

Frequency Domain Finite Field Arithmetic for Elliptic Curve Cryptography

Selçuk Baktır, Berk Sunar

{selcuk,sunar}@wpi.edu

Department of Electrical & Computer Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609

January 30, 2007

Abstract

The discrete Fourier transform (DFT) based method originally proposed for integer multiplication provides an extremely efficient method with the best asymptotic complexity, i.e. $O(m \log m \log \log m)$, for multiplication of m -bit integers or $(m - 1)^{st}$ degree polynomials. Unfortunately, this method bears significant overhead due to the conversions between the time and frequency domains. This makes the original DFT based method impractical for small operands, e.g. less than 1000 bits in length as used in many applications. In this work, we investigate the application of the number theoretic transform (NTT), which found many applications in digital signal processing, to finite field multiplication with an emphasis on elliptic curve cryptography (ECC). Furthermore, we introduce an efficient algorithm for computing Montgomery products of polynomials in the frequency domain. Our algorithm performs the entire modular multiplication (including the reduction step) in the frequency domain, and thus eliminates costly back and forth conversions improving upon the straightforward NTT approach. We show that, especially in computationally constrained platforms, multiplication of finite field elements may be achieved more efficiently in the frequency domain than in the time domain for operand sizes relevant to elliptic curve cryptography (ECC). This paper is an expanded version of the earlier paper [22] on the same topic which, for the first time, proposes the use of frequency domain arithmetic for ECC and shows that it can be efficient.

Key Words: Finite field multiplication, discrete Fourier transform, number theoretic transform, elliptic curve cryptography.

1 Introduction

Finite fields have many applications in coding theory [4, 3], cryptography [15, 6, 20], and digital signal processing [5]. Hence, efficient implementation of finite field arithmetic operations is crucial. Multiplication of field elements is commonly implemented in terms of modular multiplication of polynomials realized in two steps: ordinary polynomial multiplication and modular reduction of the result by the field generating polynomial. Unlike polynomial multiplication, modular reduction has only linear complexity, i.e. $O(m)$, for polynomials of degree $m - 1$ when a fixed special

modulus is chosen. Hence it is essential to attack the complexity of the multiplication step. The classical polynomial multiplication method for multiplication of $(m - 1)^{st}$ degree polynomials has quadratic complexity, i.e. $O(m^2)$, given in terms of coefficient multiplications and additions. The complexity may be improved to $O(m^{\log_2 3})$ using the Karatsuba algorithm [12]. However, despite the significant improvement gained by the Karatsuba algorithm, the complexity is still not optimal. Furthermore, the implementation of the Karatsuba algorithm is more burdensome due to its recursive nature and not considered practical in applications such as ECC where operand sizes are relatively small. The known fastest multiplication algorithm, introduced by Schönhage and Strassen [25], performs multiplication in the frequency domain using the Fast Fourier Transform (FFT) [9] with complexity $O(m \log m \log \log m)$ for multiplication of m -bit integers or m -coefficient polynomials [11]. Unfortunately, the FFT based algorithm becomes efficient and useful in practice only for very large operands due to the overhead associated with the forward and inverse Fourier transform operations.

When the transformation computations between the time and frequency domains are not considered, frequency domain polynomial multiplication (without modular reduction) has surprisingly low *linear*, i.e. $O(m)$, complexity. Sadly, no efficient method for performing modular reduction is known to exist in the frequency domain, and therefore one needs to convert the result of a polynomial multiplication operation back to the time domain to perform modular reduction bearing significant overhead.

This paper provides an expanded version of the work presented in [22]. In this paper we show that NTT based multiplication methods, with $O(m)$ complexity in terms of the number of coefficient multiplications, may be applied efficiently for operand sizes relevant to ECC and we provide parameters for their efficient application. Furthermore, we introduce a new algorithm, named *DFT modular multiplication* which achieves modular reduction, as well as multiplication, in the frequency domain using *DFT modular reduction*. Thus, the entire finite field multiplication, including modular reduction, can be carried out in the frequency domain which may yield significant improvement over the straightforward NTT based method. In many finite field applications a chain of arithmetic operations is performed rather than a solitary one. Using our method, after an initial conversion step from the time domain to the frequency domain all intermediary operations may be computed in the frequency domain. Therefore, there will be no need for conversion before and after every single finite field multiplication except before the very first and after the very last ones.

In the next section, we briefly give some background information on finite fields and the discrete Fourier transform in finite fields. Furthermore, we describe efficient methods for computing the DFT in finite fields and application of the DFT to finite field multiplication. In Section 3.2, we introduce a new algorithm, named *DFT modular multiplication*, for finite field modular multiplication of polynomials in the frequency domain. In Section 3.3, we present ideas for efficient implementation of DFT modular multiplication. Furthermore, in Section 3.4, we give a complexity analysis of the DFT modular multiplication algorithm, compare it with other efficient methods such as the Karatsuba algorithm, and show its relevance for elliptic curve cryptography. Finally, we present our conclusions in Section 4.

2 Background

2.1 Finite Fields and Polynomial Representation

A field with finite number of elements is called a finite field or Galois field, denoted by F_q or $GF(q)$, where q stands for the number of elements in the field [14]. The number of elements in a finite field is always a prime or a prime power, i.e., $q = p$ or $q = p^m$, where the prime number p is called

the characteristic of the finite field. When q is a prime, i.e. $q = p$, the finite field $GF(p)$ is called a prime field. The prime field $GF(p)$ is the field of residue classes modulo p and its elements are represented by the integers in $\{0, 1, 2, \dots, p - 1\}$. When q is a prime power, i.e. $q = p^m$, the finite field $GF(p^m)$ is called an extension field. The extension field $GF(p^m)$ is generated by using an m^{th} degree irreducible polynomial over $GF(p)$ and it is the field of residue classes modulo the irreducible field generating polynomial. Hence, in polynomial representation the elements of $GF(p^m)$ are represented by polynomials of degree $m - 1$ with coefficients in $GF(p)$.

2.2 Number Theoretic Transform (NTT)

The number theoretic transform over a ring, also known as *the discrete Fourier transform over a finite field*, was introduced by Pollard [21]. For a finite field $GF(q)$ and a generic sequence (a) of length d whose entries are from $GF(q)$, the forward NTT of (a) over $GF(q)$, denoted by (A) , can be computed as

$$A_j = \sum_{i=0}^{d-1} a_i r^{ij} \quad , \quad 0 \leq j \leq d - 1 . \quad (1)$$

Here we refer to the elements of (a) and (A) by a_i and A_i , respectively, for $0 \leq i \leq d - 1$. Likewise, the inverse NTT of (A) over $GF(q)$ can be computed as

$$a_i = \frac{1}{d} \cdot \sum_{j=0}^{d-1} A_j r^{-ij} \quad , \quad 0 \leq i \leq d - 1 . \quad (2)$$

We will refer to the sequences (a) and (A) as the *time and frequency domain representations*, respectively, of the same sequence. The above NTT computations over the finite field $GF(q)$ are defined by utilizing a d^{th} primitive root of unity, denoted by r , from $GF(q)$ or a finite extension of $GF(q)$.

Definition 1 *The element r is a primitive d^{th} root of unity modulo n if*

$$r^d = 1 \pmod{n}$$

and

$$r^{d/t} - 1 \neq 0 \pmod{n}$$

for any prime divisor t of d .

Note that unlike the complex number $r = e^{j2\pi/d}$ generally used as the d -th primitive root of unity in the discrete Fourier transform (DFT) computations, a finite field element r can be utilized for the same purpose in an NTT.

We would like to caution the reader that in a number theoretic transform the modulus q and the transform length d can not be chosen independently of each other. If the modulus is a composite number of the form $q = p_1^{e_1} p_2^{e_2} p_3^{e_3} \dots p_k^{e_k}$, then the length of the transform, d , must divide $\text{gcd}(p_1 - 1, p_2 - 1, p_3 - 1, \dots, p_k - 1)$. In the case of the NTT over a finite field, i.e. when q is a prime p or a prime power p^m , d must divide $p - 1$.

2.3 Multiplication in $GF(q^m)$ Using the NTT

Cyclic convolution of any two d -element sequences (a) and (b) in the time domain results in another d -element sequence (c) and can be computed as follows:

$$c_i = \sum_{j=0}^{d-1} a_j b_{i-j \bmod d}, \quad 0 \leq i \leq d-1. \quad (3)$$

The above convolution operation in the time domain is equivalent to the following computation in the frequency domain:

$$C_i = A_i \cdot B_i, \quad 0 \leq i \leq d-1. \quad (4)$$

Thus, convolution of two d -element sequences in the time domain, with complexity $O(d^2)$, has a surprisingly low $O(d)$ complexity in the frequency domain.

Multiplication of two polynomials is basically the same as the *acyclic (linear) convolution* of the polynomial coefficients. We have seen that cyclic convolution can be performed very efficiently in the frequency domain by pairwise coefficient multiplications, hence it will be wise to represent an element of $GF(q^m)$, in polynomial representation an $(m-1)^{st}$ degree polynomial with coefficients in $GF(q)$, with at least a $d = (2m-1)$ element sequence by appending zeros at the end, so that the cyclic convolution of two such sequences will be equivalent to their acyclic convolution and give us their polynomial multiplication. We can form sequences by taking the ordered coefficients of polynomials. For instance,

$$a(x) = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1},$$

an element of $GF(q^m)$ in polynomial representation, can be interpreted as the sequence

$$(a) = (a_0, a_1, a_2, \dots, a_{m-1}, 0, 0, \dots, 0), \quad (5)$$

after appending $d-m$ zeros to the right. For $a(x), b(x) \in GF(q^m)$ the cyclic convolution of (a) and (b) yields a sequence (c) whose first $2m-1$ entries can be interpreted as the coefficients of $c(x) = a(x) \cdot b(x)$. The following straightforward algorithm (Algorithm 1) realizes the polynomial multiplication $c(x) = a(x) \cdot b(x)$.

Algorithm 1 Polynomial Multiplication by Direct Application of the NTT

Input: $a(x), b(x) \in GF(q^m)$

Output: $c(x) = a(x) \cdot b(x)$

- 1: Interpret $a(x)$ and $b(x)$ as the sequences (a) and (b) with length $d \geq 2m-1$
 - 2: Convert (a) and (b) into (A) and (B) using the NTT as in (1)
 - 3: Multiply (A) with (B) to compute (C) as in (4)
 - 4: Convert (C) to (c) using the inverse NTT as in (2)
 - 5: Interpret the first $2m-1$ coefficients of (c) as the coefficients of $c(x) = a(x) \cdot b(x)$
 - 6: Return $c(x)$
-

Note that with Algorithm 1 the polynomial product $c(x) = a(x) \cdot b(x)$ is computed in the frequency domain but the final reduction by the field generating polynomial remains to be computed. One needs to convert (C) back to the time domain to do the modular reduction, so that further multiplications can be performed on it using the same method. In Section 2.4, we will show the relationship between the NTT and the *residue number system* (RNS) [29, 27, 26] and see that

the frequency domain representation of a polynomial is equivalent to the RNS representation of the same polynomial provided certain conditions are satisfied on the modulus polynomials used in the residue computations. Furthermore, we will show that Algorithm 1 which uses the NTT for multiplication in $GF(q^m)$ is equivalent to the optimal case of multiplication in $GF(q^m)$ using the RNS in terms of the number of required $GF(q)$ multiplications. In Section 2.5, we will present parameters and methods for special NTTs for performing the conversions between the time and frequency domains efficiently and in Section 3 we will introduce *DFT modular multiplication* which performs both polynomial multiplication and modular reduction in the frequency domain and thus avoids costly conversions between the time and frequency domains.

2.4 On the Relationship Between the RNS and NTT

For a polynomial $a(x)$ and a modulus polynomial

$$P(x) = \prod_{i=0}^{d-1} p_i(x) ,$$

made up of relatively prime factors $p_i(x)$, for $0 \leq i \leq d-1$, $a(x) \bmod P(x)$ can be uniquely represented by its residues modulo $p_i(x)$, for $0 \leq i \leq d-1$, using the RNS. And hence, provided that

$$\deg(a(x)) < \deg(P(x))$$

holds for the degrees of $a(x)$ and the modulus polynomial $P(x)$, $a(x)$ can be uniquely represented in this RNS by its residues as

$$(\langle a(x) \rangle_{p_0(x)}, \langle a(x) \rangle_{p_1(x)}, \dots, \langle a(x) \rangle_{p_{d-1}(x)}) ,$$

where $\langle a(x) \rangle_{p_i(x)}$ denotes $a(x) \bmod p_i(x)$.

Using Chinese Remainder Theorem (CRT), the conversion from the RNS representation back to the normal polynomial representation can be performed as follows:

$$a(x) = \sum_{i=0}^{d-1} \langle a(x) \rangle_{p_i(x)} \cdot P_i(x) , \quad (6)$$

where

$$P_i(x) = \left(\frac{P(x)}{p_i(x)} \right) \cdot \left(\left(\frac{P(x)}{p_i(x)} \right)^{-1} \bmod p_i(x) \right) , \quad 0 \leq i \leq d-1 . \quad (7)$$

Theorem 1 *Computing the d -element NTT of the sequence (a) corresponding to $a(x) \in GF(q^m)$, as described with (5) where $d \geq 2m-1$, is equivalent to computing the RNS coefficients of $a(x)$ in the RNS with the modulus polynomials $p_i(x) = x - r^i$, for $0 \leq i \leq d-1$, where r is the d^{th} primitive root of unity used in the NTT.*

Proof of Theorem 1 For (A) , denoting the NTT of (a) corresponding to $a(x)$, with elements A_i , for $0 \leq i \leq d-1$, and for the RNS representation $(\langle a(x) \rangle_{p_0(x)}, \langle a(x) \rangle_{p_1(x)}, \dots, \langle a(x) \rangle_{p_{d-1}(x)})$ of $a(x)$, we need to show that $A_i = \langle a(x) \rangle_{p_i(x)}$. Remember in (1) that the coefficients of (A) are computed as follows

$$A_i = \sum_{j=0}^{d-1} a_j r^{ji} , \quad 0 \leq i \leq d-1 . \quad (8)$$

Likewise, for $a(x)$ represented as

$$a(x) = \sum_{j=0}^{m-1} a_j x^j$$

in the standard basis, the i^{th} residue of $a(x)$, modulo $p_i(x) = x - r^i$, is

$$\langle a(x) \rangle_{p_i(x)} = a(x) \bmod x - r^i = a(r^i) = \sum_{j=0}^{m-1} a_j r^{ji} = \sum_{j=0}^{d-1} a_j r^{ji}, \quad 0 \leq i \leq d-1, \quad (9)$$

assuming $a_j = 0$, for $m \leq j \leq d-1$. Since the summations in (8) and (9) are equivalent the NTT and RNS representations of $a(x)$ are equivalent. \square

A divide-and-conquer method for multiplication is by using the RNS. If the degree of the product $c(x) = a(x) \cdot b(x)$ is less than the degree of the RNS modulus $P(x)$, then the multiplication of $a(x)$ and $b(x)$ can be achieved in the RNS representation by simply multiplying their corresponding residues. The computation of $c(x) = a(x) \cdot b(x)$ can be conducted in the RNS representation as

$$(\langle c(x) \rangle_{p_0(x)}, \langle c(x) \rangle_{p_1(x)}, \dots, \langle c(x) \rangle_{p_{k-1}(x)}) ,$$

where $\langle c(x) \rangle_{p_i(x)} = \langle \langle a(x) \rangle_{p_i(x)} \cdot \langle b(x) \rangle_{p_i(x)} \rangle_{p_i(x)}$. Thus, two polynomials could be multiplied in a completely parallel manner with minimal effort in the RNS representation. If the modulus polynomials $p_i(x)$, for $0 \leq i \leq d-1$, used in the RNS are all first degree binomials, then polynomial multiplication could be achieved in the RNS representation with the minimal number of coefficient multiplications which is only linear in the number of polynomial coefficients.

The following algorithm achieves polynomial multiplication of $a(x), b(x) \in GF(p^m)$ in the RNS representation. We would like to note that, similar to Algorithm 1, Algorithm 2 achieves only the polynomial multiplication of the finite field elements and the final modular reduction by the field generating polynomial remains to be computed.

Algorithm 2 RNS Polynomial Multiplication Using the CRT

Input: $a(x), b(x) \in GF(q^m)$

Output: $c(x) = a(x) \cdot b(x)$

- 1: Obtain $\langle a(x) \rangle_{p_i(x)}$ and $\langle b(x) \rangle_{p_i(x)}$, for $0 \leq i \leq d-1$, where $d \geq 2m-1$
 - 2: Compute $\langle c(x) \rangle_{p_i(x)} = \langle a(x) \rangle_{p_i(x)} \cdot \langle b(x) \rangle_{p_i(x)}$, for $0 \leq i \leq d-1$
 - 3: Obtain $c(x)$ from $\langle c(x) \rangle_{p_i(x)}$, for $0 \leq i \leq d-1$, using Chinese Remainder Theorem as in (6)
 - 4: Return $c(x)$
-

With Theorem 3, we prove that when the modulus polynomials used in Algorithm 2 are the first degree binomials $p_i(x) = x - r^i$, for $0 \leq i \leq d-1$, where r is the d^{th} primitive root of unity used in Algorithm 1, Algorithm 2 which performs multiplication in $GF(p^m)$ utilizing the RNS and CRT is equivalent to Algorithm 1 which utilizes the NTT. We first present the following theorem which will be used in the proof of Theorem 3.

Theorem 2 For a d^{th} primitive root of unity r , the following equality holds:

$$x^d - 1 = (x - r^0) \cdot (x - r^1) \cdot (x - r^2) \cdot \dots \cdot (x - r^{d-1}) \quad (10)$$

Proof of Theorem 2 Any polynomial of degree d is uniquely identified by its d roots. Hence, it suffices to show that $x^d - 1$ and $(x - r^0) \cdot (x - r^1) \cdot (x - r^2) \cdots (x - r^{d-1})$ have the same roots in order to prove that they are equivalent. Clearly, r^i is a distinct root of $(x - r^0) \cdot (x - r^1) \cdot (x - r^2) \cdots (x - r^{d-1})$ for $i = 0, 1, 2, \dots, d - 1$ since r is a d^{th} primitive root of unity. Since r is a d^{th} primitive root of unity, $r^d = 1$ and $(r^i)^d - 1 = (r^d)^i - 1 = 1^i - 1 = 0$ for $i = 0, 1, 2, \dots, d - 1$. Hence, r^i is also a root of the polynomial $x^d - 1$ for $i = 0, 1, 2, \dots, d - 1$, and thus the two polynomials are equivalent. \square

Theorem 3 describes the relationship between Algorithm 2, which uses the RNS and CRT, and Algorithm 1, which uses the NTT.

Theorem 3 *DFT multiplication algorithm, described with Algorithm 1, which utilizes a d^{th} primitive root of unity r for multiplication in $GF(q^m)$, where $d \geq 2m - 1$, is equivalent to RNS polynomial multiplication in $GF(q^m)$, described with Algorithm 2, which utilizes the Chinese Remainder Theorem with the relatively prime binomials $x - r^0, x - r^1, x - r^2, \dots, x - r^{d-1}$.*

Proof of Theorem 3 Due to the convolution theorem [8, 19], pairwise multiplication of the elements of two d -element sequences in the frequency domain corresponds to the cyclic convolution of the two sequences in the time domain. Hence, by pairwise multiplying the frequency domain representations of two input polynomials, Algorithm 1 yields a product which is the cyclic convolution of the input polynomials and thus equals $a(x) \cdot b(x) \bmod x^d - 1$. On the other hand, Algorithm 2 computes $a(x) \cdot b(x) \bmod P(x)$, where $P(x) = \prod_{0 \leq i \leq d-1} (x - r^i)$, which is equivalent to $a(x) \cdot b(x) \bmod x^d - 1$ due to Theorem 2. Hence, Algorithms 1 and 2 are equivalent. \square

NTT based finite field multiplication in $GF(q^m)$, presented with Algorithm 1, or the equivalent RNS based method presented with Algorithm 2, has only $O(m)$ complexity in terms of the required $GF(q)$ multiplications ignoring the conversions. In the next section, we will provide efficient parameters for the NTT based method which achieve the conversions extremely efficiently and for many cases meet the minimal theoretical bound of $2m - 1$ multiplications in $GF(q)$ for multiplication in $GF(q^m)$ where q is an odd prime [29].

2.5 Special NTTs for Efficient Multiplication in $GF(q^m)$

By selecting special values for the parameters such as $r, d, \text{etc.}$, fast NTT computations can be made possible which may yield efficient multiplication in $GF(q^m)$ using the NTT.

2.5.1 Mersenne Transform

A *number theoretic transform* of special interest is the *Mersenne transform*, which is an NTT with arithmetic modulo a Mersenne number of the form $M_n = 2^n - 1$ [23]. The Mersenne transform allows for very efficient forward and inverse DFT operations for $r = \pm 2$. Multiplication of an n -bit number with integer powers of 2 modulo M_n can be achieved with a simple bitwise left rotation of the n -bit number, e.g. multiplication of an n -bit number with 2^i modulo M_n can be achieved with a simple bitwise left rotation by $i \bmod n$ bits. Similarly, multiplication of an n -bit number with integer powers of -2 modulo M_n can be achieved with a simple bitwise left rotation of the number, in addition to a negation if the power of -2 is odd. Also, note that negation of an n -bit number modulo M_n can simply be achieved by flipping all n bits of the number. Hence, when $r = \pm 2$ all of the multiplications by powers of r in the forward and inverse DFT computations in a Mersenne transform can be achieved with simple bitwise rotations. In this case, for a transform length of d ,

the forward DFT computation can be achieved with only $(d - 1)^2$ simple rotations and $d(d - 1)$ additions/subtractions avoiding any multiplications. For the inverse DFT computation additional d constant multiplications with $1/d$ are required. Hence, when q is a Mersenne prime, multiplication in $GF(q^m)$ using Algorithm 1 has only $O(m)$ complexity in terms of $GF(q)$ multiplications and $O(m^2)$ complexity in terms of $GF(q)$ additions/subtractions and rotations. For a more detailed complexity analysis and efficient implementation ideas for the Mersenne transform, we refer the interested reader to [23].

Remember that, as in all number theoretic transforms, in a Mersenne transform the values of the sequence length d and the d^{th} primitive root of unity r are dependent on each other and can not be chosen independently. In a Mersenne transform over $GF(q)$, where $q = M_n = 2^n - 1$ is a Mersenne prime, and for $r = \pm 2$, the following equalities hold determining the relationship between d and r :

$$d = \begin{cases} n, & r = 2. \\ 2n, & r = -2. \end{cases}$$

In Table 1, we provide a list of parameters for utilizing the Mersenne transform which may yield efficient multiplication in $GF(q^m)$ in the frequency domain for operand sizes relevant to ECC.

2.5.2 Pseudo-Mersenne Transform

Similar to the Mersenne transform achieved modulo a Mersenne number, an NTT modulo an integer submultiple of a Mersenne number, e.g., $M_n/t = (2^n - 1)/t$ for an integer t , can also be performed efficiently and is called *the pseudo-Mersenne transform* [17]. In a pseudo-Mersenne transform all arithmetic operations can be achieved using Mersenne number arithmetic modulo the Mersenne number M_n and only the final result needs to be reduced modulo M_n/t . Hence, the pseudo-Mersenne transform, similar to the Mersenne transform, allows for very efficient forward and inverse DFT operations for $r = \pm 2$, since intermediary multiplications with integer powers of ± 2 modulo M_n/t can be achieved with a simple bitwise rotation, in addition to a negation if the power of $r = -2$ is odd. Also, remember that negation of an n -bit number modulo M_n can simply be achieved by flipping all n bits of the number. Thus, for a transform length of d , the forward DFT computation can be achieved with only $(d - 1)^2$ simple rotations and $d(d - 1)$ additions/subtractions avoiding any multiplications. For the inverse DFT computation additional d constant multiplications with $1/d \bmod M_n/t$ are required. Hence, when q is a pseudo-Mersenne prime, multiplication in $GF(q^m)$ using Algorithm 1 has only $O(m)$ complexity in terms of $GF(q)$ multiplications and $O(m^2)$ complexity in terms of $GF(q)$ additions/subtractions and rotations. We will see in Section 2.5.5 that for the cases when $q = M_n/t$ is a Fermat prime, with the use of the fast Fourier transform, this complexity can be further reduced to $O(m \log m)$ in terms of the number of required additions/subtractions and rotations. Although the pseudo-Mersenne transform increases the number of available transform lengths for the Mersenne transform, it has the downside of increasing the word size for the intermediary arithmetic operations from $n - \log_2 t$ to n for a pseudo-Mersenne transform modulo $M_n/t = (2^n - 1)/t$. In Table 2, we provide a list of parameters for utilizing the pseudo-Mersenne transform which may yield efficient multiplication in $GF(q^m)$ in the frequency domain for operand sizes relevant to ECC.

2.5.3 Fermat Transform

Fermat primes are also popular choices as finite field characteristics since modular reductions by Fermat primes can be achieved by simple addition/subtraction and shift operations. A number

$q = M_n = 2^n - 1$	m	d	r	equivalent binary field size
$2^{13} - 1$	11	26	-2	$\sim 2^{143}$
$2^{13} - 1$	12	26	-2	$\sim 2^{156}$
$2^{13} - 1$	13	26	-2	$\sim 2^{169}$
$2^{17} - 1$	9	17	2	$\sim 2^{153}$
$2^{17} - 1$	11	34	-2	$\sim 2^{187}$
$2^{17} - 1$	12	34	-2	$\sim 2^{204}$
$2^{17} - 1$	13	34	-2	$\sim 2^{221}$
$2^{17} - 1$	14	34	-2	$\sim 2^{238}$
$2^{17} - 1$	15	34	-2	$\sim 2^{255}$
$2^{17} - 1$	16	34	-2	$\sim 2^{272}$
$2^{17} - 1$	17	34	-2	$\sim 2^{289}$
$2^{19} - 1$	10	19	2	$\sim 2^{190}$
$2^{19} - 1$	11	38	-2	$\sim 2^{209}$
$2^{19} - 1$	12	38	-2	$\sim 2^{228}$
$2^{19} - 1$	13	38	-2	$\sim 2^{247}$
$2^{19} - 1$	14	38	-2	$\sim 2^{266}$
$2^{19} - 1$	15	38	-2	$\sim 2^{285}$
$2^{19} - 1$	16	38	-2	$\sim 2^{304}$
$2^{19} - 1$	17	38	-2	$\sim 2^{323}$
$2^{19} - 1$	18	38	-2	$\sim 2^{342}$
$2^{19} - 1$	19	38	-2	$\sim 2^{361}$
$2^{31} - 1$	11	31	2	$\sim 2^{341}$
$2^{31} - 1$	12	31	2	$\sim 2^{372}$
$2^{31} - 1$	13	31	2	$\sim 2^{403}$

Table 1: List of some $q = M_n$, m , d and $r = \pm 2$ values for efficient multiplication in $GF(q^m)$ in the frequency domain for ECC over finite fields of size 143 to 403 bits.

$q = M_n/t = (2^n - 1)/t$	m	d	r	equivalent binary field size
$(2^{15} - 1)/217$	15	30	-2	$\sim 2^{120}$
$(2^{23} - 1)/47$	9	23	2	$\sim 2^{162}$
$(2^{23} - 1)/47$	10	23	2	$\sim 2^{180}$
$(2^{23} - 1)/47$	11	23	2	$\sim 2^{198}$
$(2^{23} - 1)/47$	12	23	2	$\sim 2^{216}$
$(2^{23} - 1)/47$	17	46	-2	$\sim 2^{306}$
$(2^{23} - 1)/47$	18	46	-2	$\sim 2^{324}$
$(2^{23} - 1)/47$	19	46	-2	$\sim 2^{342}$
$(2^{23} - 1)/47$	20	46	-2	$\sim 2^{360}$
$(2^{23} - 1)/47$	21	46	-2	$\sim 2^{378}$
$(2^{23} - 1)/47$	22	46	-2	$\sim 2^{396}$
$(2^{23} - 1)/47$	23	46	-2	$\sim 2^{414}$
$(2^{27} - 1)/511$	11	27	2	$\sim 2^{209}$
$(2^{27} - 1)/511$	12	27	2	$\sim 2^{228}$
$(2^{27} - 1)/511$	13	27	2	$\sim 2^{247}$
$(2^{27} - 1)/511$	14	27	2	$\sim 2^{266}$
$(2^{32} - 1)/65535$	13	32	2	$\sim 2^{221}$
$(2^{32} - 1)/65535$	14	32	2	$\sim 2^{238}$
$(2^{32} - 1)/65535$	15	32	2	$\sim 2^{255}$
$(2^{32} - 1)/65535$	16	32	2	$\sim 2^{272}$
$(2^{33} - 1)/14329$	17	33	2	$\sim 2^{340}$
$(2^{37} - 1)/223$	11	37	2	$\sim 2^{330}$
$(2^{37} - 1)/223$	12	37	2	$\sim 2^{360}$
$(2^{37} - 1)/223$	13	37	2	$\sim 2^{390}$
$(2^{37} - 1)/223$	14	37	2	$\sim 2^{420}$
$(2^{37} - 1)/223$	15	37	2	$\sim 2^{450}$
$(2^{37} - 1)/223$	16	37	2	$\sim 2^{480}$
$(2^{37} - 1)/223$	17	37	2	$\sim 2^{510}$
$(2^{37} - 1)/223$	18	37	2	$\sim 2^{540}$
$(2^{37} - 1)/223$	19	37	2	$\sim 2^{570}$

Table 2: List of some $q = M_n/t$, m , d and $r = \pm 2$ values for efficient DFT modular multiplication in $GF(q^m)$ for ECC over finite fields of size 120 to 570 bits.

theoretic transform with arithmetic modulo a Fermat number of the form $F_n = 2^{2^n} + 1$ for a positive integer n is called *the Fermat Transform* [24, 23, 1, 2]. Fermat transforms were first defined and proposed for fast convolution and digital filtering by Agarwal and Burrus [1, 2]. In this work, we provide efficient parameters for their use in *finite field polynomial multiplication* which may find applications in cryptography.

The Fermat transform allows for very efficient forward and inverse DFT computations for $r = 2^{2^k}$, where k is a non-negative integer. For a Fermat prime p , i.e., when $p = 2^{2^n} + 1$, $2^{2^{n+1}} \equiv 1 \pmod{p}$ and $r = 2$ is a d^{th} primitive root of unity where $d = 2^{n+1}$. Likewise, $r = 2^{2^k}$ is a d^{th} primitive root of unity where $d = 2^{n+1-k}$. Thus, since d is a power of 2, as we will see in Section 2.5.5, the fast Fourier transform can be applied very efficiently for computation of the number theoretic transform of a sequence of length d , significantly reducing the complexity of Algorithm 1 for polynomial multiplication in finite fields. Also, all modular multiplications by powers of $r = 2^{2^k}$ can be achieved by simple shift and subtraction operations. In this case, modular multiplication by a power of r is slightly more complex than that in the Mersenne transform where the same operation can be achieved with a mere bitwise rotation. However, since $F_n = 2^{2^n} + 1 = \frac{2^{2^{n+1}} - 1}{2^{2^n} - 1}$, the Fermat transform modulo $F_n = 2^{2^n} + 1$ is equivalent to the corresponding pseudo-Mersenne transform modulo $M_n/t = \frac{2^{2^{n+1}} - 1}{2^{2^n} - 1}$ and can be achieved via Mersenne number arithmetic modulo the Mersenne number $2^{2^{n+1}} - 1$. Thus, one can achieve all multiplications by powers of $r = 2^{2^k}$ with simple bitwise rotations. The advantage of this approach compared with the Fermat transform using the Fermat number arithmetic is that modular multiplications with powers of r become mere rotations and can be achieved very easily, however it has the drawback of having almost twice increase in the word size of operands. Hence, when q is a Fermat number, the complexity of multiplication in $GF(q^m)$ using the Fermat transform is only $O(m)$ multiplications and $O(m \log m)$ additions/subtractions and shifts in terms of $GF(q)$ operations. In Table 3, we give a list of some Fermat primes and values for d and m that allow for application of the FFT and thus possibly efficient polynomial multiplication in $GF(p^m)$ in the frequency domain.

Efficient Fermat transform is also possible with a d^{th} primitive root of unity of the form $r = (\sqrt{2})^k$, for a positive integer k , thanks to the following relationship

$$\sqrt{2} = 2^{e/4}(2^{e/2} - 1) \pmod{2^e + 1}.$$

Thus, when $r = (\sqrt{2})^k$ in a Fermat transform modulo $F_n = 2^{2^n} + 1$ or a pseudo-Fermat transform modulo $F_n/t = \frac{2^{2^n} + 1}{t}$ (see Section 2.5.4), $r = (\sqrt{2})^{ki} = 2^a - 2^b$ for some positive integers a and b , and hence all multiplications by powers of $r = (\sqrt{2})^k$ can be carried out with at most two shifts/rotations and a subtraction.

2.5.4 Pseudo-Fermat Transform

An NTT can also be efficiently performed modulo a pseudo-Fermat prime, e.g., $F_n/t = \frac{2^n + 1}{t}$ for a positive integer t , and is called *the pseudo-Fermat transform* [18]. In a pseudo-Fermat transform, all intermediary arithmetic operations can be carried out using Fermat number arithmetic modulo $2^n + 1$ or Mersenne number arithmetic modulo the Mersenne number $M_n = 2^{n+1} - 1 = (2^n + 1) \cdot (2^n - 1)$ and only the final result needs to be reduced modulo $F_n/t = \frac{2^n + 1}{t}$. Hence, the pseudo-Fermat transform retains some of the computational advantages of the Mersenne transform and introduces further variety in the available transform lengths in the expense of increasing the word size for the intermediary arithmetic operations from $n + 1 - \log_2 t$ to $n + 1$ for a pseudo-Fermat transform modulo $F_n/t = \frac{2^n + 1}{t}$. In Table 4, we present a list of pseudo-Fermat primes and corresponding transform parameters suitable for finite field multiplication in $GF(q^m)$ for ECC.

$p = F_n = 2^{2^n} + 1$	m	d	r	equivalent binary field size
$2^{2^3} + 1$	13	32	$\sqrt{2}$	117
$2^{2^3} + 1$	14	32	$\sqrt{2}$	126
$2^{2^3} + 1$	15	32	$\sqrt{2}$	135
$2^{2^3} + 1$	16	32	$\sqrt{2}$	144
$2^{2^4} + 1$	7	16	2^2	119
$2^{2^4} + 1$	8	16	2^2	136
$2^{2^4} + 1$	13	32	2	221
$2^{2^4} + 1$	14	32	2	238
$2^{2^4} + 1$	15	32	2	255
$2^{2^4} + 1$	16	32	2	272
$2^{2^4} + 1$	29	64	$\sqrt{2}$	478
$2^{2^4} + 1$	30	64	$\sqrt{2}$	495
$2^{2^4} + 1$	31	64	$\sqrt{2}$	512
$2^{2^4} + 1$	32	64	$\sqrt{2}$	544

Table 3: List of Fermat primes $p = F_n = 2^{2^n} + 1$ and d, r, m values for efficient multiplication in $GF(p^m)$ in the frequency domain for ECC over finite fields of size 117 to 544 bits .

2.5.5 Fast Fourier Transform

An extremely efficient method for computing the DFT is the fast Fourier transform (FFT) [9]. The FFT algorithm works by exploiting the symmetry of the DFT computation and the periodicity of the d^{th} primitive root of unity r when the sequence length d is a composite number. For instance, if a sequence is of even length, i.e. if its length d is divisible by two, then by applying the FFT the DFT computation of this d -element sequence is basically reduced to the DFT computations of two $(d/2)$ -element sequences, namely the sequence comprising only the even indexed elements of the original sequence and the sequence comprising only the odd indexed elements of the original sequence. The DFT of a d -element sequence (a) , where d is divisible by 2, can be expressed as follows:

$$A_j = \sum_{i=0}^{d-1} a_i r^{ij} = \sum_{i=0}^{\frac{d}{2}-1} a_{2i} r^{2ij} + \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} r^{(2i+1)j} = \sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{ij} + r^j \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{ij}, \quad 0 \leq j \leq d-1. \quad (11)$$

Note that r^2 is a $(\frac{d}{2})^{\text{th}}$ primitive root of unity in $GF(p)$. Hence, the above d -element DFT computation of A_j , for $0 \leq j \leq d-1$, can be performed with two $(\frac{d}{2})$ -element DFTs which are the DFTs of the $(\frac{d}{2})$ -element sequences consisting of the even indexed elements and the odd indexed elements of (a) . In (11), the first and the second summations correspond to the $(\frac{d}{2})$ -element DFTs of even and odd indexed elements of (a) , respectively. Here, A_j needs to be computed for $0 \leq j \leq d-1$, not for $0 \leq j \leq \frac{d}{2}-1$. However, $(r^2)^j$ is periodic with $\frac{d}{2}$ for a d^{th} primitive root of unity r and d even and therefore $r^{j+\frac{d}{2}} = -r^j$. Thus, the equalities

$$\sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{i(j+\frac{d}{2})} = \sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{ij}$$

$q = F_n/t = (2^n + 1)/t$	m	d	r	equivalent binary field size
$(2^{13} + 1)/3$	13	26	2	$\sim 2^{156}$
$(2^{15} + 1)/99$	15	30	2	$\sim 2^{135}$
$(2^{17} + 1)/3$	17	34	2	$\sim 2^{272}$
$(2^{19} + 1)/3$	19	38	2	$\sim 2^{342}$
$(2^{19} + 1)/3$	10	19	2^2	$\sim 2^{180}$
$(2^{20} + 1)/17$	8	16	$(\sqrt{2})^5$	$\sim 2^{128}$
$(2^{20} + 1)/17$	20	40	2	$\sim 2^{320}$
$(2^{20} + 1)/17$	10	20	2^2	$\sim 2^{160}$
$(2^{21} + 1)/387$	21	42	2	$\sim 2^{273}$
$(2^{22} + 1)/1985$	22	44	2	$\sim 2^{264}$
$(2^{22} + 1)/1985$	11	22	2^2	$\sim 2^{132}$
$(2^{23} + 1)/3$	23	46	2	$\sim 2^{506}$
$(2^{27} + 1)/1539$	27	54	2	$\sim 2^{459}$
$(2^{27} + 1)/1539$	9	18	2^3	$\sim 2^{153}$
$(2^{28} + 1)/17$	14	28	2^2	$\sim 2^{336}$
$(2^{28} + 1)/17$	7	14	2^4	$\sim 2^{168}$
$(2^{32} + 1)/641$	16	32	2^2	$\sim 2^{368}$
$(2^{32} + 1)/641$	8	16	2^4	$\sim 2^{184}$

Table 4: List of some $q = F_n/t$, m , d and $r = \pm 2$ values for efficient multiplication in $GF(q^m)$ in the frequency domain for ECC over finite fields of size 128 to 506 bits.

and

$$r^{j+\frac{d}{2}} \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{i(j+\frac{d}{2})} = -r^j \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{ij}$$

hold. Therefore, once A_j is computed for $0 \leq j \leq \frac{d}{2} - 1$ as in (11) by performing two $(\frac{d}{2})$ -element NTTs, $\frac{d}{2} - 1$ multiplications for multiplications of the second summations by r^j (for $j = 0$ no multiplication is necessary for a multiplication by r^j) and $\frac{d}{2}$ additions for merging the two summations together, we can compute A_j for $\frac{d}{2} \leq j \leq d - 1$ immediately by using the same already computed summations and with only additional $\frac{d}{2}$ subtractions for merging the two summations. The inverse FFT can be computed in a similar manner as the forward FFT with the exception of the minus signs in front of the powers of r and d additional constant multiplications due to the multiplications by d^{-1} . When d is a power of two, the same approach can easily be applied recursively surprisingly reducing the $O(d^2)$ complexity of the DFT computation to $O(d \log_2 d)$. Likewise, if d is a power of three, a similar approach could be applied recursively to reduce the $O(d^2)$ complexity of the DFT computation to $O(d \log_3 d)$.

Unfortunately, full recursive application of the FFT for finite field polynomial multiplication using Algorithm 1 would find limited application since there are only a few cases in Tables 1, 2, 3 and 4 where the available sequence length d is a power of 2 or another small prime number. Also, we would like to note that for the efficient case of Algorithm 1 with the Mersenne transform where all arithmetic is performed modulo the Mersenne prime $M_n = 2^n - 1$, which is the field characteristic, and there is no redundancy due to an increase in the word size, the allowable sequence length d is either the prime number n (for $r = 2$) or $2n$ (for $r = -2$). Hence, either $d = n$ is prime and the FFT algorithm can not be applied at all or $d = 2n$ and only a single level of recursion is allowed in the FFT computation which would have limited computational advantage. In the next section, we introduce the DFT modular multiplication algorithm which achieves both multiplication and modular reduction in the frequency domain and could be more efficient than Algorithm 1 or other efficient methods, especially for multiplication in Mersenne fields.

3 Modular Multiplication in the Frequency Domain

In many finite field applications, a chain of arithmetic operations need to be performed, rather than a solitary one. For example, in elliptic curve cryptography a scalar point product is computed by applying a chain of finite field additions, subtractions, multiplications, squarings and inversions on the input point coordinates [7]. The Montgomery residue representation has proven to be useful in this computation [16, 13]. In using this method, first the operands are converted to their respective Montgomery residue representations, then utilizing Montgomery arithmetic the desired computation is implemented, and finally the result is converted back to the normal integer or polynomial representation. If there are a large number of operations performed in the Montgomery domain, due to the efficiency of the intervening computations, the forward and backward conversion operations become affordable. We introduce the same notion for the frequency domain. We present an arithmetic operation in the frequency domain that is equivalent to Montgomery multiplication in the time domain. Due to the *linearity* property of the DFT [8, 19], operations in the time domain such as addition/subtraction and multiplication by a scalar directly map to the frequency domain, i.e., for any two sequences (a) and (b) representing elements of $GF(q^m)$ in the time domain and for any two scalars $y, z \in GF(q)$,

$$\text{DFT}(y \cdot (a) \pm z \cdot (b)) = y \cdot \text{DFT}((a)) \pm z \cdot \text{DFT}((b)) .$$

Hence, if a modular multiplication algorithm in the frequency domain can also be utilized, then all finite field operations such as addition/subtraction and multiplication can be performed in the frequency domain, and thus a finite field application, e.g. ECC, can be achieved completely in the frequency domain, assuming also that for inversion a Fermat like inversion algorithm consisting of multiplications is utilized and/or projective coordinates are used to avoid inversions. In the remainder of this section, we introduce the *DFT modular multiplication* algorithm which allows for both polynomial multiplication and Montgomery modular reduction operations in the frequency domain.

3.1 Mathematical Notation

Since the DFT modular multiplication algorithm runs in the frequency domain, the parameters used in the algorithm are in their frequency domain sequence representations. These parameters are the input operands $a(x), b(x) \in GF(q^m)$, the result $c(x) = a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) \in GF(q^m)$, irreducible field generating polynomial $f(x)$, normalized irreducible field generating polynomial $f'(x) = f(x)/f(0)$, and the indeterminate x . The time domain sequence representations of these parameters are $(a), (b), (c), (f), (f')$ and (x) , respectively, and their frequency domain sequence representations, i.e. the discrete Fourier transforms of the time domain sequence representations, are $(A), (B), (C), (F), (F')$ and (X) . We will denote elements of a sequence with the name of the sequence and a subscript for showing the location of the particular element in the sequence, e.g. for the indeterminate x represented as the following d -element sequence in the time domain

$$(x) = (0, 1, 0, 0, \dots, 0) ,$$

the DFT of (x) is computed as the following d -element sequence

$$(X) = (1, r, r^2, r^3, r^4, r^5, \dots, r^{d-1}) ,$$

whose first and last elements are denoted as $X_0 = 1$ and $X_{d-1} = r^{d-1}$, respectively.

3.2 The DFT Modular Multiplication Algorithm

DFT modular multiplication presented with Algorithm 3 consists of two parts: multiplication (steps 1 – 3) and Montgomery reduction (steps 4 – 13). Multiplication is performed by simple pairwise multiplication of the coefficients of the frequency domain sequence representations of the input operands. Reduction is more complex and performed by Montgomery reduction in the frequency domain. In the reduction process the normalized field generating polynomial $f'(x) = f(x)/f_0 \bmod q$ is used. Hence, $f'(x)$ is equivalent to $f(x)$ but normalized to have $f'(0) = 1$ and thus $f'_i = f_i/f_0 \bmod q$ and $F'_i = F_i/f_0 \bmod q$, for $0 \leq i \leq d - 1$.

Algorithm 3 DFT Modular Multiplication

Input: (A) , (B) which represent $a(x), b(x) \in GF(q^m)$ in the frequency domain

Output: (C) which represents $c(x) = a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x)$ in the frequency domain

```
1: for  $i = 0$  to  $d - 1$  do
2:    $C_i \leftarrow A_i \cdot B_i$ 
3: end for
4: for  $j = 0$  to  $m - 2$  do
5:    $S \leftarrow C_0$ 
6:   for  $i = 1$  to  $d - 1$  do
7:      $S \leftarrow S + C_i$ 
8:   end for
9:    $S \leftarrow -S/d$ 
10:  for  $i = 0$  to  $d - 1$  do
11:     $C_i \leftarrow (C_i + F'_i \cdot S) \cdot X_i^{-1}$ 
12:  end for
13: end for
14: Return  $(C)$ 
```

Proof of Correctness:

DFT modular multiplication is a direct adaptation of Montgomery multiplication for the frequency domain. Polynomial multiplication part of the algorithm (steps 1 – 3) is performed via simple pairwise multiplications. As a result, the polynomial $c(x) = a(x) \cdot b(x)$ is obtained. In the modular reduction part (steps 4 – 13), S is computed such that $(c(x) + S \cdot f'(x))$ is a multiple of x . This is accomplished by computing $-c_0$, the negative of the first coefficient in the time domain sequence (c) and then by making c_0 zero by adding $S \cdot (F')$ to (C) in the frequency domain. Then again in the frequency domain, $(c(x) + S \cdot f'(x))$ is divided by x and the result, which is congruent to $c(x) \cdot x^{-1}$ modulo $f(x)$ in the time domain, is obtained. This division of $(c(x) + S \cdot f'(x))$ by x is accomplished in the frequency domain by dividing $(C) + (F') \cdot S$ by (X) (Step 11). By repeating steps 5 – 12 of the algorithm $m - 1$ times the final result (C) , which represents $a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x)$ in the frequency domain, is obtained.

□

The inputs of the DFT modular multiplication algorithm presented with Algorithm 3 are the DFTs (A) and (B) of the d -element sequences (a) and (b) which represent $a(x), b(x) \in GF(q^m)$, respectively. The output of the algorithm is (C) , the DFT of the sequence (c) , where (c) represents the d -element sequence for $c(x) = a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x)$. The extra $x^{-(m-1)}$ factor shows that the DFT modular multiplication algorithm actually computes Montgomery products of polynomials in the frequency domain. Hence, the input polynomials $a(x)$ and $b(x)$ may be viewed as the Montgomery residue representations of two polynomials $u(x)$ and $v(x)$ such that

$$a(x) = u(x) \cdot x^{m-1} \bmod f(x)$$

and

$$b(x) = v(x) \cdot x^{m-1} \bmod f(x) .$$

By DFT modular multiplication the residue representation is kept intact, i.e.,

$$a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) = (u(x) \cdot v(x)) \cdot x^{m-1} \bmod f(x)$$

which allows for further computation in the frequency domain.

For $d \approx 2m$, modular multiplication in $GF(q^m)$ with the DFT modular multiplication algorithm requires only $2m$ multiplications in addition to $4m^2 - 3m - 1$ constant multiplications and $4m^2 - 5m + 1$ additions in the ground field $GF(q)$, while the classical method requires m^2 multiplications and $(m-1)^2$ additions ignoring the cost of modular reduction. In the next section, we will see that this complexity of DFT modular multiplication may be improved dramatically by using special values for q , r , d and the irreducible field generating polynomial $f(x)$.

3.3 Utilizing Efficient Parameters for DFT Modular Multiplication

Using the smallest possible sequence length d , satisfying $d \geq 2m - 1$, will lead to the smallest number of arithmetic operations in the computation of DFT modular multiplication. Optimally, $d = 2m - 1$ will lead to the least number of arithmetic operations.

As mentioned in Section 2.5.1, using a Mersenne prime as the modulus and selection of $r = \pm 2$ will allow for extremely efficient modular multiplication with $r^i = \pm 2^i$. This modular multiplication can be achieved with a simple bitwise rotation (in addition to a negation when $r = -2$ and i is odd) which is inexpensive. In the DFT modular multiplication algorithm, this property may be exploited if q is chosen as a Mersenne prime and r is chosen as $r = \pm 2$. In that case, in step 11 of the algorithm multiplications with $X_i^{-1} = r^{-i} = (\pm 2)^{-i} = (\pm 1)^i \cdot 2^{-i \bmod d}$ become simple $(-i \bmod d)$ -bit left-rotations (in addition to a negation when $r = -2$ and i is odd), which have negligible cost compared to a regular multiplication. Also, note that when q is a Mersenne prime, negation of an element of $GF(q)$ can be achieved by simply flipping its bits. Multiplications with F'_i in step 11 can also be avoided in a similar fashion for special $f(x)$. For instance, for the binomial $f(x) = x^m \pm r^{s_0}$ with s_0 an integer, $F'_i = \pm r^{mi-s_0 \bmod d} + 1$ and hence for $r = \pm 2$ multiplications with F'_i can be achieved with only one bitwise rotation and one addition/subtraction. Likewise, for the trinomial $f(x) = x^m \pm r^{s'_m} x^{m'} \pm r^{s_0}$ or $f(x) = x^m \mp r^{s'_m} x^{m'} \pm r^{s_0}$, where s'_m and s_0 are integers, $F'_i = \pm r^{mi-s_0} + r^{m'i+s'_m-s_0} + 1$ or $F'_i = \pm r^{mi-s_0} - r^{m'i+s'_m-s_0} + 1$, respectively, and hence multiplications with F'_i can be achieved with only two bitwise rotations and two additions/subtractions. Finally, we would like to caution the reader that all these parameters q, d, r and $f(x)$ are dependent on each other and can not be chosen independently.

3.3.1 Existence of Efficient Parameters

In Tables 1, 2, 3 and 4 we give lists of parameters that would yield efficient multiplication in $GF(q^m)$ using the DFT modular multiplication algorithm for operand sizes relevant to elliptic curve cryptography. For every case listed in these tables, one can verify that there exist many special irreducible binomials of the form $x^m \pm 2^s$, or trinomials of the form $x^m \pm r^{s_1} x_1 \pm r^{s_0}$ or $x^m \pm r^{s_1} x_1 \mp r^{s_0}$, as field generating polynomials that would allow for efficient DFT modular multiplication. For some of these cases there exist efficient irreducible binomials which we present with Theorem 6 and in Appendix, whereas for other cases we were not able to find such binomials and included lists of efficient irreducible trinomials instead (see Appendix). However, note that for the computationally more desirable cases of $d = 2m$, where $m = n$ and $q = 2^n - 1$ is the Mersenne prime field characteristic, there always exist efficient irreducible binomials. We would like to first present Theorems 4 and 5, and Definition 2, which will be used in the proof of Theorem 6.

Theorem 4 [14] *Let $m \geq 2$ be an integer and $w \in GF(q)^*$. Then the binomial $x^m - w$ is irreducible in $GF(q)[x]$ if and only if the following three conditions are satisfied:*

1. each prime factor of m divides the order e of w in $GF(q)^*$;
2. the prime factors of m do not divide $\frac{q-1}{e}$;
3. $q \equiv 1 \pmod{4}$ if $m \equiv 0 \pmod{4}$.

Theorem 5 [15] Let $\alpha, \beta \in GF(q)$ and $\alpha = \beta^i$. The orders of α and β are related as

$$\text{ord}(\alpha) = \frac{\text{ord}(\beta)}{\text{gcd}(i, \text{ord}(\beta))},$$

where $\text{ord}(a)$ denotes the order of field element a and $\text{gcd}(a, b)$ denotes the greatest common denominator of a and b .

Definition 2 A Wieferich prime is an odd prime p which satisfies $2^{p-1} \equiv 1 \pmod{p^2}$.

Theorem 6 For a Mersenne prime $q = 2^n - 1$ and for $m = n$, a binomial of the form $x^m \pm 2^s$, where s is an integer not congruent to 0 modulo n , is irreducible in $GF(q)[x]$ if m is not a Wieferich prime.

Proof of Theorem 6 For a binomial of the form $x^m \pm 2^s$ to be irreducible in $GF(q)[x]$, it needs to satisfy all three conditions of Theorem 4. The third condition is satisfied since $m = n$ is a prime number and the condition $m \equiv 0 \pmod{4}$ never holds. For the first and second conditions, we consider the cases $x^m - 2^s$ and $x^m + 2^s$ separately.

Let's first consider the binomials $x^m - 2^s$. When q is a Mersenne prime of the form $q = 2^n - 1$ the order of 2 in $GF(q)$ is $n = m$ since $2^n \equiv 1 \pmod{q}$ and $2^i \not\equiv 1 \pmod{q}$ for $i < n$. Due to Theorem 5 the order of 2^s in $GF(q)$ is $\frac{\text{ord}(2)}{\text{gcd}(s, \text{ord}(2))} = \frac{m}{\text{gcd}(s, m)} = m$. The only prime factor of m , which is itself, divides m and hence the first condition of Theorem 4 is satisfied. The second condition of Theorem 4 is satisfied since $m \mid \frac{2^m - 2}{m}$ never holds unless m is a Wieferich prime.

Now, let's consider the binomials $x^m + 2^s = x^m - (-2^s)$. Let's first find the order of (-2^s) in $GF(q)$, i.e., the smallest positive integer k such that $(-2^s)^k \equiv 1 \pmod{q}$. The equality $(-2^s)^k = (-1)^k \cdot (2^s)^k \equiv 1 \pmod{q}$ can hold true in only two cases:

- Case 1: The integer k is odd, thus $(-1)^k \equiv -1 \pmod{q}$, and $(2^s)^k \equiv -1 \pmod{q}$
- Case 2: The integer k is even, thus $(-1)^k \equiv 1 \pmod{q}$, and $(2^s)^k \equiv 1 \pmod{q}$

We have $-1 \pmod{q} = 2^n - 2 = 2(2^{n-1} - 1)$ and $(2^s)^k \pmod{q} = 2^{sk \pmod{n}}$. The equality $2(2^{n-1} - 1) = 2^{sk \pmod{n}}$ never holds for any positive integer k . Hence, the equality " $(2^s)^k \equiv -1 \pmod{q}$ " in Case 1 can not hold and Case 1 is not possible. So, the integer k , which is the order of (-2^s) in $GF(q)$, satisfies Case 2, i.e., it is even and the smallest positive integer that satisfies $(2^s)^k \equiv 1 \pmod{q}$. Since in $GF(q)$ $\text{ord}(2^s) = m$ and m is odd, the smallest even k which satisfies $(2^s)^k \equiv 1 \pmod{q}$ is $2m$. Hence $2m$ is the order of (-2^s) in $GF(q)$. The first condition of Theorem 4 is satisfied for the irreducible binomials of the form $x^m + 2^s = x^m - (-2^s)$ since the only prime factor of m , which is itself, divides $2m$, the order of (-2^s) in $GF(q)$. The second condition of Theorem 4 is also satisfied since $m \mid \frac{2^m - 2}{2m}$ never holds unless m is a Wieferich prime. □

The only known Wieferich primes are 1093 and 3511. It is also known that there are no other Wieferich primes less than 4×10^{12} [10]. Hence, for the more efficient cases in Table 1 where the field characteristic $q = M_n = 2^n - 1$ is a Mersenne prime and $m = n$, m^{th} degree irreducible binomials of the form $x^m \pm 2^s$, for a nonzero integer s , always exist.

3.4 Complexity Analysis

In this section, we present the complexity of DFT modular multiplication for a practical set of parameters relevant to ECC and compare it with other efficient methods, e.g. multiplication by the direct application of the NTT and with time domain modular reduction (Algorithm 1). In our complexity analysis we assume the use of Mersenne prime finite field characteristics as q , irreducible field generating binomials of the form $f(x) = x^m \pm 2^{s_0}$, d -th primitive root of unity $r = \pm 2$ and sequence length $d \approx 2m$. The field parameters we use, such as the low Hamming weight field generating polynomial and Mersenne prime field characteristic, lead to efficient implementation of multiplication for all methods. Therefore, for the selected parameters, we can safely assume that our comparisons are fair. In Table 2, we present the complexities of multiplication in $GF(q^m)$ in terms of $GF(q)$ operations for both Algorithm 1 and Algorithm 3 when such ideal parameters are used. Note that the astonishingly low $O(m)$ complexity of both algorithms in terms of the number of $GF(q)$ multiplications is achieved under the ideal conditions with these efficient field parameters together with the choice of $r = \pm 2$.

	Algorithm 1	Algorithm 3
# Multiplication	$\approx 2m$	$\approx 2m$
# Const. Mult.	$\approx 2m - 1$	$m - 1$
# Add./Subtr.	$\approx 8m^2 - 7m$	$\approx 6m^2 - 7m + 1$
# Rotation	$\approx 8m^2 - 11m + 3$	$\approx 4m^2 - 4m$

Table 5: Complexity of multiplication in $GF(q^m)$ in terms of the number of $GF(q)$ operations when $f(x) = x^m + 2^{s_0}$, q is a Mersenne prime and $d \approx 2m$.

One could argue that for a Mersenne transform modulo a Mersenne prime $p = 2^n - 1$ and with $r = -2$, $d = 2n$ is composite, hence it is possible to utilize the FFT for one level and obtain faster computation when using the direct NTT approach of Algorithm 1. On the other hand, similar improvements also exist for DFT modular multiplication. For instance, for DFT modular multiplication in $GF(p^m)$ where $p = 2^n - 1$, $m = n$ and $r = -2$, $f(x) = x^m - 2$ is always irreducible (see Theorem 6) and could be used as the field generating polynomial. In this case, the normalized irreducible polynomial would be $f'(x) = -\frac{1}{2} \cdot x^m + 1$. The following equality holds in $GF(q)$:

$$F'_i = -\frac{1}{2} \cdot (-2)^{mi} + 1 = \begin{cases} -\frac{1}{2} + 1 = \frac{1}{2}, & i \text{ even} \\ \frac{1}{2} + 1, & i \text{ odd} \end{cases}$$

since

$$(-2)^{mi} \equiv (-2)^{ni} \equiv (-1)^{ni} \cdot (2^n)^i \equiv (-1)^{ni} \pmod{q}.$$

Note that in this case F'_i has only two distinct values, namely $-\frac{1}{2} + 1 = \frac{1}{2}$ and $\frac{1}{2} + 1$ for the irreducible field generating polynomial $f(x) = x^m - 2$. Hence, $F'_i \cdot S$ in Step 11 of the Algorithm 1 can attain only two values for any distinct value of S and these values can be precomputed outside the loop avoiding all such computations inside the loop. The precomputations can be achieved very efficiently with only one bitwise rotation and one addition. With the suggested improvement, both the number of base field additions/subtractions and the number of base field bitwise rotations required to perform

an extension field multiplication are reduced by $(d-1) \cdot (m-1) = (2m-1) \cdot (m-1) = 2m^2 - 3m + 1$. In Table 6, we present the new complexities of multiplication in $GF(q^m)$ when the single level FFT is used for the direct DFT approach (Algorithm 1) and the above mentioned improvement is utilized for DFT modular multiplication (Algorithm 3).

	Algorithm 1 (with FFT)	Algorithm 3 (improved)
# Multiplication	$\approx 2m$	$\approx 2m$
# Const. Mult.	$\approx 2m - 1$	$m - 1$
# Add./Subtr.	$\approx 4m^2 - 2$	$\approx 4m^2 - 4m$
# Rotation	$\approx 4m^2 - 7m + 3$	$\approx 2m^2 - m - 1$

Table 6: Complexity of multiplication in $GF(q^m)$ in terms of the number of $GF(q)$ operations when $f(x) = x^m - 2$, $q = 2^m - 1$ is a Mersenne prime and $d = 2m$.

Clearly, the complexity of DFT modular multiplication (Algorithm 3) is an improvement upon the direct DFT approach (Algorithm 1). Moreover, since DFT modular multiplication requires significantly less number of complex operations such as multiplication and constant multiplication, its overall performance appears to be better than the Karatsuba algorithm (see Table 7), for computationally constrained platforms where multiplication is significantly more expensive compared to simpler operations such as addition, subtraction or bitwise rotation.

	Karatsuba	Algorithm 3 (improved)
#Multiplication	$\approx m^{\log_2 3}$	$2m$
#Const. Mult.	–	$m - 1$
#Add./Subtr.	$\approx 6m^{\log_2 3} - 7m + 1$	$4m^2 - 4m$
#Rotation	$m - 1$	$2m^2 - m - 1$

Table 7: Complexity of multiplication in $GF(q^m)$ in terms of the number of $GF(q)$ operations when $f(x) = x^m - 2$, $q = 2^m - 1$ is a Mersenne prime and $d = 2m$.

Multiplication operation is inherently more complex than addition, subtraction or bitwise rotation. Therefore a multiplier circuit either runs much slower or it is designed significantly larger in area to run as fast. A straightforward low power/small area implementation of an n -bit multiplication can be achieved via n additions and n shift operations. Therefore, in serialized hardware implementation the complexity of an n -bit multiplication may be assumed to be roughly equal to the complexity of n additions and n shift operations. Using the same approach, we may assume that multiplication by a constant n -bit number can be achieved with $n/2$ shifts and $n/2$ additions on average. Under these assumptions, Table 8 gives the complexities of modular multiplication in $GF(q^{13})$ for both Algorithm 3 and the Karatsuba method when $q = 2^{13} - 1$ and $GF(q^{13})$ is constructed using the irreducible binomial $f(x) = x^{13} - 2$. The table also includes the total number of clock cycles that a single multiplication with these methods takes, assuming addition/subtraction and rotation operations all take a single clock cycle. Note that this finite field has size $\sim 2^{169}$ and is chosen to be representative for elliptic curve cryptography. We would like to note that the complexity figures in Table 8 corresponding to Algorithm 3 are obtained directly from Table 7 and the figures for the exact complexity of Karatsuba multiplication for $GF(q^{13})$ are drawn from [28].

	Karatsuba	Algorithm 2 (improved)
#Multiplication	91	26
#Const. Mult.	–	12
#Add./Subtr.	390	624
#Rotation	12	324
#Total Clock Cycles	2768	1780

Table 8: Complexity of multiplication in $GF(q^{13})$ where $f(x) = x^{13} - 2$, $q = 2^{13} - 1$, $d = 26$ and $r = -2$.

4 Conclusion

In this work, we investigated the application of the number theoretic transform, which found many applications in digital signal processing, to finite field multiplication with an emphasis on elliptic curve cryptography. We introduced the DFT modular multiplication algorithm which performs modular multiplication in the frequency domain using Montgomery reduction. By allowing for modular reductions in the frequency domain the costly overhead of back and forth conversions between the frequency and time domains is avoided, and thus efficient finite field multiplication is made possible for cryptographic operand sizes. We have shown that with the utilization of the NTT in general and with our *DFT modular multiplication* method in particular, especially in computationally constrained platforms, finite field multiplication could be achieved more efficiently in the frequency domain than in the time domain for even small finite field sizes, e.g. ~ 160 bits, relevant to elliptic curve cryptography.

References

- [1] R. C. Agarwal and C. S. Burrus. Fast Digital Convolution Using Fermat Transforms. In *Southwest IEEE Conf. Rec.*, pages 538–543, Houston, Texas, USA, April 1973.
- [2] R. C. Agarwal and C. S. Burrus. Fast Convolutions Using Fermat Number Transforms with Applications to Digital Filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-22(2):87–97, April 1974.
- [3] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, New York, USA, 1968.
- [4] R. E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, Massachusetts, USA, 1983.
- [5] R. E. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, Reading, Massachusetts, USA, 1985.
- [6] I.F. Blake, X.H. Gao, R.C. Mullin, S.A. Vanstone, and T. Yaghoobin. *Applications of Finite Fields*. Kluwer Academic, 1999.
- [7] I.F. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, London Mathematical Society Lecture Notes Series 265, 1999.

- [8] C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley & Sons, 1985.
- [9] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.
- [10] R. Crandall, K. Dilcher, and C. Pomerance. A Search for Wieferich and Wilson Primes. *Mathematics of Computation*, 66(217):433–449, 1997.
- [11] R. Crandall and C. Pomerance. *Prime Numbers*. Springer-Verlag, New York, NY, USA, 2001.
- [12] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Sov. Phys. Dokl. (English translation)*, 7(7):595–596, 1963.
- [13] Ç. K Koç and T. Acar. Montgomery Multiplication in $GF(2^k)$. *Design, Codes, and Cryptography*, 14(1):57–69, 1998.
- [14] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, Massachusetts, USA, 1983.
- [15] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 2nd edition, 1989.
- [16] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.
- [17] H. J. Nussbaumer. Digital Filtering Using Complex Mersenne Transforms. *IBM Journal of Research and Development*, 20(5), September 1976.
- [18] H. J. Nussbaumer. Digital Filtering Using pseudo Fermat Number Transforms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-25:79–83, February 1977.
- [19] A. V. Oppenheim, R. W. Schaffer, and J. R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 1999.
- [20] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, (Engl. transl.), Institute for Experimental Mathematics, University of Essen, Essen, Germany, June 1994. ISBN 3–18–332810–0.
- [21] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25:365–374, 1971.
- [22] Selçuk Baktır and Berk Sunar. Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography. In *Proceedings of the 21th International Symposium on Computer and Information Sciences (ISCIS 2006)*, volume 4263 of *Lecture Notes in Computer Science (LNCS)*, pages 991–1001, Heidelberg, October 2006. Springer.
- [23] C. M. Rader. Discrete Convolutions via Mersenne Transforms. *IEEE Transactions on Computers*, C-21(12):1269–1273, December 1972.
- [24] C. M. Rader. The Number Theoretic DFT and Exact Discrete Convolution. In *IEEE Arden House Workshop on Digital Signal Processing*, Harriman, New York, January 1972.

- [25] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
- [26] Alexander Skavantzios and Fred J. Taylor. On the Polynomial Residue Number System. *IEEE Transactions on Signal Processing*, 39(2):376–382, February 1991.
- [27] F. J. Taylor. Residue Arithmetic A Tutorial with Examples. *IEEE Computer*, 17(5):50–62, May 1984.
- [28] A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Technical report, Department of Electrical Engineering and Information Sciences, Ruhr-Universität Bochum, Germany.
- [29] S. Winograd. Some Bilinear Forms whose Multiplicative Complexity Depends on the Field of Constants. *Mathematical Systems Theory*, 10:169–180, 1977.

Appendix

Finite Field	$f(x)$		
$GF((2^{13} - 1)^{11})$	$x^{11} + 2^2x^3 + 1,$	$x^{11} + 2^3x^2 + 1,$	$x^{11} + 2^6x^5 + 1$
$GF((2^{13} - 1)^{12})$	$x^{12} + 2^5x^5 + 1,$	$x^{12} + 2^8x^5 + 1,$	$x^{12} + 2^5x^7 + 1$
$GF((2^{13} - 1)^{13})$	$x^{12} \pm 2^{s_0},$	for $1 \leq s_0 \leq 12$	
$GF((2^{17} - 1)^9)$	$x^9 + 2^6x + 1,$	$x^9 + 2^9x + 1,$	$x^9 + 2^{11}x^2 + 1$
$GF((2^{17} - 1)^{11})$	$x^{11} + 2^8x^3 + 1,$	$x^{11} + 2^{13}x^3 + 1,$	$x^{11} + 2^5x^4 + 1$
$GF((2^{17} - 1)^{12})$	$x^{12} + x + 1,$	$x^{12} + x^{11} + 1,$	$x^{12} + 2^9x^5 + 1$
$GF((2^{17} - 1)^{13})$	$x^{13} + 2x + 1,$	$x^{13} + 2^2x + 1,$	$x^{13} + 2x^2 + 1$
$GF((2^{17} - 1)^{14})$	$x^{14} + x^2 + 1,$	$x^{14} + x^6 + 1,$	$x^{14} + x^8 + 1$
$GF((2^{17} - 1)^{15})$	$x^{15} + 2^4x + 1,$	$x^{15} + 2^5x^2 + 1,$	$x^{15} + 2^6x^4 + 1$
$GF((2^{17} - 1)^{16})$	$x^{16} + 2^5x^5 + 1,$	$x^{16} + 2^5x^{11} + 1,$	$x^{16} + 2^{16}x^5 + 2^{16}$
$GF((2^{17} - 1)^{17})$	$x^{17} \pm 2^{s_0},$	for $1 \leq s_0 \leq 16$	
$GF((2^{19} - 1)^{10})$	$x^{10} \pm 2^{s_0},$	for $1 \leq s_0 \leq 18$	
$GF((2^{19} - 1)^{11})$	$x^{11} + 2^9x + 1,$	$x^{11} + 2^{13}x + 1,$	$x^{11} + 2^2x^2 + 1$
$GF((2^{19} - 1)^{12})$	$x^{12} + 2^{11}x + 1,$	$x^{12} + 2^{14}x + 1,$	$x^{12} + 2^{10}x^5 + 1$
$GF((2^{19} - 1)^{13})$	$x^{13} + 2^4x + 1,$	$x^{13} + 2^4x^2 + 1,$	$x^{13} + 2^6x^2 + 1$
$GF((2^{19} - 1)^{14})$	$x^{14} + 2^3x + 1,$	$x^{14} + 2^{17}x + 1,$	$x^{14} + 2^3x^2 + 1$
$GF((2^{19} - 1)^{15})$	$x^{15} + 2^9x + 2^0,$	$x^{15} + 2^{18}x + 1,$	$x^{15} + x^2 + 1$
$GF((2^{19} - 1)^{16})$	$x^{16} + 2^{18}x^3 + 1,$	$x^{16} + 2^{15}x^7 + 1,$	$x^{16} + 2^{15}x^9 + 1$
$GF((2^{19} - 1)^{17})$	$x^{17} + x^3 + 1,$	$x^{17} + x^{14} + 1,$	$x^{17} + 2^5x + 1$
$GF((2^{19} - 1)^{18})$	$x^{18} + x^8 + 1,$	$x^{18} + x^{10} + 1,$	$x^{18} + 2^2x + 1$
$GF((2^{19} - 1)^{19})$	$x^{19} \pm 2^{s_0},$	for $1 \leq s_0 \leq 18$	
$GF((2^{31} - 1)^{11})$	$x^{11} + 2^{20}x + 1,$	$x^{11} + 2^{29}x + 1,$	$x^{11} + 2^5x^4 + 1$
$GF((2^{31} - 1)^{12})$	$x^{12} + 2^{21}x + 1,$	$x^{12} + 2^{23}x + 1,$	$x^{12} + 2^{30}x + 1$
$GF((2^{31} - 1)^{13})$	$x^{13} + 2^8x + 1,$	$x^{13} + 2^{12}x + 1,$	$x^{13} + 2^{15}x + 1$

Table 9: Short list of efficient irreducible polynomials for construction of the finite fields listed in Table 1.

Finite Field	$f(x)$		
$GF\left(\left(\frac{2^{15}-1}{217}\right)^{15}\right)$	$x^{15} + 2^2x^3 + 2^2,$	$x^{15} + 2^3x^2 + 2^3,$	$x^{15} + 2^4x^3 + 2^4$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^9\right)$	$x^9 + 2^2x + 1,$	$x^9 + 2^6x + 1,$	$x^9 + 2^9x + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{10}\right)$	$x^{10} + 2^8x + 1,$	$x^{10} + 2^{11}x^3 + 1,$	$x^{10} + 2^{18}x^3 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{11}\right)$	$x^{11} + x^5 + 1,$	$x^{11} + x^6 + 1,$	$x^{11} + 2^4x + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{12}\right)$	$x^{12} + 2^2x + 1,$	$x^{12} + 2^3x^3 + 1,$	$x^{12} + 2^7x^3 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{17}\right)$	$x^{17} + 2^{15}x + 1,$	$x^{17} + 2^{17}x^3 + 1,$	$x^{17} + 2^5x^4 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{18}\right)$	$x^{18} + 2^9x + 1,$	$x^{18} + 2^{15}x + 1,$	$x^{18} + 2^7x^3 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{19}\right)$	$x^{19} + 2^{16}x^4 + 1,$	$x^{19} + 2^2x^6 + 1,$	$x^{19} + 2^{15}x^8 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{20}\right)$	$x^{20} + 2^{11}x^7 + 1,$	$x^{20} + 2^{17}x^9 + 1,$	$x^{20} + 2^{17}x^{11} + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{21}\right)$	$x^{21} + 2^9x + 1,$	$x^{21} + 2^4x^2 + 1,$	$x^{21} + 2^6x^7 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{22}\right)$	$x^{22} + 2^{17}x^3 + 1,$	$x^{22} + 2^{15}x^5 + 1,$	$x^{22} + 2^5x^7 + 1$
$GF\left(\left(\frac{2^{23}-1}{47}\right)^{23}\right)$	$x^{23} \pm 2^{s_0},$	for $1 \leq s_0 \leq 22$	
$GF\left(\left(\frac{2^{27}-1}{511}\right)^{11}\right)$	$x^{11} + x^3 + 1,$	$x^{11} + x^8 + 1,$	$x^{11} + 2^{25}x + 1$
$GF\left(\left(\frac{2^{27}-1}{511}\right)^{12}\right)$	$x^{12} + 2^4x + 1,$	$x^{12} + 2^{13}x + 1,$	$x^{12} + 2^{22}x + 1$
$GF\left(\left(\frac{2^{27}-1}{511}\right)^{13}\right)$	$x^{13} + 2^5x^3 + 1,$	$x^{13} + 2^7x^3 + 1,$	$x^{13} + 2x^5 + 1$
$GF\left(\left(\frac{2^{27}-1}{511}\right)^{14}\right)$	$x^{14} + 2^{25}x + 1,$	$x^{14} + 2^{14}x^5 + 1,$	$x^{14} + 2^{14}x^9 + 1$
$GF\left(\left(\frac{2^{32}-1}{65535}\right)^{13}\right)$	$x^{13} + 2^5x^3 + 1,$	$x^{13} + 2^{11}x^3 + 1,$	$x^{13} + 2^{15}x^5 + 1$
$GF\left(\left(\frac{2^{32}-1}{65535}\right)^{14}\right)$	$x^{14} + 2^2x^3 + 1,$	$x^{14} + 2^4x^3 + 1,$	$x^{14} + 2^{11}x^5 + 1$
$GF\left(\left(\frac{2^{32}-1}{65535}\right)^{15}\right)$	$x^{15} + 2^9x^4 + 1,$	$x^{15} + 2^5x^7 + 1,$	$x^{15} + 2^5x^8 + 1$
$GF\left(\left(\frac{2^{32}-1}{65535}\right)^{16}\right)$	$x^{16} + 2^{12}x^{11} + 2^{12},$	$x^{16} + x^3 + 2^3,$	$x^{16} + x^{11} + 2^3$
$GF\left(\left(\frac{2^{33}-1}{14329}\right)^{17}\right)$	$x^{17} + 2^8x + 1,$	$x^{17} + 2^9x^2 + 1,$	$x^{17} + 2^{15}x^2 + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{11}\right)$	$x^{11} + 2^{19}x + 1,$	$x^{11} + 2^{31}x + 1,$	$x^{11} + 2^{32}x + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{12}\right)$	$x^{12} + 2^{17}x + 1,$	$x^{12} + 2^{19}x + 1,$	$x^{12} + 2^{25}x + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{13}\right)$	$x^{13} + 2^2x + 1,$	$x^{13} + 2^{24}x + 1,$	$x^{13} + 2^{28}x + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{14}\right)$	$x^{14} + 2^5x + 1,$	$x^{14} + 2^{21}x^3 + 1,$	$x^{14} + 2^{30}x^3 + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{15}\right)$	$x^{15} + 2^7x + 1,$	$x^{15} + 2^{22}x + 1,$	$x^{15} + 2^{27}x + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{16}\right)$	$x^{16} + 2^{22}x + 1,$	$x^{16} + 2^3x^3 + 1,$	$x^{16} + 2^{19}x^3 + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{17}\right)$	$x^{17} + 2^7x + 1,$	$x^{17} + 2^{11}x^2 + 1,$	$x^{17} + 2^{12}x^2 + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{18}\right)$	$x^{18} + x + 1,$	$x^{18} + x^{17} + 1,$	$x^{18} + 2^{15}x + 1$
$GF\left(\left(\frac{2^{37}-1}{223}\right)^{19}\right)$	$x^{19} + x^6 + 1,$	$x^{19} + x^{13} + 1,$	$x^{19} + 2x^3 + 1$

Table 10: Short list of efficient irreducible polynomials for construction of the finite fields listed in Table 2.

Finite Field	$f(x)$		
$GF((2^{2^3} + 1)^{13})$	$x^{13} + 2^8x + 1,$	$x^{13} + 2^{15}x + 1,$	$x^{13} + 2^3x^3 + 1$
$GF((2^{2^3} + 1)^{14})$	$x^{14} + 2^6x + 1,$	$x^{14} + 2^{14}x + 1,$	$x^{14} + 2^{15}x + 1$
$GF((2^{2^3} + 1)^{15})$	$x^{15} + 2^{12}x + 1,$	$x^{15} + 2^{13}x + 1,$	$x^{15} + 2^4x^7 + 1$
$GF((2^{2^3} + 1)^{16})$	$x^{16} + 2^5x^7 + 2^5,$	$x^{16} + 2^{11}x^9 + 2^{11},$	$x^{16} - 2^5x^7 + 2^5$
$GF((2^{2^4} + 1)^7)$	$x^7 + 2^5x + 1,$	$x^7 + 2^{15}x + 1,$	$x^7 + 2^{16}x + 1$
$GF((2^{2^4} + 1)^8)$	$x^8 + x^3 + 1,$	$x^8 + x^5 + 1,$	$x^8 + 2^4x^3 + 1$
$GF((2^{2^4} + 1)^{13})$	$x^{13} + 2^{29}x + 1,$	$x^{13} + 2^5x^3 + 1,$	$x^{13} + 2^{11}x^3 + 1$
$GF((2^{2^4} + 1)^{14})$	$x^{14} + 2^2x^3 + 1,$	$x^{14} + 2^4x^3 + 1,$	$x^{14} + 2^{20}x^3 + 1$
$GF((2^{2^4} + 1)^{15})$	$x^{15} + 2^9x^4 + 1,$	$x^{15} + 2^{29}x^2 + 1,$	$x^{15} + 2^{30}x^7 + 1$
$GF((2^{2^4} + 1)^{16})$	$x^{16} + 2^{27}x^7 + 2^{27},$	$x^{16} + 2^{29}x^5 + 2^{29},$	$x^{16} + 2^{29}x^{13} + 2^{29}$
$GF((2^{2^4} + 1)^{29})$	$x^{29} + x^{11} + 1,$	$x^{29} + x^{18} + 1,$	$x^{29} + 2^8x^2 + 1$
$GF((2^{2^4} + 1)^{30})$	$x^{30} + 2^7x^7 + 1,$	$x^{30} + 2^{23}x^7 + 1,$	$x^{30} + 2^{30}x^9 + 1$
$GF((2^{2^4} + 1)^{31})$	$x^{31} + 2^{30}x^3 + 1,$	$x^{31} + 2^{31}x^3 + 1,$	$x^{31} + 2^{23}x + 1$
$GF((2^{2^4} + 1)^{32})$	$x^{32} + 2^{10}x + 2^{10},$	$x^{32} + 2^{25}x^5 + 2^{25},$	$x^{32} + 2^{27}x^5 + 2^{27}$

Table 11: Short list of efficient irreducible polynomials for construction of the finite fields listed in Table 3.

Finite Field	$f(x)$		
$GF((\frac{2^{13}+1}{3})^{13})$	$x^{13} \pm 2^{s_0},$	for $1 \leq s_0 \leq 12$	
$GF((\frac{2^{15}+1}{99})^{15})$	$x^{15} + 2,$	$x^{15} + 2^2,$	$x^{15} + 2^4$
$GF((\frac{2^{17}+1}{3})^{17})$	$x^{17} \pm 2^{s_0},$	for $1 \leq s_0 \leq 16$	
$GF((\frac{2^{19}+1}{3})^{19})$	$x^{19} \pm 2^{s_0},$	for $1 \leq s_0 \leq 18$	
$GF((\frac{2^{19}+1}{3})^{10})$	$x^{10} + 2^5x + 1,$	$x^{10} + 2^{24}x + 1,$	$x^{10} + 2x^2 + 1$
$GF((\frac{2^{20}+1}{17})^{20})$	$x^{20} + 2^2x + 2^2,$	$x^{20} + 2^{16}x + 2^{16},$	$x^{20} + 2^2x^5 + 2^2$
$GF((\frac{2^{20}+1}{17})^{10})$	$x^{10} + 2^7x + 2^7,$	$x^{10} + 2^{15}x + 2^{15},$	$x^{10} + 2x^3 + 2$
$GF((\frac{2^{21}+1}{387})^{21})$	$x^{21} + 2^3x + 2^3,$	$x^{21} + 2x^7 + 2,$	$x^{21} + 2^3x + 2^3$
$GF((\frac{2^{22}+1}{1985})^{22})$	$x^{22} + 2^6x + 2^6,$	$x^{22} + 2^{24}x + 2^{24},$	$x^{22} + 2^{23}x^3 + 2^{23}$
$GF((\frac{2^{22}+1}{1985})^{11})$	$x^{11} \pm 2^{s_0},$	for $1 \leq s_0 \leq 21$	
$GF((\frac{2^{23}+1}{3})^{23})$	$x^{23} \pm 2^{s_0},$	for $1 \leq s_0 \leq 22$	
$GF((\frac{2^{27}+1}{1539})^{27})$	$x^{27} \pm 2^{s_0},$	for $1 \leq s_0 \leq 8$	
$GF((\frac{2^{27}+1}{1539})^9)$	$x^9 \pm 2^{s_0},$	for $\gcd(s_0, 3) = 1$	
$GF((\frac{2^{28}+1}{17})^{14})$	$x^{14} + 2^6x + 2^6,$	$x^{14} + 2^{11}x + 2^{11},$	$x^{14} + 2^{16}x + 2^{16}$
$GF((\frac{2^{28}+1}{17})^7)$	$x^7 \pm 2^{s_0},$	for $\gcd(s_0, 7) = 1$	
$GF((\frac{2^{32}+1}{641})^{16})$	$x^{16} + 2^2x + 2^2,$	$x^{16} + 2^{11}x + 2^{11},$	$x^{16} + 2^{35}x + 2^{35}$
$GF((\frac{2^{32}+1}{641})^8)$	$x^8 + 2^5x + 2^5,$	$x^8 + 2^6x + 2^6,$	$x^8 + 2^{18}x + 2^{18}$

Table 12: Short list of efficient irreducible polynomials for construction of the finite fields listed in Table 4.