17x bits Elliptic Curve Scalar Multiplication over GF(2^m) using Optimal Normal Basis

TANG KO CHEUNG, SIMON

(Tang.Simon@usa.net)

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Information Engineering

May 2001

Abstract

Elliptic Curve Scalar Multiplication is the dominant computation of Elliptic Curve Cryptography, and each usual ECC cryptographic operation takes just one or two Elliptic Curve Multiplications. Using Optimal Normal Basis in $GF(2^m)$, we achieve 60ms to 88ms per iteration for bits length ranges from 173 to 179 on a Pentium II-400Mhz PC in C without using assembly. No such result using ONB has ever been reported in the literature before. The competitive performance is due to an adaptation of the Almost Inverse Algorithm for an implementation of fast field inverses for ONB. Since we have only used the standard addition-subtraction method for Scalar Multiplication, vast improvements can still be possible.

In the final chapter, we present our findings of a particular extension of the RSA public-key cryptosystem from using integers modulo n to a version using matrices whose entries are integers modulo n, where n is the product of two large primes.

Acknowledgment

I wish to thank Prof. Wei for his help, advice, generous support and genuine academic freedom. In addition, his contribution to the last chapter is very much appreciated.

簡介

橢圓曲線數量乘法是橢圓曲線密碼學之主要運算，而一個一般的橢圓曲線密碼操作只需要一至兩個橢圓曲線數量乘法。

在有限體 GF(2^m) 使用最適正規基，對 173 至 179 位元長度，一個橢圓曲線數量乘法只需要 0.060-0.088 秒。這是使用 C 語言，在奔騰二 · 四百 百萬赫個人電腦上作出的，而且沒有使用過組合語言。

使用正規基，沒有這樣的結果在文獻上出現過。

其良好的性能是由於在使用最適正規基，履行了一個快速的逆，這是把 差不多逆算法 改編而作出的。

對數量乘法，因爲我們只是使用過 加減 算法，故還有很多改良的空間。

在最後一章，我們推廣 RSA 公開密碼系統至矩陣，其元素是整數模 n ， n 係兩個大質數之乘積。

Contents

Chapter 1

Theory of Optimal Normal Bases

1.1 Introduction

In this chapter the theory of optimal normal bases in the finite fields $GF(p^m)$ is presented. We explain the use of an optimal normal basis so as to reduce the complexity of multiplying field elements. Constructions for these bases in $GF(2^m)$ and extensions of the results to $GF(p^m)$ are presented. Important applications of finite field arithmetic include: cryptography [Men93] and error correction coding [Ber68], since a reduction in the complexity of multiplying and exponentiating elements of $GF(2^m)$ is achieved for many values of m, some prime.

A normal basis in $GF(p^m)$ is a basis N of the form $N = \{\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}\}$. It is well known that a normal basis exists in every finite field. Every $B \in GF(p^m)$ may be uniquely expressed in terms of N as $B = \sum_{i=0}^{m-1} b_i \beta^{p^i}$, $b_i \in GF(p)$.

Further, let $A = \sum_{i=0}^{m-1} a_i \beta^{p^i}$, and let $C = AB = \sum_{i=0}^{m-1} c_i \beta^{p^i}$, where c_i is referred to as a product digit. Now $C = \left(\sum_{i=0}^{m-1} a_i \beta^{p^i}\right)\left(\sum_{j=0}^{m-1} b_j \beta^{p^j}\right)$

$= \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{p^i} \beta^{p^j}$. The expressions $\beta^{p^i} \beta^{p^j}$ are referred to as cross-product terms. Since N is a basis for the vector space, we can write $\beta^{p^i} \beta^{p^j} = \sum_{k=0}^{m-1} \lambda_{ijk} \beta^{p^k}$, $\lambda_{ijk} \in GF(p)$. Substitution yields

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \lambda_{ijk} a_i b_j. \tag{1.1}$$

For $X \in GF(p^m)$, let $X = (x_0, x_1, \ldots, x_{m-1})$ denote the coordinate vector for X

in the basis N. Since N is normal, we have $A^{p^k} = (a_{-k}, a_{-k+1}, \ldots, a_{-k-1})$ where the subscripts are taken modulo m.

Also $A^{p^{m-k}} B^{p^{m-k}} =$

$(c_0(A^{p^{m-k}}, B^{p^{m-k}}), c_1(A^{p^{m-k}}, B^{p^{m-k}}), \ldots, c_{m-1}(A^{p^{m-k}}, B^{p^{m-k}}))$, so equating coefficients yields $c_k(A, B) = c_0(A^{p^{m-k}}, B^{p^{m-k}})$.

Therefore viewing c_k as a bilinear form, the form c_k is obtained from c_0 by an $k-$fold cyclic shift of the variables involved.

Define a matrix T_N as follows: index the rows of T_N by the ordered pairs (i, j), $0 \leq i, j \leq m-1$. In row (i, j), column k put λ_{ijk}, the coefficient of β^{p^k} in the expansion of $\beta^{p^i} \beta^{p^j}$. Let C_N denote the number of nonzero terms in the form c_0, and therefore c_k, in the basis N.

As an example, consider the finite field $GF(2^5)$ as generated by the irreducible polynomial $f(x) = x^5 + x^2 + 1$. If we choose α to be a zero of $f(x)$ and set $\beta = \alpha^3$, then $N = \{\beta, \beta^2, \beta^{2^2}, \beta^{2^3}, \beta^{2^4}\}$ is a normal basis. The matrix T_N for this basis is given in Table 1.1a. The value of C_N in this example is 15. If $\beta = \alpha^5$, then $\{\beta, \beta^2, \beta^{2^2}, \beta^{2^3}, \beta^{2^4}\}$ is again a normal basis. Its matrix is given in Table 1.1b, and $C_N = 9$ for this basis.

Table 1.1a. T_N for a normal basis N in $GF(2^5)$ with $C_N = 15$.

		2^k				
2^i	2^j	1	2	4	8	16
1	1	0	1	0	0	0
1	2	0	1	1	1	0
1	4	1	1	1	0	1
1	8	1	0	1	1	1
1	16	1	1	1	0	0
2	1	0	1	1	1	0
2	2	0	0	1	0	0
2	4	0	0	1	1	1
2	8	1	1	1	1	0
2	16	1	1	0	1	1
4	1	1	1	1	0	1
4	2	0	0	1	1	1
4	4	0	0	0	1	0
4	8	1	0	0	1	1
4	16	0	1	1	1	1
8	1	1	0	1	1	1
8	2	1	1	1	1	0
8	4	1	0	0	1	1
8	8	0	0	0	0	1
8	16	1	1	0	0	1
16	1	1	1	1	0	0
16	2	1	1	0	1	1
16	4	0	1	1	1	1
16	8	1	1	0	0	1
16	16	1	0	0	0	0

Definition 1 *We define N to be an optimal normal basis of $GF(p^m)$ if and only if $C_N = 2m - 1$.*

Table 1.1b. T_N for an **optimal** normal basis N in $GF(2^5)$.

		2^k				
2^i	2^j	1	2	4	8	16
1	1	0	1	0	0	0
1	2	1	0	0	1	0
1	4	0	0	0	1	1
1	8	0	1	1	0	0
1	16	0	0	1	0	1
2	1	1	0	0	1	0
2	2	0	0	1	0	0
2	4	0	1	0	0	1
2	8	1	0	0	0	1
2	16	0	0	1	1	0
4	1	0	0	0	1	1
4	2	0	1	0	0	1
4	4	0	0	0	1	0
4	8	1	0	1	0	0
4	16	1	1	0	0	0
8	1	0	1	1	0	0
8	2	1	0	0	0	1
8	4	1	0	1	0	0
8	8	0	0	0	0	1
8	16	0	1	0	1	0
16	1	0	0	1	0	1
16	2	0	0	1	1	0
16	4	1	1	0	0	0
16	8	0	1	0	1	0
16	16	1	0	0	0	0

1.2 The minimum number of terms

We prove that $2m - 1$ is the minimum possible number of terms in equation (1.1), $c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \lambda_{ijk} a_i b_j$. The proof of the following theorem is based on an examination of m rows of a submatrix of T_N.

Theorem 2 *If N is a normal basis for $GF(p^m)$ with matrix T_N, then $C_N \geqslant 2m-1$.*

Proof. Let $N = \{\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}\}$ and, for simplicity, denote β^{p^i} by β_i. Since N is a normal basis, $\sum_{i=0}^{m-1} \beta_i = \text{trace } \beta$.

Let b denote trace β. Consider the $m \times m$ submatrix T_0 of T_N consisting of the m rows of T_N corresponding to the elements $\beta_0\beta_i, 0 \leq i \leq m-1$. Now $b\beta_0 = \beta_0 \sum_{i=0}^{m-1} \beta_i = \sum_{i=0}^{m-1} \beta_0\beta_i$. Therefore, the sum of the rows of T_0 is an $m-$tuple with a b in position 1 and zeros elsewhere. Hence, each column of T_0 contains at least two nonzero elements with the possible exception of column 1 because each column of T_0 must contain at least one nonzero element since the rows of T_0 are linearly independent or equivalently, $\{\beta_0\beta_i : 0 \leq i \leq m-1\}$ is a basis for $GF(p^m)$.

Therefore, the total number of nonzero elements in T_0 is at least $2m-1$. If we define T_j to be the matrix obtained from T_0 by raising each element (associated with a row) of T_0 to the p^j th power, then $\beta_0\beta_i$ in T_0 becomes $\beta_{0+j}\beta_{(i+j) \bmod m}$ in T_j for $0 \leq i, j \leq m-1$.

From the definition of a normal basis, T_j must also contain a total of at least $2m-1$ nonzero elements. Thus, the total number of nonzero elements in T_N is at least $m(2m-1)$ and since each column has C_N nonzero elements, $C_N \geqslant 2m-1$. □

Corollary 3 *Given an optimal normal basis in $GF(p^m)$, for every $0 \leq k \leq m-1$, equation (1.1), i.e., $c_k = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} \lambda_{ijk}a_ib_j$ will contain* `two` *occurrences of subscript i for every $0 \leq i \leq m-1$, and one occurrence of subscript j. This is also true for the subscripts j.*

1.3 Constructions for optimal normal bases

By appropriately choosing β, we can generate an optimal normal basis $N = \{\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}\}$ in $GF(p^m)$, for certain values of m.

Lemma 4 *Suppose that $K = GF(p^m)$ contains $(m+1)st$ roots of unity. If the m nonunit roots of unity are linearly independent, then K contains an optimal normal basis.*

Proof. Let β denote a primitive $(m+1)st$ root of unity in K. Then the conjugates of β are $\beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}$. Since $N = \{\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}\}$ is linearly independent, it is a normal basis for K. But N is the set of zeros of $p(x) = (x^{m+1}-1)/(x-1)$; that is, N is the set of nonunit roots of unity in K. Let $\beta_0 = \beta$,

and $\beta_i = \beta^{p^i}$, $i = 1, 2, \ldots, m-1$. Recall that the number of nonzero terms in the bilinear form for c_0 is also the number of nonzero terms in the expansion of the set $\{\beta_0\beta_i : 0 \leq i \leq m-1\}$ in the basis N. But if $\beta_i \neq \beta_0^{-1}$, then $\beta_0\beta_i = \beta_j$ for some exponent j (depending on i) whereas $\beta_0\beta_0^{-1} = \sum_{i=0}^{m-1} \beta_i$. Hence there are $2m-1$ nonzero terms in the expansion, and N is optimal. □

The above can be restated as below.

Theorem 5 *The field $K = GF(p^m)$ contains an optimal normal basis consisting of the nonunit $(m+1)st$ roots of unity if and only if $m+1$ is a prime and p is primitive in $\mathbb{Z}_{m+1}$.*

Proof. If $m+1$ is prime, then $m+1$ divides $p^m - 1$ and K contains a primitive $(m+1)st$ root of unity β. Since p is primitive in $\mathbb{Z}_{m+1}$, the minimal polynomial of β is $(x^{m+1}-1)/(x-1)$ and the nonunit $(m+1)st$ roots are linearly independent. Conversely if these roots are independent in K then p has order m modulo $m+1$ and $m+1$ is prime. □

The above theorem cannot produce optimal normal bases in $GF(p^m)$ for prime m unless $m = 2$. This liability can be overcome in extensions of $GF(2^m)$ in some instances by the following theorem.

Theorem 6 *If*

(a) 2 is primitive in $\mathbb{Z}_{2m+1}$, or

(b) $2m+1$ is a prime, congruent to 3 *modulo* 4 and 2 generates the quadratic residues in $\mathbb{Z}_{2m+1}$,

then there exists an optimal normal basis in $GF(2^m)$.

Proof. Since $2m+1 | 2^{2m}-1$, there exists a primitive $(2m+1)st$ root of unity, γ in $GF(2^{2m})$. Let $\beta = \gamma + \gamma^{-1}$.

Since $2^m \equiv \pm 1 \bmod (2m+1)$, either $\gamma^{-1} = \gamma^{2^m}$ or $\gamma = \gamma^{2^m}$. Now $\beta^{2^m} = (\gamma+\gamma^{-1})^{2^m} = \gamma^{2^m} + \gamma^{-2^m} = \gamma + \gamma^{-1} = \beta$.

Hence, β is an element of the subfield $GF(2^m)$.

We claim that $\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$is an optimal normal basis of the subfield.

If $\sum_{i=0}^{m-1} \lambda_i \beta^{2^i} = 0$,then $\sum_{i=0}^{m-1} \lambda_i(\gamma^{2^i} + \gamma^{-2^i}) = 0$. Now since either 2 is a generator of the multiplicative group of $GF(2m+1)$ or 2 generates the quadratic residues of $GF(2m+1)$ with $2m+1 \equiv 3 \bmod 4$ then $\sum_{i=0}^{m-1} \lambda_i(\gamma^{2^i} + \gamma^{-2^i}) = \sum_{i=0}^{m-1} \lambda_i\gamma^{2^i} +$

$\sum_{i=0}^{m-1} \lambda_i \gamma^{-2^i} = \sum_{j=1}^{2m} \mu_j \gamma^j$, where each λ_i occurs in $\{\mu_1, \mu_2, \ldots, \mu_{2m}\}$. Therefore γ is a zero of the polynomial $f(X) = \sum_{i=0}^{2m-1} \mu_{i+1} X^i$. Since $f(\gamma) = 0$, the minimal polynomial of γ, $m_\gamma(X)$ divides $f(X)$.

If hypothesis (a) holds then $m_\gamma(X) = 1 + X + X^2 + \cdots + X^{2m}$. Since $m_\gamma(X) | f(X)$ we conclude that $f(X) \equiv 0$ and all $\lambda_i = 0$.

If hypothesis (b) holds then $m_\gamma(X)$ has degree m as does $m_{\gamma^{-1}}(X)$ and $X^{2m+1} - 1 = (X-1) m_\gamma(X) m_{\gamma^{-1}}(X)$. But $m_\gamma(X) \,|f(X)$ since $f(\gamma) = 0$ and $m_{\gamma^{-1}}(X) \,|f(X)$ since $f(\gamma^{-1}) = 0$ and, hence, $1 + X + X^2 + \cdots + X^{2m} | f(X)$ implying that $f(X) \equiv 0$ and that all $\lambda_i = 0$. Therefore, we conclude that N is a normal basis for $GF(2^m)$.

The cross-product terms are $\beta^{2^i} \beta^{2^j} = (\gamma^{2^i} + \gamma^{-2^i})(\gamma^{2^j} + \gamma^{-2^j}) = (\gamma^{(2^i+2^j)} + \gamma^{-(2^i+2^j)}) + (\gamma^{(2^i-2^j)} + \gamma^{-(2^i-2^j)})$. Now if 2 is primitive modulo $2m+1$ then each nonzero residue has the form 2^k for some integer k satisfying $0 \leq k \leq 2m-1$, whereas if 2 generates the quadratic residues modulo $2m+1$ and is congruent to 3 modulo 4, then each nonzero residue has the form of either 2^k *or* -2^k for some integer k satisfying $0 \leq k \leq m-1$.

Therefore if $2^i \neq \pm 2^j \bmod (2m+1)$, then there exist integers k and k' such that $2^i + 2^j = \pm 2^k$ and $2^i - 2^j = \pm 2^{k'}$ for at least one choice of the $+$ or $-$ sign in each case. In this event, $\beta^{2^i} \beta^{2^j} = \beta^{2^k} + \beta^{2^{k'}}$.

One the other hand, if $2^i = \pm 2^j \bmod (2m+1)$, then one of $2^i + 2^j$ is not zero modulo $2m+1$, and so there exists a k such that at least one of the equations $2^i + 2^j = 2^k$, $2^i + 2^j = -2^k$, $2^i - 2^j = 2^k$, $2^i - 2^j = -2^k$ is satisfied. In this case, since we are in a field of characteristic 2, $\beta^{2^i} \beta^{2^j} = \beta^{2^k}$.

Let $\beta_i = \beta^{2^i}$, $i = 0, 1, 2, \ldots, m-1$. Then, since $(\beta_0)^2 = \beta_1$, there are at most $2m-1$ terms in the expansion of the set $\{\beta_0 \beta_i\}$ in terms of the basis N, and therefore there are precisely $2m-1$ such terms and N is an optimal normal basis. □

The minimal polynomial $M_\beta(X)$ as defined in the above Theorem can be easily determined recursively. Over $GF(2)$, define the sequence of polynomials $f_i(X)$, $i = 0, 1, 2, \ldots$ as follows. Let $f_0(X) = 1$, $f_1(X) = X + 1$, and $f_t(X) = X f_{t-1}(X) + f_{t-2}(X)$, $t \geqslant 2$. If m is such that the hypotheses of the theorem are satisfied, then $f_m(X)$ is the minimal polynomial of β. Indeed, it is easily shown by induction that $f_t(Y + Y^{-1}) = 1 + \sum_{i=1}^{t} (Y^i + Y^{-i})$. Therefore $f_m(\beta) = 1 + \sum_{i=1}^{m} (\gamma^i + \gamma^{-i}) = 1 + \sum_{i=1}^{2m} \gamma^i = 0$, since γ is a primitive $(2m+1)$st root of unity.

Definition 7 *Let* $N = \{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$ *be a normal basis over* $GF(2^m)$. *Let* $\beta^{2^i} = \beta_i$, $i = 0, 1, \ldots, m-1$. *The basis* N *will be said to be of type I if with the exception of one value of* i, $0 \leq i \leq m-1$, *there exists an integer* k_i *satisfying* $0 \leq k_i \leq m-1$ *such that* $\beta_0\beta_i = \beta_{k_i}$.
The basis N *is said to be of type II if, for every* i, *satisfying* $1 \leq i \leq m-1$, *there exists integers* k_i *and* m_i *such that* $\beta_0\beta_i = \beta_{k_i} + \beta_{m_i}$.

Clearly every optimal basis constructed by the method of Theorem 5 is a type-I basis and every optimal basis constructed by the methods of Theorem 6 is a type-II basis.

It is easily shown that every type-I basis can be obtained by the construction of Theorem 5. It can be shown that every type-II basis can be obtained by the construction of Theorem 6, [MOVW89].

1.4 Existence of optimal normal bases

It is known, see [MOVW89], that

- 2 is primitive in $\mathbb{Z}_p$ for a prime p if $p = 4q + 1$ and q is an odd prime,
- 2 is primitive in $\mathbb{Z}_p$ for a prime p if $p = 2q + 1$ where q is a prime congruent to 1 modulo 4,
- 2 is a generator of the quadratic residues in $\mathbb{Z}_p$ if $p = 2q+1$ where q is a prime congruent to 3 modulo 4, and
- 2 cannot be primitive modulo p if p is a prime congruent to 1 modulo 8.

In view of these results, the testing of the hypotheses in Theorems 5 and 6 becomes easier in certain cases. Computer searches give a complete list of $m < 1200$ for which we can construct an optimal normal basis in $GF(2^m)$, Table 1.4, has 23% of all possible values of m.

Table 1.4. Values of m for which an optimal normal basis can be constructed in $GF(2^m)$.

2	3	4	5	6	9	10	11	12	14	18	23
26	28	29	30	33	35	36	39	41	50	51	52
53	58	60	65	66	69	74	81	82	83	86	89
90	95	98	99	100	105	106	113	119	130	131	134
135	138	146	148	155	158	162	172	173	174	178	179
180	183	186	189	191	194	196	209	210	221	226	230
231	233	239	243	245	251	254	261	268	270	273	278
281	292	293	299	303	306	309	316	323	326	329	330
338	346	348	350	354	359	371	372	375	378	386	388
393	398	410	411	413	414	418	419	420	426	429	431
438	441	442	443	453	460	466	470	473	483	490	491
495	508	509	515	519	522	530	531	540	543	545	546
554	556	558	561	562	575	585	586	593	606	611	612
614	615	618	629	638	639	641	645	650	651	652	653
658	659	660	676	683	686	690	700	708	713	719	723
725	726	741	743	746	749	755	756	761	765	771	772
774	779	783	785	786	791	796	803	809	810	818	820
826	828	831	833	834	846	852	858	866	870	873	876
879	882	891	893	906	911	923	930	933	935	938	939
940	946	950	953	965	974	975	986	989	993	998	1013
1014	1018	1019	1026	1031	1034	1041	1043	1049	1055	1060	1065
1070	1090	1103	1106	1108	1110	1116	1118	1119	1121	1122	1133
1134	1146	1154	1155	1166	1169	1170	1178	1185	1186	1194	1199

This table leads to the following:

Conjecture. If m does not satisfy the criteria for Theorem 5 or Theorem 6, then $GF(2^m)$ does not contain an optimal normal basis.

It was proved in [GL92].

Finally, here are some pointers for hardware implementations.

Massey and Omura, [OM86], constructed a serial-in serial-out multiplier to exploit this particular aspect of normal bases. An architecture for a hardware implementation is given in [AMOV91], of the *Journal of Cryptology*. In this paper, G. Agnew, R. Mullin, I. Onyszchuk and S. Vanstone examine the development of a high-speed

implementation of a system to perform exponentiation in fields of the form $GF(2^m)$. The use of optimal normal bases and observations on the structure of multiplications have led to the development of an architecture which is of low complexity and high-speed. Using this architecture a multiplication can be performed in m clock cycles.

Chapter 2

Implementing Multiplication in $GF(2^m)$

2.1 Defining the Galois fields $GF(2^m)$

A normal basis can be found for any finite field $GF(p^m)$, see chapter 1. For computers we use $p = 2$, i.e., $\{\beta^{2^{m-1}}, \ldots, \beta^{2^2}, \beta^2, \beta\}$. An element $\underline{e}$ in a field $GF(2^m)$ can be written in a normal basis as: $\underline{e} = e_{m-1}\beta^{2^{m-1}} + \ldots + e_2\beta^{2^2} + e_1\beta^{2^1} + e_0\beta$ or $\sum_{i=0}^{m-1} e_i\beta^{2^i}$.

Here is the first header file, called `field2n.h`, which helps to define Galois Fields for the C code.

```
/*** field2n.h ***/

#define WORDSIZE (sizeof(int)*8)
#define NUMBITS 173

#define NUMWORD (NUMBITS/WORDSIZE)
#define UPRSHIFT (NUMBITS%WORDSIZE)
#define MAXLONG (NUMWORD+1)
#define MAXBITS (MAXLONG*WORDSIZE)
#define MAXSHIFT (WORDSIZE-1)
#define MSB (1L<<MAXSHIFT)
#define UPRBIT (1L<<(UPRSHIFT-1))
#define UPRMASK (~(-1L<<UPRSHIFT))
#define SUMLOOP(i) for(i=0; i<MAXLONG; i++)
```

```
typedef short int INDEX;
typedef unsigned long ELEMENT;
typedef struct {
        ELEMENT e[MAXLONG];
} FIELD2N;
```

WORDZISE is the number of bits in a machine word. NUMBITS is the number of bits the normal basis math will be expected to work on. NUMWORD is the maximum index of machine words into a normal basis coefficients array. UPRSHIFT is the number of left shifts needed to get to the most significant bit in the zero offset of the coefficient list. MAXLONG is the number of machine words to hold the normal basis coefficients.

The term MAXBITS is used in a few places; it is the maximum number of bits we can store in MAXLONG machine words. The term MAXSHIFT is the largest number of shifts we need to move the most significant bit to the least in a single bit block. Since we are doing a lot of shifting, this will be useful later. MSB is a mask for the most significant bit in a WORDZISE block of bits.

The term UPRSHIFT is used to compute the most significant bit position, and UPRBIT, a mask for the high-order ELEMENT UPRMASK. We will use UPRBIT for rotations and UPRMASK to clear bits after rotations and shifts.

SUMLOOP is a macro. An INDEX is used for bookkeeping. We call an unsigned long an ELEMENT, because it is the simplest thing we can work with. An ELEMENT is one machine word in size.

Finally, we define the fields storage structure FIELD2N. This comes from the mathematical symbols, $GF(2^m)$, which means a field of characteristic 2 and vector length m. Since we are going to reference each field element in the array, we use a single letter "e," for ELEMENT. The coefficients are in big-endian order. This is an arbitrary choice, so if one changes the order, make sure he changes it everywhere.

2.2 Adding and squaring normal basis numbers in $GF(2^m)$

In base 2, all the coefficients can only be 0 or 1, and addition is simply an exclusive-or, XOR operation. For $m = 8, 16, 32$,or 64 we would have perfect word-size alignment for any processor. Unfortunately, these are too small for cryptographic purposes and not mathematically optimal.

The nicest aspect of this representation is that squaring a number amounts to a rotation. There are two reasons for this: 1. $(\beta^{2^i})^2 = \beta^{2^{i+1}}$ 2. $\beta^{2^m} = \beta$. The first statement is obvious. The second statement comes from the rules of finite fields and is similar to Fermat's Little Theorem. So squaring $\underline{e}$ amounts to shifting each coefficient up to the next term and rotating the last coefficient down to 0 position. Squaring is thus very fast in a normal basis.

The first weird or interesting thing to recognize is what 1 is in a normal basis: $1=\sum_{i=0}^{m-1} \beta^{2^i}$. In other words, the fundamental constant, 1, is represented as "all bits set" in a normal basis. So, adding 1 to a normal basis number amounts to flipping all the bits—not counting as we are used to.

2.3 Multiplication formula

Multiplication over a normal basis gets a touch complicated and is the main theme of this chapter. The basics are the same in any mathematical system, just multiply coefficients and sum over all those that have the same power of x. What makes optimal normal bases slick is that most of those terms are 0.

To show how multiply works, let us recall those material in Chapter 1, section 1 first.

Take two elements in $GF(2^m)$: $A = \sum_{i=0}^{m-1} a_i \beta^{2^i}$ and $B = \sum_{i=0}^{m-1} b_j \beta^{2^j}$. The formal multiplication is $C = AB = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i} \beta^{2^j}$ but $C = \sum_{k=0}^{m-1} c_k \beta^{2^k}$ by definition of an element in a normal basis. So the double sum in the first equation has to match the single sum in the second equation. In fact, we must have each cross-product term map to a sum over the basis terms

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk} \beta^{2^k}, \ \lambda_{ijk} \in GF(2).$$

The λ_{ijk} coefficient is called "the lambda matrix" or "multiplication table."

If we substitute the multiplication table formula into the $C = AB$ formula, we get a mess. From that mess we can find the solution to each c_k coefficient of β^{2^k} in $C = \sum_{k=0}^{m-1} c_k \beta^{2^k}$, which is "only" a double sum:

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \lambda_{ijk}.$$

which is equation 1.1 in Chapter 1 section 1.

The equation $c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \lambda_{ijk}$ can be transmogrified into a form that requires only λ_{ij0}. From the discussions following equation 1.1 of Chapter 1 section 1, it is readily seen that

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0}. \tag{2.1}$$

That reduces the amount of work required to construct the λ matrix (multiplication table). What makes equation 2.1 so awesome is that all we need to do is shift the inputs by the correct amount and all coefficients can be computed in parallel.

Because there is no carry, even high-level languages can reasonably implement normal basis math using very little memory. Of course, assembler and hardware will always be faster.

An "optimal" normal basis has the minimum number of nonzero terms. This number is called the "complexity" of the multiplication table. For fields $GF(2^m)$ the optimal (minimum) complexity is $2m - 1$, chapter 1 theorem 1.

Recall that there are two types of optimal normal basis over $GF(2^m)$ mentioned in Chapter 1, Section 3. They are called Type I and Type II. The only real difference between them is the way we find which bits in the λ matrix are set. For Type I ONB we only need to store one vector; for Type II ONB we'll need to store two vectors. For the code here, we'll make them both look the same so that the multiply routine will work in either case. A Type I ONB multiply could be made quicker with a few math tricks.

2.4 Construction of `Lambda` table for Type I ONB in $GF(2^m)$

According to Chapter 1, Section 3, the rules for creating Type I Optimal Normal Basis in the field $GF(2^m)$ are:

1. $m + 1$ must be prime,
2. 2 must be primitive in $\mathbb{Z}_{m+1}$.

Rule 2 means that 2 raised to any power in the range $0 \ldots m - 1$ modulo $m + 1$ must result in a unique integer in the range $1 \ldots m$. $\mathbb{Z}$ is used to mean the set of integers.

We need to find the cross-product terms of $\beta^{2^i}\beta^{2^j}$ in $\beta^{2^i}\beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk}\beta^{2^k}$, $\lambda_{ijk} \in$

$GF(2)$ to make the multiplication work. Because we can transform the λ matrix to $k = 0$ for all cross-terms, we really only need to solve the equation: $\beta^{2^i}\beta^{2^j} = \beta^1$. There is also the special case: $\beta^{2^i}\beta^{2^j} = 1$, when $2^i + 2^j$ is congruent to 0 modulo $m + 1$.

To proceed, we need to know some math rules. For all the above to work optimally we must have β be an element of order $m + 1$ in $GF(2^m)$. Since 2 is primitive in $\mathbb{Z}_{m+1}$, $2^i \bmod m + 1$ will run through all the integers between 1 and m as i runs through all values $0, 1, \ldots, m - 1$. The combination β^{2^i} is just another way of counting through all powers of β, which is what generates our basis. The order is scrambled compared to β^j, but all powers of the generator are accounted for. These are just simple matters, see [Ros98], p.80.

The easy way to solve equations $\beta^{2^i}\beta^{2^j} = \beta^1$, and also $\beta^{2^i}\beta^{2^j} = 1$ is to rewrite them as modulo $m + 1$ and to "step into the exponent." We only need to solve:

$$\begin{aligned} 2^i + 2^j &= 1 \mod m + 1, \\ 2^i + 2^j &= 0 \mod m + 1. \end{aligned}$$

Let's start with $i = 0$. $2^0 = 1$ and $2^m = 1$, since $m + 1$ is prime. This last point comes from Fermat's Little Theorem. The first equation $2^i + 2^j = 1 \bmod m + 1$ cannot have a solution for $i = 0$. Only the second equation $2^i + 2^j = 0 \bmod m + 1$ can.

This second equation $2^i + 2^j = 0 \bmod m + 1$, can be solved, like this: If we take the square root of $2^m = 1$, we find: $2^{m/2} = \pm 1 \bmod m + 1$. But we already know that $2^0 = +1$, and, since 2 generates all the numbers mod $m + 1$, there is only one choice: $2^{m/2} = -1 \bmod m + 1$. The equation $2^i + 2^j = 0 \bmod m + 1$ thus has a solution for $i = 0$, which is: $2^0 + 2^{m/2} = 0 \bmod m+1$. We mentioned previously that the Type I ONB only needs a single vector to keep track of all the cross-product terms. The first entry in the table is at offset 0 (we are programming in C) and has value $m/2$. We don't need to store any more of the values to solutions of this equation, because we can multiply equation $2^0 + 2^{m/2} = 0 \bmod m+1$ by 2 and still have 0 on the right-hand side. For every i, the nonzero element λ_{ij0} element from the equation $2^i + 2^j = 0 \bmod m + 1$ will always be: $j = m/2 + i \bmod m$.

Now for the first equation $2^i+2^j = 1 \bmod m+1$, we do have to tabulate the values from this equation once and then use those to look up the correct shift amounts. The first value for $i = 1$ is easy, since: $2 + 2^j = 1$, or $2^j = -1$, so $j = m/2$. After that we need to use antilog and log tables so we can find $2^i \bmod m + 1$ easily and j from $1 - 2^i \bmod m + 1$ just as easily.

Let's look at a set of simple tables for $m = 4$.

The antilog table, see below, is really simple; just multiply 2^i by 2 modulo 5 for each entry.

Antilog Table for 2^i for $m = 4$

i	0	1	2	3	4
2^i	1	2	4	3	1

The log table, see below, takes each value of 2^i as the index and places i as the entry.

Log Table for 2^i for $m = 4$

2^i	1	2	3	4
i	0	1	3	2

The zero offset isn't used in this case, but we'll find a use for that storage location later. The log table is sometimes called the Zech logarithm and has uses in other places such as spread-spectrum communications.

The code for generating the single vector we need is given below. It does two things. The first is a construction of log tables for 2^i mod $m+1$. We call the prime number $m+1$ "`field_prime`" in the code. This must be set in a header file. The best thing to do is a simple addition to the `field2n.h` file with the following line:

```
#define field_prime (NUMBITS+1)
```

The second step is to create a lambda vector, which stores all the values of j for each value of i that satisfies the equation $2^i + 2^j = 1 \bmod m + 1$. The lambda vector, and the log table as well, are globals and are defined using these lines:

```
static INDEX Lambda[2][field_prime];
static INDEX log2[field_prime+1];
```

A two-dimensional vector is not needed for the Type I ONB, but it is required for the Type II ONB. To make both types work with one multiply routine, the solutions to the equation $2^i + 2^j = 0 \bmod m + 1$ is copied into the `Lambda[0]` array. The `Lambda[1]` array holds the lambda vector solution to the equation $2^i + 2^j = 1 \bmod m + 1$, which is the single vector we need.

Here's the code that creates the `Lambda` Table.

```
/* create Lambda [i,j] table. indexed by i, each entry contains the
value of j which satisfies 2^i + 2^j = 1 || 0 mod field_prime. There are
```

```
two 16 bit entries per index i except for zero.
Since 2^0 = 1 and 2^2n = 1, 2^n = -1 and the first entry would
be 2^0 + 2^n = 0. Multiplying both sides by 2, it stays congruent to
zero. So Half the table is unnecessary since multiplying exponents by
2 is the same as squaring is the same as rotation once. Lambda[0][0]
stores n=(field_prime-1)/2. The terms congruent to one must be found via
lookup in the log table. Since every entry for (i,j) also generates an
entry for (j,i), the whole 1D table can be built quickly.
*/
void genlambda()
{
        INDEX i, logof, n, index, twoexp;
        for (i=0; i<field_prime; i++) log2[i] = -1;
/* build log table first */
        twoexp = 1;
        for (i=0; i<field_prime; i++)
        {
                log2[twoexp] = i;
                twoexp = (twoexp << 1) % field_prime;
        }
```

Creating the log table takes four lines of code, only two of which are inside the loop. Each multiple of two is stored in the variable `twoexp`. The first line of code initializes `twoexp` to 1, since $2^0 = 1$. Log base 2 of 1 is 0, so the first entry in the loop sets offset 1 to the value 0.

The next line of code shifts `twoexp` left once, which is the same as multiplying by 2. That value is reduced modulo `field_prime`. The value of i is incremented at the bottom of the loop, and the next entry in the log table is log base 2 of `twoexp`, which equals i. All values of i will be stored somewhere in the log table and never overlap as long as the fundamental rule that 2 is a generator modulo `field_prime` $(m+1)$ is not broken. It is awesome to watch the vector get filled using a debugger, a very controlled chaos.

Because the lambda matrix is symmetric, we only need to do half of it. The variable `n` in the code is used more than once, so it is convenient to define it as the following:

```
/* compute n for easy reference */
```

```
        n = (field_prime - 1)/2;

/* fill in first vector with indices shifted by half table size */
        Lambda[0][0] = n;
        for (i=1; i<field_prime; i++)
        Lambda[0][i] = (Lambda[0][i-1] + 1) % NUMBITS;
```

The above code fills in the `Lambda[0]` table. Starting with $i = 0$ in equation $2^i + 2^j = 0 \bmod m + 1$, each succeeding value is just 1 plus the previous value modulo `NUMBITS`. For memory-constrained systems using only Type I ONB math, only one vector is needed.

Here is the code that generates the single important `lambda` vector.

```
/* initialize second vector with known values */
        Lambda[1][0]= -1; /* never used */
        Lambda[1][1] = n;
        Lambda[1][n] = 1;
/* loop over result space. Since we want 2^i + 2^j = 1 mod field_prime
 it's a ton easier to loop on 2^i and look up i than solve the silly
 equations. Think about it, make a table, and it'll be obvious. */
        for (i=2; i<=n; i++) {
        index = log2[i];
        logof = log2[field_prime - i + 1];
        Lambda[1][index] = logof;
        Lambda[1][logof] = index;
        }
/* last term, it's the only one which equals itself. */
        Lambda[1][log2[n+1]] = log2[n+1];
}
```

Let's see how this works. The first thing to recognize is that equation $2^i + 2^j = 1 \bmod m + 1$ is symmetric. If entry $\lambda_{ij0} = 1$, then $\lambda_{ji0} = 1$ too. The counter `i` is taken as the value of 2^i. Because we'll hit every value only once, the counter steps through every possible value of $2^i \pmod{m+1}$. The variable `index` is log base 2 of `i`, so that tells us where to store the value of i in the `Lambda` vector.

Since $1 - 2^i$ modulo `field_prime` does not change if we add `field_prime`, we can write the equation as: $2^j = \texttt{field\_prime} + 1 - 2^i$. Take log base 2 of both sides, and we have the second line of code in the loop. This eliminates negative lookups, which would give the code some major headaches.

The last two lines in the loop simply set the entry point `index` to the value of j (called `logof` in the code) and entry point j to the value of `index`. The very last line in the above listing fills in the only nonsymmetric term.

2.5 Constructing Lambda table for Type II ONB in $GF(2^m)$

2.5.1 Equations of the Lambda matrix

For a Type II optimal normal basis we have the same number of terms. But both sets are scrambled, so we end up with two sets of vectors. There are two possible Type II ONBs: let us call them Type IIa and Type IIb.

According to Chapter 1, Section 3. A Type II optimal normal basis over $GF(2^m)$ can be created if :

1 $2m+1$ is prime
and either
2a 2 is primitive in $\mathbb{Z}_{2m+1}$
or
2b $2m+1 \equiv 3 \bmod 4$ and 2 generates the quadratic residues in $\mathbb{Z}_{2m+1}$.

What does 2a mean? If we take $2^k \bmod 2m+1$ for $k = 0, 1, 2, \ldots, 2m-1$ then we get every value in the range $[1 \ldots 2m]$ back. What does 2b mean? The first part is simple: The last two bits are set in the binary representation of the prime $2m+1$. The second part means that even if $2^k \bmod 2m+1$ does not generate every element in the range $[1 \ldots 2m]$, we can at least take the square root mod $2m+1$ of 2^k.

For Type II ONB we need to modify the `field2n.h` header file again. To make life a bit simpler, we add some additional code. This allows all the modifications to work when needed. By simply changing `TYPE2` to `TYPE1` in one place the entire code package will compile correctly.

```
#define TYPE2

#ifdef TYPE2
#define field_prime ((NUMBITS<<1)+1)
#else
#define field_prime (NUMBITS+1)
#endif
```

To generate a Type II ONB we use two field elements from two different fields. First pick an element γ of order $2m+1$ in $GF(2^{2m})$. We use that to find β, which is in the field $GF(2^m)$. We won't actually have to find the γ element; we are just going to use it symbolically to help us create the λ matrix. Form the sum of $\gamma+\gamma^{-1}$. This element gives us the first element, β, of our normal basis, recall Chapter 1, Section 3.

The cross-product terms of $\beta^{2^i}\beta^{2^j}$ are $\beta^{2^i}\beta^{2^j} = (\gamma^{2^i}+\gamma^{-2^i})(\gamma^{2^j}+\gamma^{-2^j}) = (\gamma^{(2^i+2^j)}+\gamma^{-(2^i+2^j)}) + (\gamma^{(2^i-2^j)}+\gamma^{-(2^i-2^j)})$. Since $\gamma^{2^k}+\gamma^{-2^k} = (\gamma+\gamma^{-1})^{2^k}$, we get

$$\beta^{2^i}\beta^{2^j} = \begin{cases} \beta^{2^k}+\beta^{2^{k'}} & \text{if } 2^i \neq \pm 2^j \bmod (2m+1) \\ \beta^{2^k} & \text{if } 2^i = \pm 2^j \bmod (2m+1) \end{cases}$$

. k and k' are two possible solutions to the multiplication of any two basis elements. That is what makes this normal basis optimal: It has the minimum number of possible terms. In the case of $2^i = \pm 2^j$, the terms $\gamma^0+\gamma^{-0}$ cancel, because the exclusive-or of anything with itself is 0.

In the case of $2^i \neq \pm 2^j \bmod (2m+1)$, at least one of these equations:

$2^i + 2^j = 2^k \quad \bmod (2m+1)$
$2^i + 2^j = -2^k \quad \bmod (2m+1)$ will have a solution, and at least one of these equations:

$2^i - 2^j = 2^{k'} \quad \bmod (2m+1)$
$2^i - 2^j = -2^{k'} \quad \bmod (2m+1)$ also has a solution.

In the case of $2^i = \pm 2^j \bmod (2m+1)$, at least one of the following four equations has a solution:

$2^i + 2^j = 2^k \quad \bmod (2m+1)$
$2^i - 2^j = 2^k \quad \bmod (2m+1)$
$2^i + 2^j = -2^k \quad \bmod (2m+1)$
$2^i + 2^j = -2^k \quad \bmod (2m+1)$.

In the first set of equations, there are two possible solutions, and in the second set of equations, there is only one possible solution. It is easy to see that, [Ros98], p.87, the equations are all similar, so instead of working with two different sets we can combine them and work with just one group of four equations. To build our λ

matrix, we set $k = 0$ and find solutions to:

$$\begin{aligned} 2^i + 2^j &= 1 \mod (2m+1), \\ 2^i + 2^j &= -1 \mod (2m+1), \\ 2^i - 2^j &= 1 \mod (2m+1), \\ 2^i - 2^j &= -1 \mod (2m+1). \end{aligned}$$

2.5.2 An example of Type IIa ONB

As an example of Type IIa, take $2m+1 = 19$. Then our field size $m = 9$. This will be the length of the λ matrix. The first thing we need to build are log and antilog tables.

Powers of 2^i mod 19 (antilog)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2^i	1	2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1

The antilog table, see above, takes an index, i, and returns $l = 2^i \mod 2m+1$. The log table, see below, takes an index, l, and returns the value of i.

Log base 2 of i mod 19 (log table)

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Log_2(i)$	0	1	13	2	16	14	6	3	8	17	12	15	5	7	11	4	10	9

The code to compute the log table is as follows:

```
twoexp = 1;
for (i=0; i<NUMBITS; i++)
{
   log2[twoexp] = i;
   twoexp = (twoexp << 1) % field_prime;
}
```

Note that the log table is built using 2^i modulo `field_prime` as the subscript, and the loop counter `i` is the value. This builds the log table in the order of the antilog table.

To continue the example, let us start with $i = 1$ in the equations

$$\begin{aligned} 2^i + 2^j &= 1 \mod (2m+1), \\ 2^i + 2^j &= -1 \mod (2m+1), \\ 2^i - 2^j &= 1 \mod (2m+1), \\ 2^i - 2^j &= -1 \mod (2m+1). \end{aligned}$$

Writing down all four equations mod 19 and subtracting 2 from both sides gives us the following:

$$\begin{aligned} 2^i &= -1 = 18 \quad \Rightarrow j = 9 \\ 2^j &= -3 = 16 \quad \Rightarrow j = 4 \\ -2^j &= -1 \quad \Rightarrow j = 0 \\ -2^j &= -3 \quad \Rightarrow j = 13 \end{aligned}$$

Since the λ matrix has only nine entries per column, the solutions for j must be in the range of $0 \ldots 8$. Only two terms for j are less than 9. These are the two terms we need. So we have our first nonzero entries in the λ matrix: $\lambda_{1,0} = 1$ and $\lambda_{1,4} = 1$. All other entries $\lambda_{1,j}$ must be 0.

Continuing in this manner we can find two values of j for each value of i, which give us nonzero entries in the λ matrix. We'll mark these with two vectors: Λ_0 and Λ_1, and call them `Lambda[0][]` and `Lambda[1][]` in the code. Each position in the Λ table corresponds to a value of i in cross-product term $\beta^{2^i}\beta^{2^j}$, and each entry is the matching value of j, which gives one of the terms in equation

$$\beta^{2^i}\beta^{2^j} = \begin{cases} \beta^{2^k} + \beta^{2^{k'}} & \text{if } 2^i \neq \pm 2^j \bmod (2m+1) \\ \beta^{2^k} & \text{if } 2^i = \pm 2^j \bmod (2m+1) \end{cases}$$

that has k or $k' = 0$. The results are shown below.

Λ Vectors for $m = 9$

i	$\Lambda_0 = j_1$	$\Lambda_1 = j_2$
0	1	
1	4	0
2	4	7
3	6	8
4	2	1
5	6	7
6	5	3
7	2	5
8	8	3

The choice of value in either column for any row does not really matter, since we are going to combine matching coefficients in the multiply routine eventually. Note that there are a total of $2m - 1$ terms. The zero entry will always be 1 for any Type II ONB. We fill in this spot just to make the code simple, but we will take advantage of it when we do the actual multiply.

2.5.3 An example of Type IIb ONB

Now, ley us look at a Type IIb ONB. When the `field_prime` is 23, for example, we have a Type IIb ONB, which is congruent to 3 mod 4 and in which 2 generates the quadratic residues of mod 23. Look at the antilog table below to see what this

really means.

Powers of 2^i mod 23 (antilog)

i	0	1	2	3	4	5	6	7	8	9	10
2^i	1	2	4	8	16	9	18	13	3	6	12

i	11	12	13	14	15	16	17	18	19	20	21	22
2^i	1	2	4	8	16	9	18	13	3	6	12	1

Note that only half the values of $1 \ldots 22$ appear in the anti-log table. But if we take $23 - 2^i$ for all values greater than 11, we get the results shown in the following table, that is fairly straightforward, [Ros98], p.89.

Powers of 2^i mod 23 (antilog)

i	0	1	2	3	4	5	6	7	8	9	10
2^i	1	2	4	8	7	9	5	10	3	6	11

i	11	12	13	14	15	16	17	18	19	20	21	22
2^i	1	2	4	8	7	9	5	10	3	6	11	1

Now, $23 - 2^i$ modulo 23 is just -2^i. It, after building half the log table, we find that `twoexp` ($= 2^i$) equals 1, then we know we will cycle through the same values of 2^i that we just finished. To solve this problem we can restart at $i = 0$ but do the subscript on negative values of 2^i. Since subscripts need to be positive (for us to fill in a useful table relative to the rest of the code anyway), we can start with `twoexp = field_rime - 1 = 2*NUMBITS`, which is congruent to -1. This following code then fills in the rest of the log table.

```
if (twoexp == 1) /* if so, then deal with quadratic residues */
{
   twoexp = 2*NUMBITS;
   for (i=0; i<NUMBITS; i++)
   {
     log2[twoexp] = i;
     twoexp = (twoexp << 1) % field_prime;
   }
}
```

The final Log table is shown here.

Log base 2 of i mod 23 (log table)

i	1	2	3	4	5	6	7	8	9	10	11
$Log_2(i)$	0	1	8	2	6	9	4	3	5	7	10

i	12	13	14	15	16	17	18	19	20	21	22
$Log_2(i)$	10	7	5	3	4	9	6	2	8	1	0

All we are doing is bookkeeping: tracking all the coefficients we need to add in order to compute the multiplication of two normal basis numbers. Since all we had to solve for was $k = 0$, instead of a complete λ matrix, we only need to store two vectors.

2.5.4 Creating the Lambda vectors for Type II ONB

Once we have the log and antilog tables, creating the vectors is easy. Here is the code for generating the vectors for a Type II ONB.

```
/* Type 2 ONB initialization. Fills 2D Lambda matrix. */

void genlambda2()
{
        INDEX i, logof[4], n, index, j, k, twoexp;

/* build log table first. For the case where 2 generates the quadratic
 residues instead of the field, duplicate all the entries to ensure
 positive and negative matches in the lookup table (that is, -k mod
 field_prime is congruent to entry field_prime - k). */

        twoexp = 1;
        for (i=0; i<NUMBITS; i++)
        {
                log2[twoexp] = i;
                twoexp = (twoexp << 1) % field_prime;
        }
        if (twoexp == 1) /* if so, then deal with quadratic residues */
        {
                twoexp = 2*NUMBITS;
                for (i=0; i<NUMBITS; i++)
                {
                        log2[twoexp] = i;
                        twoexp = (twoexp << 1) % field_prime;
                }
        }
        else
```

```
        {
                for (i=NUMBITS; i<field_prime-1; i++)
                {
                        log2[twoexp] = i;
                        twoexp = (twoexp << 1) % field_prime;
                }
        }

/* first element in vector 0 always = 1 */
        Lambda[0][0] = 1;
        Lambda[1][0] = -1;

/* again compute n = (field_prime - 1)/2 but this time we use it to see if
 an equation applies */
        n = (field_prime - 1)/2;

/* as in genlambda for Type I we can loop over 2^index and look up index
 from the log table previously built. But we have to work with 4
 equations instead of one and only two of those are useful. Look up
 all four solutions and put them into an array. Use two counters, one
 called j to step thru the 4 solutions and the other called k to track
 the two valid ones.

 For the case when 2 generates quadratic residues only 2 equations are
 really needed. But the same math works due to the way we filled the
 log2 table.
*/
        twoexp = 1;
        for (i=1; i<n; i++)
        {
                twoexp = (twoexp<<1) % field_prime;
                logof[0] = log2[field_prime + 1 - twoexp];
                logof[1] = log2[field_prime - 1 - twoexp];
                logof[2] = log2[twoexp - 1];
                logof[3] = log2[twoexp + 1];
                k = 0;
                j = 0;
                while (k<2)
                {
```

```
                    if (logof[j] < n)
                    {
                            Lambda[k][i] = logof[j];
                            k++;
                    }
                    j++;
            }
    }
```

The genlambda2 routine is very similar to the previous genlambda routine. The main difference is that we now have to check four equations instead of one. The four equations are solved as a look up in the log table and saved in the `logof[]` array.

Since there are two solutions and four variables to check, we use two counters. The variable `j` counts over the `logof[]` array, and the variable `k` counts over the solutions. A solution is valid only if less than `n`. For Type IIb that is automatic, and only the first two equations will ever be used. But anything modulo `field_prime` will give us an index into the array in the range of `1...2*NUMBITS`, and for Type IIa we have to check that we get the right two solutions.

Going from 1 to $m-1$ and getting two solutions gives us $2m-2$ terms. The first term is already known and is set at offset `0` in `Lambda[0][]`. So we have found all the terms, Chapter 1, Section 2.

2.6 Multiplication in practice

The starting point for multiplication is the multiplication formula

$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0}$ in section 2.3.

From the previous efforts we know that there are only two values of j for each value of i (or vice versa, since multiplication is independent of order). Note that each subscript of a and b is shifted by the same value of k. This means we can shift all the a coefficients and all the b coefficients for any particular values of i and j to find one term for all the c coefficients.

An example will help. Let's take the $i = 2$ index of the Type IIa $GF(2^9)$ example worked out before.

Λ Vectors for $m = 9$

i	$\Lambda_0 = j_1$	$\Lambda_1 = j_2$
0	1	
1	4	0
2	4	7
3	6	8
4	2	1
5	6	7
6	5	3
7	2	5
8	8	3

From this table we have $\Lambda_{0,2} = 4$ and $\Lambda_{1,2} = 7$. Consider one explicit term of the multiplication formula, which looks like this: $c_k = \ldots\ldots\ldots + a_{2+k}(b_{4+k} + b_{7+k}) + \ldots\ldots\ldots$

For $k = 0$, the partial sum is $a_2(b_4 + b_7)$. For $k = 1$, the partial sum is $a_3(b_5 + b_8)$ and so one through $k = 8$. All of these bitwise manipulations can be done in parallel. What we have to do is rotate the A vector right two places and multiply it with the sum of the B vector rotated right four and seven places. As an example, graphically this appears as follows:

$$
\begin{array}{ccccccccc}
a_1 & a_0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 \\
 & & & & & & & \bullet & \\
b_3 & b_2 & b_1 & b_0 & b_8 & b_7 & b_6 & \overset{\frown}{b_5} & b_4 \\
 & & & & & & & + & \\
b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & \underset{\smile}{b_8} & b_7 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

The addition is performed by using exclusive-or XOR. The multiplication is performed using AND. Depending on machine size, we can do 8, 16, 32, or 64 coefficients simultaneously.

Since multiplication is commutative, we can choose either A or B to be summed. In the code, we shift B once for each term and use the count of that offset as the index into the lambda vector table to find each proper shift of A.

The first routine we need is a rotate. Going right, the least significant bit needs to be placed into the most significant bit. Going left we do the opposite. Here are single bit rotation routines.

```
void rot_left(a)
FIELD2N *a;
```

```
{
        INDEX i;
        ELEMENT bit,temp;
        bit = (a->e[0] & UPRBIT) ? 1L : 0L;
        for (i=NUMWORD; i>=0; i--) {
             temp = (a->e[i] & MSB) ? 1L : 0L;
             a->e[i] = ( a->e[i] << 1) | bit;
             bit = temp;
        }
        a->e[0] &= UPRMASK;
}

void rot_right(a)
FIELD2N *a;
{
        INDEX i;
        ELEMENT bit,temp;
        bit = (a->e[NUMWORD] & 1) ? UPRBIT : 0L;
        SUMLOOP(i) {
             temp = ( a->e[i] >> 1) | bit;
             bit = (a->e[i] & 1) ? MSB : 0L;
             a->e[i] = temp;
        }
        a->e[0] &= UPRMASK;
}
```

The first routine is actually a squaring operation, and the second is a square root operation in a normal basis. This is a great speed advantage over polynomial basis applications. However there are more things we have to do than just square and square root.

For every sum we need two shifts of A. So it is faster to compute all the shifts once and store them in a lookup table. This is the first `for()` loop in the `opt_mul` routine. This section of code would be much faster if implemented assembler. It might even be possible to eliminate it for those processors that have barrel shifters.

```
/* Generalized Optimal Normal Basis multiply. Assumes two dimensional
 Lambda vector already initialized. Will work for both type 1 and
 type 2 ONB. Enter with pointers  to FIELD2N a, b and result area c.
 Returns with c = a*b over GF(2^NUMBITS).
*/
```

```
void opt_mul(a, b, c)
FIELD2N *a, *b, *c;
{
        INDEX i, j;
        INDEX k, zero_index, one_index;
        ELEMENT bit, temp;
        FIELD2N amatrix[NUMBITS], copyb;

/* clear result and copy b to protect original */
        null(c);
        copy(b, &copyb);

/* To perform the multiply we need two rotations of the input a. Performing all
 the rotations once and then using the Lambda vector as an index into a table
 makes the multiply almost twice as fast.
*/
        copy( a, &amatrix[0]);
        for (i = 1; i < NUMBITS; i++)
        {
              copy( &amatrix[i-1], &amatrix[i]);
              rot_right( &amatrix[i]);
        }
```

The basic idea of the multiply is really simple: Look up the two terms we need from the shifted table, XOR them, then AND that with the present B vector. XOR this term with the C vector as a partial sum. Rotate the B vector and repeat until all m partial terms have been summed with the result.

```
/* Lambda[1][0] is non existent, deal with Lambda[0][0] as special case. */
        zero_index = Lambda[0][0];
        SUMLOOP (i) c->e[i] = copyb.e[i] & amatrix[zero_index].e[i];

/* main loop has two lookups for every position. */
        for (j = 1; j<NUMBITS; j++)
        {
             rot_right( &copyb);
             zero_index = Lambda[0][j];
             one_index = Lambda[1][j];
             SUMLOOP (i) c->e[i] ^= copyb.e[i] &
```

```
            (amatrix[zero_index].e[i] ^amatrix[one_index].e[i]);
        }
}
```

Optimizing the code for speed is obviously important but implementation dependent. For a Type I ONB, there is no need for two lookup vectors. In fact, we only need to perform the multiplication (AND) of A rotated half its length with B and take the "trace" of the result, which gets summed with the single lookup vector result for a very high speed Type I ONB algorithm. The trace function for ONB it is identical to a parity bit calculation, see [Ros98], p.95.

Chapter 3

Inversion over optimal normal basis

3.1 A straightforward method

To begin with, let us look at a very straightforward method of inversion over a normal basis. Observe that if $\alpha \in GF(2^m)$, $\alpha \neq 0$, then using Fermat's Little Theorem, $\alpha^{-1} = \alpha^{2^m-2} = (\alpha^{2^{m-1}-1})^2$. At this point we could just exponentiate directly. But this would require m squarings and $m-1$ multiplies. For m on the order of 200 this would be exceptionally slow.

The following is a way around the problem. The most efficient technique, from the point of view of minimizing the number of multiplications, was proposed by Itoh, Teechai and Tsujii in 1986.

If m is odd, then since $2^{m-1}-1 = (2^{(m-1)/2}-1)(2^{(m-1)/2}+1)$, we have $\alpha^{2^{m-1}-1} = \left(\alpha^{2^{(m-1)/2}-1}\right)^{2^{(m-1)/2}+1}$. Hence it takes only one multiplication to evaluate $\alpha^{2^{m-1}-1}$ once the quantity $\alpha^{2^{(m-1)/2}-1}$ has been computed (we are ignoring the cost of squaring).

If m is odd, then we have $\alpha^{2^{m-1}-1} = \alpha^{2(2^{(m-2)/2}-1)(2^{(m-2)/2}+1)+1}$, and consequently it takes two multiplications to evaluate $\alpha^{2^{m-1}-1}$ once $\alpha^{(2^{(m-2)/2}-1)}$ has been computed. The procedure is then repeated recursively.

Here is an example: Consider the field $GF(2^{155})$. We have

$2^{155} - 2 = 2(2^{77} - 1)(2^{77} + 1)$,

$2^{77} - 1 = 2(2^{19} - 1)(2^{19} + 1)(2^{38} + 1) + 1$,

$2^{19} - 1 = 2(2^9 - 1)(2^9 + 1) + 1,$

$2^9 - 1 = 2(2 + 1)(2^2 + 1)(2^4 + 1) + 1,$

so an inversion in $GF(2^{155})$ takes ten multiplications.

It can easily be verified by induction that this method requires exactly $I(m) = \lfloor \log_2(m-1) \rfloor + \varpi(m-1) - 1$ field multiplications, where $\varpi(m-1)$ denotes the number of 1's in the binary expansion of $m-1$, see [Men93].

3.2 High-speed inversion for optimal normal basis

3.2.1 Using the almost inverse algorithm

The following method combines polynomial basis and optimal normal basis to implement a very high speed inversion algorithm. Most research into elliptic curve crypto systems over the past few years has concentrated on finding mathematical tricks to increase throughput. Many of these tricks rely on the structure of certain fields. Others rely on finding specific irreducible polynomials.

In finite field arithmetic, most inversion routines require two to five times as long as a multiply. The routine presented here was first developed by Dave Dahm, see [Ros98]. This inversion routine is as fast as a **single** optimal normal basis multiply. But notice, however that special polynomial bases can be made much faster in general..

Dahm's inversion algorithm is based on [SOOS95] for the "almost inverse algorithm" and on the results of Chapter 1 for the irreducible polynomial, which converts from Type I and Type II ONB to polynomial basis. The "almost inverse algorithm" is based on Euclid's algorithm, but it leaves a final factor of x^k, which has to be divided out. Fortunately, this is a trivial operation for ONB, so the conversion to and from polynomial basis and the elimination of the final factor turns out to be much faster than the inversion algorithm described in section 1 of this chapter.

The basic idea of the almost inverse algorithm is the same as polynomial inversion. We keep the following formula constant, see [Ros98], p.285: $B \cdot F + C \cdot G = 1 \bmod M$, where M is the prime polynomial. For an ONB the prime polynomial is very specific; we will describe it later.

The **almost inverse algorithm** is initialized with:

$B = 1$

$C = 0$

F =Source Polynomial

$G = M$ =Prime Polynomial

$k = 0$

The variables B, C, F, and G are polynomials, and k is an integer. The **almost inverse algorithm** proceeds by repeating the following steps:

While the last bit of F is 0:
    shift F right (divide by x)
    shift C left (multiply by x)
    increment k by 1.
If $F = 1$, return B, k. (3.1)
If degree(F) < degree(G), then exchange F, G and B, C.
$F = F + G$.
$B = B + C$.
Repeat entire loop.

The reason this is an "almost" inverse routine is that we have the final result with an extra factor of x^k, which must be divided out. For optimal normal basis this factor is very easy to find, and we don't need a full-scale multiply to remove it.

There are additional tricks that Schroeppel et al. [SOOS95] include in their paper to help speed up the algorithm. These include using two separate loops rather than exchanging polynomials, using registers, and expanding structures to explicitly named variables. Many of these tricks reduce portability but increase throughput.

In Chapter 1, Section 3, an irreducible polynomial for a Type IIa optimal normal basis, for which 2 is primitive in $\mathbb{Z}_{2m+1}$, is: $M_{IIa} = 1 + x + x^2 + \ldots + x^{2m}$.

It follows that for a Type I ONB (with 2 primitive in $\mathbb{Z}_{m+1}$) an irreducible polynomial is given by: $M_I = 1 + x + x^2 + \ldots + x^m$.

For the Type IIb optimal normal basis, for which 2 generates the quadratic residues, the polynomial $M_{IIa} = 1 + x + x^2 + \ldots + x^{2m}$ has two factors, so it is not irreducible. Fortunately, the almost inverse algorithm will still work. This is due to the requirement that the source polynomial be relatively prime to the basis polynomial M. This will always be the case for sources of Type II ONB.

Let us look at how to convert a Type I normal basis representation to a polynomial basis representation. Each term in a normal basis is of the form: $a_i x^{2^i}$. But since

2 is a generator modulo $m+1$, we can also write this term as: $a_i x^k$. Since $M_I = 1 + x + x^2 + \ldots + x^m$, we can move the i th coefficient of Type I ONB to the k position to convert form normal to polynomial basis. This is just a permutation of all the bits, and we only have to move the ones that are set.

The only problem we have is when $2^i = m$. This term is not represented in the polynomial basis, because it has the power of the most significant coefficient in $M_I = 1 + x + x^2 + \ldots + x^m$. In the normal basis case, there is no representation for x^0. For this case, we map the coefficient of the most significant bit of the ONB to the least significant bit of the polynomial representation, and nothing is lost. Here is Rosing's explanation: As vector spaces, the basis vectors are exactly the same (except for order) at all places except 1. Each set of basis vectors contains $x, x^2, x^3 \ldots, x^{m-1}$. The only difference is that ONB contains x^m and polynomial basis contains 1. In the Type I field these two representations are related by: $1 + x + x^2 + \ldots + x^m = 0$.

For a Type II optimal normal basis, things are almost as easy, [Ros98], p.287. The polynomial basis is twice as long as the normal basis. For each bit in the ONB we will have 2 bits in the polynomial basis. It turns out that the 2 bits are palindromes; if bit i is set, then so is bit $2m + 1 - i$.

In chapter 1, section 3, we took the basis to be of the form: $\gamma + \gamma^{-1} = \beta$. The combination of these two terms created a new one, which was the basis for the Type II ONB. Using this, along with the polynomial $M_{IIa} = 1 + x + x^2 + \ldots + x^{2m}$, we can derive a simple conversion scheme to go from Type II ONB to polynomial representation and back by flipping just 2 bits in a known permutation. Since the permutation is predefined for any ONB, we can create a lookup table at initialization, along with the creation of the multiplication vectors.

For simplicity, let $p = 2m + 1$. Since we are doing base 2 field math, we have: $(\gamma + \gamma^{-1})^{2^i} = \gamma^{2^i} + \gamma^{-2^i} = \gamma^{2^i} + \gamma^{p-2^i}$. Just as with the Type I ONB, we have a permutation form the coefficient of each term to a corresponding one in the polynomial representation. But there are now 2 bits that we have to map: one that maps 2^i to k and one that maps $p - 2^i$ to $p - k$.

By creating the permutation map as a set of indices and bit masks, the conversion is very fast. The almost inverse algorithm is then simple to invoke. The whole process is identical for both Type I and Type II (other than the number of bits needed).

3.2.2 Faster inversion, preliminary subroutines

There are several steps to creating this faster inversion routine. The first is a pair of tables used in the conversion from normal basis to polynomial and back. Another is the multiplication of x^{-k} as the final step in the almost inverse algorithm. Then there is the inversion routine itself. The latter has been expanded using some of the suggestions in [SOOS95].

Let us start with some new constants, which are defined in a header file:

```
#define LONGWORD  (field_prime/WORDSIZE)
#define LONGSHIFT ((field_prime-1)%WORDSIZE)
#define LONGBIT   (1L<<(LONGSHIFT-1))
#define LONGMASK  (~(-1L<<LONGSHIFT))
```

These are used to create the polynomial basis representation, as we will see below. `LONGWORD` is the number of ELEMENTS needed to hold the polynomials we will be using. `LONGSHIFT` is the number of left shifts needed to get to the most significant bit in the most significant `ELEMENT` of the polynomial representation. `LONGBIT` is the most significant bit we will need in the polynomial basis, and `LONGMASK` is a mask that keeps the most significant bits in the most significant `ELEMENT` of the polynomial basis.

Next comes some additional initialization code, which needs to be called only once. The initializations are for the following arrays, which need to be added to the .c file (chapter 3 subroutines):

```
static INDEX      log2[field_prime+1];
static INDEX      two_inx[field_prime];
static ELEMENT    two_bit[field_prime];
static unsigned char shift_by[256];
```

The variable `log2` is the same; it is just global for use in the conversion process. The two arrays `two_*` are used to find specific bits. Rather than save the bit position, as in the `log2` array, we save the `ELEMENT` index and bit offset within an `ELEMENT`. This speeds execution with only a minor increase in memory requirements. The array `shift_by` is used as one of the speed enhancements. Instead of shifting the F polynomial only once and incrementing k as stated in the algorithm, we do several shifts at once if possible.

We have seen the `genlambda` routines before; the only change there is the removal of variable `log2`, because it is now global. The routine `init_two` fills in the arrays

two_inx and two_bit. It's pretty simple too, [Ros98], p.288.

```
static void init_two(void)
{
        INDEX n, i, j;
        j = 1;
        n = (field_prime-1)/2;
        for ( i=0; i<n; i++ ) {
              two_inx[i] = LONGWORD-(j / WORDSIZE);
              two_bit[i] = 1L << (j % WORDSIZE);
              two_inx[i+n] = LONGWORD-((field_prime-j) / WORDSIZE);
              two_bit[i+n] = 1L << ((field_prime-j) % WORDSIZE);
              j = (j << 1) % field_prime;
        }
        two_inx[field_prime-1] = two_inx[0];
        two_bit[field_prime-1] = two_bit[0];

        for ( i=1; i<256; i++ )
             shift_by[i] = 0;
        shift_by[0] = 1;
        for ( j=2; j<256; j+=j )
        for ( i=0; i<256; i+=j )
             shift_by[i]++;
}
```

The variable shift_by is really simple, [Ros98], p.289. By masking the last 8 bits of a FIELD2N ELEMENT, and using that as an index into the array, it tells us how many bits are clear.

The initialization code, which was put in the main routine in Chapter 3, is now combined with the above initialization routine.

```
void init_opt_math()
{
#ifdef TYPE2
       genlambda2();
#else
       genlambda();
#endif
       init_two();
}
```

There is yet another type of variable. It is used to hold double-size arrays for Type II optimal normal basis conversions to "customary" polynomial basis representations. For Type I optimal normal basis, it is the same size as FIELD2N, but this makes the code more general.

```
typedef struct {
        ELEMENT e[LONGWORD+1];
} CUSTFIELD;
```

Basic operations on this structure are similar to operations on FIELD2N and DBLFIELD structures. To copy values from one CUSTFIELD to another, we use the following code segment:

```
void copy_cust (a,b)
CUSTFIELD *a,*b;
{
        INDEX i;
        for (i=0; i<=LONGWORD; i++) b->e[i] = a->e[i];
}
```

And to clear out a variable, the following routine is used.

```
void null_cust(a)
CUSTFIELD *a;
{
        INDEX i;
        for (i=0; i<=LONGWORD; i++) a->e[i] = 0;
}
```

The last step of the almost inverse algorithm is the multiplication of the extra factor x^{-k}. Let us see how this routine works.

```
/* set b = a * u^n, where n>0 and n <= field_prime */

void cus_times_u_to_n(CUSTFIELD *a, int n, CUSTFIELD *b)
{
#define SIZE (2*LONGWORD+2)
        ELEMENT w, t[SIZE+1];
        INDEX i, j, n1, n2, n3;
```

The constant SIZE is used here to make the dimension of the array t hold twice as many bits as the normal basis representation. The check for n being equal to field_prime is robust code; in practice, it should never happen.

```
if ( n == field_prime ) {
     copy_cust(a, b);
     return;
}
```

The next line of code clears the t array. Since this array is special, this could be done with a pointer, too. The variables n1 and n2 determine the index into the t array and the bit within an ELEMENT of that array. The offset is from the end of the array rather than the start-this is why SIZE - n1 appears everywhere.

```
for ( j=0; j<=SIZE; j++ ) t[j] = 0;
n1 = n / WORDSIZE;
j = SIZE-n1;
n2 = n & (WORDSIZE-1);
```

The next block of code then shifts every word of the input. If the bit position is 0, then whole ELEMENTS can be moved. Otherwise, the variable n3 is used to shift a word up a portion, and n2 is used to shift a word down a portion. Each word is put into t shifted by the amount SIZE - n1. Note that j is decremented after being used as the subscript, so that the second line in the for loop uses the previous value of j.

```
if ( n2 ) {
   n3 = WORDSIZE-n2;
   for ( i=LONGWORD; i>=0; i-- ) {
        t[j--] |= a->e[i] << n2;
        t[j] |= a->e[i] >> n3;
   }
} else {
   for ( i=LONGWORD; i>=0; i-- ) {
   t[j--] |= a->e[i];
}
}
```

At this point we have actually multiplied the input by x^n (presumably the value for n was computed modulo field_prime). This is just a shift by n bit positions, since each coefficient is multiplied by its appropriate power of x.

The next block of code then shifts the upper portion by the correct amount to account for the fact that the field size does not fill a complete word. This moves the upper portion of an ELEMENT at an offset, which holds the most significant bits of the result back to the least significant ELEMENT position. The code works because: $x^p - 1 = (x-1)M$, where M is M_{I} or M_{IIa} as discussed before. Now $x^p = 1 + (x-1)M$. Multiply both sides by x^k and reduce modulo M, we have the relationship: $x^{p+k} = x^k$. So every x^{p+k} reduced modulo M will simply be x^k.

```
        n3 = LONGSHIFT+1;
        i = SIZE-LONGWORD;
        for ( j=SIZE; j>=SIZE-n1; j-- ) {
            t[j] |= t[i--] >> n3;
            t[j] |= t[i] << (WORDSIZE-n3);
        }
```

The final step is to move the data from the t array to the output b array. This includes one important check: If the coefficient to x^{p-1} is set, we can reduce this power by adding in the rest of M (since $M = 0 \bmod M$). This is just all bits set, and the variable w is used to contain these bits for each ELEMENT.

```
        w = t[SIZE-LONGWORD] & (1L << LONGSHIFT ) ? ~0 : 0;
        for ( i=0; i<=LONGWORD; i++ )
            b->e[i] = t[i+SIZE-LONGWORD] ^w;
        b->e[0] &= LONGMASK;
#undef SIZE
}
```

Finally, the upper bits are cleared to complete the operation and clean up the output. (The term SIZE was previously defined at the beginning of the routine. Since the name is common, it is good to limit the scope.)

3.2.3 Faster inversion, the code

The fast inversion algorithm does not translate into pretty code. It uses all the methods described previously, including the conversion from optimal normal basis to polynomial basis and back using M_{I} or M_{IIa}. The almost inverse algorithm is included, along with several speed ups mentioned in [SOOS95].

```
/* This algorithm is the Almost Inverse Algorithm of Schroeppel, et al.
   given in ''Fast Key Exchange with Elliptic Curve Systems''
```

```
*/
void opt_inv(FIELD2N *a, FIELD2N *dest)
{
        CUSTFIELD f, b, c, g;
        INDEX i, j, k, m, n, f_top, c_top;
        ELEMENT bits, t, mask;

 /* f, b, c, and g are not in optimal normal basis format: they are held
    in 'customary format', i.e. a0 + a1*u^1 + a2*u^2 + ...; For the
    comments in this routine, the polynomials are assumed to be
    polynomials in u. */
```

The first thing is to initialize G to the prime polynomial M, as in the algorithm 3.1. This is all bits set, including 1 extra bit past the defined limit of `LONGSHIFT`.

```
        /* Set g to polynomial (u^p-1)/(u-1) */

        for ( i=1; i<=LONGWORD; i++ )
            g.e[i] = ~0;
        g.e[0] = LONGMASK | (1L << LONGSHIFT);
```

The next chunk of code converts the input value a from normal basis to polynomial basis using the predefined offset and bit masks created in `init_two`. For a Type II normal basis we need 2 bits set.

```
        /* Convert a to 'customary format', putting answer in f */

        null_cust(&f);
        j = 0;
        for ( k=NUMWORD; k>=0; k-- ) {
            bits = a->e[k];
            m = k>0 ? WORDSIZE : UPRSHIFT;
            for ( i=0; i<m; i++ ) {
                if ( bits & 1 ) {
                     f.e[two_inx[j]] |= two_bit[j];
#ifdef TYPE2
                     f.e[two_inx[j+NUMBITS]] |= two_bit[j+NUMBITS];
#endif
                }
```

```
            j++;
            bits >>= 1;
        }
    }
```

After initializing the remaining variables of algorithm 3.1. We then eliminates powers of x (which called `u here`), as stated in the first step of the almost inverse algorithm. The variables `c_top` and `f_top` are used to track the unused (zeroed out) ELEMENTS in `b`, `c` and `f`, `g`, respectively. This also helps speed up the code, since it eliminates loop executions, which have null results.

```
    /* Set c to 0, b to 1, and n to 0 */
    null_cust(&c);
    null_cust(&b);
    b.e[LONGWORD] = 1;
    n = 0;

/* Now find a polynomial b, such that a*b = u^n */

/* f and g shrink, b and c grow. The code takes advantage of this.
c_top and f_top are the variables which control this behavior */

    c_top = LONGWORD;
    f_top = 0;
    do {
        i = shift_by[f.e[LONGWORD] & 0xff];
        n+=i;
/* Shift f right i (divide by u^i) */
        m = 0;
        for ( j=f_top; j<=LONGWORD; j++ ) {
            bits = f.e[j];
            f.e[j] = (bits>>i) | ((ELEMENT)m << (WORDSIZE-i));
            m = bits;
        }
    } while ( i == 8 && (f.e[LONGWORD] & 1) == 0 );
```

Everything is now initialized, and we're ready for the main loop.

If $F = 1$, then the routine is finished. This will happen on occasion if you enter with a single bit set; the above code shifts the bit down and the main routine would not be needed. We check for that here.

```
for ( j=0; j<LONGWORD; j++ )
    if ( f.e[j] ) break;
if ( j<LONGWORD || f.e[LONGWORD] != 1 )
{
```

Assuming F is not equal to 1, we enter the "almost inverse algorithm's" main loop.

```
/* There are two loops here: whenever we need to exchange f with g and
   b with c, jump to the other loop which has the names reversed! */
    do
    {
    /* Shorten f and g when possible */
       while ( f.e[f_top] == 0 && g.e[f_top] == 0 ) f_top++;
    /* f needs to be bigger - if not, exchange f with g and b with c.
      (Actually jump to the other loop instead of doing the exchange)
       The published algorithm requires deg f >= deg g, but we don't
       need to be so fine */
       if ( f.e[f_top] < g.e[f_top] ) goto loop2;
loop1:
    /* f = f+g, making f divisible by u */
       for ( i=f_top; i<=LONGWORD; i++ )
           f.e[i] ^= g.e[i];
    /* b = b+c */
       for ( i=c_top; i<=LONGWORD; i++ )
           b.e[i] ^= c.e[i];
       do {
          i = shift_by[f.e[LONGWORD] & 0xff];
          n+=i;
    /* Shift c left i (multiply by u^i), lengthening it if needed */
          m = 0;
          for ( j=LONGWORD; j>=c_top; j-- ) {
              bits = c.e[j];
              c.e[j] = (bits<<i) | m;
              m = bits >> (WORDSIZE-i);
          }
          if ( m ) c.e[c_top=j] = m;
    /* Shift f right i (divide by u^i) */
          m = 0;
          for ( j=f_top; j<=LONGWORD; j++ ) {
```

```
            bits = f.e[j];
            f.e[j] = (bits>>i) | ((ELEMENT)m << (WORDSIZE-i));
            m = bits;
        }
    } while ( i == 8 && (f.e[LONGWORD] & 1) == 0 );
/* Check if we are done (f=1) */
    for ( j=f_top; j<LONGWORD; j++ )
        if ( f.e[j] ) break;
} while ( j<LONGWORD || f.e[LONGWORD] != 1 );
```

There are two loops here that are identical; only the variable names have been flipped to do the correct operations. The last step in the loop is to check to see if $F = 1$. If it is, then the above `while` loop ends. Note that if we exit the loop this way, the value of `j` will always be equal to `LONGWORD`. The use of `goto`'s may violate most C programming styles, but it is very useful here.

```
if ( j>0 )
goto done;
do {
/* Shorten f and g when possible */
    while ( g.e[f_top] == 0 && f.e[f_top] == 0 ) f_top++;
/* g needs to be bigger - if not, exchange f with g and b with c.
   (Actually jump to the other loop instead of doing the exchange)
   The published algorithm requires deg g >= deg f, but we don't
   need to be so fine */
    if ( g.e[f_top] < f.e[f_top] ) goto loop1;
loop2:
/* g = f+g, making g divisible by u */
    for ( i=f_top; i<=LONGWORD; i++ )
        g.e[i] ^= f.e[i];
/* c = b+c */
    for ( i=c_top; i<=LONGWORD; i++ )
        c.e[i] ^= b.e[i];
    do
    {
        i = shift_by[g.e[LONGWORD] & 0xff];
        n+=i;
/* Shift b left i (multiply by u^i), lengthening it if needed */
        m = 0;
        for ( j=LONGWORD; j>=c_top; j-- ) {
```

```
                bits = b.e[j];
                b.e[j] = (bits<<i) | m;
                m = bits >> (WORDSIZE-i);
            }
            if ( m ) b.e[c_top=j] = m;

        /* Shift g right i (divide by u^i) */
            m = 0;
            for ( j=f_top; j<=LONGWORD; j++ ) {
                bits = g.e[j];
                g.e[j] = (bits>>i) | ((ELEMENT)m << (WORDSIZE-i));
                m = bits;
            }
        } while ( i == 8 && (g.e[LONGWORD] & 1) == 0 );

        /* Check if we are done (g=1) */
        for ( j=f_top; j<LONGWORD; j++ )
            if ( g.e[j] ) break;
    } while ( j<LONGWORD || g.e[LONGWORD] != 1 );
      copy_cust(&c, &b);
    }
```

The guts of the routine are straightforward executions of the almost inverse algorithm. The variable `c_top` and `f_top` are both adjusted along the way to reduce the number of execution loops as the procedure progresses. If we exit the last loop, then we have to swap `b` and `c` so we can finish the algorithm correctly. The shifting is done using the least significant byte of `f` or `g` as an index into the `shift_by` array. Instead of calling a subroutine, we has made the code inline for this shift.

The final stage is to multiply b by the appropriate power of x and then convert that value back to normal basis so we'll get the right answer.

```
done:
    /* Now b is a polynomial such that a*b = u^n, so multiply b by u^(-n) */
       cus_times_u_to_n(&b, field_prime - n % field_prime, &b);

    /* Convert b back to optimal normal basis form (into dest) */

        if ( b.e[LONGWORD] & 1 )
            one(dest);
```

```
        else
            null(dest);
            j = 0;
            for ( k=NUMWORD; k>=0; k-- ) {
                 bits = 0;
                 t = 1;
                 mask = k > 0 ? ~0 : UPRMASK;
                 do {
                    if ( b.e[two_inx[j]] & two_bit[j] ) bits ^= t;
                    j++;
                    t <<= 1;
                 } while ( t&mask );
                 dest->e[k] ^= bits;
            }

} /* opt_inv */
```

This version of the inverse algorithm requires about as much time to compute as a normal basis multiply. This routine is at least ten times faster than the straightforward method found in section 1 of this chapter. So, although it looks messy, it is much quicker to execute. As usual, it takes up more code space to reduce run time, but inversion is called by every elliptic sum or doubling, see chapter 4. It is extremely useful to use this if one has ROM to spare.

Chapter 4

Elliptic Curve Cryptography over $GF(2^m)$

Elliptic curve cryptography (ECC) was proposed independently by Victor Miller and Neil Koblitz in the mid-eighties. 160bit ECC are considered to be of the same security level as 1024bit RSA. Elliptic curve Cryptography offers many advantages over RSA. For the same level of security, especially over 160bit, ECC is faster, requires less computing power, permits reduction in key and certificate size which saves memory and bandwidth.

ECC has received increased commercial acceptance as evidenced by its inclusion in standards by accredited standards organizations such as IEEE (Institute of Electrical and Electronics Engineers), ISO (International Standards Organization), NIST (National Institute of Standards and Technology), and ANSI (American National Standards Institute), see [ECC standard]. It is being promoted as the best method for implementing digital signatures for banking smartcard applications.

4.1 Mathematics of elliptic curves

There are several kinds of defining equations for elliptic curves, but the most common are the Weierstrass equations. For the binary finite fields $GF(2^m)$, the Weierstrass equation is

$$y^2 + xy = x^3 + ax^2 + b \tag{4.1}$$

where a and b are elements of $GF(2^m)$ with $b \neq 0$.

There is another kind of Weierstrass equation over $GF(2^m)$, giving what are called supersingular curves. However, these curves are cryptographically weak, see section

3 below; thus they are omitted.

Given a Weierstrass equation, the elliptic curve E consists of the solutions (x, y) over $GF(2^m)$ to the defining equation, along with an additional element called the point at infinity (denoted ϑ). The points other than ϑ are called finite points. The number of points on E (including ϑ) is called the order of E and is denoted by $\#E(GF(2^m))$.

There is an addition operation on the points of an elliptic curve which possesses the algebraic properties of ordinary addition (e.g. commutativity and associativity). This operation can be described geometrically as follows.

Define the inverse of the point $P = (x, y)$ to be $(x, x + y)$. Then the sum $P + Q$ of the points P and Q is the point R with the property that P, Q, and $-R$ lie on a common line. The point at infinity ϑ plays a role analogous to that of the number 0 in ordinary addition. Thus $P + \vartheta = P$, $P + (-P) = \vartheta$ for all points P. Under this addition operation, it can be shown that E forms a group.

When implementing the formulae for elliptic curve addition, it is necessary to distinguish between doubling (adding a point to itself) and adding two distinct points that are not inverses of each other, because the formulae are different in the two cases. Besides this, there are also the special cases involving ϑ. By full addition is meant choosing and implementing the appropriate formula for the given pair of points. Algorithms for full addition are given here:

Full Addition	
Input:	a field $GF(2^m)$; coefficients a, b for an elliptic curve $E : y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$; points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E.
Output:	the point $P_2 := P_0 + P_1$.
1.	If $P_0 = \vartheta$, then output $P_2 \leftarrow P_1$ and stop
2.	If $P_1 = \vartheta$, then output $P_2 \leftarrow P_0$ and stop
3.	If $x_0 \neq x_1$ then
	3.1 set $\lambda \leftarrow (y_0 + y_1)/(x_0 + x_1)$
	3.2 set $x_2 \leftarrow a + \lambda^2 + \lambda + x_0 + x_1$
	3.3 go to step 7
4.	If $y_0 \neq y_1$ then output $P_2 \leftarrow \vartheta$ and stop
5.	If $x_1 = 0$ then output $P_2 \leftarrow \vartheta$ and stop
6.	Set
	6.1 $\lambda \leftarrow x_1 + y_1/x_1$
	6.2 $x_2 \leftarrow a + \lambda^2 + \lambda$
7.	$y_2 \leftarrow (x_1 + x_2)\lambda + x_2 + y_1$
8.	$P_2 \leftarrow (x_2, y_2)$

The above algorithm requires 2 general multiplications, a squaring, and a multiplicative inversion. To subtract the point $P = (x, y)$, one adds the point $-P = (x, x + y)$.

Elliptic curve points can be added but not multiplied. It is, however, possible to perform scalar multiplication, which is another name for repeated addition of the same point. If k is a positive integer and P a point on an elliptic curve, the scalar multiple kP is the result of adding k copies of P. Thus, for example, $5P = P + P + P + P + P$. The notion of scalar multiplication can be extended to zero and the negative integers via $0P = \vartheta$, $(-k)P = k(-P)$.

Scalar multiplication can be performed efficiently by the addition-subtraction method outlined below.

Elliptic Curve Scalar Multiplication

Input: an integer k and an elliptic curve point P.

Output: the elliptic curve point kP.

1. If $k = 0$ then output ϑ and stop.
2. If $k < 0$ the set $Q \leftarrow (-P)$ and $K \leftarrow (-k)$, else set $Q \leftarrow P$ and $K \leftarrow k$.
3. Let $h_l h_{l-1} \ldots h_1 h_0$ be the binary representation of $3K$, where the most significant bit h_l is 1.
4. Let $K_l K_{l-1} \ldots K_1 K_0$ be the binary representation of K.
5. Set $S \leftarrow Q$.
6. For i from $l-1$ downto 1 do
   Set $S \leftarrow 2S$.
   If $h_i = 1$ and $K_i = 0$ then compute $S \leftarrow S + Q$.
   If $h_i = 0$ and $K_i = 1$ then compute $S \leftarrow S - Q$.
7. Output S.

There are several modifications that improve the performance of this algorithm. These methods are summarized in [Gor98]. Elliptic Curve Multiplication is the computation that dominates the operations of Elliptic Curve Cryptography.

4.2 Elliptic Curve Cryptography

Let us explain some more terms of Elliptic Curves first.

If u is the order of an elliptic curve over $GF(2^m)$, then the Hasse bound is $2^m + 1 - 2\sqrt{2^m} \leq u \leq 2^m + 1 + 2\sqrt{2^m}$. Thus the order of an elliptic curve over $GF(2^m)$, $\#E(GF(2^m))$, is approximately 2^m.

The order of a point P on an elliptic curve is the smallest positive integer n such that $nP = \vartheta$. The order always exists and divides the order of the curve $\#E(GF(2^m))$. If k and l are integers, then $kP = lP$ if and only if $k \equiv l \bmod n$.

Suppose that the point P on E has large prime order n where n^2 does not divide the order of the curve $\#E$. P is called a base point of the curve. Then a key pair can the defined as follows. (However, one needs to know what are good and secure curves, see section 4 below.)

EC key pair generation	
Input:	elliptic curve E with base point P of order n.
Output:	private key d and the corresponding public key Q
1.	select a statistically unique and unpredictable integer d in the interval $[1 \ldots n-1]$
2.	compute the point $Q = dP$ on the curve E.
3.	the private key is d, the public key is Q.

Of course, public key is public, the curve is also public, and a user must kept his own secret key secret.

It is necessary to compute an elliptic curve discrete logarithm, see section 3 below, in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the elliptic curve discrete logarithm problem, ECDLP. 160 bits Elliptic Curve Cryptography are being viewed as equivalent to 1024bits RSA in terms of security. It is still extremely difficult to break them in the year 2000 even if one hundred thousands of computers can be used together, see section 3 below.

Now we **briefly** outline how elliptic curves can be useful in cryptography. See the standards [ECC standard] for more details, nevertheless these are simple protocols.

For encrypting or decrypting a file, it is important to note that these are essentially carried out by traditional secret key algorithm, such as DEA. The key for use with the traditional secret key algorithm is generated randomly in a particular session, and is called a session key. The role of ECC is to encrypt or decrypt this session key only. For 160bit ECC, we should choose a 80bit session key for a comparable security level, see [ECC standard].

The advantage of public key cryptography here is that no key management is necessary. When Alice wants to send a message to Bob, he only needs to find out the public key of Bob, which is of course publicly available. Alice can then encrypt messages using this public key, none can decrypt them except Bob, as only Bob processes the private key of Bob. The reader can then imagine what would the situation be if traditional secret key algorithm was used when each of his Bob's (girl) friends wanted to communicate to Bob.

Digital Signature are the electronic equivalent of traditional handwritten signatures. Digital Signature is hard to forge. One of its main function is authentication, another is data integrity.

How can Bob proves that a message is written by Bob himself?

Answer: Bob applies his private key, $Private_{bob}$, to the message using a digital signature algorithm to generate a digital signature, see below. He then sends the message along with the digital signature to Alice. Alice checks, or verifies, the signature by applying Bob's public key, $Public_{bob}$, to the signature using a digital signature verification algorithm. Since only Bob is in possession of $Private_{bob}$, Alice knows: if in case the signature verification passes, the message was from Bob and that the data has not been changed.

How can Bob proves that he is Bob, to Alice?

Answer: By the process of random challenge. Alice randomly chooses a message and ask Bob to sign it. Alice can then verify the signature. Since none except Bob, as only Bob processes $Private_{bob}$, can produce a valid signature, Alice knows that he is Bob if the verification passes.

ECDSA signature generation

Input:	message M, private key d,
	elliptic curve E with base point P of order n.
Output:	signature for M.
1.	select a statistically unique and unpredictable integer k
	in the interval $[1 \ldots n-1]$
2.	compute $kP = (x_1, y_1)$
3.	compute $r = x_1 \bmod n$
4.	compute e =Hash(M)
5.	compute $s = k^{-1}(e + dr) \bmod n$
6.	the signature for M is (r, s)

ECDSA signature verification

Input:	message M, signature for message (r, s), public key Q,
	elliptic curve E with base point P of order n.
Output:	accept or reject the signature.
1.	compute e =Hash(M)
2.	compute $s^{-1} \bmod n$
3.	compute $u_1 = e \cdot s^{-1} \bmod n$
4.	compute $u_2 = r \cdot s^{-1} \bmod n$
5.	compute $u_1 P + u_2 Q = (x_1, y_1)$
6.	compute $v = x_1 \bmod n$
7.	accept the signature if $v = r$

Hash is a one-to-one function that maps a message M to a integer of length 160bit, it is also called a message digest function and runs very fast, see [ECC standard].

Since hash functions or modulus computations are relatively cheap, it is clear that Elliptic Curve Scalar Multiplication is the computation that dominates elliptic curve cryptography.

Signature	~1 single EC point multiplication,
Signature verification	~1.5 times of a single EC point multiplication,
Encryption	~2 EC point multiplications
Decryption	~slightly > a single EC point multiplication,

The reason for the number 1.5 rather than 2 in signature verification is due to the time taken to do a simultaneous elliptic curve multiplications.

4.3 Elliptic curve discrete log problem

Let E be an elliptic curve defined over a finite field $GF(2^m)$. Let $P \in E(GF(2^m))$ be a point of order n, where n is a prime number and $n > 2^{160}$.

The elliptic curve discrete logarithm problem (ECDLP) is the following: given the curve E, P and $Q \in E(GF(2^m))$, determine the integer l, $0 \leq l \leq n-1$, such that $Q = lP$, provided that such an integer exists.

The best general algorithms known to date for ECDLP are the Pollard-ρ method and the Pollard-λ method, [Pol78]. The Pollard-ρ method takes about $\sqrt{\pi n/2}$ steps, where each step is an elliptic curve addition. The Pollard-ρ can be parallelized (see [OW99]) so that if m processors are used, then the expected number of steps by each processor before a single discrete logarithm is obtained is $\sqrt{\pi n/2}/m$. The Pollard-λ method takes about $3.28\sqrt{n}$ steps. It can also be parallelized (see [OW99]) so that if m processors are used, then the expected number of steps by each processor before a single discrete logarithm is obtained is about $2\sqrt{n}/m$.

Some special classes of elliptic curves, including supersingular curves, see section 1, have been prohibited in the ECC standards, see [ECC standard], by the requirement of the MOV condition [MOV93]. These curves have been prohibited because there is a method for efficiently reducing the discrete logarithm problem in these curves to the discrete logarithm problem in a finite field, which is much easier.

Also, the special class of elliptic curves called $GF(2^m)$-anomalous curves have been prohibited by the requirement of the Anomalous condition (see section 4 below) because there is an efficient algorithm for computing discrete logarithms in $E(GF(2^m))$ where E is an anomalous curve over $GF(2^m)$. (i.e. $\#E(GF(2^m)) = 2^m$).

Assume that a 1 MIPS (Million Instructions Per Second) machine can perform 4×10^4 elliptic curve additions per second. (This estimate is indeed high, an ASIC (Application Specific Integrated Circuit) built for performing elliptic curve operations over the field $GF(2^{155})$ has a 40 MHz clock-rate and can perform roughly $40,000$ elliptic additions per second.) Then, the number of elliptic curve additions that can be performed by a 1 MIPS machine in one year is $(4 \times 10^4) \cdot (60 \times 60 \times 24 \times 365) \approx 1.261 \times 10^{12}$.

The following table shows the computing power required to compute a single discrete logarithm for various values of n with the Pollard-ρ method.

Field size (in bits)	Size of n (in bits)	$\sqrt{\pi n/4}$	MIPS years
163	160	2^{80}	8.5×10^{11}
191	186	2^{93}	7.0×10^{15}
239	234	2^{117}	1.2×10^{23}
359	354	2^{177}	1.3×10^{41}
431	426	2^{213}	9.2×10^{51}

Note: The strength of any cryptographic algorithm relies on the best methods that are known to solve the hard mathematical problem that the cryptographic algorithm is based upon, The discovery and analysis of the best methods for any hard mathematical problem is a continuing research topic. The state of the art in solving the ECDLP is subject to change as time goes by.

To put the numbers in the above table into some perspective, the following table, [Odl95], shows the computing power required to factor integers with the 1995 versions of the general number field sieve.

Size of integer to be factored (in bits)	MIPS years
512	3×10^4
768	2×10^8
1024	3×10^{11}
1280	1×10^{14}
1546	3×10^{16}
2048	3×10^{20}

Odlyzko, [Odl95] has estimated that if 0.1% of the world's computing power were available for one year to work on a collaborative effort to break some challenge cipher, then the computing power available would be 10^8 MIPS years in 2004 and 10^{10} to 10^{11} MIPS years in 2014.

As an example, if 10,000 computers each rated at 1,000 MIPS are available, and $n \approx 2^{160}$, then an elliptic curve discrete logarithm can be computed in 85,000 years. Take a look of

http://cristal.inria.fr/~harley/ecdl7/readMe.html
http://cristal.inria.fr/bin/ecdldb?m=AP
http://cristal.inria.fr/bin/ecdldb?m=P;i=501

to see how this can be possible in the future.

A Pentium II-400 is a many-splendored Thing but it is at most a 800 MIPS machine. The fact that the chip has 400 Million cycles per second and that at most 2 instructions can be started per second means I have a machine capable of 800

MIPS by definition of the numbers. (The Pentium has two pipelines for processing instructions, so 2 instructions can start in each cycle.) But MIPS is a bad number because it only tells how many instructions begin in a second, and has nothing to do with how long it takes fully execute the instructions.

On the other hand, how weak is 32bits ECC?

On January 2000, I wrote a program in MAPLE implementing the Pollard-ρ method. It broke the 32 bits ECDLP over GF(p), where p is a prime, in 2.5 minutes and for 40bits in 1.5 hours on a Pentium II-400. **Since 32 bits is the usual wordsize of a processor, it might be the case that many students in many parts of the world are still going on to implementing 32bits Elliptic Curve Cryptography. Yet no explicit evidence as to how weak 32bits ECC are, probably because it is too small, that the usual research journals simply chose to forget it!** *But this was why I wrote this program.* **So although this MAPLE program was rather simple, at least on a second thought, there is still of value in trying to publish it.**

4.4 Finding good and secure curves

4.4.1 Avoiding weak curves

To guard against existing attacks on ECDLP, one should select an elliptic curve E over $GF(2^m)$ such that:

1. The order $\#E(GF(2^m))$ is divisible by a large prime $n > 2^{160}$;
2. The MOV condition holds; and
3. The Anomalous condition holds.

The MOV Condition

The reduction attack of Menezes, Okamoto and Vanstone, [MOV93] reduces the discrete logarithm problem in an elliptic curve over $GF(2^m)$ to the discrete logarithm in the finite field $GF(2^{mB})$ for some $B \geq 1$. The attack is only practical if B is small; this is not the case for most elliptic curves. The MOV condition ensures that an elliptic curve is not vulnerable to these reduction attacks. Most elliptic curves over a field $GF(2^m)$ will indeed satisfy the MOV condition.

Before performing the algorithm, it is necessary to select an MOV threshold. This is a positive integer B such that taking discrete logarithms over $GF(2^{mB})$ is at least as difficult as taking elliptic discrete logarithms over $GF(2^m)$. A value of $B \geq 20$ is

required. Selecting $B \geq 20$ also limits the selection of curves to non-supersingular curves, section 1. Suppose E is an elliptic curve defined over $GF(2^m)$, and n is a prime divisor of $\#E(GF(2^m))$.

The MOV Condition	
Input:	An MOV threshold B, $q = 2^m$, and a prime n.
Output:	The message true if the MOV condition is satisfied for an elliptic curve over $GF(2^m)$ with a base point of order n; the message false otherwise.
1.	Set $t = 1$
2.	For i from 1 to B do 2.1 Set $t = t \cdot q \bmod n$ 2.2 If $t = 1$, then output "false" and stop
3.	Output "true"

The Anomalous Condition

Smart [Sma99] and Satoh and Araki [SA98] showed that the elliptic curve discrete logarithm problem in anomalous curves can be efficiently solved. An elliptic curve E defined over $GF(2^m)$ is said to be $GF(2^m)$-anomalous if $\#\mathrm{E}(GF(2^m)) = 2^m$. The Anomalous condition checks that $\#E(GF(2^m)) \neq 2^m$; this ensures that an elliptic curve is not vulnerable to the Anomalous attack. Most elliptic curves over a field $GF(2^m)$ will indeed satisfy the Anomalous condition.

4.4.2 Finding curves of appropriate order

In order to perform EC-based cryptography it is necessary to be able to find an elliptic curve. The task is as follows: given a field size 2^m and lower and upper bounds r_{min} and r_{max} for base point order, find an elliptic curve E over $GF(2^m)$ and a prime n in the interval $r_{min} \leq n \leq r_{\max}$, which is the order of a point on E. Since factor large numbers are difficult in general, a trial division bound l_{max} is chosen and the search is restricted to nearly prime curve orders.

There are four approaches to selecting such a curve:

1. Select curve coefficients with particular desired properties, compute the curve order by using formulae, and repeat the process until an appropriate order is found.

2. If m is divisible by a small integer d, then select a curve defined over $GF(2^d)$ and compute its order over $GF(2^m)$ by using formulae. Repeat if possible until an appropriate order is found.

3. Select a curve at random, compute its order directly, and repeat the process

until an appropriate order is found.

4. Search for an appropriate order, and construct a curve of that order.

These material are highly technical and the mathematics involved is far from being trivial, especially if one demands a program that can produce reliable results in a reasonable amount of time. See [SSB99] and the standards [ECC standard] for more details.

For the first and second approach, attention must be paid to a new kind of attack, [GHS2000], since m might be composite here. The fourth approach is implemented using the complex multiplication (or CM) method. The third approach: Selecting a curve at random, is considered to be the best. Since, to guard against possible future attacks against special classes of non-supersingular curves, it is prudent to select an elliptic curve at random.

We **briefly** outline how "Selecting a curve at random" can be accomplished. The main difficulty is point-counting, which is omitted. To explain them need another thesis. Despite this, even an outline can be quite cumbersome.

We explain the input parameters first:

1. r_{min} shall be selected so that $r_{min} > 2^{160}$. The security level of the resulting elliptic curve discrete logarithm problem can be increased by selecting a larger r_{min} (e.g. $r_{min} > 2^{200}$).

2. The order u of an elliptic curve E over $GF(2^m)$ satisfies $2^m + 1 - 2\sqrt{2^m} \leq u \leq 2^m + 1 + 2\sqrt{2^m}$, and u is even. Hence for a given 2^m, r_{min} should be $\leq (2^m + 1 - 2\sqrt{2^m})/2$.

3. l_{max} is typically a small integer (e.g. $l_{max} = 255$).

Selecting an appropriate curve and point at random	
Input:	A field size $q = 2^m$, lower bound r_{min}, and trial division bound l_{max}.
Output:	Field elements $a, b \in GF(2^m)$ which define an elliptic curve over $GF(2^m)$, a point P of prime order $n \geq r_{min}$, $n \geq 4\sqrt{2^m}$on the curve, and the cofactor $h = \#E(GF(2^m))/n$.
1.	Generate an elliptic curve verifiably at random, see note 1 below
2.	Compute the order u of the curve defined by a and b, see note 2 below
3.	Verify that $b \neq 0$, the curve equation is $E : y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$
4.	Test u for near primality, if the result is not nearly prime, then go to step 1. Otherwise, $u = h \cdot n$, where h is l_{max}-smooth, and $n \geq r_{min}$, $n \geq 4\sqrt{2^m}$ is probably prime, see note 3 below
5.	Check the MOV condition, with inputs $B \geq 20$, $q = 2^m$, and n, if the result is false, then go to step 1
6.	Check the Anomalous condition, if the result is false, then go to step 1
7.	Find a point P on E of order n, see note 4 below
8.	Output the curve E, the point P, the order n, and the cofactor h

Note:

1. select parameters $(SEED, a, b)$ arbitrarily, see [ECC standard], to understand what "verifiably at random" means.

2. the order $\#E(GF(2^m))$ can be computed by using Schoof's algorithm, [Sch87] and its modifications. Although the basic algorithm is quite *inefficient*, several dramatic improvements and extensions of this method have been discovered in recent years. In 1998, it was feasible to compute orders of elliptic curves over $GF(2^m)$ where m was as large as 1300. Cryptographically suitable elliptic curves over fields as large as $GF(2^{196})$ could be randomly generated in about 5 hours on a workstation, [Ler97]. In 2000, M. Fouquet, P. Gaudry and R. Harley could count the points on a curve of cryptographic size in seconds. Their program could also be combined with an early-abort strategy to examine lots of random curves until a secure one is found. E.g., generating a secure 163-bit curve took about 20 seconds on an Alpha 750 server, [FGH2000].

3. Given a trial division bound l_{max}, a positive integer h is said to be l_{max}-smooth if every prime divisor of h is at most l_{max}. Given a positive integer r_{min}, the positive integer u is said to be nearly prime if $u = h \cdot n$ for some probable prime value of n such that $n \geq r_{min}$ and some l_{max}-smooth integer h.

4. If the order $u = \#E(GF(2^m))$ of an elliptic curve E is nearly prime, the following algorithm efficiently produces a random point on E whose order is the large prime factor n of $u = hn$.

1. Generate a random point R (not ϑ) on E
2. Set $P = hR$
3. If $P = \vartheta$, then go to step 1,
   otherwise output P.

Chapter 5

The performance of 17x bit Elliptic Curve Scalar Multiplication

5.1 Choosing finite fields

There are specific values of m for which an optimal normal basis exists in $GF(2^m)$. From chapter 1, section 4; chapter 2, section 4 and 5, we have the following:

m for which a Type I ONB exists in $GF(2^m)$
100, 106,
130, 138, 148,
162, 172, 178, 180,
196, 210,
226,
268,
292, 316,
346, 348,
372, 378,
388,
418, 420, 442
460, 466

m for which a Type II ONB exists in $GF(2^m)$
105, **113**, 119,
131, 134, 135, 146, 155, 158,
173, 174, **179**, 183, 186, 189, **191**,
194, 209, 221,
230, 231, **233**, **239**, 243, 245, **251**, 254,
261, 270, 273, 278, **281**,
293, 299, 303, 306, 309,
323, 326, 329, 330, 338, 350,
354, **359**, 371, 375,
386, 393, 398, 410, 411, 413, 414,
419, 426, 429, **431**, 438, 441, **443**,
453, 470, 473

Those $m < 100$ have all been deleted since the Certicom ECDL challenge has been solved up to ECC2K-108 on 4th April 2000.

See http://www.certicom.com/research/ecc_challenge.html

for the number of machines, time and computing power that was involved.

How about 160bit ECC? Would the news of ECC2K-108 influence our belief towards the security of 160bit ECC?

Answer: $\sqrt{2^{160}}/\sqrt{2^{108}} = 2^{26} \approx 67,000,000$. See section 3 of this chapter.

Those $m < 160$ are included here for reference only. Notice that I have classified the table into rows. The rationale for this arrangement is after taking into account of the size of a 32bit processor, so that we have a category of 161 to 192 bits, the next row would then be 193 to 224 bits.

It should be pointed out that a new kind of attack was discovered recently, [GHS2000] and they recommended to choose m that was prime so that this attack would not work. However, this will eliminate many many interesting field sizes, such as **all** field sizes with Type I ONB exists. Whether their attack will work practically to **some or all** curves on these field sizes is not clear. For this reason, in the above two tables, I have marked those field sizes with m prime with a **bold**-faced type.

Now for our work on testing the "The performance of Elliptic Curve Scalar Multiplication":

For type II ONB, we choose $GF(2^m)$ with $m = 173, 179$.

For type I ONB, we choose $m = 178$. Note that $178 = 2 \cdot 89$ which is nearly prime.

5.2 17x bit test vectors for onb

According to the information in the section "Finding good and secure curves," we found secure curves according to the standard, [ECC standard].

Numbers are presented in hexadecimal form. Each hexadecimal digit expands in the natural way to four bits, except possibly the most significant digit, which expands to the approximate number of bits. Once expanded, the bits are from high orders to low orders, left to right.

Example 8 *m=173 bits*

use Type II ONB in $GF(2^{173})$
The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{173})$

$a =$	1FFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
$b =$	D20	65676EBA	2BD7E13A	EE225240	52D30C43	B66CFC30

Base point P

$x =$	1F51	4AC1C477	27B9B985	57195120	7A62EE52	66A287DC
$y =$	1EF2	A6F631E0	7D953066	289BC9AA	F04EF67F	82137C2E

Order of P **(size is 173 bits)**

$n =$	1000	00000000	00000000	005A7FC9	654DD68B	04AFBC97

$h =$ 2

Example 9 *m=173 bits*

use Type II ONB in $GF(2^{173})$
The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{173})$

$a =$						0
$b =$	C69	7988DD3E	2FE085BD	2C1472AB	067B8F0C	F119704E

Base point P

$x =$	171F	E4505005	1C356898	A5D9508B	81C0B3C7	CF9F7A2E
$y =$	C5F	0165A54D	14B2590D	FC6C4EB6	4B3347AB	F50CFCF4

Order of P **(size is 171 bits)**

$n =$	7FF	FFFFFFFF	FFFFFFFF	FFE2DC3A	3FBDB80E	5A93E6B3

$h =$ 4

Example 10 *m=178 bits*

use Type I ONB in $GF(2^{178})$

The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{178})$

$a =$ 1

$b =$ 2C7E5 439E4C8B 270D7F84 475F527A D5B2AB28 5804C450

Base point P

$x =$ 1CAAC 5F88C818 29548641 B2908BB2 EE5AAE1A CE99E7AB

$y =$ D2CD A4044CA1 0EB0E98D 7C4E2934 1FF24D75 49861AD5

Order of P **(size is** 178 **bits)**

$n =$ 20000 00000000 00000000 01A9C35E A2EBCADC A3E11E47

$h =$ 2

Example 11 *m=178 bits*

use Type I ONB in $GF(2^{178})$

The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{178})$

$a =$ 0

$b =$ 21BE6 CB54480A E8FF68F3 CFBD12B5 C1DDEF59 77635C79

Base point P

$x =$ 104CB 99623987 82FDE024 19DD68ED 63E5463E 2425A7E6

$y =$ F175 A82716AE 6B52049E 317B951A 8C90C9D0 C3A3E564

Order of P **(size is** 177 **bits)**

$n =$ 10000 00000000 00000000 00AD9B22 331C2A11 F16A0F29

$h =$ 4

Example 12 *m=179 bits*

use Type II ONB in $GF(2^{179})$
The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{179})$

$a =$	7FFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
$b =$	5FB16	DAD38D11	EAFD5A08	EEBE9C56	C231CFB2	AFB1BA1F

Base point P

$x =$	30804	3E46FB55	2577734C	6245AEBB	DB575C38	06D3D1E5
$y =$	4345	999023A7	952127AD	F8D81B4E	DE0D60F9	6F0937B6

Order of P **(size is** 179 **bits)**

$n =$	40000	00000000	00000000	0225CBA7	682E2598	9A153953
$h =$	2					

Example 13 *m=179 bits*

use Type II ONB in $GF(2^{179})$
The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{179})$

$a =$						0
$b =$	AB46	EEAE9E1D	2298F20D	C3D98307	772D0DFC	9FC25A6D

Base point P

$x =$	4F8B4	E1272856	F022CAD2	E9091089	CFAEEAFC	4E8146B1
$y =$	EF71	3BE26E57	567F3F43	AB8B0E3B	1C718157	542ECDE5

Order of P **(size is** 177 **bits)**

$n =$	1FFFF	FFFFFFFF	FFFFFFFF	FED96217	1E865BCD	CA029BD7
$h =$	4					

There appeared a question about ONB test vectors in the IEEE P1363 mailing list just a few days ago, and that was my question a while ago. Today is 22 Jan 2001, nobody has reacted to this question yet. Note that the first draft of IEEE P1363 dated at least back to 1995. The answer has been given above.

```
>Date: Thu, 18 Jan 2001 12:14:38 +0100 (MET)
>From: Birgit Henhapl <birgit@cdc.Informatik.TU-Darmstadt.DE>
>To: P1363 Diskussionsgruppe <stds-p1363-discuss@majordomo.ieee.org>
>Subject: P1363: examples for EC Domain Parameters in ONB representation
>Sender: owner-stds-p1363-discuss@ieee.org
```

```
>X-Resent-To: Multiple Recipients <stds-p1363-discuss@majordomo.ieee.org>
>X-Info: [Un]Subscribe requests to majordomo@majordomo.ieee.org
>X-Moderator-Address: stds-p1363-discuss-approval@majordomo.ieee.org
>-----------------------------------------------------------------
>This is a stds-p1363-discuss broadcast. See the IEEE P1363 web page
>(http://grouper.ieee.org/groups/1363/) for more information. For list
>info, see http://grouper.ieee.org/groups/1363/WorkingGroup/maillist.html
>-----------------------------------------------------------------
>
>Hi,
>I'm implementing ECDSA over GF(2^n) in ONB-representation. I need to find
>some testvectors in order to verify my implementations.
>P1363 unfortunatly does not include any testvectors.
>The only examples I found are those of X9.62-1998, but I'm not able to
>extract the y-coordinate based on the given (compressed) basepoint. My
>finite-field-arithmetic seems to be correct (I got some testvectors).
>Can either anybody give me the y-coordinates of the X9.6* examples (J4.3,
>exc. 4 + 5, p. 109/110) or any other examples WITH y-coordinate or at
>least confirm that the given x-coordinates of the base points are not
>correct.
>
>Many thanks,
>   Birgit Henhapl
>-----------------------------------------------------------------
>Birgit Henhapl birgit@cdc.informatik.tu-darmstadt.de
>Technische Universitaet Darmstadt phone: +49 6151 16 5541
>FB Informatik fax: +49 6151 16 6036
>Institut fuer Theoretische Informatik
>Lehrstuhl Prof. J. Buchmann
>Alexanderstr. 10
>64283 Darmstadt
>Germany]
```

5.3 Testing methodology and sample runs

For each bit length,

- Two SECURE, according to the IEEE P1363 November 1999 documents, curves were generated.

- Verify the sample curve parameters matched with each other. See note 1 below.
- 1000 times of $(rn) \cdot P$, P is the base point, where rn is a random number satisfies $0 <= rn <$ order of P, were performed.
- Several runs on each of the two curves were then followed. See note 2 below.

Note:

- 1. Check that the order of P was correctly specified in the data of a sample curve.

Now $P = (x, y)$, and let *pnt_order* be the order of P as specified in the data of a sample curve. It suffices to check that $(pnt\_order - 1) \cdot P = (x, x + y)$.

Reason: If $(pnt\_order - 1) \cdot P = (x, x + y)$ then $(pnt\_order - 1) \cdot P = -P$ by the inverse formula. Therefore $(pnt\_order) \cdot P = (pnt\_order - 1) \cdot P + P = -P + P = \vartheta$. Since *pnt_order* is a prime number, so from simple group theory, the actual order of P is either 1 or *pnt_order*. Obviously P is not ϑ, therefore the actual order of P is indeed *pnt_order*.

- 2. But no significant variance was observed.

Here is a sample run of the "Elliptic Curve Scalar Multiplication" C program for the 178 bit curve of example 3 in the last section on a Pentium II-400Mhz PC.

```
point P
x : 1caac 5f88c818 29548641 b2908bb2 ee5aae1a ce99e7ab
y :  d2cd a4044ca1 0eb0e98d 7c4e2934 1ff24d75 49861ad5

x XOR y
  : 11861 fb8c84b9 27e46fcc cedea286 f1a8e36f 871ffd7e

(pnt_order -1) P
x : 1caac 5f88c818 29548641 b2908bb2 ee5aae1a ce99e7ab
y : 11861 fb8c84b9 27e46fcc cedea286 f1a8e36f 871ffd7e

User time: 0.0500000000 seconds
Real time: 0.0000000000 seconds
```

```
---------------------------------- random seed is 1046851076
I am running ... (random)P ... for 1000 times

User time: 59.6650000000 seconds
Real time: 60.0000000000 seconds

FYI, the last rn is
  :  c1c6 af8c5ca1 105348ab 78a8d9f0 ea4e4672 5f35caab

FYI, the last (rn)P is
x : 398e9 daf7da6b 448a3b6b befce5c5 50125580 414fbe52
y : 19361 4ba20af5 bebdb124 ad61b0b5 72bc0ad9 c7244fa9


---------------------------------- random seed is 975226737
I am running ... (random)P ... for 1000 times

User time: 59.7760000000 seconds
Real time: 59.0000000000 seconds

FYI, the last rn is
  : 168ef 95ca9911 029b1e1e b8209691 5d224c8d 2f814c26

FYI, the last (rn)P is
x : 36424 05a78ce8 a291512c 6602839c 19a6232b 50192993
y :  729d afe77a19 75bdaf94 24009e23 13756bc6 7036f514


---------------------------------- random seed is 2632629705
I am running ... (random)P ... for 1000 times

User time: 59.8160000000 seconds
Real time: 60.0000000000 seconds

FYI, the last rn is
  : 15ec6 c889e808 cfec1841 9334e3de 60e6665a 01b4ebb7

FYI, the last (rn)P is
x :  a336 692cf5ff 9aafd2f1 31d390d6 a589e724 a3d09639
```

```
y : 14641 b82b0083 dbf2f57d 260ee831 43199f65 885c030e


---------------------------------- random seed is 1527782800
I am running ... (random)P ... for 1000 times

User time: 59.6660000000 seconds
Real time: 60.0000000000 seconds

FYI, the last rn is
  :  ac88 3f79b1fd 68588745 d9b50ea8 56c9f7b8 6a548ad9

FYI, the last (rn)P is
x :  f379 56ea7a2e 7b5b00ad 3379c485 346caaaf 759bb29f
y : 23908 c6158fa3 02229a57 a46f271f 2ce1ded7 dbb8c38b


---------------------------------- random seed is < stuff deleted>
I am running ... (random)P ... for 1000 times

User time: 59.8460000000 seconds
Real time: 60.0000000000 seconds

FYI, the last rn is
 : < for ECDL problem >

FYI, the last (rn)P is
x : 35b7a 1c281adf 93562c4c f0506c1b ec8dfb28 d0031c81
y : 342fc 94043645 1283a0d2 e8a35d41 2a3a4207 03ae1d98

------ stuff deleted -----
```

Notice that $(random) \cdot P$ took 0.060 seconds per iteration.

5.4 Proposing an elliptic curve discrete log problem for an 178bit curve

Once we have the "Elliptic Curve Scalar Multiplication" C program working, it is very easy to propose an elliptic curve discrete log problem—simply look at the last few lines of the previous section. This 178 bit curve is considered secure according to the standard of IEEE P1363 November 1999 documents. But **why** do we need to propose an elliptic curve discrete log problem? Because there was a new kind of attack discovered by Nigel Smart, et al. in January 2000. The following is an excerpt, see [GHS2000] for details of the attack.

>From nigel_smart@hplb.hpl.hp.com Sat Jan 15 08:34:24 2000
>Date: Fri, 14 Jan 2000 15:38:18 +0000
>From: Nigel Smart <nigel_smart@hplb.hpl.hp.com>
>To: stds-p1363-discuss@majordomo.ieee.org, nsma@hplb.hpl.hp.com
>Subject: P1363: ECC Stuff
>
> I think it would be a good idea for the P1363 document to recommend
>that in ECC systems in char 2 that the finite field used should be
>chosen to be of the PRIME degree over F_2.
>
> It is has been suspected by the experts for some time that curves
>over fields of composite degree over F_2 could be weaker. Indeed G.
>Frey gave a talk in Waterloo in 1998 which mentioned this idea, as
>have a number of other people in other meetings over the last couple
>of years.
>
> Just before Xmass at a meeting in Cirencester (UK) on "Coding and
>Cryptography", Steven Galbraith presented a joint paper with me
>describing further details of the possible problems with such
>curves. (The proceedings are available as an LNCS volume).
>
> At this conference we also announced that Florian Hess (Uni.
>Sydney), Pierrick Gaudry (\'{E}cole Polytechnique) and myself have
>discovered the following fact....
>

Let q=2^t and fix an integer n>=4.

Consider an elliptic curve over F_{q^n}. Then for "most" such curves one can solve the dlog problem on E(F_{q^n}) in

time O(q^{2+epsilon}) this should be compared to Pollard rho which would give a time of O(q^{n/2}).

>It should be pointed out that "random" curves defined over
> i) A prime field F_p
>or
> ii) A field of prime degree over F_2, F_{2^p}
>
>ARE NOT AFFECTED IN ANY WAY by this new result. Neither are
>Koblitz type curves defined over fields of any degree over F_2.
>
>Making P1363 recommend only prime fields or those of prime degree
>over F_2 would bring P1363 into line with the recently recommended
>NIST curves and the way the ANSI standards are progressing. Hence
>such a move would make sense not only from the security perspective
>but also from the standards and interoperability perspective.
>
>Nigel
>
>Nigel P. Smart | mail: nigel_smart@hpl.hp.com

The important points have been high-lighted by me. Notice that it was a very serious and strong attack, **provided it worked!** However, there have been no further discussions on this attack and its improvements in the IEEE P1363 mailing list since then. Notice that $178 = 2 \cdot 89$ which is not prime, nearly prime though, and may as well be subject to this attack. It would be **valuable** to know the answer because the program under consideration on Scalar Multiplication over a field with Type I ONB exists is always faster than that of over a field with Type II ONB exists, for similar bit lengths, see section 5 below. The reason is due to the differences in their polynomial basis representation for the implementation of the Fast Inverse, chapter 3, section 3.2.1.

Now we propose the following elliptic curve discrete log problem:

ECDLP on a 178bit curve $(178 = 2 * 89)$

Find the integer l, $0 \leq l \leq n-1$, **such that** $Q = lP$, **where** n, Q, P **and the curve are defined as follows:**

use Type I ONB in $GF(2^{178})$

The curve is $y^2 + xy = x^3 + ax^2 + b$ **over** $GF(2^{178})$

$a =$ 1

$b =$ 2C7E5 439E4C8B 270D7F84 475F527A D5B2AB28 5804C450

Base point P

$x =$ 1CAAC 5F88C818 29548641 B2908BB2 EE5AAE1A CE99E7AB

$y =$ D2CD A4044CA1 0EB0E98D 7C4E2934 1FF24D75 49861AD5

Order of P **(size is 178 bits)**

$n =$ 20000 00000000 00000000 01A9C35E A2EBCADC A3E11E47

$h =$ 2

Point Q

$x =$ 35b7a 1c281adf 93562c4c f0506c1b ec8dfb28 d0031c81

$y =$ 342fc 94043645 1283a0d2 e8a35d41 2a3a4207 03ae1d98

Let us know the answer so that we will then stop to use $GF(2^{178})$.

5.5 Results and further explorations

Platform

Pentium II-400MHz, compiled with Microsoft VC++ Version 6.0, with standard /O2 compiler optimization. No assembly was being used.

The performance of Elliptic Curve Scalar Multiplication over GF(2^m)

bit length	onb	**Elliptic Curve Scalar Multiplication** $r \cdot P$
173	type II	0.082 seconds per iteration
178	type I	0.060 seconds per iteration
179	type II	0.088 seconds per iteration

This result was also posted on http://itec.erg.cuhk.edu.hk/wb/r/wb.html

It is the scalar multiplication that dominates the computations of elliptic curve cryptography, chapter 4, section 2. Since an usual elliptic curve cryptographic operation only requires 1 or 2 elliptic curve multiplications, ECC based on this scalar multiplication program is thus fully functional on a Pentium II-400Mhz Pc.

Yet only the addition-subtraction method was used for scalar multiplication, one

obvious way to improve the performance is to use better methods of Multiplication, [Gor98]. Other techniques that might as well benefit normal basis arithmetic such as point-halving and Frobenius expansions, [SSB99] might also be valuable.

However, our next target is to make Elliptic Curve Cryptography runs smoothly on embedded devices, such as a smart card, PDA or mobile phone. That is perhaps a long journey.

Chapter 6

On matrix RSA

In this chapter, we present our findings of a particular extension of the RSA public-key cryptosystem from using integers modulo n to a version using matrices whose entries are integers modulo n, where n is the product of two large primes. In contrast to the methods found in the existing literature, our schemes do not require matrices of a special form, such as triangular matrices, and essentially work on all plaintext message matrices, whether non-singular or not. The derivation of the theorems on matrix powers depends on the consideration of eigenvalues of matrices over $GF(p)$ and $GF(n)$, rather than on counting the number of matrices that form a group under matrix multiplication. Several numbers are encrypted and decrypted in a single run. By using the Cayley-Hamilton theorem, our method reduces the computation of a matrix exponentiation with a large exponent to the computation of an exponentiation of x modulo the characteristic polynomial of a matrix, which can be accomplished by the square and multiply method or any other methods of fast exponentiations. Thus computational saving can be achieved.

6.1 Introduction

In the RSA public-key cryptosystem, integers modulo n, where n is the product of two large primes p and q, are used. The public key consists of the integer n and the encryption exponent e. The private key consists of the decryption exponent d. It is required that $d \cdot e \equiv 1 \bmod (p-1)(q-1)$. Note that a slightly different version requires that $d \cdot e \equiv 1 \bmod$ l.c.m.$[(p-1), (q-1)]$. Let integer x, $0 \leq x < n$ denote the plaintext. The encryption process computes the cyphertext $y \equiv x^e \bmod n$, $0 \leq y < n$. Given a cyphertext y, the decryption process computes $x \equiv y^d \bmod n$.

Note that the encryption process or the decryption process computes the exponentiation of integers modulo the integer n.

In the matrix extension of RSA, we compute the exponentiations of matrices, rather than integers, in the encryption or the decryption process. The central idea is to use matrices whose entries are integers modulo n. The plaintext shall take on the form of a k-by-k matrix X, and the cyphertext shall also take on the form of a k-by-k matrix Y. The entries of either matrix are integers modulo n where n is the product of two large primes p and q. Given a plaintext matrix X, the encryption process computes the cyphertext $Y \equiv X^{\bar{e}} \bmod n$. Given a cyphertext Y, the decryption process computes the plaintext $X \equiv Y^{\bar{d}} \bmod n$.

The motivation of using matrix extensions of RSA public-key cryptography is that it has the potential of speeding up the RSA encryption and decryption computations, particularly when multiple rounds of integer encryption and decryption are present. One may use the square and multiply technique for matrices exponentiation mod n [VO85], but because of the Cayley-Hamilton Theorem, exponentiation of matrices with a large exponent can be reduced to the computation of a matrix polynomial of moderate degree, [CD90]. By using the Cayley-Hamilton theorem, our method reduces the computation of a matrix exponentiation with a large exponent to the computation of an exponentiation of x modulo the characteristic polynomial of a matrix, which can be accomplished by the square and multiply method or any other methods of fast exponentiations, section 6.

Attention must be paid to ensure that the encryption and the decryption process returns the original plaintext under all conditions of relevance. For example, the relationship between the encryption exponent $\bar{e}$ and the decryption exponent $\bar{d}$ become modified. Additional constraints on usable plaintext X also arise.

Our proposed 2 by 2 matrix RSA scheme 1 is our first step towards an extension of RSA scheme to matrices. Three numbers can be encrypted and decrypted in each single run. However, it requires the sender to check whether a plaintext message matrix is non-singular or not: if it is singular, he cannot encrypt that particular message. This is not very appropriate for cryptographic application although the proportion of singular matrices over $\mathbb{Z}_n$, where for instance $n = p \cdot q$ and p, q are large distinct primes, is very much close to zero. It is not good to restrict arbitrary plaintext matrices to be non-singular. This problem was also mentioned in [VO85] and [CD90]. In order to solve this problem, they used upper triangular matrices.

In our proposed 2 by 2 matrix RSA scheme 2 and 3, we can still use matrices without a special form over $\mathbb{Z}_n$. The condition for the sender to check is just one or two gcd computations of n with an integer depending on the entries on the plaintext

matrix. What are required are only those gcd's would not equal to one. This will always be the case for otherwise n would get factored, as we know that factoring large numbers is hard. Hence it is not a demanding assumption.

In section 3, some theorems of matrix powers over $GF(p)$ and $GF(n)$, are explained. These theorems look similar to Fermat's little theorem. But unlike the usual approach by counting the number of matrices that form a group, [VO85], the derivation of these theorems depends on the consideration of eigenvalues of matrices.

Given a square matrix A, the characteristic polynomial of A is $\det(\lambda I - A)$, where det denotes the determinant of A. For example, if $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then the characteristic polynomial of A is $\det(\lambda I - A)$, which is $\lambda^2 - (a+d)\lambda + (ad - bc)$.

The Cayley-Hamilton Theorem: Every matrix satisfies its only characteristic polynomial. For example: if A=$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then $A^2 - (a+d)A + (ad-bc)I = O$, where I here denotes 2 by 2 identity matrix, and O denotes 2 by 2 zero matrix.

Throughout this chapter, we let p, q be large primes, and $n = p \times q$.

6.2 2 by 2 matrix RSA scheme 1

Key preparation:
1. Choose two large strong primes p and q. Compute $n = p \times q$.
2. Choose a random large integer $\overline{e}$ such that $0 < \overline{e} < 2(p-1)(q-1)$ and is relatively prime to $2(p-1)(q-1)$.
3. Calculate $\overline{d}$ such that $\overline{e} \cdot \overline{d} \equiv 1 \bmod 2(p-1)(q-1)$.
Note: e stands for encrypt, d stands for decrypt. Public key is $(n, \overline{e})$. Secret key is $\overline{d}$. p and q should be discarded or kept secret.

Encryption:
1. Suppose a, b, c are non-negative integers less than n, with $a^2 + bc \neq 0 \bmod n$. Let $X = \begin{pmatrix} a & b \\ c & -a \end{pmatrix}$ be the plaintext message matrix.
2. X is encrypted to $Y \equiv X^{\overline{e}} \bmod n$, where modulo n is taken element-wisely.

Decryption:
$Y^{\overline{d}} \bmod n$.

Note that three integers are encrypted and decrypted in a single run. For matrix exponentiations mod n, we may use the standard square and multiply method. However, notice that $X^2 = (a^2 + bc)I$. So $Y \equiv X^{\overline{e}} \equiv (a^2 + bc)^{\frac{\overline{e}-1}{2}} X \bmod n$.

Moreover, $Y^{\overline{d}} \equiv (X^{\overline{e}})^{\overline{d}} \equiv ((a^2+bc)^{\frac{\overline{e}-1}{2}})^{\overline{d}} X^{\overline{d}} \bmod n$. But $X^{\overline{d}} \equiv (a^2+bc)^{\frac{\overline{d}-1}{2}} X \bmod n$, so $Y^{\overline{d}} \equiv ((a^2+bc)^{\frac{\overline{e}-1}{2}})^{\overline{d}} (a^2+bc)^{\frac{\overline{d}-1}{2}} X \bmod n$. Now $((a^2+bc)^{\frac{\overline{e}-1}{2}})^{\overline{d}} (a^2+bc)^{\frac{\overline{d}-1}{2}} \equiv (a^2 + bc)^{\frac{\overline{e}\overline{d}-1}{2}} \equiv 1 \bmod n$ if $a^2 + bc \neq 0 \bmod n$. This last fact comes from the definitions of $\overline{e}$, $\overline{d}$ and the theory of basic RSA scheme.

In this way, encryption and decryption of a matrix take just a single exponentiation of numbers modulo n.

6.3 Theorems on matrix powers

We let p, q be large primes, and $n = p \times q$. I denotes 2 by 2 identity matrix.

Lemma 14 *Let A be a 2 by 2 matrix whose entries are elements of $GF(p)$. Let $f(x) \in GF(p)[x]$. Then*

$$f(A) = \begin{cases} \frac{f(\lambda_2)-f(\lambda_1)}{\lambda_2-\lambda_1}A + \frac{\lambda_2 f(\lambda_1)-\lambda_1 f(\lambda_2)}{\lambda_2-\lambda_1}I & \text{if } \lambda_1 \neq \lambda_2 \\ f'(\lambda_1)A + (f(\lambda_1) - \lambda_1 f'(\lambda_1))I & \text{if } \lambda_1 = \lambda_2 \end{cases}$$

where λ_1 and λ_2 are eigenvalues of A in $GF(p^2)$ and $f'(x)$ is the formal derivative of $f(x)$.

Proof: Suppose $\lambda_1 \neq \lambda_2$, there exists $Q(x) \in GF(p)[x]$, $e, h \in GF(p)$ such that $f(x) = (x - \lambda_1)(x - \lambda_2)Q(x) + ex + h$. Thus $f(\lambda_1) = e\lambda_1 + h$, $f(\lambda_2) = e\lambda_2 + h$, and $e = \frac{f(\lambda_2)-f(\lambda_1)}{\lambda_2-\lambda_1}$, $h = \frac{\lambda_2 f(\lambda_1)-\lambda_1 f(\lambda_2)}{\lambda_2-\lambda_1}$. The result then follows from the Cayley-Hamilton theorem.

For the case $\lambda_1 = \lambda_2$, there exists $Q(x) \in GF(p)[x]$, $e, h \in GF(p)$ such that $f(x) = (x - \lambda_1)^2 Q(x) + ex + h$. So $f'(x) = 2(x - \lambda_1)Q(x) + (x - \lambda_1)^2 Q'(x) + e$. Thus $f(\lambda_1) = e\lambda_1 + h$, $f'(\lambda_1) = e$. Therefore $h = f(\lambda_1) - \lambda_1 f'(\lambda_1)$, and the result follows from the Cayley-Hamilton theorem. □

Theorem 15 *Let A be a 2 by 2 matrix whose entries are elements of $GF(p)$. Suppose A has no zero eigenvalue,*
(1) Then $A^{p^2-1} \equiv I \bmod p$ if A has distinct eigenvalues in $GF(p^2)$, and $A^{p(p-1)} \equiv I \bmod p$ if A has repeated eigenvalues in $GF(p^2)$.
(2) $A^{p(p^2-1)} \equiv I \bmod p$.

Proof: Let $f(x) = x^k$ in the above lemma.
Suppose $\lambda_1 \neq \lambda_2$, where λ_1, $\lambda_2 \in GF(p^2)$. $A^k = \frac{\lambda_2^k-\lambda_1^k}{\lambda_2-\lambda_1}A + \frac{\lambda_2\lambda_1^k-\lambda_1\lambda_2^k}{\lambda_2-\lambda_1}I$. Since λ_1, $\lambda_2 \in GF(p^2)$, $\lambda_1 \neq 0$, $\lambda_2 \neq 0$, then $\lambda_1^{(p^2-1)} = \lambda_2^{(p^2-1)} = 1$ in $GF(p^2)$. We have $A^{p^2-1} = \frac{1-1}{\lambda_2-\lambda_1}A + \frac{\lambda_2-\lambda_1}{\lambda_2-\lambda_1}I = I$. Thus $A^{p^2-1} \equiv I \bmod p$.
Suppose $\lambda_1 = \lambda_2$, in this case λ_1 actually equals to $\frac{a+d}{2}$, which is in $GF(p)$. Using the above lemma, we have $A^k = k \cdot \lambda_1^{k-1}A + (\lambda_1^k - \lambda_1 \cdot k \cdot \lambda_1^{k-1})I = k \cdot \lambda_1^{k-1}A + \lambda_1^k(1-k)I$. Thus $A^{p(p-1)} \equiv 0 \cdot \lambda_1^{k-1}A + (\lambda_1^{p-1})^p(1-0)I \equiv I \bmod p$.
This proves (1). (2) follows easily from (1). □

Theorem 16 *Let A be a 2 by 2 matrix whose entries are non-negative integers. A p-eigenvalue of A means an eigenvalue of $A \bmod p$, where modulo p is taken element-wisely, in $GF(p^2)$. Let p, q be two large primes, $n = p \cdot q$.*
(1) If the p-eigenvalues of A are non-zero and the q-eigenvalues of A are non-zero, then $A^{p(p^2-1)q(q^2-1)} \equiv I \bmod n$.
(2) If the p-eigenvalues of A are non-zero and distinct, and the q-eigenvalues of A

are non-zero and distinct, then $A^{(p^2-1)(q^2-1)} \equiv I \bmod n$.

Proof: Let $A_p \equiv A \bmod p$. Since A_p has no zero eigenvalue, from the above theorem, $A_p^{p(p^2-1)} \equiv I \bmod p$. Thus $A^{p(p^2-1)} \equiv A_p^{p(p^2-1)} \equiv I \bmod p$. Similarly $A^{q(q^2-1)} \equiv A_q^{q(q^2-1)} \equiv I \bmod q$. Now $A^{p(p^2-1)q(q^2-1)} \equiv I \bmod p$, and $A^{p(p^2-1)q(q^2-1)} \equiv I \bmod q$. Hence $A^{p(p^2-1)q(q^2-1)} \equiv I \bmod pq$. This proves (1). The proof of (2) is similar. □

Remark 1 *Suppose* a_1, a_2, a_3, a_4 *are non-negative integers, and* $A = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$. *Let* p, q *be two large primes,* $n = p \cdot q$.
If $gcd\{n, a_1a_4 - a_2a_3\} = 1$ *then the* p*-eigenvalues and* q*-eigenvalues of* A *are non-zero.*

Proof: The characteristic equation of $A \bmod p$ is obtained from $\lambda^2 - (a_1 + a_4)\lambda + (a_1a_4 - a_2a_3) \equiv 0$ by taking each coefficients mod p. If A has zero p-eigenvalue, then $a_1a_4 - a_2a_3 \equiv 0 \bmod p$. So $gcd\{n, a_1a_4 - a_2a_3\} \neq 1$. □

Remark 2 *Suppose* a_1, a_2, a_3, a_4 *are integers, and* $A = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$. *Let* p, q *be two large odd primes,* $n = p \cdot q$.
If $gcd\{n, (a_1 - a_4)^2 + 4a_2a_3\} = 1$ *then the* p*-eigenvalues (and* q*-eigenvalues as well) of* A *are distinct.*

Proof: The characteristic equation of $A \bmod p$ is obtained from $\lambda^2 - (a_1 + a_4)\lambda + (a_1a_4 - a_2a_3) \equiv 0$ by taking each coefficients mod p. Suppose A has repeated p-eigenvalues, $\lambda^2 - (a_1 + a_4)\lambda + (a_1a_4 - a_2a_3) \equiv (\lambda - \lambda_1)(\lambda - \lambda_1) \bmod p$. So $\lambda_1 + \lambda_1 \equiv a_1 + a_4$, $\lambda_1 \cdot \lambda_1 \equiv a_1a_4 - a_2a_3 \bmod p$. Thus $(a_1 + a_4)^2 - 4(a_1a_4 - a_2a_3) \equiv 0$ or $(a_1 - a_4)^2 + 4a_2a_3 \equiv 0 \bmod p$. Thus $gcd\{n, (a_1 - a_4)^2 + 4a_2a_3\} \neq 1$. □

6.4 2 by 2 matrix RSA scheme 2

	Key preparation:
1.	Choose two large strong primes p and q. Compute $n = p \times q$.
2.	Choose a random large integer $\overline{e}$ such that $0 < \overline{e} < p(p^2-1)q(q^2-1)$ and is relatively prime to $p(p^2-1)q(q^2-1)$.
3.	Calculate $\overline{d}$ such that $\overline{e} \cdot \overline{d} \equiv 1 \bmod p(p^2-1)q(q^2-1)$.

Note: e stands for encrypt, d stands for decrypt.
Public key is $(n, \overline{e})$. Secret key is $\overline{d}$.
p and q should be discarded or kept secret.

	Encryption:
1.	Suppose a_1, a_2, a_3, a_4 are non-negative integers less than n, with $gcd\{n,\ a_1a_4 - a_2a_3\} = 1$. Let $X = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ be the plaintext message matrix.
2.	X is encrypted to $Y \equiv X^{\overline{e}} \bmod n$, where modulo n is taken element-wisely.

	Decryption:
	$Y^{\overline{d}} \bmod n$.

From theorem 16, part (1) in the last section, it is easily seen that $Y^{\overline{d}} \equiv X^{\overline{e}\cdot\overline{d}} \equiv X \cdot I \equiv X \bmod n$ if $gcd\{n,\ a_1a_4 - a_2a_3\} = 1$. Thus this method works.

Notice that in the encryption step, the assumption on the gcd is not demanding. The probability of plaintexts not satisfying the assumption is securely small. The cost of the adversary exploiting this deficiency is high. There are also additional techniques to shore up this shortcoming. Reserve a few bits in the plaintext group $\{a_1, a_2, a_3, a_4\}$ for redundant random bits. For example, let the last few bits of a_4 be bits randomly generated by the encoder which do not carry any useful information. The encoder checks if the plaintexts $\{a_1, a_2, a_3, a_4\}$ satisfy the encryption assumption. If the assumption is not met, the encoder re-generate the redundant bits anew using random methods. The process is iterated until the encryption conditions are met. This way, the encryption-decryption process are guaranteed to recover the original plaintexts. The random and redundant bits are then discarded after decryption. Another alternative is to use our matrix RSA encryption without checking whether the assumption is satisfied. The errors that are caused in the rare

event when the assumption fails can be controlled by some higher-level error-control protocols. Or, this scheme can be used in non-critical application when some rare errors can be tolerated, such as in the transmission of audio or video signals.

6.5 2 by 2 matrix RSA scheme 3

Key preparation:

1. Choose two large strong primes p and q. Compute $n = p \times q$.
2. Choose a random large integer $\overline{e}$ such that $0 < \overline{e} < (p^2-1)(q^2-1)$ and is relatively prime to $(p^2-1)(q^2-1)$.
3. Calculate $\overline{d}$ such that $\overline{e} \cdot \overline{d} \equiv 1 \bmod (p^2-1)(q^2-1)$.

Note: e stands for encrypt, d stands for decrypt.
Public key is $(n, \overline{e})$. Secret key is $\overline{d}$.
p and q should be discarded or kept secret.

Encryption:

1. Suppose a_1, a_2, a_3, a_4 are non-negative integers less than n, with
$gcd\{n,\ a_1a_4 - a_2a_3\} = 1$ and $gcd\{n,\ (a_1-a_4)^2 + 4a_2a_3\} = 1$.
Let $X = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ be the plaintext message matrix.
2. X is encrypted to $Y \equiv X^{\overline{e}} \bmod n$, where modulo n is taken element-wisely.

Note: the assumptions on the gcd's are not demanding, for otherwise n could be factored, as we know factoring large integers is hard.

Decryption:

$Y^{\overline{d}} \bmod n$.

Recall theorem 16 part (2), $X^{(p^2-1)(q^2-1)} \equiv I \bmod n$ if X has non-zero distinct p-eigenvalues and non-zero distinct q-eigenvalues. The remarks following the theorem tell us that $gcd\{n,\ a_1a_4 - a_2a_3\} = 1$ ensures that the p-eigenvalues and q-eigenvalues of X are non-zero; $gcd\{n,\ (a_1-a_4)^2 + 4a_2a_3\} = 1$ ensures that the p-eigenvalues (and q-eigenvalues as well) of X are distinct. Hence $Y^{\overline{d}} \equiv X^{\overline{e}\cdot\overline{d}} \equiv X \cdot I \equiv X \bmod n$.

6.6 An example and conclusion

We give an example of the 2 by 2 matrix RSA scheme 3 of which n is of 309 decimal digits or 1026 bits.

	Key preparation:
1.	Choose two large strong primes p and q. Compute $n = p \times q$.

p =

589696576894595765950684885484845658456584647546475652309059659 6575967549565759575895965965568958657045739923959705657348464358 456348684568232904355043

q =

699347304785623909273578347346078473893465899346524894634745645 894654734658945764588849406903263736467405643902342376922645893 406234567893478547893464580380 27

n =

412402711692543971112606093180864899333408902305762865443878200 584909633167784895865449828731924440630989008353066692872673945 719731855649428232614364137831806886263658783141826218279111754 350906438590852377466171643091994631517651549573932214003341748 500990925748983912506799324936191992750415914160322016 1

2. Choose a random large integer $\overline{e}$ such that $0 < \overline{e} < (p^2-1)(q^2-1)$ and is relatively prime to $(p^2-1)(q^2-1)$.

$\overline{e}$ =

547896596794795669758495624745745947590598346734848468548464854 645636458462373647464579579

3. Calculate $\overline{d}$ such that $\overline{e} \cdot \overline{d} \equiv 1 \bmod (p^2-1)(q^2-1)$.

$\overline{d}$ =

99232402119093657788488775289515135799226041946725727270199404568570790644969068642922856211895221823541150145226513378196320128397325310217577675825206583732637545471963428401555255968744715214199550526081898123875997413111012622203032051646907631651459976134826814726292911624908280764774329459542546838255418437615505082724013206478464743531653445896817581502468354584360868902538230455170359572192025199103985192696701282735061210352058515226512580395121854814468020236757564466643666393093837073145085485797342732072452375676578912347501395433648178444740365830576777270977068980148177871561262555536450906903795

Encryption:

---

1. Suppose a_1, a_2, a_3, a_4 are non-negative integers less than n, with $gcd\{n,\ a_1a_4 - a_2a_3\} = 1$ and $gcd\{n,\ (a_1-a_4)^2 + 4a_2a_3\} = 1$.

   Let $X = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ be the plaintext message matrix.

   Now we let X = $\begin{pmatrix} 11454 & 12565 \\ 21566 & 22383 \end{pmatrix}$.

2. X is encrypted to $Y \equiv X^{\overline{e}} \bmod n$, where modulo n is taken element-wisely.

   The characteristic polynomial of X is $x^2 - 33837 \cdot x - 14601908$, by taking each coefficients modulo n.

   The remainder when $x^{\overline{e}}$ is divided by $x^2 - 33837 \cdot x - 14601908$, is:

   19508292887702220187136293643934317171259954182115614873483864177998443221197765240714027731325654830308510663222278016130569737120764820367009868340230247362373843790047528162866033488920659807189777920840106077052946825800588686639042463820265787505676028360091252750549270855109232607187081515594149604306 4 · x +

   25139875065492153932735101332870538986790678578503153085320049135062914548305648648416756665365023519291716222337612263216339872389737364475147923076487748192786494322471322685350306285437391154047130542752661363429849182866016787094562393136819506051823153183180888049133116023668172078666401430453919782236 4

   Computation is carried out by the square and multiply method, and by taking modulo n.

Using the Cayley-Hamilton theorem,

$Y = X^{\overline{e}} \bmod n = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$, where b_1, b_2, b_3, b_4 is given by:

b_1 =

33337415786398628581861213563936959757362611268435416476291267025192319841652135886173184312753516003128127078775810608018317902984875939604015587448842264093083489023571837738447252155189749494679150607146556209920519475895270914591351425725679490471673620832681898002519651702314166582776327629228628245122

b_2 =

30768575099514619145728458646685583036413657967829952027938788938644082735356636734925228724157875829668819160387698450482805685849629786927009267826986973799022157023043154538495555526934369902595029920253576712201970460301738216409440022491798148807954955474864599259560386059411920292763521662634984368233 7

b_3 =

23838218621975623811833071287200305381750649396651322252862446048109713893632908893351764332950182761453542088314296457174922048088016555117419726276944594692268400581745530599625556843943268258170575552625125306920063778259504494132571065336925258232769005294391433454093401372036312078552972563080921585586 3

b_4 =

27268440438729988577064273614936369084999375530588934296413165971897936567237289212220805565899350223523846899572033726585037794525104023110904061562623097035220287337098286459778749561310788085474532761874372214181593038284080934175141287864073863165237445183860961927840674681197217927870561275175023465920 8

Decryption:

$Y^{\overline{d}} \bmod n$.

By taking each coefficients modulo n,

the characteristic polynomial of Y is $x^2+g\cdot x+h$, where

g =

21874686113380177063595731457299651024319793662128222316071207119891670224667554074695975867732021899545827692265494239931433446436363748583936816217262914535309996166647442085426434841722971121659193812683826508241173669809911186536606434274674653046585935965308638037464687215138487873338611933730434353599 2

h =

297152960979313069825387130751471177502788609433588666578359927 1

135908651028323010794993982172748966209198532251844680419599217 0

655771054754599066264302312424567567320374517389135281489641853 7

793996402764375622391924298334946526082291502926919455949248173 7

761969184976985794660204273913047879882094221477863 31

The remainder when $x^{\overline{d}}$ is divided by $x^2+g\cdot x+h$, is:

472443182276735770863385826950045152567897959456908635463332551 6

830989129881560996829745796875796944654919489372034894387848379 5

787639304249289648667044303184865681329993497350372345894901270 5

889992139399578294566316135764528464370863613645997430476799281 7

938186939249776026343317978574476357366436121775532 1 · x +

163985893642735012587515140876961178090849188266064626069403899 8

452016584092197336561617076268344279462530671473587511058578269 1

227812082897206727876947832620802795838732593183751685392719012 3

522491350041754678787784098406614685567270587148739646342843902 3

251540684842760372599806337808598364479086281844022 93

Computation is carried out by the square and multiply method, and by taking modulo n.

Using the Cayley-Hamilton theorem,

$$Y^{\overline{d}} \bmod n = \begin{pmatrix} 11454 & 12565 \\ 21566 & 22383 \end{pmatrix}.$$

Essentially all plaintext message matrices can be used as the assumptions on the gcd's are not demanding, for otherwise n could be factored, as we know factoring large integers is hard. Four numbers are encrypted and decrypted in each single run. A matrix exponentiation with a large exponent can be computed by using the Cayley-Hamilton theorem, of which one exponentiation of x modulo the characteristic polynomial of a matrix is required, which can be accomplished by the square and multiply method or any other methods of fast exponentiations. Thus computational saving can be achieved.

Bibliography

[ABV89] D. W. Ash, I. F. Blake, and S. A. Vanstone, "Low Complexity Normal Bases," in *Discrete Applied Math*, vol. 25 (Amsterdam: Elsevier Science Publishers/North-Holland, 1989), 191-210.

[AMOV91] G. Agnew, R. Mullin, I. Onyszchuk and S. Vanstone, "An implementation for a fast public-key cryptosystem", *Journal of Cryptology*, 3 (1991), 63-69.

[Ber68] E. Berlekamp, *Algebraic Coding Theory*, McGraw-Hill, 1968.

[CD90] C.C. Chuang and J.G. Dunham, "Matrix extensions of the RSA algorithm", in advances in cryptology: *Crypto '90* (Santa Barbara, CA, 1990), 140–155, Lecture Notes in Computer Science, Springer, Berlin, 1991.

[ECC standard] Elliptic Curve Cryptography Standard

a) ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

b) ANSI X9.63, *Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Protocols*, working draft, August 1999.

c) IEEE P1363, *Standard Specications for Public-Key Cryptography*, 2000.

d) ISO/IEC 14888-3, *Information Technology – Security Techniques – Digital Signatures with Appendix – Part 3: Certicate Based-Mechanisms*, 1998.

e) ISO/IEC 15946, *Information Technology – Security Techniques – Cryptographic Techniques Based on Elliptic Curves, Committee Draft (CD)*, 1999.

f) National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, February 2000.

[FGH2000] M. Fouquet, P. Gaudry and R. Harley, "On Satoh's algorithm and its implementation", *Ecole Polytechnique Research Report* Lix/RR/00/06, June 29, 2000.

[GL92] S. H. Gao and H. W. Lenstra, "Optimal normal bases," *Designs, Codes and Cryptography 2* (1992), 315-323.

[Gor98] D. M. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms* 27 (1998), 129- 146.

[GHS2000] P. Gaudry, F. Hess, and N. P. Smart. "Constructive and Destructive Facets of Weil Descent on Elliptic Curves," to appear in *Journal of Cryptology.*

[Ler97] R. Lercier, "Finding good random elliptic curves for cryptosystems defined over $GF(2^m)$," In Advances in Cryptology: *EuroCrypt 97*, pages 379-392, 1997.

[Men93] A.J. Menezes, *Elliptic Curve Public Key Cryptosystems* (Boston: Kluwer Academic Publishers, 1993).

[MOV93] A. Menezes, T. Okamoto and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field", *IEEE Transactions on Information Theory* 39 (1993), 1639-1646.

[MOVW89] R. C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R. M. Wilson, "Optimal Normal Bases in $GF(p^m)$," in *Discrete Applied Math*, vol.22 (Amsterdam: Elsevier Science Publishers/North-Holland, 1988), 149-161.

[Odl95] A. Odlyzko, "The Future of Integer Factorization", *Cryptobytes*, volume 1, number 2, summer 1995, 5-12.

[OM86] J. Omura and J. Massey, "Computational method and apparatus for finite field arithmetic", *U.S. patent number 4,587,627*, May 1986.

[OW99] P. van Oorschot and M. Wiener, "Parallel collision search with cryptanalytic applications", *Journal of Cryptology*, (1999) 12:1-28.

[Pol78] J. Pollard, "Monte Carlo methods for index computation mod p", *Mathematics of Computation*, 32 (1978), 918-924.

[Ros98] M. Rosing, *Implementing Elliptic Curve Cryptography*, Manning Publications Company, 1998.

[Sch87] R. Schoof. "Elliptic curves over finite fields and the computation of square roots mod p". *Mathematics of Computation*, 44, pages 483-494, 1987.

[SA98] T. Satoh and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", *Commentarii Mathematici Universitatis Sancti Pauli* 47 (1998), 81-92. Errata: ibid. 48(1999), 211-213.

[Sma99] N.P. Smart, "The Discrete Logarithm Problem on Elliptic Curves of Trace One", *Journal of Cryptology,* 1999, 12:193-196.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck "Fast Key Exchange with Elliptic Curve Systems", *Crypto '95* (New York: Springer-Verlag, 1995), 43-56.

[SSB99] G. Seroussi, N. Smart and I.F. Blake, *Elliptic Curves in Cryptography,* London Mathematical Society Lecture Note Series, Cambridge University Press, 1999.

[VO85] V. Varadharajan and R. Odoni, "Extension of RSA cryptosystems to matrix rings", *Cryptologia* 9 (1985), no. 2, 140–153.

The End