Fast Multiplication in Finite Fields GF(2^N)

Joseph H. Silverman

Mathematics Department, Brown University, Providence, RI 02912 USA
NTRU Cryptosystems, Inc., 3 Leicester Way, Pawtucket, RI 02860 USA
jhs@math.brown.edu, jhs@ntru.com, http://www.ntru.com

Abstract. A method is described for performing computations in a finite field GF(2^N) by embedding it in a larger ring R_p where the multiplication operation is a convolution product and the squaring operation is a rearrangement of bits. Multiplication in R_p has complexity $N+1$, which is approximately twice as efficient as optimal normal basis multiplication (ONB) or Montgomery multiplication in GF(2^N), while squaring has approximately the same efficiency as ONB. Inversion and solution of quadratic equations can also be performed at least as fast as previous methods.

Introduction

The use of finite fields in public key cryptography has blossomed in recent years. Many methods of key exchange, encryption, signing and authentication use field operations in either prime fields GF(p) or in fields GF(2^N) whose order is a power of 2. The latter fields are especially pleasant for computer implementation because their internal structure mirrors the binary structure of a computer.

For this reason there has been considerable research devoted to making the basic field operations in GF(2^N) (especially squaring, multiplication, and inversion) efficient. Innovations include:

- use of optimal normal bases [15];
- use of standard bases with coefficients in a subfield GF(2^r) [26];
- construction of special elements $\alpha \in \mathrm{GF}(2^N)$ such that powers α^e can be computed very rapidly [5,6,8];
- an analogue of Montgomery multiplication [14] for the fields GF(2^N) [13].

The discrete logarithm problem in finite fields can be used directly for cryptography, for example in Diffie-Hellman key exchange, ElGamal encryption, digital signatures, and pseudo-random number generation. (See [3,9,16,17,22] for a discussion of the difficulty of solving the discrete logarithm problem in GF(2^N).) An alternative application of finite fields to cryptography, as independently suggested by Koblitz and Miller, uses elliptic curves. In this situation the finite fields are much smaller (fields of order 2^{155} and 2^{185} are suggested in [10]), but the field operations are used much more extensively. Various methods have been suggested to efficiently implement elliptic curve cryptography over GF(2^N) in hardware [1] and in software [11,23].

Ç.K. Koç and C. Paar (Eds.): CHES'99, LNCS 1717, pp. 122–134, 1999.
© Springer-Verlag Berlin Heidelberg 1999

The standard way to work with GF(2^N) is to write its elements as polynomials in GF(2)[X] modulo some irreducible polynomial $\Phi(X)$ of degree N. Operations are performed modulo the polynomial $\Phi(X)$, that is, using division by $\Phi(X)$ with remainder. This division is time-consuming, and much work has been done to minimize its impact. Frequently one takes $\Phi(X)$ to be a trinomial, that is a polynomial $X^N + aX^M + b$ with only three terms, so as to simplify the division process. See, for example, [23] or [10, §6.3,6.4]. Montgomery multiplication replaces division by an extra multiplication [13], although this also exacts a cost.

A second way to work with GF(2^N) is via normal bases, especially optimal normal bases [15], often abbreviated ONB. Using ONB, elements of GF(2^N) are represented by exponential polynomials $a_0\beta + a_1\beta^2 + a_2\beta^4 + \cdots + a_{N-1}\beta^{2^{N-1}}$. Squaring is then simply a shift operation, so is very fast, and with an "optimal" choice of field, multiplication is computationally about the same as for a standard representation. More precisely, the computational complexity of multiplication is measured by the number of 1 bits in the multiplication transition matrix (λ_{ij}). The minimal complexity possible for a normal basis is $2N - 1$, and optimal normal bases are those for which the complexity is exactly $2N - 1$. (The ONB's described here are so-called Type I ONB's; the Type II ONB's are similar, but a little more complicated. Both types of ONB have complexity $2N - 1$.)

In this note we present a new way to represent certain finite fields GF(2^N) that allows field operations, especially multiplication, to be done more simply and rapidly than either the standard representation or the normal basis representation. We call this method GBB, which is an abbreviation for *Ghost Bit Basis*, because as we will see, the method adds one extra bit to each field element. The fields for which GBB works are the same as those for which Type I ONB works, but the methods are quite different. Most importantly, the complexity of the multiplication transition matrix for GBB is $N + 1$, so multiplication using GBB is almost twice as fast (or, for hardware implementations, half as complex) as multiplication using ONB. Further, squaring in GBB is a rearrangement of bits that is different from the squaring rearrangement (cyclic shift) used by ONB. (We refer the reader to [24] for a description of all fields having a GBB-multiplication.)

[*Important Note.* The GBB construction is originally due to Ito and Tsujii [28]. See the note "Added in Proof" at the end of this article.]

1 Cyclotomic Rings over GF(2)

We generate the field GF(2^N) in the usual way as a quotient GF(2)[X]/($\Phi(X)$), where we choose an irreducible cyclotomic polynomial of degree N,

$$\Phi(X) = X^N + X^{N-1} + X^{N-2} + \cdots + X^2 + X + 1.$$

As is well known, $\Phi(X)$ is irreducible in GF(2)[X] if and only if

- $p = N + 1$ is prime.
- 2 is a primitive root modulo p.

The second condition simply means that the powers $1, 2, 2^2, 2^3, \ldots, 2^{p-1}$ are distinct modulo p, or equivalently, that

$$2^{N/\ell} \not\equiv 1 \pmod{p} \text{ for every prime } \ell \text{ dividing } N.$$

There are many N's that satisfy these properties, including for example the values $N = 148$, 162, 180, 786, and 1018. (See Section 4 for a longer list.)

Remark 1. As noted above, the primes satisfying conditions (1) and (2) are exactly the primes for which there exists a Type I ONB. However, ONB's use these properties to find a basis for $\mathrm{GF}(2^N)$ of the form $\beta, \beta^2, \beta^4, \beta^8, \ldots, \beta^{2^N}$. For GBB, we will simply be using the fact that $\Phi(X)$ is irreducible.

We now observe that the field $\mathrm{GF}(2^N)$, when represented in the standard way as the set of polynomials modulo $\Phi(X)$, sits naturally as a subring of the ring of polynomials modulo $X^p - 1$. (Remember, $N = p - 1$.) In mathematical terms, there is an isomorphism

$$\frac{\mathrm{GF}(2)[X]}{(X^p - 1)} \cong \mathrm{GF}(2^N) \times \mathrm{GF}(2).$$

This is an isomorphism of rings, not fields, but as we will see, the distinction causes few problems.

For notational convenience, we let R_p denote the ring of polynomials modulo $X^p - 1$,

$$R_p = \frac{\mathrm{GF}(2)[X]}{(X^p - 1)}.$$

We interchangeably write polynomials as $a = a_N X^N + \cdots + a_1 X + a_0$ and as a list of coefficients $a = [a_N, a_{N-1}, \ldots, a_0]$.

Remark 2. Our method works more generally for fields $\mathrm{GF}(q^N)$ for any prime power q provided $p = N + 1$ is prime and q is a primitive root modulo p. In this setting, the ring $\mathrm{GF}(q)[X]/(X^N - 1)$ is isomorphic to $\mathrm{GF}(q^N) \times \mathrm{GF}(q)$. We leave to the reader the small adaptations necessary for $q \geq 3$. For most computer applications, $q = 2$ is the best choice, but depending on machine architecture other values could be useful, especially $q = 2^k$ for $k \geq 2$.

We now briefly discuss the complexity of operations in R_p. More generally, let R be any ring that is a $\mathrm{GF}(2)$-vector space of dimension n, so for example R could be R_p (with $n = p$), or R could be a field $\mathrm{GF}(2^N)$ (with $n = N$). Let $\mathcal{B} = \{\beta_0, \ldots, \beta_{n-1}\}$ be a basis for R as a $\mathrm{GF}(2)$-vector space. Then each product $\beta_i \beta_j$ can be written as a linear combination of basis elements,

$$\beta_i \beta_j = \sum_{k=0}^{n-1} \lambda_{ij}^{(k)} \beta_k.$$

The *complexity of multiplication relative to the basis* $\mathcal{B}$ is measured by the number of $\lambda_{ij}^{(k)}$'s that are equal to 1,

$$\mathrm{C}(\mathcal{B}) = \frac{1}{n}\#\{(i,j,k) : \lambda_{ij}^{(k)} = 1\}.$$

It is easy to see that $\mathrm{C}(\mathcal{B}) \geq 1$, and that if R is a field, then $\mathrm{C}(\mathcal{B}) \geq n$. A more interesting example is given by a normal basis for $R = \mathrm{GF}(2^N)$, in which case it is known [15] that $\mathrm{C}(\mathcal{B}) \geq 2N - 1$. A normal basis for $\mathrm{GF}(2^N)$ is called *optimal* if its complexity equals $2N - 1$. A complete description of all fields that possess an optimal normal basis is given in [7].

The complexity of the basis $\mathcal{B} = \{1, X, \ldots, X^{p-1}\}$ for the ring R_p is clearly $\mathrm{C}(\mathcal{B}) = p$, since $\lambda_{ij}^{(k)} = 1$ if and only if $i + j \equiv k \mod p$. In other words, for each pair (i, j) there is exactly one k with $\lambda_{ij}^{(k)} = 1$. So taking $p = N + 1$ as usual, we see that an optimal normal basis for $\mathrm{GF}(2^N)$ has complexity $2N - 1$, while the standard basis for R_p has complexity $N + 1$, making multiplication in R_p approximately twice as fast (or half as complicated) as in $\mathrm{GF}(2^N)$. It is thus advantageous to perform $\mathrm{GF}(2^N)$ multiplication by first moving to R_p and then doing the multiplication in R_p.

A second important property of a basis for finite field implementations, especially in hardware, is a sort of symmetry whereby the n^3 multipliers $\lambda_{ij}^{(k)}$ are determined by the n^2 multipliers $\lambda_{ij}^{(1)}$ by a simple transformation. We say that $\mathcal{B}$ is a *permutation basis* if there are permutations σ_k, τ_k such that

$$\lambda_{ij}^{(k)} = \lambda_{\sigma_k(i)\tau_k(j)}^{(1)} \quad \text{for all } 1 \leq i, j, k \leq n.$$

In practical terms, this means that the circuitry used to compute the first coordinate of a product ab can be used to compute all of the other coordinates simply by rearranging the order of the inputs.

To see why this is true, we write $a = \sum a_i \beta_i$ and $b = \sum b_j \beta_j$. Then (after a little algebra) the product ab is equal to

$$ab = \sum_{k=0}^{n-1} \left(\sum_{i,j=0}^{n-1} a_i b_j \lambda_{ij}^{(k)} \right) \beta_k.$$

If $\mathcal{B}$ is a permutation basis, we can rewrite this as

$$ab = \sum_{k=0}^{n-1} \left(\sum_{i,j=0}^{n-1} a_i b_j \lambda_{\sigma_k(i)\tau_k(j)}^{(1)} \right) \beta_k = \sum_{k=0}^{n-1} \left(\sum_{i,j=0}^{n-1} a_{\sigma_k^{-1}(i)} b_{\tau_k^{-1}(j)} \lambda_{ij}^{(1)} \right) \beta_k.$$

Thus the k^{th} coordinate of ab is computed by first using the permutations σ_k and τ_k to rearrange the bits of a and b respectively, and then feeding the rearranged bit strings into the circuit that computes the first coordinate of ab.

It turns out that both ONB and GBB are permutation bases, but the corresponding permutations are slightly different. For ONB one has the relation

$\lambda_{ij}^{(k)} = \lambda_{i-k,j-k}^{(0)}$, while for GBB (i.e., for the standard basis in R_p) the relation is $\lambda_{ij}^{(k)} = \lambda_{i,j-k}^{(0)}$, where in both formulas we are taking the subscripts modulo n. Thus the permutation for GBB is a little easier to implement than ONB because only one of the inputs needs to be shifted.

2 Overview of Operations in R_p

In this section we briefly describe some of the advantages of working in the ring R_p. In Section 3 we will discuss these in more detail.

2.1 Moving between GF(2^N) and R_p

An element of GF(2^N) is simply a list of N bits, and similarly an element of R_p is a list of $N+1$ bits,

$$[a_{N-1}, \dots, a_1, a_0] \in \mathrm{GF}(2^N) \qquad \text{and} \qquad [a_N, a_{N-1}, \dots, a_0] \in R_p.$$

We call the extra bit in R_p the "ghost bit". In order to do a computation in GF(2^N), we first move to R_p, next do all computations in R_p, and finally move the final answer back to GF(2^N). Movement between GF(2^N) and R_p is extremely fast, at most a single complement operation.

More precisely, the map from GF(2^N) to R_p is given by

$$\mathrm{GF}(2^N) \longrightarrow R_p, \quad a = [a_{N-1}, \dots, a_1, a_0] \longmapsto [0, a_{N-1}, \dots, a_1, a_0].$$

That is, we simply pad a by setting the ghost bit equal to zero. Moving in the other direction is almost as easy. If the ghost bit is zero, we drop it, while if the ghost bit is one, we first take the complement:

$$R_p \to \mathrm{GF}(2^N), \quad [a_N, a_{N-1}, \dots, a_1, a_0] \mapsto \begin{cases} [a_{N-1}, \dots, a_1, a_0] & \text{if } a_N = 0, \\ \sim [a_{N-1}, \dots, a_1, a_0] & \text{if } a_N = 1. \end{cases}$$

Here $\sim$ means take the complement, that is, flip every bit. If this isn't available as a primitive operation, one can XOR with $1111\cdots111$.

2.2 Addition in R_p

Addition in R_p is the usual addition of vectors over GF(2). That is, the coordinates are added using the rules $0+0 = 1+1 = 0$ and $0+1 = 1+0 = 1$.

2.3 Squaring in R_p

The squaring operation in R_p is very fast. One simply interleaves the top order bits and the bottom order bits. Thus if $a = [a_N, a_{N-1}, \dots, a_0] \in R_p$, then

$$a^2 = [a_{N/2}, a_N, a_{N/2-1}, a_{N-1}, \dots a_2, a_{N/2+2}, a_1, a_{N/2+1}, a_0].$$

Fast squaring can be implemented using the operation that takes a w-bit word $[b_w, b_{w-1}, \dots, b_1]$ and returns the two words $[b_w, 0, b_{w-1}, 0, \dots, 0, b_{w/2+1}, 0]$ and $[b_{w/2}, 0, \dots, b_2, 0, b_1, 0]$. This is trivial to implement in hardware, while in software it might be quickest to implement at the word level using a look-up table.

2.4 Multiplication in R_p

Multiplication in R_p is extremely fast, because the transition matrix for multiplication has complexity p (i.e., it is a p-by-p matrix with p entries equal to 1 and the rest 0). Multiplication in R_p is simply the convolution product of the coefficient vectors:

$$a(X)b(X) = \sum_{k=0}^{N} \Big(\sum_{i+j=k} a_i b_j \Big) X^k,$$

where we understand that the indices on a and b are taken modulo p. We will discuss in more detail below various ways in which to optimize the multiplication process.

Remark 3. Since multiplication $c = ab$ is simply the convolution product of the vectors a and b, it would be nice to use Fast Fourier Transforms to compute these convolutions. Unfortunately the vectors have dimension p, which is prime, so FFT does not help. On the other hand, some speed-up may be possible using the standard trick (Karatsuba multiplication) of splitting polynomials in half and replacing multiplications by additions, see for example [2, §3.1.2]).

2.5 Inversion in R_p

Inversion in R_p and in GF(2^N) are extremely fast. Not all elements of R_p are invertible, but we are really interested in computing inverses in GF(2^N). (Aside: $a \in R_p$ is invertible if and only if $a \neq \Phi$ and a has an odd number of 1 bits.) An especially efficient way to compute these inverses is the "Almost Inverse Algorithm" described in [23, §4.4]. Given a polynomial $a(X) \in \mathrm{GF}(2)[X]$, the almost inverse algorithm efficiently finds a polynomial $A(X)$ so that

$$a(X)A(X) \equiv X^k \pmod{\Phi(X)}$$

for some exponent $0 \leq k < 2N$. Then $a(X)^{-1} = X^{-k}A(X)$, where the product $X^{-k}A(X)$ is easily computed as a cyclic right shift in R_p. Compare with [23], where the final step of dividing by X^k requires more work. (Use of the almost inverse algorithm is also efficient for computing inverses using ONB's, especially of Type I, see [21, §11.1].)

We also mention the well-known alternative method of inversion via multiplication using the relation

$$a(X)^{2^N-1} \equiv 1 \pmod{X^p - 1} \quad \text{for all invertible } a(X) \in R_p.$$

Thus we can compute the inverse of an invertible $a(X)$ by repeated squaring and multiplication

$$a(X)^{-1} \equiv a(X)^{2^N-2} \pmod{X^p - 1}.$$

2.6 Quadratic Equations in R_p

For elliptic curve applications, it is important to be able to solve the equation $z^2 + z + c = 0$ in $\mathrm{GF}(2^N)$, see [21, §6.5]. Not all such equations are solvable, the necessary and sufficient condition being $\mathrm{Tr}(c) = 0$, where Tr is the trace map $\mathrm{GF}(2^N) \to \mathrm{GF}(2)$. The analogous condition for R_p says that $z^2 + z + c = 0$ has a solution (actually 4 solutions) in R_p if and only if $c_0 + c_1 + \cdots + c_N = 0$ and $c_0 = 0$. If there is a solution, then a solution may be computed using a recursion coming from the formula

$$z = z^{1/2} + c^{1/2}.$$

The recursion is very simple because the squaring and square root operations in R_p are so simple. We also note that if $c_0 + c_1 + \cdots + c_N = c_0 = 1$, then there will still be a solution to $z^2 + z + c = 0$ in $\mathrm{GF}(2^N)$. This solution may be found by first replacing c with its complement $\sim c$, next solving $z^2 + z + c = 0$ in R_p, and finally mapping the result back to $\mathrm{GF}(2^N)$ in the usual way.

Remark 4. It is possible to use an (automatically "optimal") normal basis in the ring R_p. To do this, write each element of R_p in terms of the basis

$$X, X^2, X^4, \cdots, X^{2^{p-1}},$$

where it is understood that the exponents are reduced modulo p. All of the usual comments that apply to normal bases in $\mathrm{GF}(2^N)$ apply to using a normal basis in the ring R_p. (See [15], [19], or [21, chapter 4] for information about optimal normal bases.) In particular, if a normal basis is used in R_p, then the complexity has the usual "optimal" value of $2p - 1$, and it is necessary to use a log table and an anti-log table to sort out the exponents when doing multiplications. Thus using a normal basis in R_p leads to slower multiplications than using the standard polynomial basis. On the other hand, squaring using a normal basis is simply a shift of bits, while squaring with the polynomial basis is interspersion of bits, so it is conceivable that situations or architectures might exist for which the normal basis is preferable.

Remark 5. For Diffie-Hellman key exchange, ElGamal encryption, and similar applications, there is no reason to move back and forth between $\mathrm{GF}(2^N)$ and R_p. One could do all the work in R_p and move back to $\mathrm{GF}(2^N)$ at the end. (Even in R_p, only one bit is exposed, namely evaluation at $X = 1$. Thus for the discrete logarithm problem $a(X)^k = b(X)$ in R_p, an attacker only deduces either $0^k = 0$ or $1^k = 1$ in $\mathrm{GF}(2)$, so he gains no information about the exponent k.)

A similar comment applies when working with elliptic curves, keeping in mind that not all elements of R_p have inverses. Thus when computing the reciprocal of $a(X) \in R_p$, if a has an even number of 1 bits, then a must first be replaced by its complement.

Remark 6. For certain finite fields, essentially Type II ONB fields [8] and their generalizations [5,6], it is possible to construct special elements called Gauss periods whose powers can be computed extremely rapidly. An interesting feature of these constructions is that the exponentiation process makes use of a "redundant representation" (see [6, page 345]), which is analogous to our "ghost bit". However, the bases used in [5,6,8] are normal bases and the fast exponentiation operation only applies to special elements, while the GBB construction in this paper gives a fast multiplication for arbitrary elements. Thus the two constructions are fundamentally different, as is also apparent from the fact that the fields to which they apply are different.

3 Bit-Level Description of Operations in GF(2^N) and R_p

In this section we give bit-level descriptions of the basic operations described in Section 2. It is relatively straightforward to give analogous word-level descriptions, although for full efficiency it is important to use all of the usual programming tricks.

Remark 7. The algorithms in this section take as input an element of GF(2^N) and return an element of GF(2^N) using the standard basis for GF(2)$[X]/\Phi(X)$. As noted above, in practice one could do all computations in R_p and only move the answer back to GF(2^N) as the very last step.

Remark 8. Polynomials in the following algorithms are written in the form

`a[N]X^N+a[N-1]X^(N-1)+...+a[2]X^2+a[1]X+a[0].`

Thus `a[i]` refers to the coefficient of X^i. We stress this point because when implementing these algorithms, it can be confusing (at least to the author) if the vector of coefficients is stored from high-to-low, instead of from low-to-high. We also note that $a(X) + \Phi(X)$ is the complement $\sim a(X)$ of $a(X)$. This is correct because $\Phi(X) = X^N + \cdots + X + 1 = [1, 1, \ldots, 1]$ has all of its bits set equal to 1. We also remind the reader again that $p = N + 1$.

Addition is simply addition of vectors with coordinates in GF(2), so there is nothing further to say.

The squaring operation is an interleaving permutation of the coefficients.

Bit-Level Procedure for Squaring in GF(2^N)

```
Input:  a(X)
Output: c(X)=a(X)^2 mod Phi(X)
Step 1: b(X):=a[0]+a[1]X^2+a[2]X^4+...+a[N/2]X^N
Step 2: c(X):=a[N/2+1]X+a[N/2+2]X^3+...+a[N]X^(N-1)
Step 3: c(X):=b(X)+c(X)
Step 4: if c[N]=1 then c(X):=c(X)+Phi(X)
```

The multiplication operation in R_p is what is commonly known as a convolution product. Here is how multiplication works at the bit level.

Bit-Level Procedure for Multiplication in $\mathrm{GF}(2^N)$

```
Input:  a(X), b(X)
Output: c(X)=a(X)b(X) mod Phi(X)
Step 1: c(X):=0
Step 2: for i=0 to p-1 do
Step 3:    if a[i]=1 then c(X):=c(X)+b(X)
Step 4:    cyclic shift c(X) right 1 bit
Step 5: if c[p-1]=1 then c(X):=c(X)+Phi(X)
```

The most time-consuming part of multiplication is Step 3, since this step is inside the main loop and requires p additions (i.e., each of the p coefficients of b must be added or XOR'd to the corresponding coefficient of c). For comparison purposes, the analogous routine using ONB has the equivalent of two Step 3's, so it takes approximately twice as long (or alternatively requires twice as complicated a circuit). Similarly, Montgomery multiplication over $\mathrm{GF}(2^N)$ has the equivalent of two Step 3's, so also takes twice as long (cf. [13]).

Computation of inverses is relatively straightforward. We give below a slight adaptation of Schroeppel, Orman, O'Malley, and Spatscheck's "Almost Inverse Algorithm" [23] (with an improvement suggested by Schroeppel) that works quite well. The speed of the Inversion Procedure can be significantly enhanced by a number of implementation tricks, such as expanding the operations on b, c, f, g into inline loop-unrolled code. We refer the reader to [23] for a list of practical suggestions.

Bit-Level Inversion Procedure in $\mathrm{GF}(2^N)$

```
Input:   a(X)
Output:  b(X)=a(X)^(-1) mod Phi(X)
Step 1:  Initialization:
            k:=0; b(X):=1; c(X):=0;
            f(X):=a(X); g(X):=Phi(X);
Step 2:  do while f[0]=0
Step 3:     f(X):=f(X)/X; k:=k+1;
Step 4:  do while f(X)!=1
Step 5:     if deg(f) < deg(g) then
Step 6:        exchange f and g; exchange b and c;
Step 7:     f(X):=f(X)+g(X)
Step 8:     b(X):=b(X)+c(X)
Step 9:     do while f[0]=0
Step 10:       f(X):=f(X)/X; c(X):=c(X)*X; k:=k+1;
Step 11: b(X):=b(X)/X^k  modulo  X^p-1
Step 12: if b[p-1]=1 then b(X):=b(X)+Phi(X)
```

Note that in Steps 3 and 10 of the inversion routine, $f(X)/X$ is f shifted right one bit and $c(X) * X$ is c shifted left one bit. Further, Step 11 in the Inversion Procedure is simply the cyclic shift

$$[b_{p-1}, b_{p-2}, \ldots, b_1, b_0] \longmapsto [b_{k-1}, b_{k-2}, \ldots, b_0, b_{p-1}, \ldots, b_{p-k+1}, b_{p-k}].$$

It is instructive to compare the simplicity of this step with the description [23, page 51] of how to compute $X^{-k}b(X)$ when $X^p - 1$ is replaced by a trinomial, even if the trinomial is selected to make this operation as simple as possible.

Finally, we describe how to solve a quadratic equation $z^2 + z + c = 0$ in GF(2^N). The equivalent formula $z = z^{1/2} + c^{1/2}$ shows that the solution may be obtained recursively using the relation

$$z_i = z_{2i} + c_{2i}, \quad 0 \le i \le N,$$

where it is understood that the indices are always reduced modulo p into the range $[0, N]$. In particular, putting $i = 0$ shows that a necessary condition for a solution to exist in R_p is $c_0 = 0$.

Bit-Level Quadratic Formula in GF(2^N)

```
Input:   c(X)
Output:  z(X) satisfying z(X)^2+z(X)+c(X)=0 mod Phi(X)
Step 1:  if c[0]=1 then c(X):=c(X)+Phi(X)
Step 2:  z[0]:=0; z[1]:=1; j:=1;
Step 3:  do N-1 times
Step 4:     i:=j
Step 5:     j:=2*j mod p
Step 6:     z[j]:=z[i]+c[j]
Step 7:  if z[N/2+1]=z[1]+c[1] then
Step 8:     if z[N]=1 then z(X):=z(X)+Phi(X)
Step 9:     return z(X)
Step 10: else
Step 11:    return "Error: z^2+z+c=0 not solvable"
```

4 Selection of Good Fields GF(2^N)

Our first requirement in choosing GF(2^N) is that $p = N + 1$ is prime and 2 is a primitive root modulo p, since this ensures that the cyclotomic polynomial

$$\Phi(X) = X^N + X^{N-1} + \cdots + X^2 + X + 1 = \frac{X^{N+1} - 1}{X - 1}$$

is irreducible in GF(2)[X].

Table 1 lists all primes in the intervals [100, 300], [650, 850], and [1000, 1200] for which $\Phi(X)$ is irreducible in GF(2)[X]. It is clear that there are lots of primes with this property. (Mathematical Aside: A conjecture of Emil Artin says that there are infinitely many primes p with this property. Artin's conjecture has not been proven unconditionally, but Hooley [12] has shown that Artin's conjecture would follow from the Riemann hypothesis.)

If one is merely interested in working in a field GF(2^N) having a very fast multiplication method, then any of the primes in Table 1 will work (taking $N = p-1$). For example, this is the case for the many cryptographic applications

Table 1. Some primes p with $\Phi(X)$ irreducible in GF(2)[X]

101, 107, 131, 139, 149, 163, 173, 179, 181, 197, 211, 227, 269, 293
653,659,661,677,701,709,757,773,787,797,821,827,829
1019, 1061, 1091, 1109, 1117, 1123, 1171, 1187

that use elliptic curves over finite fields. For elliptic curve cryptography, one might take N to be one of the values 162, 172, 178, 180, or 196.

On the other hand, if one wishes to use the discrete logarithm problem (DLP) in GF(2^N), for example with Diffie-Hellman key exchange or the ElGamal public key cryptosystem, then there is another very important issue to consider. The group of non-zero elements in the field GF(2^N) is a cyclic group of order $2^N - 1$, and if $2^N - 1$ factors as a product of small primes, the Pohlig-Hellman algorithm [20] gives a reasonably efficient way to solve the DLP in GF(2^N).

To investigate prime divisors of $2^N - 1$, we begin with the factorization of $X^N - 1$ as a product of cyclotomic polynomials,

$$X^N - 1 = \prod_{d|N} \Phi_d(X).$$

Here $\Phi_d(X)$ is the d^{th} cyclotomic polynomial. That is, $\Phi_d(X)$ is the polynomial whose roots are the primitive d^{th} roots of unity,

$$\Phi_d(X) = \prod_{1\le k\le d,\ \gcd(k,d)=1} \left(X - e^{2\pi ik/d}\right).$$

The polynomial $\Phi_d(X)$ has integer coefficients and is irreducible in $\mathbb{Q}[X]$. We will not need to use any special properties of the Φ_d's, but for further information on cyclotomic polynomials, see for example [25].

For cryptographic purposes, we want to choose a value for N so that $2^N - 1$ is divisible by a large prime. We always have the factorization

$$2^N - 1 = \prod_{d|N} \Phi_d(2),$$

so we look for cyclotomic polynomial values $\Phi_d(2)$ that have large prime divisors.

The problem of factoring numbers of the form $2^N - 1$ has a long history. Indeed, the Cunningham Project set itself the long-term task of factoring numbers of the form $b^N \pm 1$. Current results on the Cunningham Project are available on the web at [4]. The following two examples were devised using material from that site, but we include sufficient information here so that the interested reader can check that our examples have the stated properties.

Example 1. For our first example we take $p = 787$ and $N = 786$. Since 786 is divisible by 393, we see that $2^{786} - 1$ is divisible by $\Phi_{393}(2) = \frac{(2^{393}-1)}{(2^3-1)(2^{131}-1)}$. The Cunningham Project archive says that $\Phi_{393}(2)$ factors into primes as

$$\Phi_{393}(2) = 36093121 \cdot 51118297 \cdot 58352641 \cdot q.$$

Here $q \approx 2^{183} \approx 10^{55}$ is a prime. Hence $2^{786} - 1$ is divisible by the large prime q, so GF(2^{786}) is a suitable field for use with Diffie-Hellman and other schemes that depend on the intractability of the discrete logarithm problem.

Example 2. As a second example, consider $p = 1019$ and $N = 1018$. Since 1018 is divisible by 509, we see that $2^{1018} - 1$ is divisible by $\Phi_{509}(2) = 2^{509} - 1$. From the Cunningham Project archive, we find that $2^{509} - 1$ factors into primes as

$$2^{509} - 1 = 12619129 \cdot 19089479845124902223 \cdot 64712571564388487675905 7 \cdot q.$$

Here $q \approx 2^{242.26} \approx 10^{103.03}$ is prime. Hence $2^{1018} - 1$ is divisible by the large prime q, so GF(2^{1018}) is a suitable field for use with Diffie-Hellman and other schemes that depend on the intractability of the discrete logarithm problem.

Remark 9. There are many other p's listed in Table 1 with the property that $2^{p-1} - 1$ is divisible by a large prime. We have merely presented two examples for which GF(2^N) has approximately the same number of elements as the "First and Second Oakley Groups" described in [10]. However, we note that the discrete logarithm problem in GF(2^N) may be easier to solve than in GF(p) for $p \approx 2^N$, see for example [3,9,16,17,22].

Acknowledgments. I would like to thank John Platko for a serendipitous airborne conversation that piqued my interest in the problem of fast implementations of finite field arithmetic, and Andrew Odlyzko, Michael Rosing, Richard Schroeppel, Igor Shparlinski, and the referees for numerous helpful comments.

Added in Proof. Immediately prior to this article being sent to the publisher for printing, the author discovered a 1989 paper of Ito and Tsujii [28] containing the GBB construction, as well as the related papers [27] and [29]. Thus the GBB construction described here should be credited to Ito and Tsujii.

References

1. Agnew, G.B., Mullin, R.C., Vanstone, S.A.: An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. IEEE Journal on Selected Areas in Communications. **11(5)** (June 1993) 804-813
2. Cohen, H. A course in computational algebraic number theory. Graduate Texts in Math., vol. 138. Springer Verlag, Berlin (1993)
3. Coppersmith, D.: Fast evaluation of discrete logarithms in fields of characteristic two. IEEE Transactions on Information Theory **30** (1984) 587–594
4. Cunningham Project. `<ftp://sable.ox.ac.uk/pub/math/cunningham>`
5. Gao, S., von zur Gathen, J., Panario, D.: Gauss periods and fast exponentiation in finite fields. In: Baeza-Yates, R., Goles, E., Poblete, P.V.(eds.): Latin American Symposium on Theoretical Informatics–LATIN '95. Lecture Notes in Computer Science, Vol. 911. Springer-Verlag, New York (1995) 311–322
6. Gao, S., von zur Gathen, J., Panario, D.: Gauss periods: orders and cryptographical applications. Mathematics of Computation **67** (1998) 343–352
7. Gao, S., Lenstra, H.W., Jr.: Optimal normal bases. Design, Codes, and Cryptography. **2** (1992) 315–323

8. Gao, S., Vanstone, S.A.: On orders of optimal normal basis generators. Mathematics of Computation **64** (1995) 1227–1233
9. Gordon, D.M., McCurley, K.S.: Massively parallel computation of discrete logarithms. In: Brickell, E.F. (ed.): Advances in cryptology–CRYPTO '92. Lecture Notes in Computer Science, Vol. 740. Springer-Verlag, New York (1993) 16–20
10. Harkins, D., Carrel, D.: The Internet Key Exchange, RFC 2409 Network Working Group (November 1998), <http://anreg.cpe.ku.ac.th/rfc/rfc2409.html>
11. Harper, G., Menezes, A., Vanstone, S.: Public-key cryptosystems with vey small key lengths. In: Rueppel, R.A. (ed.): Advances in Cryptology — EUROCRYPT 92. Lecture Notes in Computer Science, Vol. 658. Springer-Verlag, New York (1992) 163–173
12. Hooley, C.: On Artin's conjecture. J. Reine Angew. Math. **225** (1997) 209-220
13. Koç, Ç.K., Acar, T.: Montgomery multiplication in GF(2^k). Design, Codes and Cryptography. **14** (1998) 57–69
14. Montgomery, P.L.: Modular multiplication without trial division. Math. Comp. **44** (1985) 519–521
15. Mullin, R., Onyszchuk, I., Vanstone, S., Wilson, R.: Optimal normal bases in GF(p^n). Discrete Applied Mathematics. **22** (1988) 149–161
16. Odlyzko, A.M.: Discrete logarithms and smooth polynomials. In: Mullen, G.L., Shiue, P. (eds.): Finite Fields: Theory, Applications and Algorithms. Amer. Math. Soc., Contemporary Math. #168 (1994) 269–278
17. Odlyzko, A.M.: Discrete logarithms: The past and the future. Preprint, July 1999. <http://www.research.att.com/~amo/doc/crypto.html>
18. Omura, J., Massey, J.: Computational method and apparatus for finite field arithmetic. United States Patent 4587627 (May 6), 1986
19. Onyszchuk, I., Mullin, R., Vanstone, S.: Computational method and apparatus for finite field multiplication. United States Patent 4745568 (May 17), 1988
20. Pohlig, S.C., Hellman, M.E.: An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. IEEE Transactions on Information Theory. **24** (1978) 106–110
21. Rosing, M.: Implementing Elliptic Curve Cryptography. Manning Publications Greenwich, CT (1999)
22. Semaev, I. A.: An algorithm for evaluation of discrete logarithms in some nonprime finite fields. Math. Comp. **67** (1998) 1679–1689
23. Schroeppel, R., O'Malley, S., Orman, H., Spatscheck, O.: Fast key exchange with elliptic curve systems In: Coppersmith, D. (ed.): Advances in Cryptology — CRYPTO 95. Lecture Notes in Computer Science, Vol. 973. Springer-Verlag, New York (1995) 43–56
24. Silverman, J.H.: Low Complexity Multiplication in Rings. Preprint, April 14, 1999
25. Washington, L.: Introduction to Cyclotomic Fields. Springer-Verlag New York (1982)
26. De Win, E., Bosselaers, A., Vandenberghe, S., De Gersem, P., Vandewalle, J.: A fast software implementation for arithmetic operations in GF(2^n). In: Advances in Cryptology — ASIACRYPT '96. Lecture Notes in Computer Science, Vol. 1163. Springer-Verlag New York (1996) 65–76
27. Drolet, G.: A new representation of elements of finite fields GF(2^m) yielding small complexity arithmetic circuits. *IEEE Trans. Comput.* **47** (1998), no. 9, 938–946
28. Ito, B., Tsujii, S.: Structure of a parallel multipliers for a class of fields GF(2^m). *Information and Computers* **83** (1989), 21–40
29. Wolf, J.K.: Efficient circuits for multiplying in GF(2^m). *Topics in Discrete Mathematics* **106/107** (1992), 497–502