A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by

$$A = (A_{23} \,||\, A_{22} \,||\, \cdots \,||\, A_0).$$

The expression for B is:

$$B = T + 2S_1 + S_2 + S_3 + S_4 + S_5 + S_6 - D_1 - D_2 - D_3 \bmod p,$$

where the 384-bit terms are given by:

$T = (A_{11} \,||\, A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_7 \,||\, A_6 \,||\, A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_1 \,||\, A_0)$

$S_1 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0)$

$S_2 = (A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, A_{20} \,||\, A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12})$

$S_3 = (A_{20} \,||\, A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{23} \,||\, A_{22} \,||\, A_{21})$

$S_4 = (A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{20} \,||\, 0 \,||\, A_{23} \,||\, 0)$

$S_5 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, A_{20} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0)$

$S_6 =$ (0 || 0 || 0 || 0 || 0 || 0 || A_{23} || A_{22} || A_{21} || 0 || 0 || A_{20})

$D_1 =$ (A_{22} || A_{21} || A_{20} || A_{19} || A_{18} || A_{17} || A_{16} || A_{15} || A_{14} || A_{13} || A_{12} || A_{23})

$D_2 =$ (0 || 0 || 0 || 0 || 0 || 0 || 0 || A_{23} || A_{22} || A_{21} || A_{20} || 0)

$D_3 =$ (0 || 0 || 0 || 0 || 0 || 0 || 0 || A_{23} || A_{23} || 0 || 0 || 0).

D.2.5 Curve P-521

The modulus for this curve is $p = 2^{521} - 1$. Every integer A less than p^2 can be written

$$A = A_1 \cdot 2^{521} + \mathrm{A}_0,$$

where each A_i is a 521-bit integer. As a concatenation of 521-bit words, this can be denoted by

$$A = (A_1 \,||\, A_0).$$

The expression for B is:

$$B = (A_0 + A_1) \bmod p.$$

D.3 Normal Bases

The elements of $GF(2^m)$ are expressed in terms of the type T normal *basis*[5] B for $GF(2^m)$, for some T. Each element has a unique representation as a bit string:

$$(\, a_0\, a_1 \;\ldots\; a_{m-1}\,).$$

The arithmetic operations are performed as follows.

Addition: addition of two elements is implemented by bit-wise addition modulo 2. Thus, for example,

$$(1100111) + (1010010) = (0110101).$$

Squaring: if

$$\alpha = (\, a_0\, a_1 \;\ldots\; a_{m-1}\,)$$

then

$$\alpha^2 = (a_{m-1}\, a_0\, a_1 \;\ldots\; a_{m-2}\,).$$

Multiplication: to perform multiplication, a function $F(\underline{u},\underline{v})$ is constructed on inputs

[5] It is assumed in this section that m is odd and T is even, since this is the only case considered in this Standard.

$$\underline{u} = (\, u_0\, u_1 \ldots\; u_{m-1} \,) \qquad \text{and} \qquad \underline{v} = (\, v_0\, v_1 \ldots\; v_{m-1} \,)$$

as follows.

```
1. Set $p \leftarrow Tm + 1$.
2. Let $u$ be an integer having order $T$ modulo $p$.
3. Compute the sequence $F(1), F(2), \ldots, F(p-1)$ as follows:
   3.1 Set $w \leftarrow 1$.
   3.2 For $j$ from 0 to $T-1$ do
       3.2.1 Set $n \leftarrow w$.
       3.2.2 For $i = 0$ to $m-1$ do
             3.2.2.1 Set $F(n) \leftarrow i$.
             3.2.2.2 Set $n \leftarrow 2n \bmod p$.
       3.2.3 Set $w \leftarrow uw \bmod p$.
4. Output the formula:
             $F(u,v) := \sum_{k=1}^{p-2} u_{F(k+1)} v_{F(p-k)}$.
```

This computation need only be performed once per basis.

Given the function F for B, the product

$$(\, c_0\, c_1 \ldots\; c_{m-1} \,) = (\, a_0\, a_1 \ldots\; a_{m-1} \,) * (\, b_0\, b_1 \ldots\; b_{m-1} \,)$$

is computed as follows:

```
1. Set $( u_0\, u_1 \ldots u_{m-1} ) \leftarrow ( a_0\, a_1 \ldots a_{m-1} )$.
2. Set $( v_0\, v_1 \ldots v_{m-1} ) \leftarrow ( b_0\, b_1 \ldots b_{m-1} )$.
3. For $k = 0$ to $m - 1$ do
   3.1 Compute
             $c_k = \mathrm{F}(\underline{u}, \underline{v})$.
   3.2 Set $u \leftarrow$ **LeftShift** $(u)$ and $v \leftarrow$ **LeftShift** $(v)$, where **LeftShift** denotes the circular
       left shift operation.
4. Output $c = ( c_0\, c_1 \ldots c_{m-1} )$.
```

Example: For the type 4 normal basis for $GF(2^7)$, $p = 29$ and $u = 12$ or 17. Thus, the values of F are given by:

$F(1) = 0$	$F(8) = 3$	$F(15) = 6$	$F(22) = 5$
$F(2) = 1$	$F(9) = 3$	$F(16) = 4$	$F(23) = 6$
$F(3) = 5$	$F(10) = 2$	$F(17) = 0$	$F(24) = 1$
$F(4) = 2$	$F(11) = 4$	$F(18) = 4$	$F(25) = 2$
$F(5) = 1$	$F(12) = 0$	$F(19) = 2$	$F(26) = 5$
$F(6) = 6$	$F(13) = 4$	$F(20) = 3$	$F(27) = 1$
$F(7) = 5$	$F(14) = 6$	$F(21) = 3$	$F(28) = 0$

Therefore,

$$F(\underline{u}, \underline{v}) = u_0 v_1 + u_1 (v_0 + v_2 + v_5 + v_6) + u_2 (v_1 + v_3 + v_4 + v_5) + u_3 (v_2 + v_5) + \\ u_4 (v_2 + v_6) + u_5 (v_1 + v_2 + v_3 + v_6) + u_6 (v_1 + v_4 + v_5 + v_6).$$

Thus, if

$$a = (1\ 0\ 1\ 0\ 1\ 1\ 1) \text{ and } b = (1\ 1\ 0\ 0\ 0\ 0\ 1),$$

then

$$c_0 = F(\ (1\ 0\ 1\ 0\ 1\ 1\ 1), (1\ 1\ 0\ 0\ 0\ 0\ 1)\ ) = 1,$$
$$c_1 = F(\ (0\ 1\ 0\ 1\ 1\ 1\ 1), (1\ 0\ 0\ 0\ 0\ 1\ 1)\ ) = 0,$$
$$\vdots$$
$$c_6 = F(\ (1\ 1\ 0\ 1\ 0\ 1\ 1), (1\ 1\ 1\ 0\ 0\ 0\ 0)\ ) = 1,$$

so that $c = ab = (1\ 0\ 1\ 1\ 0\ 0\ 1)$.

D.4 Scalar Multiplication on Koblitz Curves

This section describes a particularly efficient method of computing the scalar multiple nP on the Koblitz curve E_a over $GF(2^m)$.

The operation τ is defined by:

$$\tau(x, y) = (x^2, y^2).$$

When the normal basis representation is used, then the operation τ is implemented by performing right circular shifts on the bit strings representing x and y.

Given m and a, define the following parameters:

- C is some integer greater than 5.

- $\mu = (-1)^{1-a}$.
- For $i = 0$ and $i = 1$, define the sequence $s_i(m)$ by:

$$s_i(0) = 0, \quad s_i(1) = 1 - i,$$

$$s_i(m) = \mu \bullet \ s_i(m-1) - 2\, s_i(m-2) + (-1)^i$$

- Define the sequence $V(m)$

$$V(0) = 2, \quad V(1) = \mu$$

$$V(m) = \mu \bullet v(m-1) - 2V(m-2).$$

For the example curves, the quantities $s_i(m)$ and $V(m)$ are as follows.

Curve K-163:

$s_0(163) =$ 2579386439110731650419537

$s_1(163) =$ –755360064476226375461594

$V(163) =$ –4845466632539410776804317

Curve K-233:

$s_0(233) =$ –27859711741434429761757834964435883

$s_1(233) =$ –44192136247082304936052160908934886

$V(233) =$ –137381546011108235394987299651366779

Curve K-283:

$s_0(283) =$ –665981532109049041108795536001591469280025

$s_1(283) =$ 1155860054909136775192281072591609913945968

$V(283) =$ 7777244870872830999287791970962823977569917

Curve K-409:

$s_0(409) =$ –18307510456002382137810317198756461378590542487556869338419259

$s_1(409) =$ –8893048526138304097196653241844212679626566100996606444816790

$V(409) =$ 10457288737315625927447685387048320737638796957687575791173829

Curve K-571:

$s_0(571) =$ –3737319446876463692429385892476115567147293964596131024123406420\
2352419167299832613 05

$s_1(571) =$ –3191857706446416099583814595948959674131968912148564658610565117\
58982848515832612248752

$V(571) = -1483809269816914138996191402970514903645425741804939362329123395\backslash$

3420851682897311459843

The following algorithm computes the scalar multiple nP on the Koblitz curve E_a over $GF(2^m)$. The average number of elliptic additions and subtractions is at most ~ $1 + (m/3)$, and is at most ~ $m/3$ with probability at least $1 - 2^{5-C}$.

```
1. For $i = 0$ to 1 do
   1.1  $n' \leftarrow \lfloor n / 2^{a-C+(m-9)/2} \rfloor$.
   1.2  $g' \leftarrow s_i(m) \cdot n'$.
   1.3  $h' \leftarrow \lfloor g' / 2^m \rfloor$.
   1.4  $j' \leftarrow V(m) \cdot h'$.
   1.5  $l' \leftarrow \text{Round}((g' + j') / 2^{(m+5)/2})$.
   1.6  $\lambda_i \leftarrow l' / 2^C$.
   1.7  $f_i \leftarrow \text{Round}(\lambda_i)$.
   1.8  $\eta_i \leftarrow \lambda_i - f_{i.}$.
   1.9  $h_i \leftarrow 0$.
2. $\eta \leftarrow 2\,\eta_0 + \mu\,\eta_1$.
3. If $(\eta \geq 1)$,
   then
       if $(\eta_o - 3\,\mu\eta_1 < -1)$
           then set $h_1 \leftarrow \mu$
           else set $h_0 \leftarrow 1$.
   else
       if $(\eta_0 + 4\,\mu\,\eta_1 \geq 2)$
           then set $h_1 \leftarrow \mu$.
4. If $(\eta < -1)$
         then
       if $(\eta_0 - 3\,\mu\,\eta_1 \geq 1)$
           then set $h_1 \leftarrow -\mu$
           else set $h_0 \leftarrow -1$.
```

```
else
    if ($\eta_0 + 4\,\mu\,\eta_1 < -2$)
        then set $h_1 \leftarrow -\mu$.
5.  $q_0 \leftarrow f_0 + h_0$.
6.  $q_1 \leftarrow f_1 + h_1$.
7.  $r_0 \leftarrow n - (s_0 + \mu\,s_1)\,q_0 - 2s_1\,q_1$.
8.  $r_1 \leftarrow s_1\,q_0 - s_0\,q_1$.
9.  Set $Q \leftarrow O$.
10. $P_0 \leftarrow P$.
11. While (($r_0 \neq 0$) or ($r_1 \neq 0$))
    11.1 If ($r_0$ odd), then
        11.1.1 set $u \leftarrow 2 - (r_0 - 2\,r_1 \bmod 4)$.
        11.1.2 set $r_0 \leftarrow r_0 - u$.
        11.1.3 if ($u = 1$), then set $Q \leftarrow Q + P_0$.
        11.1.4 if ($u = -1$), then set $Q \leftarrow Q - P_0$.
    11.2 Set $P_0 \leftarrow \tau P_0$.
    11.3 Set $(r_0, r_1) \leftarrow (r_1 + \mu r_0/2, -r_0/2)$.
        Endwhile
12. Output $Q$.
```

D.5 Generation of Pseudo-Random Curves (Prime Case)

Let *l* be the bit length of *p*, and define

$$v = \lfloor (l - 1)/160 \rfloor$$

$$w = l - 160v - 1.$$

1. Choose an arbitrary 160-bit string *s* as the domain parameter seed.
2. Compute $h = \text{SHA-1}(s)$.
3. Let h_0 be the bit string obtained by taking the *w* rightmost bits of *h*.
4. Let *z* be the integer whose binary expansion is given by the 160-bit string *s*.
5. For *i* from 1 to *v* do:

5.1 Define the 160-bit string s_i to be binary expansion of the integer

$(z + i) \bmod (2^{160})$.

5.2 Compute $h_i = \text{SHA-1}(s_i)$.

6. Let h be the bit string obtained by the concatenation of h_0, h_1, . . . , h_v as follows:

$$h = h_0 \;||\; h_1 \;||\; \ldots \;||\; h_v.$$

7. Let c be the integer whose binary expansion is given by the bit string h.

8. If (($c = 0$ or $4c + 27 \equiv 0 \pmod{p}$)), then go to Step 1.

9. Choose integers $a, b \in GF(p)$ such that

$$c\, b^2 \equiv a^3 \pmod{p}.$$

(The simplest choice is $a = c$ and $b = c$. However, one may want to choose differently for performance reasons.)

10. Check that the elliptic curve E over $GF(p)$ given by $y^2 = x^3 + ax + b$ has suitable order. If not, go to Step 1.

D.6 Verification of Curve Pseudo-Randomness (Prime Case)

Given the 160-bit domain parameter seed value s, verify that the coefficient b was obtained from s via the cryptographic hash function SHA-1 as follows.

Let l be the bit length of p, and define

$$v = \lfloor (l - 1)/160 \rfloor$$

$$w = l - 160v - 1.$$

1. Compute $h = \text{SHA-1}(s)$.

2. Let h_0 be the bit string obtained by taking the w rightmost bits of h.

3. Let z be the integer whose binary expansion is given by the 160-bit string s.

4. For $i = 1$ to v do

   4.1 Define the 160-bit string s_i to be binary expansion of the integer

   $(z + i) \bmod (2^{160})$.

   4.2 Compute $h_i = \text{SHA-1}(s_i)$.

5. Let h be the bit string obtained by the concatenation of h_0, h_1, . . . , h_v as follows:

$$h = h_0 \;||\; h_1 \;||\; \ldots \;||\; h_v.$$

6. Let c be the integer whose binary expansion is given by the bit string h.

7. Verify that $b^2 c \equiv -27 \pmod{p}$.

D.7 Generation of Pseudo-Random Curves (Binary Case)

Let:

$$v = \lfloor (m-1)/B \rfloor$$

$$w = m - Bv.$$

1. Choose an arbitrary 160-bit string s as the domain parameter seed.
2. Compute h = SHA-1(s).
3. Let h_0 be the bit string obtained by taking the w rightmost bits of h.
4. Let z be the integer whose binary expansion is given by the 160-bit string s.
5. For i from 1 to v do:
   - 5.1 Define the 160-bit string s_i to be binary expansion of the integer $(z + i) \bmod (2^{160})$.
   - 5.2 Compute h_i = SHA-1(s_i).
6. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \parallel h_1 \parallel \ldots \parallel h_v.$$

7. Let b be the element of $GF(2^m)$ which binary expansion is given by the bit string h.
8. Choose an element a of $GF(2^m)$.
9. Check that the elliptic curve E over $GF(2^m)$ given by $y^2 + xy = x^3 + ax^2 + b$ has suitable order. If not, go to Step 1.

D.8 Verification of Curve Pseudo-Randomness (Binary Case)

Given the 160-bit domain parameter seed value s, verify that the coefficient b was obtained from s via the cryptographic hash function SHA-1 as follows.

Define

$$v = \lfloor (m-1)/160 \rfloor$$

$$w = m - 160v$$

1. Compute h = SHA-1(s).
2. Let h_0 be the bit string obtained by taking the w rightmost bits of h.

3. Let z be the integer whose binary expansion is given by the 160-bit string s.
4. For i = 1 to v do

   4.1 Define the 160-bit string s_i to be binary expansion of the integer $(z + i) \bmod (2^{160})$.

   4.2 Compute h_i = SHA-1(s_i).

5. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \parallel h_1 \parallel \ldots \parallel h_v.$$

6. Let c be the element of $GF(2^m)$ which is represented by the bit string h.
7. Verify that $c = b$.

D.9 Polynomial Basis to Normal Basis Conversion

Suppose that α is an element of the field $GF(2^m)$. Let p be the bit string representing α with respect to a given polynomial basis. It is desired to compute n, the bit string representing α with respect to a given normal basis. This is done via the matrix computation

$$p\,\Gamma = n,$$

where Γ is an m-by-m matrix with entries in $GF(2)$. The matrix Γ, which depends only on the bases, can be computed easily given its second-to-last row. The second-to-last row for each conversion is given the below.

Degree 163:

```
         3 e173bfaf 3a86434d 883a2918 a489ddbd 69fe84e1
```

Degree 233:

```
     0be 19b89595 28bbc490 038f4bc4 da8bdfc1 ca36bb05 853fd0ed
0ae200ce
```

Degree 283:

```
 3347f17 521fdabc 62ec1551 acf156fb 0bceb855 f174d4c1 7807511c
9f745382 add53bc3
```

Degree 409:

```
 0eb00f2 ea95fd6c 64024e7f 0b68b81f 5ff8a467 acc2b4c3 b9372843
6265c7ff a06d896c ae3a7e31 e295ec30 3eb9f769 de78bef5
```

Degree 571:

```
 7940ffa ef996513 4d59dcbf e5bf239b e4fe4b41 05959c5d 4d942ffd
46ea35f3 e3cdb0e1 04a2aa01 cef30a3a 49478011 196bfb43 c55091b6
1174d7c0 8d0cdd61 3bf6748a bad972a4
```

Given the second-to-last row r of Γ, the rest of the matrix is computed as follows. Let β be the element of $GF(2^m)$ whose representation with respect to the normal basis is r. Then the rows of Γ, from top to bottom, are the bit strings representing the elements

$$\beta^{m-1}, \beta^{m-2}, \ldots, \beta^2, \beta, 1$$

with respect to the normal basis. (Note that the element 1 is represented by the all-1 bit string.)

Alternatively, the matrix is the inverse of the matrix described in Appendix D.10.

More details of these computations can be found in Annex A.7 of the IEEE Standard 1363-2000 standard.

D.10 Normal Basis to Polynomial Basis Conversion

Suppose that α is an element of the field $GF(2^m)$. Let n be the bit string representing α with respect to a given normal basis. It is desired to compute p, the bit string representing α with respect to a given polynomial basis. This is done via the matrix computation

$$n\,\Gamma = p,$$

where Γ is an m-by-m matrix with entries in $GF(2)$. The matrix Γ, which depends only on the bases, can be computed easily given its top row. The top row for each conversion is given below.

Degree 163:

```
      7 15169c10 9c612e39 0d347c74 8342bcd3 b02a0bef
```

Degree 233:

```
    149 9e398ac5 d79e3685 59b35ca4 9bb7305d a6c0390b cf9e2300
253203c9
```

Degree 283:

```
 31e0ed7 91c3282d c5624a72 0818049d 053e8c7a b8663792 bc1d792e
ba9867fc 7b317a99
```

Degree 409:

```
 0dfa06b e206aa97 b7a41fff b9b0c55f 8f048062 fbe8381b 4248adf9
2912ccc8 e3f91a24 e1cfb395 0532b988 971c2304 2e85708d
```

Degree 571:

```
 452186b bf5840a0 bcf8c9f0 2a54efa0 4e813b43 c3d41496 06c4d27b
487bf107 393c8907 f79d9778 beb35ee8 7467d328 8274caeb da6ce05a
eb4ca5cf 3c3044bd 4372232f 2c1a27c4
```

Given the top row r of Γ, the rest of the matrix is computed as follows. Let β be the element of

$GF(2^{m})$ whose representation with respect to the polynomial basis is r. Then the rows of Γ, from top to bottom, are the bit strings representing the elements

$$\beta, \beta^{2}, \beta^{2^{2}}, \ldots, \beta^{2^{m-1}}$$

with respect to the polynomial basis.

Alternatively, the matrix is the inverse of the matrix described in Appendix D.9.

More details of these computations can be found in Annex A.7 of the IEEE Std 1363-2000 standard.

Appendix E: A Proof that *v* = *r* in the DSA (Informative)

The purpose of this appendix is to show that if $M' = M$, $r' = r$ and $s' = s$ in the signature verification, then $v = r'$. Let **Hash** be an approved hash function. The following result is needed.

Lemma: Let p and q be primes such that q divides $(p - 1)$, let h be a positive integer less than p, and let $g = (h^{(p-1)/q} \bmod p)$. Then $(g^q \bmod p) = 1$, and if $(m \bmod q) = (n \bmod q)$, then $(g^m \bmod p) = (g^n \bmod p)$.

Proof:

$$\begin{aligned} g^q \bmod p &= (h^{(p-1)/q} \bmod p)^q \bmod p \\ &= h^{(p-1)} \bmod p \\ &= 1 \end{aligned}$$

by Fermat's Little Theorem. Now let $(m \bmod q) = (n \bmod q)$, i.e., $m = (n + kq)$ for some integer k. Then

$$\begin{aligned} g^m \bmod p &= g^{n+kq} \bmod p \\ &= (g^n g^{kq}) \bmod p \\ &= ((g^n \bmod p)\,(g^q \bmod p)^k) \bmod p \\ &= g^n \bmod p, \end{aligned}$$

since $(g^q \bmod p) = 1$.

Proof of the main result:

Theorem: If $M' = M$, $r' = r$, and $s' = s$ in the signature verification, then $v = r'$.

Proof:

$$w = (s')^{-1} \bmod q = s^{-1} \bmod q$$

$$u1 = ((\mathbf{Hash}(M'))w) \bmod q = ((\mathbf{Hash}(M))w) \bmod q$$

$$u2 = ((r')w) \bmod q = (rw) \bmod q.$$

Now $y = (g^x \bmod p)$, so that by the lemma,

$$\begin{aligned} v &= ((g^{u1}\, y^{u2}) \bmod \mathrm{p}) \bmod q \\ &= ((g^{\mathbf{Hash}(M)w}\, y^{rw}) \bmod p) \bmod q \\ &= ((g^{\mathbf{Hash}(M)w}\, g^{xrw}) \bmod p) \bmod q \\ &= ((g^{(\mathbf{Hash}(M)\,+\,xr)w)} \bmod p) \bmod q. \end{aligned}$$

Also:

$$s = (k^{-1}\ (\mathbf{Hash}(M) + xr)) \bmod q.$$

Hence:

$$w = (k\ (\mathbf{Hash}(M) + xr)^{-1}) \bmod q$$

$$(\mathbf{Hash}(M) + xr)w \bmod q = k \bmod q.$$

Thus, by the lemma:

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

Appendix F: Calculating the Required Number of Rounds of Testing Using the Miller-Rabin Probabilistic Primality Test

(Informative)

F.1 The Required Number of Rounds of the Miller-Rabin Primality Tests

The ideas of paper [1] were applied to estimate $p_{k,t}$, the probability that an odd k-bit integer that passes t rounds of Miller-Rabin (M-R) testing is actually composite. The probability $p_{k,t}$ is understood as the ratio of the number of odd composite numbers of a binary length k that can be expected to pass t rounds of M-R testing (with randomly generated bases) to the sum of that value and the number of odd prime integers of binary length k. This is equivalent to assuming that candidates selected for testing will be chosen uniformly at random from the entire set of odd k-bit integers. Following Pomerance, et al., $p_{k,t}$ can be (over) estimated by the ratio of the expected number of odd composite numbers of binary length k that will pass t rounds of M-R testing (with randomly generated bases) to the total number of odd primes of binary length k. From the perspective of a party charged with the responsibility of generating a k-bit prime, the objective is to determine a value of t such that $p_{k,t}$ is no greater than an acceptably small target value $p_{t\arg et}$.

Using [1], it is possible to compute an upper bound for $p_{k,t}$ as a function of k and t. From this, an upper bound can be computed for t as a function of k and p_{target}, the maximum allowed probability of accidentally generating a composite number. The following is an algorithm for computing t:

```
1. For $t = 1, 2 \ldots \lceil -\log_2(p_{target})/2 \rceil$
   1.1 For $M = 3, 4 \ldots \lfloor 2\sqrt{k-1} - 1 \rfloor$          (1)
       1.1.1 Compute $p_{k,t}$ as in (2).
       1.1.2 If $p_{k,t} \le p_{target}$
             1.1.2.1 Accept $t$.
             1.1.2.2 Stop.
```

In (1), k is the bit length of the candidate primes and (2) is as follows:

$$p_{k,t} = 2.00743 \cdot \ln(2) \cdot k \cdot 2^{-k} \left[2^{k-2-Mt} + \frac{8(\pi^2-6)}{3} 2^{k-2} \sum_{m=3}^{M} 2^{m-(m-1)t} \sum_{j=2}^{m} \frac{1}{2^{\left(j+\frac{(k-1)}{j}\right)}} \right] . \qquad (2)$$

Using this expression for t, the following methodologies are used for testing the DSA and RSA candidate primes.

F.2 Generating DSA Primes

For DSA, the maximum possible care must be taken when generating the primes p and q that are used for the domain parameters. The same primes p and q are used by many parties. This means that any weakness that these numbers may possess would affect multiple users. It also means that the primes are not generated very often; typically, an entire system uses the same set of domain parameters for an extended period of time. Therefore, in this case, some additional care is called for.

With this in mind, it may be too optimistic to simply subject candidate primes to t rounds of M-R testing, where the minimal acceptable value for t is determined according to (1) and (2) in Appendix F.1. This might be the case, for example, if there is a reason to doubt that the assumptions made in [1] have been satisfied during the process of selecting candidates for primality testing. One may gain more confidence in the process by performing some additional (different) primality test(s) on the candidates that survive the M-R testing. As another option, one could, of course, perform additional rounds of M-R testing. These considerations lead to the following alternatives: either (A) use the number of rounds of M-R testing determined according to (1) and (2) in Appendix F.1, and follow that with a single Lucas test (as recommended in ANS X9.31), or (B) use a (much) more conservative approach when determining t (e.g., as described below) and subject candidate primes to additional rounds of M-R testing.

One approach for strategy (B) would be to adopt the viewpoint of the majority of system users, who have no part in generating the (supposed) prime, but who must rely upon its primality for their security. Such parties may be concerned that the candidates for M-R testing have been selected in a fashion that deviates significantly from the uniform distribution – which was assumed when determining t according to (1) and (2) in Appendix F.1. In cases where the selection process could be unusually biased in some way, it is important to minimize the probability that a composite number will survive testing. It can be shown that for any k-bit odd composite number (regardless of how it was selected), the probability that it will pass t rounds of M-R testing with randomly chosen bases is less than 4^{-t} (although this is not a particularly tight bound). Selecting t such that $4^{-t} \le p_{target}$ is equivalent to choosing $t \ge -\log_2(p_{target})/2$. To ensure that a composite number has a probability no greater than p_{target} of surviving the M-R tests, the number of rounds can be set at $t = \lceil -\log_2(p_{target})/2 \rceil$. Even if the method of selecting candidates were so biased that it offered nothing but composite numbers for testing, it is reasonable to expect that it would take at least $1/p_{target}$ attempts (which is greater than 4^t) before a composite number would slip through the t-round M-R testing process.

WARNING: As the discussion above illustrates, care must be taken when using the phrase “error probability” in connection with the recommended number of rounds of M-R testing. The probability that a composite number survives t rounds of Miller-Rabin testing is <u>not</u> the same as $p_{k,t}$, which is the probability that a number surviving t rounds of Miller-Rabin testing is composite. Ordinarily, the latter probability is the one that should be of most interest to a party responsible for generating primes, while the former may be more important to a party responsible for validating the primality of a number generated by someone else. However, for sufficiently large k (e.g., $k \geq 51$), it can be shown that $p_{k,t} \leq 4^{-t}$ under the same assumptions concerning the selection of candidates as those made to obtain formula (2) in Appendix F.1 (see [1].) In such cases, $t = \lceil -\log_2(p_{target})/2 \rceil$ rounds of Miller-Rabin testing can be used both in generating and validating primes, with p_{target} serving as an upper bound on both the probability that the generation process yields a composite number and the probability that a composite number would survive an attempt to validate its primality.

Table C.1 in Appendix C.3 identifies the minimum values for t when generating the primes p and q for DSA using either strategy (A) or (B) above. To obtain the t values shown in the column titled "M-R Tests Only", the conservative strategy (B) was followed; those t values are sufficient to validate the primality of p and q. The t values shown in the column titled "M-R Tests when followed by One Lucas Test" result from following strategy (A) using computations (1) and (2) in Appendix F.1.

F.3 Generating Primes for RSA Signatures

When generating primes for the RSA signature algorithm, it is still very important to reduce the probability of errors in the M-R testing procedure. However, since the (probable) primes are used to generate a user's key pair, if a composite number survives the testing process, the consequences of the error may be less dramatic than in the case of generating DSA domain parameters; only one user's transactions are affected, rather than a domain of users. Furthermore, if the p or q value generated for some user is composite, the problem will not be undiscovered for long, since it is almost certain that signatures generated by that user will not be verifiable.

Therefore, when generating the RSA primes p and q, it is sufficient to use the number of rounds derived from (1) and (2) in Appendix F.1 as the minimum number of M-R tests to be performed. However, if the definition of $p_{k,\,t}$ is not considered to be sufficiently conservative when testing p and q, it is recommended that the t rounds of Miller-Rabin tests be followed by a single Lucas test.

The lengths for p and q that are recommended for use in RSA signature algorithms are 512, 1024 and 1536 bits; recall that $n = pq$, so the corresponding lengths for n are 1024, 2048 and 3072 bits, respectively. As currently specified in SP 800-57, Part 1, these lengths correspond to security strengths of 80, 112 and 128 bits, respectively. Hence, it makes sense to match the number of rounds of Miller-Rabin testing to the target error probability values of 2^{-80}, 2^{-112}, and 2^{-128}. A probability of 2^{-100} is included for all prime lengths, since this probability has often been used in the past and may be acceptable for many applications.

When generating the RSA primes p and q with conditions, it is sufficient to use the value t derived from (1) and (2) as the minimum number of M-R tests to be performed when generating the auxiliary primes p_1, p_2, q_1 and q_2. It is not necessary to use an additional Lucas test on these numbers. In the extremely unlikely event that one of the numbers p_1, p_2, q_1 or q_2 is composite, there is still a high probability that the corresponding RSA prime (p or q) will satisfy the requisite conditions.

The sizes of p_1, p_2, q_1, and q_2 were chosen to ensure that, for an adversary with significant but not overwhelming resources, Lenstra's elliptic curve factoring method [2] (against which there is no protection beyond choosing large p and q) is a more effective factoring algorithm than either the Pollard P–1 method [2], the Williams P+1 method [3] or various cycling methods [2]. For an adversary with overwhelming resources, the best all-purpose factoring algorithm is assumed to be the General Number Field Sieve [2].

Tables C.2 and C.3 in Appendix C.3 specify the minimum number of rounds of M-R testing when generating primes to be used in the construction of RSA signature key pairs.

References

[1] I. Damgard, P. Landrock, and C. Pomerance, C. "Average Case Error Estimates for the Strong Probable Prime Test," Mathematics of Computation, v. 61, No, 203, pp. 177-194, 1993.

[2] A.J Menezes, P.C. Oorschot, and S.A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[3] H.C. Williams. "A p+1 Method of factoring". *Math. Comp*. 39, 225-234, 1982.

[4] D.E. Knuth, The Art of Computer Programming, Vol. 2, 3rd Ed., Addison-Wesley, 1998, Algorithm P, page 395.

[5] R. Baillie and S.S. Wagstaff Jr., Mathematics of Computation, V. 35 (1980), pages 1391 – 1417.