Acta Appl Math
DOI 10.1007/s10440-006-9046-1

Efficient Software-Implementation of Finite Fields with Applications to Cryptography

Jorge Guajardo · Sandeep S. Kumar · Christof Paar · Jan Pelzl

Abstract In this work, we present a survey of efficient techniques for software implementation of finite field arithmetic especially suitable for cryptographic applications. We discuss different algorithms for three types of finite fields and their special versions popularly used in cryptography: Binary fields, prime fields and extension fields. Implementation details of the algorithms for field addition/subtraction, field multiplication, field reduction and field inversion for each of these fields are discussed in detail. The efficiency of these different algorithms depends largely on the underlying micro-processor architecture. Therefore, a careful choice of the appropriate set of algorithms has to be made for a software implementation depending on the performance requirements and available resources.

Mathematics Subject Classifications 12-02 · 12E30 · 12E10

Key words field arithmetic · cryptography · efficient implementation · binary field arithmetic · prime field arithmetic · extension field arithmetic · optimal extension fields

J. Guajardo (✉)
Information and System Security Department, Philips Research, Eindhoven, The Netherlands
e-mail: Jorge.Guajardo@philips.com

S. S. Kumar · C. Paar · J. Pelzl
Horst-Görtz Institute for IT-Security, Ruhr-University Bochum, Bochum, Germany

S. S. Kumar
e-mail: kumar@crypto.rub.de

C. Paar
e-mail: cpaar@crypto.rub.de

J. Pelzl
e-mail: pelzl@crypto.rub.de

1. Introduction

Finite field (or Galois field) and finite ring arithmetic are an integral part of many cryptographic algorithms. The main application domain is asymmetric algorithms (also known as public-key algorithms), for instance algorithms based on the *Discrete Logarithm (DL)* problem (the Diffie–Hellman (DH) key exchange protocol [16] and the Digital Signature Algorithm (DSA) [36]), and in the group operations for *Elliptic Curve Cryptography (ECC)* and *Hyperelliptic Curve Cryptography (HECC)*. A second application domain for finite fields in cryptography are inversions in small fields which occur in the context of block ciphers, e.g., within the S-box of the *Advanced Encryption Standard (AES)*.

Though various efficient algorithms exist for finite field arithmetic for signal processing applications, the algorithms suitable for practical cryptographic implementations vary due to the relatively large finite field structures used in asymmetric cryptographic algorithms. For cryptographic applications, three different types of finite fields and their special versions are most popular (as shown in Figure 1):

– Binary fields $\mathbb{F}_{2^m}$,
– Prime fields $\mathbb{F}_p$,
– Extension fields $\mathbb{F}_{p^m}$

The security considerations determine the size of the finite fields used, and restrictions on the field parameters for each cryptographic application. Table 1 provides the appropriate field sizes (in bit-length) used for different applications. *Elliptic Curve Cryptography (ECC)* is, however, restricted to prime fields, binary fields, and extension fields with only prime extensions because of the Weil descent attack on the composite extensions [20]. There are also other restrictions on the binary and extension fields based on Weil and Tate pairing attacks [23]. Also special fields are generally chosen to allow a faster implementation of the algorithms. For prime fields, normally generalized-Mersenne primes are chosen. For prime extensions fields, a special class of fields called *Optimal Extension Fields (OEFs)* [2] are chosen which allows fast implementation in software.

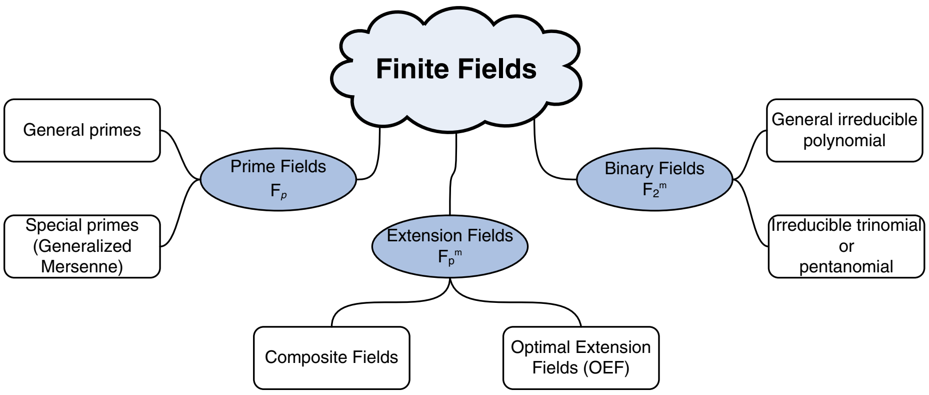

Finite Fields
General primes
Special primes (Generalized Mersenne)
Prime Fields F_p
Extension Fields F_{p^m}
Binary Fields F_{2^m}
General irreducible polynomial
Irreducible trinomial or pentanomial
Composite Fields
Optimal Extension Fields (OEF)

Figure 1 Finite field choices for cryptography.

Table 1 Bit-length of finite fields used for different applications

Application	Prime fields	Binary fields	Extension fields
ECC	160–512	160–512	160–512
HECC	40–256	40–256	40–256
DL (DH, DSA)	1,024–4,096	1,024–4,096	1,024–4,096
Block ciphers (AES)	–	8	4
RS codes	–	8–16	–
Signal processing	–	8–16	–

The paper presents different finite field arithmetic algorithms suitable for the cryptographic implementations in software on a standard or embedded microprocessors. This paper is organized as follows: Section 2 introduces the notion of fields and elements used in this paper. In Sections 3, 4, and 5 we describe software implementation techniques for $\mathbb{F}_{2^m}$, $\mathbb{F}_p$, and $\mathbb{F}_{p^m}$, respectively. Section 6 concludes this contribution.

2. Basis Representation

For the discussion that follows, it is important to point out that there are several possibilities to represent elements of a finite field. The *standard polynomial basis* representation is used mainly in this paper. In general, given an irreducible polynomial

$$F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$$

where $g_i \in \mathbb{F}_p$. Let α be a root of $F(x)$, then one can represent an element $A \in \mathbb{F}_{p^m}$, p prime, as a polynomial in α as

$$A(\alpha) = a_{m-1}\alpha^{m-1} + a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0, \quad a_i \in \mathbb{F}_p. \tag{1}$$

The set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is then said to be a polynomial basis (or standard basis) for the finite field $\mathbb{F}_{p^m}$ over $\mathbb{F}_p$.

Another type of basis is called a *normal basis*. Normal bases are of the form $\{\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}}\}$ for an appropriate element $\beta \in \mathbb{F}_{p^m}$. Then, an element $B \in \mathbb{F}_{p^m}$ can be represented as:

$$B(\beta) = b_{m-1}\beta^{p^{m-1}} + b_{m-2}\beta^{p^{m-2}} + \cdots + b_1\beta^p + b_0\beta, \quad b_i \in \mathbb{F}_p. \tag{2}$$

It can be shown that for any field $\mathbb{F}_p$ and any extension field $\mathbb{F}_{p^m}$, there exists always a normal basis of $\mathbb{F}_{p^m}$ over $\mathbb{F}_p$ (see [30, Theorem 2.35]).

3. Software Implementation Techniques Over $\mathbb{F}_{2^m}$

In $\mathbb{F}_{2^m}$, the polynomial in standard polynomial basis (Equation (1)) consists of bit coefficients and therefore can also be represented as a bit vector: $(a_{m-1}, ...a_1, a_0)$. In software, this bit string is grouped into $s = \lceil \frac{m}{w} \rceil$ words where w is the word-length of the processor (w is normally 8, 16, 32 or 64). Thus we can represent

$$A(\alpha) = \sum_{i=0}^{s-1} A_i \alpha^{wi}$$
$$\text{where } A_i = \begin{cases} (a_{wi+(w-1)} \cdots a_{wi+1} a_{wi}) & \text{for } i \in \{0, \cdots, (s-2)\} \\ \\ (0...0 a_{m-1} a_{m-2} \cdots a_{w(s-1)}) & \text{for } i = s-1 \end{cases} \tag{3}$$

A is then stored as a word string $(A_{s-1}...A_1 A_0)$ in the processor memory as shown in Figure 2.

The $\mathbb{F}_{2^m}$ field arithmetic is implemented as polynomial arithmetic modulo $F(x)$. Notice that by assumption $F(\alpha) = 0$ since α is a root of $F(x)$. Therefore,

$$\alpha^m = -G(\alpha) = \sum_{i=0}^{m-1} g_i \alpha^i \quad , \quad g_i \in \mathbb{F}_2 \tag{4}$$

gives an easy way to perform modulo reduction whenever we encounter powers of α greater than $m-1$. Throughout the text, we will write $A \bmod F(\alpha)$ to mean *explicitly* the reduction step. Normally $F(x)$ is chosen with least number of coefficients g_i to minimize the complexity of the arithmetic operations.

3.1. Addition

$\mathbb{F}_{2^m}$ addition is the simplest of all operations, since it is a bitwise addition in $\mathbb{F}_2$ which maps to an XOR operation ($\oplus$) over the words in software:

$$\begin{aligned} C &\equiv A + B \bmod F(\alpha) \\ &\equiv (A_{s-1} \oplus B_{s-1})\alpha^{w(s-1)} + \ldots + (A_1 \oplus B_1)\alpha^w + (A_0 \oplus B_0) \end{aligned}$$

Such a word level XOR operation is widely available in most micro-processors. No reduction is required, as the size of the polynomial does not exceed $m-1$ after this operation. The carry free addition makes this operation much more efficient to implement compared to $\mathbb{F}_p$ addition.

3.2. Multiplication

The multiplication of two elements $A, B \in \mathbb{F}_{2^m}$, with $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$ and $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i$ is given as

$$C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i \equiv A(\alpha) \cdot B(\alpha) \bmod F(\alpha) \tag{5}$$

where the multiplication is a polynomial multiplication, and all α^t, with $t \geq m$ are reduced with Equation (4).

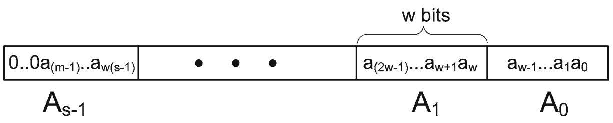


Figure 2 Representation of binary field element in software.

The simplest algorithm for field multiplication is the shift-and-add method [28] with the reduction step inter-leaved (Algorithm 1).

Algorithm 1. Shift-and-Add Most Significant Bit (MSB) first $\mathbb{F}_{2^m}$ multiplication

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$ where $a_i, b_i \in \mathbb{F}_2$.
Output: $C \equiv A \cdot B \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$ where $c_i \in \mathbb{F}_2$.
  1 : $C \leftarrow 0$
  2 : for $i = m-1$ downto 0 do
  3 :    $C \leftarrow C + b_i \cdot (\sum_{i=0}^{m-1} a_i\alpha^i)$
  4 :    $C \leftarrow (\sum_{i=0}^{m-1} c_i\alpha^i) \cdot \alpha \bmod F(\alpha)$
  5 : end for
  6 : Return $(C)$
```

The shift-and-add method is not particularly suitable for software implementations as the bitwise shifts (in Step 4, Algorithm 1) are hard to implement across the words on a processor. A more efficient method for implementing the multiplier in software is the Comb method [31]. Here the multiplication is implemented efficiently in two separate steps, first performing the polynomial multiplication to obtain $2n$-bit length polynomial and then reducing it using special reduction polynomials.

Algorithm 2 shows the polynomial multiplication using the Comb method. The operation $\text{SHIFT}(A << k) = \sum_{i=0}^{m-1} a_i\alpha^{(i+k)}$, performs a k-bit shift across the words without reduction. Comb method is more efficient in software because $\text{SHIFT}(A << w.i)$ where w is the word-length of the processor, are the same original set of bytes referenced with a different memory pointer and therefore requires no actual shifts.

Algorithm 2. Comb method polynomial multiplication on a w-bit processor.

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$ where $a_i, b_i \in \mathbb{F}_2$ and $s = \lceil \frac{m}{w} \rceil$.
Output: $C = A \cdot B = \sum_{i=0}^{2m-2} c_i\alpha^i$, where $c_i \in \mathbb{F}_2$
  1 : $C \leftarrow 0$
  2 : for $j = 0$ to $w-1$ do
  3 :    for $i = 0$ to $s-1$ do
  4 :       $C \leftarrow b_{wi+j} \cdot \text{SHIFT}(A << w.i) + C$
  5 :    end for
  6 :    $A \leftarrow \text{SHIFT}(A << 1)$
  7 : end for
  8 : Return $(C)$
```

3.3. Squaring

Field Squaring is much simpler in $\mathbb{F}_{2^m}$ when represented in polynomial basis as shown here:

$$\begin{aligned} C &\equiv A^2 \bmod F(\alpha) \\ &\equiv (a_{m-1}\alpha^{2(m-1)} + a_{m-2}\alpha^{2(m-2)} + \ldots + a_1\alpha^2 + a_0) \bmod F(\alpha) \end{aligned} \tag{6}$$

Polynomial squaring is implemented by expanding C to double its bit-length by interleaving 0 bits in between the original bits of C and then reducing the double length result. The interleaving step is most efficiently implemented in software using a precomputed table as shown in Algorithm 3. The size of the precomputed table can be chosen based on the available memory and performance requirements. Normally a precomputed table of all possible k bit expansion to $2k$ bits is used, where w is a multiple of k.

Algorithm 3. Polynomial squaring in software for a w bit processor

```
Input: $A = \sum_{i=0}^{s-1} A_i\alpha^{wi}$.
Output: $C = A^2 = \sum_{i=0}^{2s-1} C_i\alpha^{wi}$.
1 : Pre-compute: For each possible $(d_k, \cdots, d_1, d_0)$, compute $2k$ bit quantity
    $T(d) = (0, d_k, \cdots, 0, d_1, 0, d_0)$.
2 : for $i = 0$ downto $s - 1$ do
3 :   Let $A_i = (u_{\frac{w}{k}}, \cdots, u_1)$ where each $u_j$ are $k$ bits.
4 :   $C_{2i} \leftarrow (T(u_{\frac{w}{2k}}), \cdots, T(u_1))$
5 :   $C_{2i+1} \leftarrow (T(u_{\frac{w}{k}}), \cdots, T(u_{\frac{w}{k}+1}))$
6 : end for
7 : Return $(C)$
```

3.4. Field Reduction

Field reduction of a $(2n-1)$ bit size polynomial $\acute{C}(\alpha)$ using the Equation (4) can be viewed as a linear mapping of the $2n-1$ coefficients of $\acute{C}(\alpha)$ into the reduced n coefficient polynomial $C(\alpha)$ represented as:

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} \acute{c}_0 \\ \acute{c}_1 \\ \vdots \\ \acute{c}_{n-1} \end{pmatrix} + \begin{pmatrix} r_{0,0} & \cdots & r_{0,n-2} \\ r_{1,0} & \cdots & r_{1,n-2} \\ \vdots & \ddots & \vdots \\ r_{n-1,0} & \cdots & r_{n-1,n-2} \end{pmatrix} \begin{pmatrix} \acute{c}_n \\ \acute{c}_{n+1} \\ \vdots \\ \acute{c}_{2n-2} \end{pmatrix}$$

where $r_{i,j} \in \mathbb{F}_2$ depends on the reduction polynomial $F(x)$. Hence, the reduction step consists of simple $\mathbb{F}_2$ additions. Therefore, choosing an appropriate reduction polynomial $F(x)$, like a trinomial or pentanomial, can reduce the complexity of this operation i.e.,

$$F(x) = x^m + x^k + 1$$
or
$$F(x) = x^m + x^j + x^k + x^l + 1$$

Such polynomials are widely recommended in all the major standards [1, 25, 36]. For software implementation, reduction polynomials with the middle terms close to each other are more suitable. Implementation techniques using different reduction polynomials can be found in [7]. We present here as an example, the reduction of $C(\alpha)$ with degree at most 160 in the field $\mathbb{F}_{2^{81}}$ with irreducible polynomial $F(x) = x^{81} + x^4 + 1$ on a 32 bit processor. Consider the sixth word C_5 of $C(\alpha)$:

$$\begin{aligned} x^{160} &\equiv x^{83} + x^{79} \bmod f(x) \\ x^{161} &\equiv x^{84} + x^{80} \bmod f(x) \\ &\vdots \\ x^{191} &\equiv x^{115} + x^{111} \bmod f(x) \end{aligned}$$

By considering columns on the right side of the above congruence, it follows that reduction of C_5 can be performed by adding C_5 twice to C, with the rightmost bit of C_5 added to bits 83 and 79 of C. This leads to Algorithm 4 for modular reduction which can easily be extended to other reduction polynomials and different degrees.

Algorithm 4. Modular reduction, one word at a time

```
Input: A binary polynomial $C(\alpha)$ of degree at most 160.
Output: $C(\alpha)$ mod $F(\alpha)$, where $F(x) = x^{81} + x^4 + 1$.
1 : for $i = 5$ downto 4 do
2 :     $T \leftarrow (C_i << 31) \oplus (C_{i-1} >> 1)$
3 :     $C_{i-3} \leftarrow C_{i-3} \oplus (T >> 12) \oplus (T >> 16)$
4 :     $C_{i-4} \leftarrow C_{i-4} \oplus (T << 20) \oplus (T << 16)$
5 : end for
6 : $T \leftarrow (C_3 << 31) \oplus ((C_2$ AND 0xFFFE0000$) >> 1)$
7 : $C_0 \leftarrow C_0 \oplus (T >> 12) \oplus (T >> 16)$
8 : $C_2 \leftarrow C_2$ AND 0x0001FFFF   {Clear unused bits of $C_2$}
9 : Return $(C = (C_2, C_1, C_0))$
```

3.5. Inversion

In the following, we will discuss practical relevant methods to compute multiplicative inverses in finite fields in software: The *Binary Extended Euclidean Algorithm*, the *Almost Inverse Algorithm*, *Fermat's little theorem*, and *look-up tables*.

Binary Extended Euclidean Algorithm (BEA). Algorithm 5 shows the binary variant of the *Extended Euclidean Algorithm* (*EEA*, cf. [32, Algorithm 2.221]). The main difference to the EEA is that the BEA clears bits of u and v from right to left and uses only divisions by 2 which can be implemented by simple shifts by one.

Algorithm 5. BEA for Inversion in $\mathbb{F}_{2^m}$ ([24])

```
Input: $A \in \mathbb{F}_{2^m}$,   $A \neq 0$.
Output: $A^{-1} \bmod f(x)$.
 1 : $b \leftarrow 1,\ c \leftarrow 0,\ u \leftarrow A,\ v \leftarrow f.$
 2 : while $x$ divides $u$ do
 3 :   $u \leftarrow u/x.$
 4 :   if $x$ divides $b$ then
 5 :     $b \leftarrow b/x.$
 6 :   else
 7 :     $b \leftarrow (b+f)/x.$
 8 :   end if
 9 : end while
10 : if $u = 1$ then
11 :   return $b$
12 : end if
13 : if $\deg(v) < \deg(v)$ then
14 :   $u \leftrightarrow v,\ b \leftrightarrow c.$
15 : end if
16 : $u \leftarrow u + v,\ b \leftarrow b + c.$
17 : goto step 2
```

The BEA maintains the invariants $ba + df = u$ and $ca + ef = v$ for some d and e which are not explicitly computed. The algorithm terminates when $\deg(u) = 0$, in which case $u = 1$ and $ba + df = 1$; hence $b = a^{-1} \bmod f(x)$.

Remark. A speed-up of the algorithm can be obtained by avoiding the change of variables in step 14 by using two different subroutines for each case [41].

Almost Inverse Algorithm (AIA). Algorithm 6 is a modification of the binary Euclidean algorithm and computes $A^{-1}x^k \bmod f$ as an intermediate result. The inverse A^{-1} is finally obtained by the reduction of x^k (step 6). With a suitable polynomial $f(x)$, the reduction can be performed efficiently (see, e.g., [19]). For such polynomials, AIA outperforms BEA since the polynomials b and c grow more slowly in AIA.

Algorithm 6. AIA for Inversion in $\mathbb{F}_{2^m}$ ([40])

```
Input: $A \in \mathbb{F}_{2^m},\ A \neq 0$.
Output: $A^{-1} \bmod f(x)$.
 1 : $b \leftarrow 1,\ c \leftarrow 0,\ u \leftarrow A,\ v \leftarrow f, k \leftarrow 0$.
 2 : while $x$ divides $u$ do
 3 :     $u \leftarrow u/x,\ c \leftarrow x \cdot c,\ k \leftarrow k+1$.
 4 : end while
 5 : if $u = 1$ then
 6 :     return $b \cdot x^{-k}$
 7 : end if
 8 : if $deg(v) < deg(v)$ then
 9 :     $u \leftrightarrow v,\ b \leftrightarrow c$.
10 : end if
11 : $u \leftarrow u + v,\ b \leftarrow b + c$.
12 : goto step 2
```

Fermat's Little Theorem. This method has a higher computational complexity than the Euclidean algorithms but can, nevertheless, be relevant in certain situations, e.g., if a fast exponentiation unit is available or if an algorithm with a simple control structure is desired. From Fermat's little theorem it follows immediately that for any element $A \in \mathbb{F}_{2^m}$, $A \neq 0$, the inverse can be computed as $A^{-1} = A^{(2^m-2)}$. In this case, the use of addition chains allows to dramatically reduce the number of multiplications (though not the number of squarings) required for computing the exponentiation to the $(2^m - 2)$th power. One such efficient method is the *Itoh–Tsujii Inversion* (cf. Section 5).

Look-up Tables. A conceptually simple method is based on look-up tables. In this case, the inverses of all field elements are precomputed once with one of the methods mentioned above, and stored in a table. Assuming the table entries can be accessed by an appropriate method, e.g., by the field elements themselves in a binary representation, the inverses are available quickly. The drawback of this method are the storage requirements, since 2^m memory locations are needed for fields $\mathbb{F}_{2^m}$. Since the storage requirements are too large for the finite fields commonly needed in public-key cryptography, inversion based on look-up tables is mainly useful in cases of small finite fields, e.g., $\mathbb{F}_{2^8}$, which have applications in block ciphers, e.g. AES, or which are subfields of larger extension fields.

4. Software Implementation Techniques Over $\mathbb{F}_p$

Arithmetic with odd primes continues to be the most widely used form of finite field in cryptography. In software, in particular, this might be due to the widespread existence of integer multipliers in general purpose and embedded processors and to the lack of support (or inherent inefficiency) for operations typically needed in fields of characteristic two, such as bitwise operations. In addition, there seems to be a belief that such fields are inherently stronger (from the cryptographic point of view) than their counterparts in characteristic two.[1] We also notice that, although the methods described in this section focus on finite field arithmetic and, therefore, assume a prime modulus, many of these techniques can also be applied to other non-prime moduli and, in particular, to RSA-type moduli, probably the most widely used type of moduli in practical applications today. The literature covering the material in this section is substantial and there are several textbooks covering the methods described here. We refer the reader for example to [28, 32, 43]. Also see [6] for a comprehensive list of references and a more mathematical treatment.

We begin this section by briefly describing multi-precision integer arithmetic as this is the basis of many modulo arithmetic algorithms. Then, we describe naive methods for modular reduction, specialized algorithms for reduction modulo general primes (moduli), and finally techniques for moduli of special form.

Notation. We will refer to multi-precision integers with capital letters and to their digits in radix-b representation with lower-case letters. For example, we would write an n-digit integer in radix b as $X = \sum_{i=0}^{n-1} x_i b^i$ with $b \geq 2$ and $0 \leq x_i < b$. For purposes of multi-precision integer arithmetic, integers can be represented in signed magnitude representation, i.e., just add an additional bit that indicates whether the integer is positive or negative or two's complement representation [32, Section 14.2.1].

4.1. Integer Multi-precision Arithmetic

Addition and Subtraction. Addition and subtraction of multi-precision integers is performed via the method taught in primary school: Add (subtract) the digits in the integers and keep track of carries. Addition and subtraction routines are depicted in Algorithms 7 and 8.

[1] For example, the Weil descent [20] attack which renders most binary fields of composite degree unsuitable for cryptographic applications, does not have yet a counterpart in the prime field case.

Algorithm 7. Multi-precision Addition

```
Input: $X = \sum_{i=0}^{n-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$
Output: $Z = X + Y = \sum_{i=0}^{n} z_i b^i$
 1 : $c \leftarrow 0$
 2 : for $i = 0$ to $n - 1$do
 3 :     $z_i \leftarrow x_i + y_i + c \bmod b$
 4 :     if $x_i + y_i + c \geq b$ then
 5 :         $c \leftarrow 1$
 6 :     else
 7 :         $c \leftarrow 0$
 8 :     end if
 9 :     $z_n \leftarrow c$
10 : end for
11 : Return($Z$)
```

Also in Algorithms 7 and 8, the value of the radix b is usually chosen as to be compatible with the underlying hardware. For example, on a 32-bit architecture, b would be chosen to be 2^{32}. Notice also that certain processors have hardware support for the operation of adding with carry shown in Step 7 of Algorithm 7.

Algorithm 8. Multi-precision Subtraction

```
Input: $X = \sum_{i=0}^{n-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$, $X > Y$
Output: $Z = X - Y = \sum_{i=0}^{n-1} z_i b^i$
 1 : $c \leftarrow 0$
 2 : for $i = 0$ to $n - 1$ then
 3 :     $z_i \leftarrow x_i - y_i + c \bmod b$
 4 :     if $x_i - y_i + c < 0$ then
 5 :         $c \leftarrow -1$
 6 :     else
 7 :         $c \leftarrow 0$
 8 :     end if
 9 : end for
10 : Return($Z$)
```

Algorithm 8 requires that $X > Y$ which is often not known in advance. In general, to avoid this requirement, if the value of c equals -1 at the end of the computation, one can perform the operation again with swapped operands. Alternatively, the implementer can choose for two's complement representation, which inherently takes care of the sign problem.

School Book Method of Multiplication. A multiplication algorithm can be derived by observing that for X and $Y = \sum_{i=0}^{n-1} y_i b^i$, the product $X \cdot Y$ can be written as:

$$X \cdot Y = \sum_{i=0}^{n-1} (X \cdot y_i) b^i = b(\cdots (b(0 + X \cdot y_{n-1}) + X \cdot y_{n-2}) + \cdots) + X \cdot y_0 \tag{7}$$

The multiplication of each term $X \cdot y_i$ can be similarly unrolled resulting in Algorithm 9. Algorithm 9 requires in Step 5, a digit multiplication $(x_j \cdot y_i)$ and two additions, the result of which can be proven to fit in two base-b digits. We also point out that if instead of initializing Z to 0 in Step 1 of Algorithm 9, we initialize it to $A \cdot 2^{-(n-1)}$, then, the final result will be $Z = X \cdot Y + A$. This operation is useful in certain applications and it is obtained at virtually no extra cost. Finally, notice that normally the base b is chosen so that the hardware multiplier in the processor can multiply two base-b digits and generate a double precision result.

Algorithm 9. Multi-precision Multiplication

```
Input: $X = \sum_{i=0}^{m-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$
Output: $Z = X \cdot Y = \sum_{i=0}^{m+n-1} z_i b^i$
 1 : $Z \leftarrow 0$
 2 : for $i = 0$ to $n - 1$ do
 3 :     $c \leftarrow 0$
 4 : for $j = 0$ to $m - 1$ do
 5 :         $(u, v)_b \leftarrow z_{i+j} + x_j \cdot y_i + c$
 6 :         $z_{i+j} \leftarrow v, \quad c \leftarrow u$
 7 :     end for
 8 : end for
 9 : $z_{n+m-1} \leftarrow c$
10 : Return($Z$)
```

Squaring is a special case of multiplication and it can be shown to require half the number of single-precision multiplications that regular multiplication requires. The traditional squaring algorithm presented in [32, Chapter 14] requires an intermediate value that is three single-precision digits in radix-b representation. To accommodate for this inconvenience, [21] proposes modifications which only require a double precision register as in the case of multiplication (see also [10] for a discussion about the relative performance of the methods). In general, multiplication should not be more than twice as fast as squaring thanks to the following identity:

$$X \cdot Y = \frac{(X+Y)^2 - (X-Y)^2}{4}$$

Karatsuba Multiplication. The Karatsuba multiplication algorithm was originally introduced in [27]. In [27], it was credited to Karatsuba alone who was the first to observe that multiplication of large integers could be done in complexity less than $\mathcal{O}(n^2)$. The basic idea is as follows. Given two integers $X = \sum_{i=0}^{n-1} x_i b^i$ and $Y = \sum_{i=0}^{n-1} y_i b^i$, define $m = n/2$ where it is assumed that n is even (if not simply pad the integer with zeros to the left until n is even). Then, we can write X and Y as:

$$X = \sum_{i=0}^{n-1} x_i b^i = X_H b^m + X_L$$

$$Y = \sum_{i=0}^{n-1} y_i b^i = Y_H b^m + Y_L$$

where $X_L = \sum_{i=0}^{m-1} x_i b^i$, $X_H = \sum_{i=m}^{n-1} x_i b^{i-m}$ and similarly for Y. Then we can compute $Z = X \cdot Y$ as

$$Z = D_0 + (D_1 - D_0 - D_2)\, b^m + D_2 b^{2m}$$

where

$$\begin{aligned} D_0 &= X_L \cdot Y_L \\ D_1 &= (X_L + X_H) \cdot (Y_L + Y_H) \\ D_2 &= X_H \cdot Y_H \end{aligned}$$

Thus one can multiply X times Y with three products of $n/2$ radix-b integers instead of four. Two sums and two differences of integers of size $n/2$ are also needed. This algorithm if run in a recursive manner has complexity $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.5849})$. Reference [6], attributes to [28] the following variant of the Karatsuba algorithm:

$$Z = D_0 + (D_0 + D_2 - D_1)\, b^m + D_2 b^{2m}$$

with $D_1 = (X_L - X_H) \cdot (Y_L - Y_H)$. The Karatsuba algorithm can be easily generalized to polynomials (simply interpret X and Y as polynomials, the only difference being that during addition no carries will be generated).

Division. Division can be fully characterized by the basic division equation

$$X = Q \cdot M + R, \qquad 0 \le R < M \tag{8}$$

where X is the dividend, M is the divisor, Q is the quotient, and R is the residue. Just like multiplication can be expressed in terms of repeated additions and shifts, division can be written in terms of subtractions and shifts. However, in practice, division is not used in the finite field setting as there are more efficient methods to compute the remainder (i.e. the modulo operation) or to compute the inverse of an element in the finite field (through the extended Euclidean algorithm). Thus, we do not describe division any further in this paper and refer the interested reader to [32, Chapter 14].

4.2. Basic Modular Arithmetic Methods

Addition and Subtraction. Field addition over $\mathbb{F}_p$ is performed using multi-precision integer addition followed by a reduction if required as shown in Algorithm 10. Similarly subtraction in $\mathbb{F}_p$ is a multi-precision integer subtraction followed by an additional addition with p if the result is negative (Algorithm 11).

Algorithm 10. Addition in $\mathbb{F}_p$

```
Input: $A, B \in \mathbb{F}_p$.
Output: $C \equiv A + B \bmod p$.
1: $C \leftarrow A + B$   {multi-precision integer addition}
2: if $C \geq p$ then
3:     $C \leftarrow C - p$
4: end if
5: Return($C$)
```

Algorithm 11. Subtraction in $\mathbb{F}_p$

```
Input: $A, B \in \mathbb{F}_p$.
Output: $C \equiv A - B \bmod p$.
1: $C \leftarrow A - B$   {multi-precision integer subtraction}
2: if $C < 0$ then
3:     $C \leftarrow C + p$
4: end if
5: Return($C$)
```

School-book Method for Modular Multiplication. Modular multiplication and squaring can be done by first performing a multi-precision integer multiplication or squaring, respectively, and then reducing the double bit-length result modulo p. The most naive method to perform modular multiplication is known as the multiply first and then divide method. In other words, to compute the product $Z = X \cdot Y \bmod p$, one first computes $Z' = X \cdot Y$ and then computes

$$Z = Z' \bmod p = Z' - \lfloor Z'/p \rfloor p. \tag{9}$$

Interleaved Multiplication Reduction Method. The details of this method are sketched in [8, 45]. The method is based on combining (7) with modular reduction and making use of the distributivity property of modular reduction. Thus, Equation (7) becomes:

$$\begin{aligned} X \cdot Y \bmod p &= \sum_{i=0}^{n-1} (X \cdot y_i) b^i \bmod p \\ &= b(\cdots (b(0 + X \cdot y_{n-1} \bmod p) + X \cdot y_{n-2} \bmod p) \\ &\quad + \cdots) + X \cdot y_0 \bmod p \end{aligned} \tag{10}$$

Algorithm 12. Interleaved Multiplication Reduction Method

```
Input: $X, M, Y = \sum_{i=0}^{n-1} y_i 2^i$ with $X, Y < p$
Output: $Z = X \cdot Y \bmod p$
1 : $Z \leftarrow 0$
2 : for $i = 0$ to $n - 1$ do
3 :     $Z \leftarrow b \cdot Z + X \cdot y_{n-1-i}$
4 :     $Z \leftarrow Z \bmod b$
5 : end for
6 : Return($Z$)
```

Algorithm 12 follows easily from (10). We notice that restricting $X, Y < p$ does not have any practical impact as in most cryptographic applications it is a requirement. For the case $b = 2$, since $Z, X, Y < p$ at the beginning of every loop iteration i, the Z in Step 3 of Algorithm 12 is

$$Z^{(i)} = 2 \cdot Z^{(i-1)} + X \cdot y_{n-1-i} \leq 2(M - 1) + (M - 1) \leq 3M - 3$$

Thus, in Step 4 we need to subtract p at most twice from Z to obtain $Z \bmod p$. The case where $b > 2$, is somewhat more complicated in that it would require more than two subtractions to obtain the result modulo p. Thus, alternative methods are required to perform the modular reduction. This is the subject of the next section.

4.3. General Moduli Algorithms

Barret Modular Reduction. Barret reduction was originally introduced in [4], in the context of implementing RSA on a DSP processor. Algorithm 13 summarizes

Algorithm 13. Barret Modular Reduction

```
Input: $X = \sum_{i=0}^{2n-1} x_i b^i$, $p = \sum_{i=0}^{n-1} p_i b^i$, with $p_{n-1} \neq 0$, $\mu = \lfloor b^{2n}/p \rfloor$, $b > 3$
Output: $R = X \bmod p$
 1 : $Q_1 \leftarrow \lfloor X/b^{n-1} \rfloor$
 2 : $Q_2 \leftarrow Q_1 \cdot \mu$
 3 : $Q_3 \leftarrow \lfloor Q_2/b^{n+1} \rfloor$
 4 : $R_1 \leftarrow X \bmod b^{n+1}$
 5 : $R_2 \leftarrow Q_3 \cdot p \bmod b^{n+1}$
 6 : $R \leftarrow R_1 - R_2$
 7 : if $R < 0$ then
 8 :     $R \leftarrow R + b^{n+1}$
 9 : end if
10 : while $R \geq p$ do
11 :     $R \leftarrow R - p$
12 : end while
13 : Return($R$)
```

Barret's modular reduction. To clarify Algorithm 11, consider re-writing X as $X = Q \cdot p + R$ with $0 \leq R < p$, which is a well known identity from the division algorithm [32, Definition 2.82]. Thus

$$R = X \bmod p = X - Q \cdot p \tag{11}$$

Barret's basic idea is that one can write Q in (11) as:

$$Q = \lfloor X/p \rfloor = \left\lfloor \left(X/b^{n-1}\right) \left(b^{2n}/p\right) \left(1/b^{n+1}\right) \right\rfloor \tag{12}$$

and in particular Q can be approximated by

$$\widehat{Q} = Q_3 = \left\lfloor \left\lfloor \left(X/b^{n-1}\right) \right\rfloor \left(b^{2n}/p\right) \left(1/b^{n+1}\right) \right\rfloor$$

We notice that Q_3 can be at most 2 smaller than Q [32, Fact 14.43] and that the quantity $\mu = b^{2n}/p$ can be precomputed when performing many modular reductions with the same modulus, as is the case in cryptographic algorithms. Finally, Step 11 in Algorithm 13 is repeated at most twice [4].

From the efficiency point of view, notice that all divisions by a power of b are simply performed by right-shifts and modular reduction modulo b^i, is equivalent to truncation. The complexity of Algorithm 13 is basically given by the number of multiplications. We notice that there are only two multi-precision multiplications: One to compute Q_2 (Step 2) and one to compute R_2 (Step 5). Both are "partial" multiplications, i.e., we don't need to compute all digits of the result. In the case of Q_2 the $n-1$ least significant digits need not to be computed and in the case of R_2 only the $n+1$ significant digits are needed. It can be shown that Algorithm 13 needs at most $\frac{(n^2+5n+2)}{2} + \binom{n+1}{2} + n = n^2 + 4n + 1$ single-precision multiplications (where single-precision multiplication means multiplication of two digits) [32, Note 14.45].

Barret's algorithm can be further improved as shown in [13, 14]. The basic idea is to re-write the quotient in (12) as:

$$Q = \lfloor X/p \rfloor = \left\lfloor \frac{\frac{X}{2^{n+\beta}} \frac{2^{n+\alpha}}{p}}{2^{\alpha-\beta}} \right\rfloor$$

where Barret's algorithm in radix $b = 2$ corresponds to the case $\alpha = n$ and $\beta = -1$. Then, the quotient can be estimated as

$$\widehat{Q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^{n+\beta}} \right\rfloor \left\lfloor \frac{2^{n+\alpha}}{p} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor$$

Then, for a given modulus p, $\mu = \frac{2^{n+\alpha}}{p}$ can be precomputed. It is shown in [14], that

$$\left\lfloor \frac{X}{p} \right\rfloor - 2^{\gamma-\alpha} - 2^{\beta+1} - 1 + 2^{\beta-\alpha} < \widehat{Q} \leq \left\lfloor \frac{X}{p} \right\rfloor \tag{13}$$

for some $\gamma > 0$. Equation (13) implies that the estimated quotient $\widehat{Q}$ is always smaller or equal to the real quotient and that one needs to choose α, β, and γ to minimize the error of the estimate $\widehat{Q}$. In particular, [14] shows that to minimize the error one must choose $\beta \leq -2$ and $\alpha > \gamma$. Following [14], the error is at most 1, thus improving over Barret's algorithm (Algorithm 13) where $\widehat{Q}$ could be at most 2.

Quisquater's Modular Reduction. Quisquater's algorithm, originally introduced in [38, 39], can be thought of as an improved version of Barret's reduction algorithm. References [5, 47] have proposed similar methods. In addition, the method is used in the Phillips smart-card chips P83C852 and P83C855, which use the CORSAIR crypto-coprocessor [12, 35] and the P83C858 chip, which uses the FAME crypto-coprocessor [18]. Quisquater's algorithm, as presented in [12], is a combination of the interleaved multiplication reduction method (Algorithm 12) and a method that makes easier and more accurate the estimation of the quotient Q in (11). Step 4 of Algorithm 12 can then be performed following (11). In particular, assume that we want to compute $R = X \bmod p = X - Q \cdot p$, then the quotient Q can be written as:

$$Q = \left\lfloor \frac{X}{p} \right\rfloor = \left\lfloor \frac{X}{2^{n+c}} \cdot \frac{2^{n+c}}{p} \right\rfloor$$

From the above, we can write

$$\widehat{Q}\delta = \left(\left\lfloor \frac{X}{2^{n+c}} \right\rfloor\right) \cdot \left(\left\lfloor \frac{2^{n+c}}{p} \right\rfloor\right) \tag{14}$$

where $\widehat{Q}$ is an approximation of the quotient Q. Thus, (14) allows us to write an approximation for $R = X \bmod p$ as

$$\widehat{R} = X - \widehat{Q} \cdot p' = X - \left\lfloor \frac{X}{2^{n+c}} \right\rfloor \cdot p'$$

where we effectively are performing a reduction modulo $p' = \delta p = \lfloor 2^{n+c}/p \rfloor p$. We notice that p' has its most significant c bits equal to 1 and that the computation of the approximate quotient $\widehat{Q}$ is immediate, i.e., it is just the most significant bits of X.

Since the objective of the modular reduction is to obtain $R = X \bmod p$, we need a way to obtain R from $\widehat{R}$ which is known in the literature as de-normalization.[2]

Since we have reduced modulo $p' = \delta p$, a multiple of p, we have that $R = X \bmod p = \widehat{R} \bmod p = (X \bmod p') \bmod p$. Notice that we can write $\delta \widehat{R} \bmod p'$ as:

$$\delta \widehat{R} \bmod p' = \left[\delta \left(X - \left\lfloor \frac{X}{p'} \right\rfloor p'\right)\right] \bmod p' = \delta X - \left\lfloor \frac{\delta X}{p'} \right\rfloor p' = \delta X - \left\lfloor \frac{X}{p} \right\rfloor (\delta p) \quad (15)$$

Thus, from (15), we obtain the following relation to obtain R from $\widehat{R}$:

$$R = \frac{(\delta \cdot \widehat{R}) \bmod p'}{\delta}$$

The only step that is left is to compute $\delta = \lfloor 2^{n+c}/p \rfloor$. It is shown in [15] that one can approximate δ within 1. Further, details about the algorithm and the choice of δ can be found in [14].

Montgomery Modular Reduction. The Montgomery algorithm, originally introduced in [33], is a technique that allows efficient implementation of the modular multiplication without explicitly carrying out the modular reduction step. The Montgomery reduction algorithm is shown in Algorithm 14. The idea behind Montgomery's algorithm is to transform the integers in M-residues[3] and compute the multiplication with these M-residues. At the end, one transforms back to the normal representation. As with Quisquater's method, this approach is only beneficial if we compute a series of multiplications in the transform domain (for example, in the case of modular exponentiation). Notice that Algorithm 14 is just the reduction step involved in a modular multiplication. The multiplication step can be accomplished, for example, with Algorithm 9. To see that Z in Step 2 is an integer, observe that $Q = T \cdot p' + k \cdot R$ and $p \cdot p' = -1 + l \cdot R$, for some integers k and l. Then, $(T + Q \cdot p)/R = (T + (T \cdot p' + k \cdot R)p)/R = l \cdot T + k \cdot p$.

In practice R in Algorithm 14 is a multiple of the word size of the processor and a power of two. This means that p, the modulus, has to be odd (because of the restriction $\gcd(p, R) = 1$) but this does not represent a problem as p is a prime or the product of two primes (RSA) in most practical cryptographic applications. In addition, choosing R a power of 2 simplifies Steps 1 and 2 in Algorithm 14, as they become simply truncation (modular reduction by R in Step 1) and right shifting (division by R in Step 2). Notice that $p' \equiv -p^{-1} \bmod R$. In [17] it is shown that if $p = \sum_{i=0}^{n-1} p_i b^i$, for some radix b typically a power of two, and $R = b^n$, then p' in Step 1 of Algorithm 14 can be substituted by $p'_0 = -p^{-1} \bmod b$. The authors of [17] notice that although the resulting sum $T + A \cdot p$ (in Step 2 of Algorithm 14) might not be the same, the effect is, namely making $T + A \cdot p$ a multiple of R.

[2] Notice that Quisquater's algorithm is intended for cases in which many modular reductions have to be performed such as in the case of an RSA exponentiation, thus the de-normalization is only performed at the end of the exponentiation.

[3] The M-residue of X is defined to be $X \cdot R \bmod M$, where $R > M$ and $\gcd(R, M) = 1$, as in the Montgomery reduction algorithm.

As with previous algorithms, one can interleave multiplication and reduction steps. The result is Algorithm 15 where the trick from [17] is also used.

Algorithm 14. Montgomery Reduction

```
Input: $0 \leq T < R \cdot p$, $R > p$, $\gcd(p, M) = 1$, and $R \cdot R^{-1} - p \cdot p' = 1$
Output: $Z = T \cdot R^{-1} \bmod p$
1 : $Q \leftarrow (T \bmod R)\, p' \bmod R$
2 : $Z \leftarrow \frac{T+Q\cdot p}{R}$
3 : if $Z \geq p$ then
4 :     $Z \leftarrow Z - p$
5 : end if
6 : Return($Z$)
```

Montgomery's multiplication algorithm has received a lot of attention since its introduction in 1985. These has lead to multiple variants depending on the granularity with which each operand is processed. We refer to [29] for an excellent treatment of several variants and a thorough comparison of each method's complexity.

Other Modular Reduction Algorithms. We have explicitly left out two other proposed methods for modular reduction because they are aimed at improving the efficiency of the modular reduction operation in hardware. These methods are the Sedlak's modular reduction algorithm [44], used by Infineon (previously Siemens Semiconductors) in the SLE44C200 and SLE44CR80S micro-processors and their derivatives [35] and Brickell's method, originally introduced in [9], which is dependent on the utilization of carry-delayed adders.

Algorithm 15. Montgomery Multiplication

```
Input: $X = \sum_{i=0}^{n-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$, $p = \sum_{i=0}^{n-1} p_i b^i$, with $0 \leq X, Y < p$, $b > 1$,
    $p' = -p_0^{-1} \bmod b$, $R = b^n$, $\gcd(b, p) = 1$
Output: $Z = X \cdot Y \cdot R^{-1} \bmod p$
1 : $Z \leftarrow 0$ {where $Z = \sum_{i=0}^{n} z_i b^i$}
2 : for $i = 0$ to $n - 1$ do
3 :     $q \leftarrow (z_0 + x_i \cdot y_0)\, p' \bmod b$
4 :     $Z \leftarrow (Z + x_i \cdot Y + q \cdot p)\, / b$
5 : end for
6 : if $Z \geq p$ then
7 :     $Z \leftarrow Z - p$
8 : end if
9 : Return($Z$)
```

Table 2 Special prime families [46]

Family name	Prime form	Comments
Mersenne primes	$2^k - 1$	k must be prime
Crandall numbers	$2^{dk} - 3$	–
Generalized Mersenne primes	$2^{dk} - 2^{ck} - 1$	where $0 < 2c \leq d$ and $\gcd(c, d) = 1$ or $3d < 6c < 4d$ and $\gcd(c, d) = 1$
	$2^{dk} - 2^{(d-1)k} + 2^{(d-2)k} - \cdots - 2^k + 1$	for d even and $k \neq 2 \bmod 4$
	$2^{dk} - 2^{ck} + 1$	$0 < 2c < d$ and $\gcd(c, d) = 1$
	$2^{4k} - 2^{3k} + 2^{2k} + 1$	–

4.4. Field Reduction for Special Primes and Other Tricks

Field reduction can be performed very efficiently if the modulus p is of special form. An example of such primes are the Crandall's primes [11] of the form $2^n - c$, for c positive and small enough to fit into one processor word. In [46], a different family of primes is introduced which accepts efficient reduction. These primes have received the name of Generalized Mersenne (GM) primes. The family of primes described in [46] (including those introduced in [11]) are shown in Table 2. These primes have been adopted for use in different standards such as NIST [36], ANSI [1] and SEC [42]. Table 3 summarizes the primes included in [36, 42] Fast reduction is possible using these primes since the powers of 2 translate naturally to bit locations in the underlying hardware. For example $2^{160} \equiv 2^{31} + 1 \bmod p_{160}$ and therefore each of the higher bits can be wrapped to the lower bit locations based on the equivalence. The steps required to compute the fast reduction using GM primes are given in NIST [36]. Two other techniques are worth mentioning. Solinas [46] notices that in some cases it might be advantageous to perform the modular reduction modulo a slightly larger

Table 3 Standardized primes [36, 42]

Prime name	Prime	Source
p_{112}	$(2^{128} - 3)/76, 439$	[42]
p_{128}	$2^{128} - 2^{97} - 1$	[42]
p_{160}	$2^{160} - 2^{31} - 1$	[42]
p_{192}	$2^{192} - 2^{64} - 1$	[36, 42]
p_{224}	$2^{224} - 2^{96} + 1$	[36, 42]
p_{256}	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	[36]
p_{384}	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	[36]
p_{521}	$2^{521} - 1$	[36]

prime than the modulus, such that the larger prime is a GM prime. This is the case of for example p_{112} in Table 3. Finally in applications where the integers do not fit exactly in k words and require a few bits of an extra word for their representation, the techniques of [50] can be useful. Notice that similar techniques were applied in [49] to an implementation of optimal extension fields in an 8051-based microcontroller, with substantial performance gains.

5. Inversion

Similar to the case of fields $\mathbb{F}_{2^m}$, the multiplicative inverse in $\mathbb{F}_p$ can be computed by variants of the EEA and Fermat's little theorem. The *binary Extended Euclidian Algorithm* (BEA, cf. Section 3) is the most general and in many cases most efficient method. When using the Montgomery residue system, the *Montgomery Inverse Algorithm* is preferable.

Binary Euclidean Algorithm (BEA). Let u be the element whose inverse is to be computed and v the modulus. Note that u and v must be relatively prime in order for the inverse to exist. The EEA computes coefficients s and t such that

$$us + vt = \gcd(u, v) = 1.$$

The parameter s is the inverse of u modulo v. In contrast to the Euclidean algorithm for $\mathbb{F}_p$ (cf. [32, Algorithm 2.107]), the BEA does not require integer divisions but only simple operations such as shifts and additions as shown in Algorithm 16. This usually leads to a faster execution on digital computers.

Montgomery Inverse. The Montgomery inverse computes the modular inverse in the Montgomery domain, i.e.,

$$\mathrm{MontInv}(A) = A^{-1}2^n (\bmod p), \qquad \text{where } n = \lceil \log_2 p \rceil.$$

The corresponding algorithm is given in Algorithm 17.

Almost Inverse. The Almost Inverse results from an intermediate step (Step 13, Algorithm 17) of the computation of the Montgomery Inverse, i.e.,

$$\mathrm{AlmMontInv}(A) = A^{-1}2^k (\bmod p).$$

where $2 \leq k \leq 2n$. Then, modular inverse A^{-1} can be obtained from the almost inverse by dividing out 2^k, an efficient method for which is a repeated division by 2^w and a final division by $2^{k-w\lfloor k/w \rfloor}$, where w is the wordsize of the processor.

Algorithm 16. BEA for Inversion in $\mathbb{F}_p$

```
Input: Prime $p$ and $A \in \mathbb{F}_p$.
Output: $A^{-1} \pmod{p}$.
1 : $u \leftarrow x, v \leftarrow p, b \leftarrow 1, c \leftarrow 0$.
2 : while $u \neq 1$ and $v \neq 1$ do
3 :     while $u$ is even do
4 :         $u \leftarrow u/2$.
5 :         if $b$ is even then
6 :             $b \leftarrow b/2$.
7 :         else
8 :             $b \leftarrow (b+p)/2$.
9 :         end if
10 :    end while
11 :    while $v$ is even do
12 :        $v \leftarrow v/2$.
13 :        if $c$ is even then
14 :            $c \leftarrow c/2$.
15 :        else
16 :            $c \leftarrow (c+p)/2$.
17 :        end if
18 :    end while
19 :    if $u \geq v$ then
20 :        $u \leftarrow u - v, b \leftarrow b - c$.
21 :    else
22 :        $v \leftarrow v - u, c \leftarrow c - b$.
23 :    end if
24 : end while
25 : if $u = 1$ then
26 :     return $b \pmod{p}$.
27 : else
28 :     return $c \pmod{p}$.
29 : end if
```

Algorithm 17. Montgomery Inversion in $\mathbb{F}_p$

```
Input: $A \in \mathbb{F}_p$.
Output: $r \in \mathbb{F}_p$ and $k$, where $r \equiv A^{-1}2^n \pmod{p}$ and $n \leq k \leq 2n$.
1 : $u \leftarrow p, v \leftarrow A, r \leftarrow 0, s \leftarrow 1$, and $k \leftarrow 0$.
2 : while $v > 0$ do
3 :     if $u$ is even then
4 :         $u \leftarrow u/2, s \leftarrow 2s$
5 :     else if $u > v$ then
6 :         $u \leftarrow (u - v)/2, r \leftarrow r + s, s \leftarrow 2s$
7 :     else if $v \geq u$ then
8 :         $u \leftarrow (v - u)/2, s \leftarrow r + s, r \leftarrow 2r$
9 :     end if
10 :    $k \leftarrow k + 1$
11 : end while
12 : if $r \geq p$ then
13 :     $r \leftarrow r - p$ (Almost Inverse)
14 : end if
15 : for $i = 1$ to $k - n$ do
16 :     if $r$ is even then
17 :         $r \leftarrow r/2$
18 :     else
19 :         $r \leftarrow (r + p)/2$
20 :     end if
21 : end for
22 : return $r$ (Montgomery Inverse)
```

6. Software Implementation Techniques Over $\mathbb{F}_{p^m}$

For elliptic curve cryptosystems, only prime extension fields are of interest due to the Weil descent attack for composite fields. An *Optimal Extension Field*, which is a special version of extension fields, has properties that allow an efficient implementation in software.

DEFINITION 1. *An Optimal Extension Field is a extension field $\mathbb{F}_{p^m}$ such that*

1. The prime p is a pseudo-Mersenne (PM) prime of the form $p = 2^n \pm c$ with $log_2(c) \leq \lfloor n/2 \rfloor$.
2. An irreducible binomial $P(x) = x^m - \omega$ exists over $\mathbb{F}_p$.

We represent the elements of $\mathbb{F}_{p^m}$ as polynomials of degree at most $m-1$ with coefficients from the subfield $\mathbb{F}_p$, i.e. any element $A \in \mathbb{F}_{p^m}$ can be written as

$$A(t) = \sum_{i=0}^{m-1} a_i \cdot t^i = a_{m-1} \cdot t^{m-1} + \cdots + a_2 \cdot t^2 + a_1 \cdot t + a_0 \quad \text{with } a_i \in \mathbb{F}_p \tag{16}$$

where t is the root of $P(x)$ (i.e. $P(t) = 0$). The prime p is generally selected to be a pseudo-Mersenne prime that fits into a single processor word. Consequently, we can store the m coefficients of $A \in \mathbb{F}_{p^m}$ in an array of m single-precision words, represented as the vector $(a_{m-1}, \ldots, a_2, a_1, a_0)$.

The construction of an OEF requires a binomial $P(x) = x^m - \omega$ which is irreducible over $\mathbb{F}_p$. Reference [3] describes a method for finding such irreducible binomials. The specific selection of p, m, and $P(x)$ leads to a fast subfield and extension field reduction, respectively.

Addition and Subtraction. Addition and subtraction of two field elements A, $B \in \mathbb{F}_{p^m}$ is accomplished in a straightforward way by addition/subtraction of the corresponding coefficients in $\mathbb{F}_p$.

$$C(t) = A(t) \pm B(t) = \sum_{i=0}^{m-1} c_i \cdot t^i \quad \text{with } c_i \equiv a_i \pm b_i \bmod p \tag{17}$$

A reduction modulo p (i.e. an addition or subtraction of p) is necessary whenever the sum or difference of two coefficients a_i and b_i is outside the range of $[0, p-1]$ (shown in Algorithms 10 and 11). There are no carries propagating between the coefficients which is an advantage for software implementations.

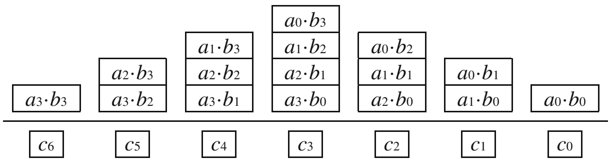
Multiplication and Squaring. A multiplication in the extension field $\mathbb{F}_{p^m}$ can be performed by ordinary polynomial multiplication over $\mathbb{F}_p$ and a reduction of the product polynomial modulo the irreducible polynomial $P(t)$. The product of two polynomials of degree at most $m-1$ is a polynomial of degree at most $2m-2$.

$$\begin{aligned} C(t) &= A(t) \cdot B(t) = \Big(\sum_{i=0}^{m-1} a_i \cdot t^i\Big) \cdot \Big(\sum_{j=0}^{m-1} b_j \cdot t^j\Big) \\ &\equiv \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} (a_i \cdot b_j \bmod p) \cdot t^{(i+j)} = \sum_{k=0}^{2m-2} c_k \cdot t^k \end{aligned} \tag{18}$$

There are several techniques to accomplish a polynomial multiplication. The standard algorithm moves through the coefficients b_j of $B(t)$, starting with b_0, and multiplies b_j by any coefficient a_i of $A(t)$. This method, which is also referred to as *operand scanning* technique, requires exactly m^2 multiplications of coefficients $a_i, b_j \in \mathbb{F}_p$. However, there are two advanced multiplication techniques which typically perform better than the standard algorithm. The *product scanning* technique reduces the number of memory accesses (in particular store operations), whereas Karatsuba's algorithm [27] requires fewer coefficient multiplications [3, 48].

The *product scanning* technique employs a 'multiply-and-accumulate' strategy [23] and forms the product $C(t) = A(t) \cdot B(t)$ by computing each coefficient c_k of $C(t)$ at a time. Therefore, the coefficient-products $a_i \cdot b_j$ are processed in a 'column-by-

Figure 3 Multiply-and-accumulate strategy ($m = 4$).



column' fashion, as depicted in Figure 3 (for $m = 4$), instead of the 'row-by-row' approach used by the operand scanning technique. More formally, the product $C(t)$ and its coefficients c_k are computed as follows.

$$C(t) = A(t) \cdot B(t) = \sum_{k=0}^{2m-2} c_k \cdot t^k \quad \text{with} \quad c_k \equiv \sum_{i+j=k} a_i \cdot b_j \bmod p \quad (0 \le i, j \le m-1) \tag{19}$$

The product scanning technique requires exactly the same number of coefficient multiplications as its operand scanning counterpart (namely m^2), but minimizes the number of store operations since a coefficient c_k is only written to memory after it has been completely evaluated. In general, the calculation of coefficient-products $a_i \cdot b_j$ and the reduction of these modulo p can be carried out in any order. However, it is usually advantageous to compute an entire column sum first and perform a single reduction thereafter, instead of reducing each coefficient-product $a_i \cdot b_j$ modulo p. The former approach results in m^2 reduction operations, whereas the latter requires only one reduction per coefficient c_k, which is $2m - 1$ reductions altogether.

When $A(t) = B(t)$, the coefficient-products of the form $a_i \cdot b_j$ appear once for $i = j$ and twice for $i \neq j$. Therefore squaring of a polynomial $A(t)$ of degree $m - 1$ can be obtained with only $m \cdot (m + 1)/2$ coefficient multiplications

Subfield Reduction. An integral part of both polynomial multiplication and polynomial squaring is the subfield reduction which is the reduction of a coefficient-product (or a sum of several coefficient-products) modulo the prime p. Pseudo-Mersenne primes are a family of numbers highly suited for modular reduction due to their special form [11]. They allow to employ very fast reduction techniques that are not applicable to general primes. The efficiency of the reduction operation modulo a PM prime $p = 2^n - c$ is based on the relation

$$2^n \equiv c \bmod p \quad (\text{for } p = 2^n - c) \tag{20}$$

which means that any occurrence of 2^n in an integer $z \ge 2^n$ can be substituted by the much smaller offset c. To give an example, let us assume that z is the product of two integers $a, b < p$, and thus $z < p^2$. Furthermore, let us write the $2n$-bit product z as $z_H \cdot 2^n + z_L$, whereby z_H and z_L represent the n most and least significant bits of z, respectively. The basic reduction step is accomplished by multiplying z_H and c together and 'folding' the product $z_H \cdot c$ into z_L.

$$z = z_H \cdot 2^n + z_L \equiv z_H \cdot c + z_L \bmod p \quad (\text{since } 2^n \equiv c \bmod p) \tag{21}$$

This leads to a new expression for the residue class with a bit-length of at most $1.5n$ bits. Repeating the substitution a few times and performing final subtraction of

p yields the fully reduced result $x \bmod p$. A formal description of the reduction modulo $p = 2^n - c$ is given in Algorithm 18.

Algorithm 18. Fast reduction modulo a pseudo-Mersenne prime $p = 2^n - c$ with $\log_2(c) \le n/2$

```
Input: $n$-bit modulus $p = 2^n - c$ with $\log_2(c) \le n/2$, operand $y \ge p$.
Output: Residue $z \equiv y \bmod p$.
1 : $z \leftarrow y$
2 : while $z \ge 2^n$ do
3 :     $z_L \leftarrow z \bmod 2^n$   { the $n$ least significant bits of $z$ are assigned to $z_L$ }
4 :     $z_H \leftarrow \lfloor z/2^n \rfloor$   { $z$ is shifted $n$ bits to the right and assigned to $z_H$ }
5 :     $z \leftarrow z_H \cdot c + z_L$
6 : end while
7 : if $z \ge p$ then $z \leftarrow z - p$ end if
8 : return $z$
```

Finding the integers z_L and z_H is especially easy when n equals the word-size of the target processor. In this case, no bit-level shifts are needed to align z_H for the multiplication by c.

Extension Field Reduction. Polynomial multiplication and squaring yields a polynomial $C(t)$ of degree $2m - 2$ with coefficients $c_k \in \mathbb{F}_p$ after subfield reduction. This polynomial must be reduced modulo the irreducible polynomial $P(t) = t^m - \omega$ in order to obtain the final polynomial of degree $m - 1$. The extension field reduction can be accomplished in linear time since $P(t)$ is a monic irreducible binomial. Given $P(t) = t^m - \omega$, the following congruences hold: $t^m \equiv \omega \bmod x(t)$. We can therefore reduce $C(t)$ by simply replacing all terms of the form $c_k \cdot t^k$, $k \ge m$, by $c_k \cdot \omega \cdot t^{k-m}$, which leads to the following equation for the residue:

$$\begin{aligned} R(t) &\equiv C(t) \bmod P(t) \\ &\equiv \sum_{l=0}^{m-1} r_l \cdot t^l \quad \text{with} \quad r_{m-1} = c_{m-1} \\ \text{and} \quad r_l &\equiv (c_{l+m} \cdot \omega + c_l) \bmod p \quad \text{for} \quad 0 \le l \le m-2 \end{aligned} \tag{22}$$

The entire reduction of $C(t)$ modulo the binomial $P(t) = t^m - \omega$ costs at most $m - 1$ multiplications of coefficients c_k by ω and the same number of subfield reductions [2].

In summary, the straightforward way of multiplying two elements in OEF requires $m^2 + m - 1$ coefficient multiplications and $3m - 2$ reductions modulo p. Special optimizations, such as Karatsuba's method or the 'interleaving' of polynomial multiplication and extension field reduction, allow to minimize the number of subfield operations (see [23] for details).

6.1. Inversion

In the case of extension fields $GF(q^m)$, $m \geq 2$, inversion in the field $GF(q^m)$ can be reduced to inversion in the field $GF(q)$. This reduction comes at the cost of extra operations (multiplications and additions) in the field $GF(q^m)$. If the inversion in the subfield $GF(q)$ is sufficiently inexpensive computationally compared to extension field inversion, the method described in *Itoh–Tsujii Inversion* can have a low over-all complexity. The method was introduced for fields in normal basis representation in [26] and generalized to fields in polynomial basis representation in [22]. The method can be applied iteratively in fields with multiple field extensions, sometimes referred to as tower fields. In the case of fields $GF(2^m)$, m a prime, the Itoh–Tsujii algorithm degenerates into inversion based on Fermat's little theorem (cf. Section 3). It should be stressed that this method is not a complete inversion algorithm since it is still necessary to eventually perform an inversion in the subfield. However, inversion in a (small) subfield can often be done fast with one of the methods described above. Inversion in an OEF can be accomplished either with the extended Euclidean algorithm or via a modification of the *Itoh–Tsujii Algorithm (ITA)* [26], which reduces the problem of extension field inversion to subfield inversion [2].

Another method is *direct inversion* which is applicable to extension fields $GF(q^m)$, and mainly relevant for fields where m is small, e.g., $m = 2, 3, 4$.

Itoh–Tsujii Inversion. The ITA computes the inverse of an element $A \in \mathbb{F}_{p^m}$ as

$$A^{-1}(t) \equiv (A^r(t))^{-1} \cdot A^{r-1}(t) \bmod P(t)$$
$$\text{where } r = \frac{p^{m-1}}{p-1} = p^{m-1} + \cdots + p^2 + p + 1.$$

Efficient calculation of $A^{r-1}(t)$ is performed by using an addition-chain constructed from the p-adic representation of $r - 1 = (111\ldots110)_p$. This approach requires the field elements to be raised to the p^ith powers, which can be done with help of the ith iterate of the Frobenius map [3].

The other operation is the inversion of $A^r(t) = A^{r-1}(t) \cdot A(t)$. Computing the inverse of $A^r(t)$ is easy due to the fact that for any element $\alpha \in \mathbb{F}_{p^m}$, the rth power of α, i.e. $\alpha^{(p^m-1)/(p-1)}$ is always an element of the subfield $\mathbb{F}_p$. Thus, the computation of $(A^r(t))^{-1}$ requires just an inversion in $\mathbb{F}_p$ which can be done using a single-precision variant of the extended Euclidean algorithm.

In summary, the efficiency of the ITA in an OEF relies mainly on the efficiency of the extension field multiplication and the subfield inversion (see [3, 23]).

Direct Inversion. Similar to Itoh–Tsujii inversion, direct inversion also reduces extension field inversion to subfield inversion.

As an example, we demonstrate the method for fields $GF(q^2)$, introduced in [34]. Let us consider a non-zero element $A = a_0 + a_1 x$ from $GF(q^m)$, where $a_0, a_1 \in GF(q)$. Let us assume the irreducible field polynomial has the form $P(x) = x^2 + x + p_0$, where $p_0 \in GF(q)$. If the inverse is denoted as $B = A^{-1} = b_0 + b_1 x$, the equation

$$A \cdot B = [a_0 b_0 + p_0 a_1 b_1] + [a_0 b_1 + a_1 b_0 + a_1 b_1]x = 1$$

must be satisfied, which is equivalent to a set of two linear equations in b_0, b_1 over $GF(q)$ with the solution:

$$\left.\begin{aligned} b_0 &= (a_0 + a_1)/\Delta \\ b_1 &= a_1/\Delta \end{aligned}\right\}, \text{ where } \Delta = a_0(a_0 + a_1) + p_0 a_1^2.$$

The advantage of this algorithm is that all operations are performed in $GF(q)$. Note that there is one inversion in the subfield $GF(q)$ of the parameter Δ required. The algorithm can be applied recursively. The relationship between direction inversion and the Itoh–Tsujii method is sketched in [37].

7. Summary

We presented here different finite field arithmetic algorithms that are suitable for software implementation of cryptographic algorithms. The performance of the various algorithms depends vastly on the underlying micro-processor architecture on which it is implemented. Therefore a careful choice of the appropriate set of algorithms have to made for an implementation depending on the performance requirements and available resources.

References

1. American National Standards Institute, New York, USA. ANSI X9.62: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA) (1999)
2. Bailey, D.V., Paar, C.: Optimal extension fields for fast arithmetic in public-key algorithms. In: Krawczyk, H. (ed.) Advances in Cryptology – CRYPTO '98, volume 1462 of Lecture Notes in Computer Science, pp. 472–485. Springer, Berlin Heidelberg New York (August 1998)
3. Bailey, D.V., Paar, C.: Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. J. Cryptology **14**(3), 153–176 (2001)
4. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology – CRYPTO '86, volume 263 of LNCS, pp. 311–323. Springer, Berlin Heidelberg New York (August 1986)
5. Benaloh, J., Dai, W.: Fast modular reduction. Rump session of CRYPTO '95
6. Bernstein, D.J.: Multidigit multiplication for mathematicians. Advances in Applied Mathematics (2001). Available at http://cr.yp.to/papers/m3.pdf (Accepted for publication but later withdrawn)
7. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic Curves in Cryptography, volume 265 of London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, UK (1999)
8. Blakley, G.R.: A computer algorithm for the product AB modulo M. IEEE Trans. Comput. **32**(5), 497–500 (May 1983)
9. Brickell, E.F.: A fast modular multiplication algorithm with applications to two key cryptography. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology – CRYPTO '82, pp. 51–60. Plenum, New York, USA (1982)
10. Brown, M., Hankerson, D., López, J., Menezes, A.: Software implementation of the nist elliptic curves over prime fields. In: Naccache, D. (ed.) Topics in Cryptology – CT-RSA 2001, volume 2020 of LNCS, pp. 250–265. Springer, Berlin Heidelberg New York (2001)
11. Crandall, R.E.: Method and apparatus for public key exchange in a cryptographic system. U.S. Patent #5,159,632, US Patent and Trade Office (Oct 1992)
12. De Waleffe, Quisquater, J.-J.: CORSAIR: A smart card for public key cryptosystems. In: Menezes, A.J., Vanstone, S.A. (eds.) Advances in Cryptology – CRYPTO '90, volume 537 of LNCS, pp. 502–514. Springer, Berlin Heidelberg New York (1990)
13. Dhem, J.-F.: Modified version of the Barret modular multiplication algorithm. UCL Technical Report CG-1994/1, Université catholique de Louvain (18 July 1994)

14. Dhem, J.-F.: Design of an efficient public-key cryptographic library for RISC-based smart cards. PhD thesis, UCL – Université catholique de Louvain, Louvain-la-Neuve, Belgium (May 1998)
15. Dhem, J.-F., Joye, M., Quisquater, J.-J.: Normalisation in diminished-radix modulus transform. Electron. Lett. **33**(23), 1931 (1997)
16. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inform. Theory **IT-22**(6), 644–654 (November 1976)
17. Dussé, S.R., Kaliski, B.S.: A cryptographic library for the Motorola DSP56000. In: Damgård, I.B. (ed.) Advances in Cryptology – EUROCRYPT '90, volume 473 of LNCS, pp. 230–244. Springer, Berlin Heidelberg New York (May 1990)
18. Ferreira, R., Malzahn, R., Marissen, P., Quisquater, J.-J., Wille, T.: FAME: A 3rd generation coprocessor for optimising public key cryptosystems in smart card applications. In: Hartel, P.H., Paradinas, P., Quisquater, J.-J. (eds.) Smart Card Research and Advanced Applications – CARDIS 1996, pp. 59–72, CWI, Amsterdam, The Netherlands. Stichting Mathematisch Centrum (16–18 Sept 1996)
19. Fong, K., Hankerson, D., López, J., Menezes, A.: Field inversion and point halving revisited. Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada (2003)
20. Gaudry, P., Hess, F., Smart, N.P.: Constructive and destructive facets of weil descent on elliptic curves. J. Cryptology **15**(1), 19–46 (2002)
21. Guajardo, J., Paar, C.: A Modified Squaring Algorithm (1999). Available at http://citeseer.ist.psu.edu/672729.html (Unpublished Manuscript)
22. Guajardo, J., Paar, C.: Itoh–Tsujii inversion in standard basis and its application in cryptography and codes. Des. Codes Cryptogr. **25**, 207–216 (2002)
23. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Berlin Heidelberg New York (2004)
24. Hankerson, D., López Hernandez, J., Menezes, A.: Software implementation of elliptic curve cryptography over binary fields. In: Koç, Ç.K., Paar, C. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES '99), volume 1717 of Lecture Notes in Computer Science, pp. 1–24. Springer, Berlin Heidelberg New York (August 2000)
25. IEEE Computer Society Press, Silver Spring, MD, USA. IEEE P1363-2000: IEEE Standard Specifications for Public-Key Cryptography (2000)
26. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Inform. and Comput. **78**, 171–177 (1988)
27. Karatsuba, A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Physics – Doklady (English translation) **7**, 595–596 (1963)
28. Knuth, D.E.: The Art of Computer Programming, vol. 2: Seminumerical Algorithms, vol. 2, 2nd edn. Addison-Wesley, Massachusetts, USA (1973)
29. Koç, Ç.K., Acar, T., Kaliski, B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE MICRO **16**(3), 26–33 (June 1996)
30. Lidl, R., Niederreiter, H.: Finite Fields, volume 20 of Encyclopedia of Mathematics and its Applications, Second edition. Cambridge University Press, Cambridge, UK (1997)
31. López, J., Dahab, R.: High-speed software multiplication in F_{2^m}. In: Roy, B., Okamoto, E. (eds.) International Conference in Cryptology in India–INDOCRYPT 2000, volume 1977 of Lecture Notes in Computer Science, pp. 203–212. Springer, Berlin Heidelberg New York (December 2000)
32. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. The CRC Press Series on Discrete Mathematics and Its Applications. CRC, Florida, USA (1997)
33. Montgomery, P.L.: Modular multiplication without trial division. Math. Comp. **44**(170), 519–521 (April 1985)
34. Morii, M., Kasahara, M.: Efficient construction of gate circuit for computing multiplicative inverses over $GF(2^m)$. Trans. Inst. Electron. Inf. Commun. Eng. **E**(72), 37–42 (January 1989)
35. Naccache, D., M'Raïhi, D.: Cryptographic smart cards. IEEE MICRO **16**(3), 14–24 (1996)
36. National Institute for Standards and Technology, Gaithersburg, MD, USA. FIPS 186-2: Digital Signature Standard (DSS). 186-2 (February 2000). Available for download at http://csrc.nist.gov/encryption
37. Paar, C.: Some Remarks on Efficient Inversion in Finite Fields. In: IEEE International Symposium on Information Theory, pp. 17–22. IEEE, Whistler, B.C. Canada (September 1995)
38. Quisquater, J.-J.: Fast modular exponentiation without division. Rump session of EUROCRYPT '90

39. Quisquater, J.-J.: Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system. US Patent 5,166,978, 24 (November 1992)
40. Schroeppel, R., Orman, H., O'Malley, S., Spatscheck, O.: Fast key exchange with elliptic curve systems. In: Coppersmith, S.D. (ed) Advances in Cryptology – CRYPTO '95, volume 963 of Lecture Notes in Computer Science, pp. 43–56. Springer, Berlin Heidelberg New York (August 1995)
41. Schroeppel, R., Orman, H., O'Malley, S., Spatscheck, O.: Fast key exchange with elliptic curve systems. In: Coppersmith, D. (ed) Advances in Cryptology – CRYPTO '95, volume 963 of Lecture Notes in Computer Science, pp. 43–56. Springer, Berlin Heidelberg New York (August 1995)
42. SEC 2. Standards for Efficient Cryptography Group: Recommended Elliptic Curve Domain Parameters. Version 1.0 (2000)
43. Sedgewick, R.: Algorithms, Second edition. Addison-Wesley, Massachusetts, USA (1988)
44. Sedlak, H.: The RSA cryptography processor. In: Chaum, D., Price, W.L. (eds.) Advances in Cryptology – EUROCRYPT '87, volume 304 of LNCS, pp. 95–105. Springer, Berlin Heidelberg New York (1987)
45. Sloan, Jr., K.R.: Comments on "A computer algorithm for the product AB modulo M". IEEE Trans. Comput. **34**(3), 290–292 (March 1985)
46. Solinas, J.: Generalized mersenne numbers. Technical Report, CORR 99-39, Department of Combinatorics and Optimization, University of Waterloo, Canada (1999)
47. Walter, C.D.: Faster modular multiplication by operand scaling. In: Feigenbaum, J. (ed.) Advances in Cryptology – CRYPTO '91, volume 576 of LNCS, pp. 313–323. Springer, Berlin Heidelberg New York (1991)
48. Weimerskirch, A., Paar, C.: Generalizations of the Karatsuba Algorithm for Polynomail Multiplication. Technical report, Ruhr-University Bochum, Germany (2003). Available at http://www.crypto.rub.de/Publikationen/texte/kaweb.pdf
49. Woodbury, A., Bailey, D.V., Paar, C.: Elliptic curve cryptography on smart cards without coprocessors. In: Ferrer, J.D., Chan, D., Watson, A. (eds.) Smart Card Research and Advanced Applications-CARDIS 2000, volume 180 of IFIP Conference Proceedings, pp. 71–92, Bristol, UK. Kluwer (September 2000)
50. Yanik, T., Savas, E., Koç, Ç.K.: Incomplete reduction in modular arithmetic. IEE Proc., Comput. Digit. Tech. **149**(2), 46–52 (March 2002)