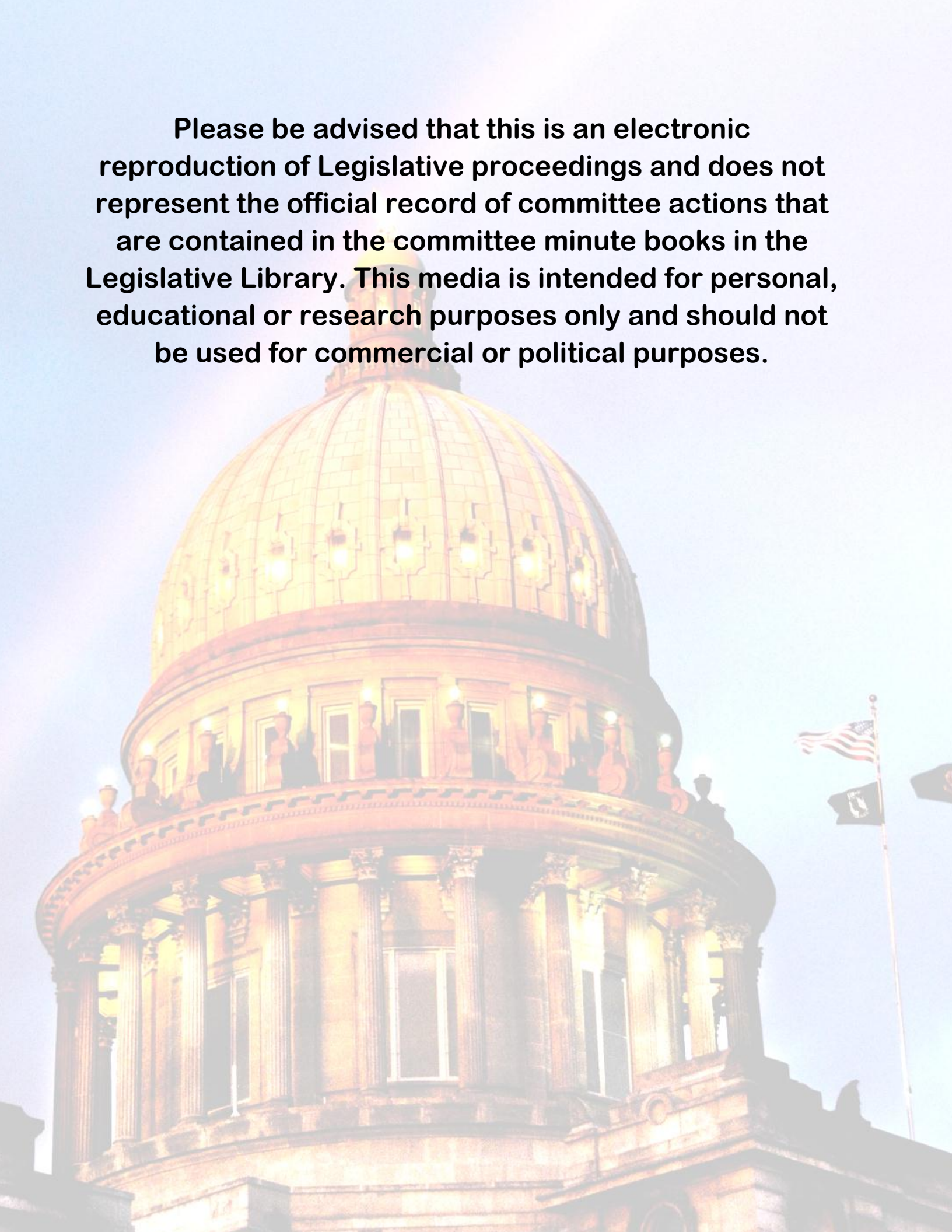

Please be advised that this is an electronic reproduction of Legislative proceedings and does not represent the official record of committee actions that are contained in the committee minute books in the Legislative Library. This media is intended for personal, educational or research purposes only and should not be used for commercial or political purposes.

MINUTES

SENATE EDUCATION COMMITTEE

DATE:	Friday, November 04, 2016
TIME:	3:00 P.M.
PLACE:	Room WW55
MEMBERS PRESENT:	Chairman Mortimer, Vice Chairman Thayn, Senators Nonini, Patrick, Souza, Den Hartog, Anthon, Buckner-Webb, Ward-Engelking
ABSENT/ EXCUSED:	None
NOTE:	The sign-in sheet, testimonies and other related materials will be retained with the minutes in the committee's office until the end of the session and will then be located on file with the minutes in the Legislative Services Library.

_____

Senator Mortimer
Chair

_____

Kelly Reister
Secretary

A Fast Software Implementation for Arithmetic Operations in GF(2^n) (PREPRINT)

Erik De Win*, Antoon Bosselaers, Servaas Vandenberghe,
Peter De Gersem*, Joos Vandewalle

Katholieke Universiteit Leuven, ESAT-COSIC
K. Mercierlaan 94, B-3001 Heverlee, Belgium
tel. +32-16-32.10.50, fax. +32-16-32.19.86
{erik.dewin,antoon.bosselaers,servaas.vandenberghe,
peter.degersem,joos.vandewalle}@esat.kuleuven.ac.be

Abstract. We present a software implementation of arithmetic operations in a finite field GF(2^n), based on an alternative representation of the field elements. An important application is in elliptic curve cryptosystems. Whereas previously reported implementations of elliptic curve cryptosystems use a standard basis or an optimal normal basis to perform field operations, we represent the field elements as polynomials with coefficients in the smaller field GF(2^{16}). Calculations in this smaller field are carried out using pre-calculated lookup tables. This results in rather simple routines matching the structure of computer memory very well. The use of an irreducible trinomial as the field polynomial, as was proposed at Crypto'95 by R. Schroeppel et al., can be extended to this representation. In our implementation, the resulting routines are slightly faster than standard basis routines.

1 Introduction

Elliptic curve public key cryptosystems are rapidly gaining popularity [M93]. The use of the group of points of an elliptic curve in cryptography was first suggested by Victor Miller [M85] and Neal Koblitz [K87]. The main advantage of using this particular group is that its discrete logarithm problem seems to be much harder than in other candidate groups (e.g., the multiplicative group of a finite field). The reason is that the various subexponential algorithms that exist for these groups up to now cannot be applied to elliptic curves. The best known algorithm for computing logarithms on a non-supersingular (see [M93]) elliptic curve is the Pohlig-Hellman attack [PH78]. Because of the difficulty of the discrete logarithm problem, the length of blocks and keys can be considerably smaller, typically about 200 bits.

Although the group of points of an elliptic curve can be defined over any field, the finite fields GF(2^n) of characteristic 2 are of particular interest for cryptosystems, because they give rise to very efficient implementations in both hardware

* N.F.W.O. research assistant, sponsored by the National Fund for Scientific Research (Belgium).

(e.g., [AMV93]) and software (e.g., [HMV92] and [SOOS95]). The group operation consists of a number of elementary arithmetic operations in the underlying field: addition/subtraction, squaring, multiplication, and inverse calculation. The speed with which these elementary operations can be executed is a crucial factor in the throughput of encryption/decryption and signature generation/verification.

To do calculations in a finite field GF(2^n), the field elements are represented in a basis. Most implementations use either a standard basis or an optimal normal basis. In a *standard basis*, field elements are represented as polynomials of the form $a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$, where all a_i are elements of GF(2), i.e., they are 0 or 1, and addition is done modulo 2. Field operations on these elements consist of operations on polynomials, e.g., a field multiplication can be calculated as a multiplication of polynomials followed by a reduction of the result modulo a fixed irreducible polynomial of degree n. In a *normal basis* an element is represented as $b_0\beta + b_1\beta^2 + b_2\beta^{2^2} + \cdots + b_{n-1}\beta^{2^{n-1}}$, where β is a fixed element of the field and all b_i are elements of GF(2). A normal base allows for a very fast squaring; multiplication is more complex than in standard basis, but this does not deteriorate efficiency if an *optimal* normal basis [MOVW88] is used. The optimal normal basis representation seems to be more appropriate for hardware, but the fastest software implementations that have been reported (e.g., [SOOS95]) use a standard basis.

The implementation presented in this paper uses a third representation of field elements that has some advantages in software. Before introducing this representation in Sect. 3, we describe elliptic curve operations in a little more detail in Sect. 2. In Sect. 4 we discuss field operations in the new representation and we compare them to standard basis in Sect. 5. We conclude the paper with some timing results.

Part of the results in this paper are based on [DD95] and [V96].

2 The Elliptic Curve Group Operation

An elliptic curve is the set of solutions (x, y) of a bivariate cubic equation over a field. In the context of public key cryptosystems, the field is often GF(2^n) and the equation is of the form $y^2 + xy = x^3 + ax^2 + b$, where a and b are elements of the field and $b \neq 0$. An "addition"-operation can be defined on the set of solutions if the *point at infinity* O is added to this set. Let $P = (x_1, y_1)$ be an element with $P \neq O$, then the inverse of P is $-P = (x_1, x_1 + y_1)$. Let $Q = (x_2, y_2)$ be a second element with $Q \neq O$ and $Q \neq -P$ then the sum $P + Q = (x_3, y_3)$ can be calculated as (see e.g., [SOOS95]):

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

These formulas are valid only if $P \neq Q$; for $P = Q$ they are a little different:

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

The point at infinity O serves as the identity element. A multiple of P, i.e., P multiplied by a natural number k, can be calculated by repeated doubling and adding. The inverse operation of this, i.e., deriving k when P and kP are given, is the elliptic curve discrete log problem, which is considered to be a very hard operation, since its running time is approximately $\mathcal{O}(2^{n/2})$.

The equations show that an elliptic curve addition can be calculated with a number of additions, multiplications, squarings, and inversions in the underlying field GF(2^n). We will see that the addition and squaring of elements of GF(2^n) are simple operations and that they require negligible time relative to the multiplication and inversion. Thus, doubling a point or adding two points on an elliptic curve takes approximately two field multiplications and one field inversion.

In an actual implementation of an elliptic curve cryptosystem, other operations are needed as well. E.g., a quadratic equation has to be solved when *point compression* [MV96] is applied. Some cryptographic algorithms require the order of the group to be known, which can be calculated by Schoof's algorithm [S85] or one of its improved versions [LM95]. However, we will concentrate in this paper on basic arithmetic operations in GF(2^n): addition, squaring, multiplication, and inversion.

3 An Alternative Representation for the Field Elements

It is well known that a field can be considered as a vector space over one of its subfields. The proper subfields of GF(2^n) are the fields GF(2^r), with $r|n$ and $0 < r < n$. Most implementations take $r = 1$, i.e., they choose a basis $\{\gamma_0, \gamma_1, \ \ldots, \gamma_{n-1}\} \subset \mathrm{GF}(2^n)$ and the field elements are represented as $a_0\gamma_0 + a_1\gamma_1 + \cdots + a_{n-1}\gamma_{n-1}$, where all $a_i \in \mathrm{GF}(2)$. The software implementations in this kind of bases are characterized by a large number of bitwise operations, e.g., testing a single bit and shifting a word over a number of bits. Although these operations are available, standard microprocessors are more suited for word operations.

Generally, r can be chosen to be any divisor of n. In [HMV92] a polynomial basis over GF(2^8) is suggested. We examined the slightly more general case where r is a multiple of 8. This limits the possible values of n to multiples of r, but if r is not too large, this causes no practical limitation.

In principle there are no restrictions on the kind of basis that is used (polynomial, normal ...). Although more work has to be done on this, we believe that a polynomial basis is most suited because a number of the advantages of (optimal) normal bases disappear when $r > 1$.

If we define $m = n/r$, then an element of $\mathrm{GF}(2^n)$ can be represented as a polynomial $\alpha_0 + \alpha_1 x + \cdots + \alpha_{m-1} x^{m-1}$, where the α_i are elements of $\mathrm{GF}(2^r)$. An important benefit of this basis is that each coefficient is represented by r bits and fits nicely in a computer word if $r = 8$, 16, or 32 (or even 64 on 64-bit processors). Arithmetic with these polynomials is identical to that with ordinary polynomials, except that operations on the coefficients are carried out in $\mathrm{GF}(2^r)$.

To calculate in $\mathrm{GF}(2^r)$, a basis has to be chosen too, but this can be simplified by the use of lookup tables. After choosing a particular basis for $\mathrm{GF}(2^r)$, we look for a generator γ and calculate all pairs (α, i) such that $\alpha = \gamma^i$ ($\alpha \in \mathrm{GF}(2^r) \setminus \{0\}$; $0 \le i < 2^r - 1$). These pairs are stored in two tables: a `log`-table sorted on α and an `alog`-table sorted on i. Each of them takes about 2^r words of r bits, resulting in a total memory requirement of about 512 bytes for $r = 8$ and 256 Kbytes for $r = 16$. The option $r = 32$ (and a fortiori $r = 64$) is excluded because of excessive memory needs. These tables can be used to efficiently calculate in $\mathrm{GF}(2^r)$, e.g., the product of two elements $\alpha, \beta \in \mathrm{GF}(2^r) \setminus \{0\}$ is

$$\alpha\beta = \mathtt{alog}[(\mathtt{log}[\alpha] + \mathtt{log}[\beta]) \bmod (2^r - 1)] \ ,$$

and also an inversion operation can be calculated with only two table lookups:

$$\alpha^{-1} = \mathtt{alog}[-\mathtt{log}[\alpha] \bmod (2^r - 1)] \ .$$

If we want polynomials to represent finite field elements, all operations have to be done modulo a fixed irreducible polynomial of degree m. In principle this polynomial can also have coefficients in $\mathrm{GF}(2^r)$, but if we can find an irreducible polynomial with coefficients in $\mathrm{GF}(2) \subset \mathrm{GF}(2^r)$, many table lookups can be saved in the reduction operation. The search for such an irreducible polynomial can be simplified by using the fact that an m-th degree polynomial that is irreducible over $\mathrm{GF}(2)$, is also irreducible over $\mathrm{GF}(2^r)$ if $\gcd(m, r) = 1$ [LN83, p. 107]. This limits m to odd numbers because r is a power of 2. We will show that the reduction operation can be speeded up even further if an irreducible *trinomial* is used [SOOS95], where the requirements for the middle term are a little different than in [SOOS95]. These special irreducible polynomials are easy to find, Table 1 lists all irreducible trinomials $1 + x^t + x^m$ with $7 \le m \le 15$ and $t \le \lfloor m/2 \rfloor$.[3]

4 Field Operations

In a polynomial basis, field operations are reduced to operations on polynomials. E.g., a field multiplication consists of a multiplication of the two polynomials representing the multiplicands, followed by a reduction of the result modulo the irreducible polynomial. Therefore, we will consider mainly operations on polynomials.

[3] It is easy to show that $1 + x^t + x^m$ is irreducible iff $1 + x^{m-t} + x^m$ is irreducible.

Table 1. List of all irreducible trinomials $1 + x^t + x^m$ with $7 \leq m \leq 15$ and $t \leq \lfloor m/2 \rfloor$. The corresponding field size is given for $r = 16$.

degree	field size	trinomial
7	112	$1 + x + x^7$ $1 + x^3 + x^7$
9	144	$1 + x + x^9$ $1 + x^4 + x^9$
11	176	$1 + x^2 + x^{11}$
15	240	$1 + x + x^{15}$ $1 + x^4 + x^{15}$ $1 + x^7 + x^{15}$
17	272	$1 + x^3 + x^{17}$ $1 + x^5 + x^{17}$ $1 + x^6 + x^{17}$

4.1 Representation of Polynomials in Memory

It is natural to store the coefficients of a polynomial in consecutive r-bit words of computer memory. To keep the routines as general as possible, we also used one word to store the length of the polynomial. In summary, a k-th degree polynomial A is stored in an array of length $k + 2$, where the first array-element A_0 contains the number of coefficients $k + 1$ (rather than the degree k), A_1 contains the constant coefficient, ... and $A_{k+1} = A_{A_0}$ contains the coefficient of the highest power of x. A zero polynomial is represented by $A_0 = 0$.

4.2 Addition

Addition in a finite field with characteristic 2 is easy: just add the corresponding bits modulo 2. Note that addition and subtraction modulo 2 are the same operations; they both correspond to a binary exclusive or (exor, $\oplus$) operation.

4.3 Multiplication

Multiplication of polynomials can be done using the shift-and-add method, where the addition is replaced by an exor. Below is an algorithm that computes the product of A and B and stores it in C (the notation of Sect. 4.1 is used).

```
1  if A_0 = 0 or B_0 = 0 then
2     C_0 = 0
3  else {
```

```
4     initialize C to zero
5     for i = 1 to A_0 do
6       if A_i ≠ 0 then
7         for j = 1 to B_0 do
8           if B_j ≠ 0 then
9             C_{i+j-1} = C_{i+j-1} ⊕ alog[(log[A_i] + log[B_j]) mod (2^r − 1)]
10    C_0 = A_0 + B_0 − 1
11  }
```

This is a very simple algorithm, although it might look a little complicated with the tests $A_i \neq 0$ and $B_j \neq 0$, which are necessary because the `log` of zero is undefined. No bit manipulations are needed. The complexity is linear in the product of the lengths of the multiplicands.

A number of optimizations are possible in an actual implementation. E.g., when the test in line 6 is successful, $\texttt{log}[A_i]$ can be stored in a register during the execution of the inner loop. Also, the `log`'s of the words of B can be stored in a temporary array at the beginning of the algorithm to reduce the number of table lookups in the inner loop.

4.4 Squaring

The square of a polynomial with coefficients in $\mathrm{GF}(2^r)$ can be calculated in a more efficient way than multiplying it by itself. The reason is that the square of a sum equals the sum of the squares because the cross-term vanishes modulo 2. The square of a polynomial is then given by

$$\left(\sum_{i=0}^{m-1} \alpha_i x^i\right)^2 = \sum_{i=0}^{m-1} \alpha_i^2 x^{2i} \quad .$$

This results in the following algorithm to compute $B = A^2$.

```
if A_0 = 0 then
  B_0 = 0
else {
  for i = 1 to A_0 − 1 do {
    if A_i ≠ 0 then
      B_{2i−1} = alog[2 log[A_i] mod (2^r − 1)]
    else
      B_{2i−1} = 0
    B_{2i} = 0
  }
  B_{2A_0−1} = alog[2 log[A_{A_0}] mod (2^r − 1)]
  B_0 = 2A_0 − 1
}
```

The complexity of this algorithm is linear in the length of the argument. For practical lengths it is much faster than multiplication.

4.5 Modular Reduction

In most cases, the result of a polynomial multiplication or squaring has to be reduced modulo an irreducible polynomial. In general, a reduction of a polynomial A modulo a polynomial B will cancel the highest power of A, say A_i, by adding (or subtracting) a multiple of B of the form $\alpha B x^{i-B_0}$ to A, where $\alpha = A_i B_{B_0}^{-1}$. This operation is repeated for decreasing values of i, until the degree of A is smaller than the degree of B.

A much simpler algorithm is obtained when B is a trinomial with coefficients in GF(2), because the calculation of α and αB is considerably simplified. The resulting algorithm is given below. A is the polynomial to be reduced, m is the degree, and t is the middle term of the irreducible trinomial, i.e., all B_i are zero, except for B_1, B_{t+1} and B_{m+1}, which are 1.

```
for i = A_0 downto m' do {
  A_{i-m} = A_{i-m} ⊕ A_i
  A_{i+t-m} = A_{i+t-m} ⊕ A_i
  A_i = 0
}
update A_0
```

Each time the loop is executed, r bits of A are cancelled. If the word length of the processor, denoted by w, is larger than r, then it is more efficient to eliminate w bits at the same time, but this induces some restrictions on the trinomial. E.g., if $r = 16$ and $w = 32$, A_i and A_{i-1} can be eliminated in one loop-operation if there is no overlap between A_{i-1} and A_{i+t-m}. This condition is satisfied if $m - t \geq w/r$.

4.6 Inversion

In general $B = A^{-1} \bmod M$ iff there exists an X such that $BA + XM = 1$, where A, B, X and M are polynomials in our case. B (and also X) can be computed with an extension of Euclid's algorithm for finding the greatest common divisor, a high level description of which is given below (deg() denotes the degree of a polynomial).

initialize polynomials $B = 1$, $C = 0$, $F = A$ *and* $G = M$

```
1  if deg(F) = 0 then return B/F_1
2  if deg(F) < deg(G) then exchange F,G and exchange B,C
3  j = deg(F) − deg(G), α = F_{F_0}/G_{G_0}
4  F = F + αx^j G, B = B + αx^j C
5  goto 1
```

This algorithm maintains the invariant relationships $F = BA + XM$ and $G = CA + YM$ (there is no need to store X and Y). In each iteration the degree of the longer of F and G is decreased by adding an appropriate multiple of the shorter.

The invariant relationships are preserved by performing the same operation on B and C. These operations are repeated until F or G is a constant polynomial.

In step 4 of this algorithm, the degree of F is decreased by at least one unit, but there is also a chance that the second-highest power of x is cancelled in F, etc. If all F_i can be considered as random, it can be shown that the degree of F is lowered by $q/(q-1)$ on average, where $q = 2^r$ is the size of the subfield. This number equals 2 for a standard basis, but quickly approximates 1 for larger subfields. On the other hand, for fixed n, convergence is faster for larger r, because in each step the length of A is decreased by about r bits. Therefore this algorithm is faster in a polynomial basis over $\mathrm{GF}(2^r)$ than in a standard basis.

In [SOOS95] the *almost inverse* algorithm is proposed to calculate inverses in standard basis. It finds a polynomial B and an integer k satisfying $BA + XM = x^k$. The inverse can be found by dividing x^k into B modulo M. The algorithm can be generalized to polynomials over larger subfields. A high level description is given below.

initialize integer $k = 0$, *and polynomials* $B = 1$, $C = 0$, $F = A$, $G = M$

0 `while` F *contains factor* x `do` $F = F/x,\ C = Cx,\ k = k + 1$
1 `if` $\deg(F) = 0$ `then return` $B/F_1,\ k$
2 `if` $\deg(F) < \deg(G)$ `then` *exchange* F,G *and exchange* B,C
3 $\alpha = F_0/G_0$
4 $F = F + \alpha G,\ B = B + \alpha C$
5 `goto 0`

The algorithm maintains the invariant relationships $x^k F = BA + XM$ and $x^k G = CA + YM$ (again, there is no need to store X and Y). Line 0 removes any factor x from F while preserving the invariant relationships. Note that after line 0 neither F nor G have a factor x. Line 2 makes sure that $\deg(F) \geq \deg(G)$. Line 4 is crucial: it adds a multiple of G to F (and the same multiple of C to B and implicitly the same multiple of Y to X to preserve the invariant relationships), such that F has a factor x again, which will be extracted in the next loop.

When analyzing the almost inverse algorithm, we observe that its behaviour is very similar to the Euclidean algorithm given above. The main difference is that it cancels powers of x from lower degree to higher degree, whereas the Euclidean algorithm moves from higher degree to lower degree. In standard basis, the former has two important benefits. Firstly, many bitwise shift operations are saved in line 4 because there is no multiplication by x^j. Secondly, if $\deg(F) = \deg(G)$ before line 4, which happens in roughly 20 % of the cases, the addition of G will decrease the degree of F. This reduces the number of iterations and hence the execution time.

These two advantages of the almost inverse algorithm are irrelevant for polynomials over larger subfields: there are no bitwise shift operations and, if $\deg(F) = \deg(G)$, the probability that the degree of F is decreased in line 4 is very small (approximately $1/q$). The somewhat surprising conclusion is that the Euclidean algorithm and the almost inverse algorithm have a comparable speed

for polynomials over GF(2^r), the former is even slightly more efficient because there is no division by x^k.

5 Comparison with Standard Basis

An important advantage of working with coefficients in GF(2^r) is that no bit operations are needed. A disadvantage is that the word size is limited because of the memory requirements for the lookup tables. This limitation can be bypassed in the modular reduction and in the final division step after the almost inverse algorithm, but for the multiplication and the almost inverse algorithm, which represent the majority of the total execution time of an elliptic curve addition, we see no obvious way to handle w bits in one step. Therefore, an increase in processor word size is likely to result in a larger speed gain for standard basis implementations than for implementations based on polynomials over GF(2^r).

We compare our multiplication algorithm in a little more detail to the basic multiplication algorithm in standard basis given below. We use << and >> to denote a bitwise shift-operation to the left and to the right respectively, and we let the line numbers start from 21 to avoid confusion with the multiplication algorithm of Sect. 4.3.

```
21 if A_0 = 0 or B_0 = 0 then
22   C_0 = 0
23 else {
24   for i = 1 to A_0 do
25     for j = 0 to w - 1 do
26       if j-th bit of A_i is 1 then {
27         lower = B_1 << j
28         higher = B_1 >> (w - j)
29         C_i = C_i ⊕ lower
30         for k = 2 to B_0 do {
31           C_{k+i-1} = C_{k+i-1} ⊕ higher
32           lower = B_k << j
33           higher = B_k >> (w - j)
34           C_{k+i-1} = C_{k+i-1} ⊕ lower
35         }
36         C_{B_0+i} = C_{B_0+i} ⊕ lower
37       }
38   update C_0
39 }
```

The loop formed by lines 24 and 25 is iterated n times. The test on line 26 is successful in 50 % of the cases on average, such that the loop on lines 30 to 35 is executed $n/2$ times. This loop runs over n/w values, such that lines 31 to 34 are executed $n^2/(2w)$ times. In the multiplication algorithm of Sect. 4.3, the loop on line 5 is iterated n/r times and the test on line 6 is almost always true (i.e., with probability $(2^r - 1)/2^r$). The same reasoning can be repeated for lines 7 and 8,

such that the inner loop on line 9 is executed about $(n/r)^2$ times. So we can state approximately that the algorithm in Sect. 4.3 will be faster if executing line 9 once takes less time than executing lines 31 to 34 $r^2/(2w)$ times (for $r = 16$ and $w = 32$ this factor equals 4). Which one is faster depends heavily on the programming language, compiler, microprocessor, and cache size. Similar comparisons can be made for the inversion (the other field operations constitute only a negligible part of the execution time of an elliptic curve operation).

Note that the standard basis algorithm given above can be optimized further. One important optimization, which we used for our timings, is to precalculate a table of shifted versions of B to avoid the shift operations in the inner loop.

6 Timings

We timed our routines for $r = 8$ and $r = 16$. For $r = 8$ the lookup tables take only 512 bytes and will fit in the first-level cache memory of any present day microprocessor; for $r = 16$ this is not the case, but the number of words is smaller for fixed n. The latter was considerably faster in our tests, therefore we will only give timings for this case.

The routines were written in ANSI-C. We used the WATCOM C 10.6 compiler and executed the tests in protected mode on a Pentium/133 based PC. Table 2 gives detailed timing results for GF(2^{177}) in standard basis and GF(2^{176}) in a polynomial basis over GF(2^{16}). We used an irreducible trinomial for the modular reduction and inversion. For standard basis, the word size w equals 32 bits. The listed figures are for the fastest routines, e.g., the almost inverse algorithm for standard basis and the extended Euclidean algorithm for polynomials over GF(2^{16}). All routines have a comparable optimization level, although we put a little less effort in the reduction, squaring and addition routines, since they have a minor impact on the overall elliptic curve operations.

Table 2 also contains timing estimates for some elliptic curve operations. These estimates were calculated by adding the times needed for the various suboperations. For the exponentiation (i.e., the repeated elliptic curve group operation), a simple double-and-add/subtract algorithm was assumed. With this algorithm, 176 doublings and on average 59 additions/subtractions are needed for one exponentiation.

The figures in table 2 show a small but significant advantage for the representation of the field elements as polynomials over GF(2^{16}). However, the proportions might change and even be reversed depending on the implementation, computer platform, field size, optimization level, etc. In addition to the possible speed gain, the routines for this alternative representation tend to be more readable and less error prone than for standard basis.

7 Conclusion

We have presented a software implementation of basic arithmetic operations in finite fields of characteristic 2. We have shown that other representations than

Table 2. Times for basic operations on polynomials over GF(2) and over GF(2^{16}). The lengths of the polynomials are suited for field operations in GF(2^{177}) and GF(2^{176}) respectively. The tests were run on a Pentium/133 based PC using the WATCOM 10.6 ANSI-C compiler.

	standard basis	pol. over GF(2^{16})
mult. 177/176 bits × 177/176 bits	71.8 μs	62.7 μs
inversion 177/176 bits	225 μs	160 μs
mod. red. 353/351 bits to 177/176 bits	8.1 μs	1.8 μs
squaring 177/176 bits	2.7 μs	5.9 μs
addition 177/176 bits	1.1 μs	1.2 μs
EC addition (est.)	404 μs	306 μs
EC doubling (est.)	411 μs	309 μs
EC exponentiation 177 bit exponent (est.)	96 ms	72 ms

standard basis and (optimal) normal basis can be used and can have some important benefits. An interesting result is that the almost inverse algorithm offers no advantages for calculating inverses of polynomials over a subfield larger than GF(2).

Acknowledgment

We would like to thank R. Schroeppel for helpful comments on his Crypto'95 paper.

References

[AMV93] G.B. Agnew, R.C. Mullin and S.A. Vanstone, "An implementation of elliptic curve cryptosystems over $F_{2^{155}}$," *IEEE Journal on Selected Areas in Communications*, Vol. 11, no. 5 (June 1993), pp. 804–813.

[BCH93] H. Brunner, A. Curiger and M. Hofstetter, "On computing multiplicative inverses in GF(2^n)," *IEEE Transactions on Computers*, Vol. 42, no. 8 (1993), pp. 1010–1015.

[DD95] E. De Win and P. De Gersem, *Studie en implementatie van arithmetische bewerkingen in* GF(2^n), Master Thesis K.U.Leuven, 1995. (in Dutch)

[HMV92] G. Harper, A. Menezes and S. Vanstone, "Public-key cryptosystems with very small key lengths," *Advances in Cryptology, Proc. Eurocrypt'92, LNCS 658*, R.A. Rueppel, Ed., Springer-Verlag, 1993, pp. 163–173.

[K87] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, Vol. 48, no. 177 (1987), pp. 203–209.

[LM95] R. Lercier and F. Morain, "Counting the number of points on elliptic curves over finite fields: strategies and performances," *Advances in Cryptology, Proc. Eurocrypt'95, LNCS 921*, L.C. Guillou and J.J. Quisquater, Eds., Springer-Verlag, 1995, pp. 79–94.

[LN83] R. Lidl and H. Niederreiter, *Finite fields,* Addison-Wesley, Reading, Mass., 1983.

[M93] A. Menezes, *Elliptic curve public key cryptosystems,* Kluwer Academic Publishers, 1993.

[M85] V.S. Miller, "Use of elliptic curves in cryptography," *Advances in Cryptology, Proc. Crypto'85, LNCS 218*, H.C. Williams, Ed., Springer-Verlag, 1985, pp. 417–426.

[MOVW88] R. Mullin, I. Onyszchuk, S. Vanstone and R. Wilson, "Optimal normal bases in GF(p^n)," *Discrete Applied Mathematics*, Vol. 22 (1988/89), pp. 149–161.

[MV96] A. Menezes and S. Vanstone, "Standard for RSA, Diffie-Hellman and related public key cryptography," Working draft of IEEE P1363 Standard, Elliptic Curve Systems, February 15, 1996.

[PH78] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, Vol. 24 (1978), pp. 106–110.

[S85] R. Schoof, "Elliptic curves over finite fields and the computation of square roots mod p," *Mathematics of Computation*, Vol. 44 (1985), pp. 483–494.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast key exchange with elliptic curve systems," *Advances in Cryptology, Proc. Crypto'95, LNCS 963*, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 43–56.

[V96] S. Vandenberghe, *Snelle basisbewerkingen voor publieke sleutelsystemen gebaseerd op elliptische curven over* GF(2^n), Master Thesis K.U.Leuven, 1996. (in Dutch)

This article was processed using the LaTeX macro package with LLNCS style