Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields GF(2^m)

Johann Großschädl and Guy-Armand Kamendje
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
{Johann.Groszschaedl,Guy-Armand.Kamendje}@iaik.at

Abstract

The performance of elliptic curve (EC) cryptosystems depends essentially on efficient arithmetic in the underlying finite field. Binary finite fields GF(2^m) have the advantage of "carry-free" addition. Multiplication, on the other hand, is rather costly since polynomial arithmetic is not supported by general-purpose processors. In this paper we propose a combined hardware/software approach to overcome this problem. First, we outline that multiplication of binary polynomials can be easily integrated into a multiplier datapath for integers without significant additional hardware. Then, we present new algorithms for multiple-precision arithmetic in GF(2^m) based on the availability of an instruction for single-precision multiplication of binary polynomials. The proposed hardware/software approach is considerably faster than a "conventional" software implementation and well suited for constrained devices like smart cards. Our experimental results show that an enhanced 16-bit RISC processor is able to generate a 191-bit ECDSA signature in less than 650 msec when the core is clocked at 5 MHz.

1. Introduction

Smart cards play an important role in today's public-key infrastructures acting as a key generator and providing storage for the secret keys and certificates. A processor-based smart card is actually a complete "computer in your hand" with an embedded processor core, RAM, nonvolatile memory, and a card operating system. High-end smart cards offer the ability to perform public-key cryptographic algorithms on a specialized co-processor. The crucial point is that all computations that require secret keys (e.g. generation of digital signatures or encryption of data) have to be performed on the card so that the secret key must never be revealed. In other words, smart cards can be used to isolate security-critical computations from other parts of a system that do not have a need to know.

Public-key cryptography includes encryption schemes (like RSA), digital signature schemes (like DSA), key agreement schemes (like Diffie-Hellman), as well as other types of schemes [21]. The security of these cryptosystems is is based on either the integer factorization problem or the discrete logarithm problem. Another common characteristic of the "traditional" public-key schemes is that their basic operation is modular exponentiation on very long integers (1024-4096 bits). In general, the cost for performing a modular exponentiation scales with the third power of the operand length. Computing a 1024-bit modular exponentiation is no problem on current-generation desktop computers, but predictably results in an unacceptably long delay on constrained devices like smart cards. Note that the

Proc. of the 14th IEEE Conf. on Application-specific Systems, Architectures and Processors (ASAP 2003), pp. 455–468.

majority of the smart cards on the market today are equipped with an 8 or 16-bit processor clocked at a mere 5 MHz and have between 128 and 1024 bytes of RAM [3]. To address this problem, many smart card manufacturers include a co-processor to accelerate long integer arithmetic operations. However, a co-processor adds to die size and increases the cost of each card since hardware accelerators for public-key cryptography are extremely cost sensitive.

Elliptic curve (EC) cryptography [2] is emerging as a serious alternative to RSA and DSA for use in constrained environments such as smart cards [3]. The mathematical basis for the security of EC cryptosystems is the computational intractability of the EC discrete logarithm problem (ECDLP). A major attraction of EC cryptography over competing techniques like RSA, DSA, or Diffie-Hellman is the absence of a subexponential-time algorithm that could solve the ECDLP on a properly chosen curve. Thus, key sizes can be much smaller than for RSA while maintaining comparable levels of security[1]. The result is faster implementations, bandwidth and storage savings, and reduced energy consumption; features which are especially attractive for security applications in restricted computing environments.

The performance of EC cryptosystems is significantly determined by an efficient implementation of the arithmetic in the underlying finite field [28]. Most standardization organizations recommended to use either *prime fields* GF(p) or *binary extension fields* GF(2^m) [1, 25]. However, the problem with GF(2^m) is that polynomial arithmetic is not supported by most of the recent smart card processors. In particular, an instruction for word-level multiplication of binary polynomials is lacking, although such an instruction would greatly facilitate the software performance of arithmetic in GF(2^m) when *polynomial basis representation* [18] is used. Previous work [17] suggested to "emulate" this instruction by `SHIFT` and `XOR` operations or by table look-up.

In this paper we present a combined hardware/software approach for efficient arithmetic in binary finite fields GF(2^m) of large order. First, we propose to extend the processor's instruction set by an additional instruction that implements the operation described before. This extra instruction, which we call `MULGF2` as in [17], performs the multiplication two binary polynomials of degree $w-1$ over GF(2), yielding a binary polynomial of degree $2w-2$, whereby w denotes the native word-size of the processor (usually 8, 16, or 32 bits). However, the multiplication of binary polynomials does not necessarily require a separate multiplier datapath. Instead, we demonstrate that polynomial arithmetic can be easily integrated into a multiplier datapath for integers, resulting in a so-called *unified* multiplier datapath. That way, the additional hardware cost for the `MULGF2` instruction is very low (significantly lower than the cost for a dedicated co-processor for GF(2^m)-arithmetic).

This little enhancement of the processor's multiplier allows to achieve significant performance gains in the processing of multiple-precision arithmetic on binary polynomials. Based on the availability of a `MULGF2` instruction, we present efficient arithmetic algorithms for multiplication, squaring and reduction in binary fields GF(2^m) of large order. The most important feature of these algorithms is that they avoid bit-level operations which are slow on microprocessors and perform word-level operations (e.g. `MULGF2`) which are significantly faster. We also analyze and compare the number of `MULGF2`, `XOR`, `LOAD`, and `STORE` instructions executed for each of the algorithms. Systems which do cryptography in software instead of hardware have the significant advantage that they are able to respond to flaws discovered in the implemented algorithms or to changes in emerging standards. With such a combined hardware/software approach, binary fields GF(2^m) and polynomial basis representation become a very attractive option for the implementation of EC cryptosystems on smart cards without co-processor[2].

[1] A widely accepted rule of thumb says that a properly chosen 160-bit EC cryptosystem offers a higher level of security than a 1024-bit RSA or discrete logarithm system.

[2] The general concepts presented in this paper are applicable to virtually all general-purpose processors. However, the high level of flexibility and cost effectiveness make this approach especially advantageous for smart cards.

2. Elliptic Curve Cryptography

In this section we briefly summarize the basics of EC cryptography (more detailed information can be found in [2]). EC cryptosystems may be viewed as elliptic curve analogues of the older discrete logarithm cryptosystems in which the group $\mathbb{Z}_p^*$ is replaced by the group of points on an elliptic curve over a finite field. Let K be any field. Then, an elliptic curve over K can be defined as the set of all solutions $(x,y) \in K \times K$ to the general (affine) Weierstraß equation of the form

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

where the a_i lie in K. When K is a *finite field* (or Galois field) GF(q), the number of points on the EC is also finite. The order q of a finite field is always a positive prime power, i.e. $q = p^m$. When we specify the characteristic of the underlying finite field GF(q), the general Weierstraß equation can be transformed into a simpler form that is isomorphic to it. For fields of characteristic two, i.e. binary extension fields GF(2^m), we can simplify Equation (1) by an appropriate change of variables and get the following short Weierstraß form:

$$y^2 + xy = x^3 + ax^2 + b, \quad a,b \in \text{GF}(2^m) \tag{2}$$

The set of points, i.e. the set of all pairs $(x,y) \in \text{GF}(2^m) \times \text{GF}(2^m)$ satisfying Equation (2), has some interesting properties. In particular, the set of points (x,y) together with a special point $\mathcal{O}$ (called the "point at infinity") allows the definition of an operation for which the group axioms are fulfilled. That way, we obtain an *abelian group* whose zero element is the point at infinity $\mathcal{O}$. The group operation is the addition of points, which is defined as follows. Let $P = (x_1, y_1) \neq \mathcal{O}$ be a point on the EC. The inverse of P is $-P = (x_1, x_1 + y_1)$. Let $Q = (x_2, y_2) \neq \mathcal{O}$ be a second point with $Q \neq -P$. Then, the sum $P + Q = (x_3, y_3)$ can be calculated as

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a,\ y_3 = \lambda(x_1 + x_3) + x_3 + y_1,\ \lambda = \tfrac{y_1+y_2}{x_1+x_2} \ \ldots \text{ if } P \neq Q \tag{3}$$

$$x_3 = \lambda^2 + \lambda + a,\ y_3 = \lambda(x_1 + x_3) + x_3 + y_1,\ \lambda = x_1 + \tfrac{y_1}{x_1} \ \ldots \text{ if } P = Q \tag{4}$$

Equation (3) describes a *point addition*, whereas Equation (4) describes a *point doubling*. Both point addition and doubling require two multiplications, one squaring, and one inversion in GF(2^m).

In cases where field inversions are significantly more expensive than multiplications, it is generally beneficial to represent the EC points with *projective coordinates* [2]. Any point $P = (x,y)$ in affine coordinates can be viewed as a triple (X,Y,Z) in projective coordinates. The conversion of an affine point (x,y) to a projective point is simply performed by setting $X = x$, $Y = y$, and $Z = 1$. Similarly, to convert a projective point (X,Y,Z) with nonzero Z to an affine point, we compute $x = X/Z$ and $y = Y/Z$. The main advantage of projective coordinates is that point addition/doubling can be done without inversion, but for the cost of an increased number of multiplications. The decision whether to use projective or affine coordinates is strongly determined by implementation aspects such as the availability of memory for storing temporary values and the relative performance of the field inversion and multiplication operations.

2.1. Point Multiplication

The fundamental building block of virtually all EC cryptosystems is a computation of the form $Q = k \cdot P$, which is nothing else than adding a point k-times to itself, i.e. $k \cdot P = P + P + \cdots + P$. This operation is called *point multiplication* or *scalar multiplication*, and dominates the execution time of

EC cryptosystems. Scalar multiplication on an EC is analogous to exponentiation in a multiplicative group. The inverse operation, i.e. to recover the integer k when the points P and $Q = k \cdot P$ are given, is the elliptic curve discrete logarithm problem (ECDLP).

The basic technique for computing $k \cdot P$ is the binary method (*double-and-add* algorithm), which works very similar to the square-and-multiply algorithm for exponentiation. A point multiplication according to the binary method requires h point additions and n doublings, whereby h is the Hamming weight of k and $n = \log_2(k)$. By using projective coordinates, an entire point multiplication can be computed without field inversion. However, one inversion needs to be performed at the end of a point multiplication if the final result has to be given in affine coordinates.

In [23], P. L. Montgomery introduced an ingenious technique to compute multiples of points on an EC. His method is based on the fact that the sum of two points whose difference is a known point can be computed without the y-coordinate of the two points. Reference [19] describes the algorithm and shows how to recover the y-coordinate of $Q = k \cdot P$. For an EC over GF(2^m), Montgomery's method requires $6 \cdot \log_2(k)$ field multiplications and $5 \cdot \log_2(k)$ squarings when using projective coordinates.

2.2. Finite Fields

An advantage of EC systems is that the underlying finite field GF(q) and a representation for its elements can be selected so that the field arithmetic (addition, multiplication, and inversion) can be optimized. Examples of finite fields with "good" arithmetic properties are prime fields GF(p) and binary extension fields GF(2^m). The *prime field* GF(p) is the field of residue classes modulo p, where the field elements are the integers $0, 1, \ldots, p-1$. The field operations are modulo operations, i.e. addition and multiplication modulo the prime p. One of the most widely used generic algorithms for modulo multiplication is due to P. L. Montgomery [22]. The *binary extension field* GF(2^m) can be viewed as a vector space of dimension m over GF(2). That means, there exists a set of m elements $B = \{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ with $\beta_i \in \text{GF}(2^m)$, called a *basis*, such that any element $a \in \text{GF}(2^m)$ may be written uniquely in the form

$$a = \sum_{i=0}^{m-1} a_i \beta_i = a_{m-1}\beta_{m-1} + \cdots + a_1\beta_1 + a_0\beta_0 \quad \text{with } a_i \in \{0,1\} \tag{5}$$

For a given basis B, we can represent the element a by the binary string $(a_{m-1}, \ldots, a_1, a_0)$. Addition in GF($2^m$) is nothing else than component-wise XOR, while the implementations of multiplication and inversion depend on the basis chosen (see Section 3). The two most common types of bases used in EC cryptography are the *polynomial (standard) basis* and the *normal basis* [18].

3. Arithmetic in Binary Extension Fields

Finite fields of characteristic two are attractive to implementers due to the "carry-free" arithmetic. Another advantage is the availability of different equivalent basis representations of the field, which can be adapted and optimized for the computational environment at hand [2]. A *polynomial basis* is a basis of the form $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, where α is a root of an *irreducible polynomial* $p(t)$ of degree m with coefficients from GF(2). Any element of the finite field GF(2^m) can be expressed as a binary polynomial of degree at most $m-1$. When using a polynomial basis representation, the multiplication of two field elements is performed modulo the irreducible polynomial $p(t)$.

In the remainder of this section we discuss arithmetic operations in GF(2^m) by means of polynomial basis representation. As mentioned before, the elements of GF(2^m) are the binary polynomials

of degree up to $m-1$. Thus, we can write $a(t) \in \mathrm{GF}(2^m)$ according to Equation (6). The irreducible polynomial $p(t)$ has a degree of exactly m, i.e. $p(t) = t^m + \sum_{i=0}^{m-1} p_i \cdot t^i$. Research on efficient implementation of arithmetic operations in $\mathrm{GF}(2^m)$ has a long history. Here we outline the fundamental algorithms for addition, multiplication, squaring, reduction, and inversion.

$$a(t) = \sum_{i=0}^{m-1} a_i \cdot t^i = a_{m-1} \cdot t^{m-1} + \cdots + a_1 \cdot t + a_0 \quad \text{with } a_i \in \{0,1\} \tag{6}$$

Addition: The addition of two elements of $\mathrm{GF}(2^m)$ is performed by adding the coefficients modulo 2, which is nothing else than bit-wise XOR-ing the coefficients of equal powers of t.

$$a(t) + b(t) = \sum_{i=0}^{m-1} (a_i + b_i \bmod 2) \cdot t^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) \cdot t^i$$

Thus, the addition of two elements in $\mathrm{GF}(2^m)$ is realized by m additions in the subfield GF(2). There is no carry propagation from less to more significant bits.

Multiplication: Multiplication in $\mathrm{GF}(2^m)$ involves multiplying the two binary polynomials and then finding the residue modulo the field polynomial $p(t)$. A simple way to compute the product $a(t) \cdot b(t)$ is to scan the coefficients of $b(t)$ from b_{m-1} to b_0 and add the partial-product $a(t) \cdot b_i$ to a running sum. The partial-product $a(t) \cdot b_i$ is either 0 (if $b_i = 0$) or the multiplicand-polynomial $a(t)$ (if $b_i = 1$). After each partial-product addition, the intermediate result must be multiplied by t (i.e. shifted one bit to the left) to align it for the next partial-product.

$$r(t) = a(t) \cdot b(t) = a(t) \cdot \sum_{i=0}^{m-1} b_i \cdot t^i = \sum_{i=0}^{m-1} a(b) \cdot b_i \cdot t^i \tag{7}$$

$$= [[\cdots [[a(t) \cdot b_{m-1}] \cdot t + a(t) \cdot b_{m-2}] \cdot t + \cdots + a(t) \cdot b_2] \cdot t + a(t) \cdot b_1] \cdot t + a(t) \cdot b_0 \tag{8}$$

This classical technique is referred to as "shift-and-xor" algorithm [27]. A software implementation of this algorithm employs a very large number of shift and XOR operations since the binary polynomials are represented by arrays of computer words. Han *et al.* [12] introduced an improved version of the classical algorithm which requires fewer shift operations. Other variations are based on a window technique and need some extra storage for multiples of $a(t)$ in order to reduce the number of both shift and XOR operations [14, 13, 20].

Reduction: Once the product $r(t) = a(t) \cdot b(t)$ has been formed it needs to be reduced modulo $p(t)$ to obtain the final result (i.e. to get a polynomial of degree less than m). Reduction modulo $p(t)$ is equivalent to replacing all occurrences of $p(t)$ by 0. Thus, considering $p(t) \equiv 0 \bmod p(t)$ leads to

$$t^m \equiv \sum_{i=0}^{m-1} p_i \cdot t^i \bmod p(t) \text{ over GF(2)} \quad \text{and} \quad t^k \equiv \sum_{i=0}^{m-1} p_i \cdot t^{i+k-m} \bmod p(t) \text{ for } k \geq m \tag{9}$$

since $p(t)$ is a polynomial of degree m. The first congruence of Equation (9) can be used to reduce a degree-m polynomial $x(t)$ modulo $p(t)$ via substitution, i.e. via addition of $p(t)$ to $x(t)$. The degree of the resulting polynomial is $m-1$ since $x_m = p_m = 1$ and consequently $x_m \oplus p_m = 0$.

In general, the product-polynomial $r(t) = a(t) \cdot b(t)$ has a degree of $2m-2$. The reduction of $r(t)$ modulo $p(t)$ is easily realized through shifts and additions (XORs). For instance, a polynomial of

degree $k \geq m$ can be reduced to a polynomial of degree $k-1$ by addition of $t^{k-m} \cdot p(t)$. This operation is repeated for decreasing values of k until the degree of $r(t)$ is less than m. In other words, we can use the second congruence of Equation (9) to reduce $r(t)$ mod $p(t)$ one bit at a time, starting with the leftmost bit r_{2m-2}. The basic idea is to zero out the upmost $m-1$ coefficients of $r(t)$, which is achieved by adding $r_k \cdot t^{k-m} \cdot p(t)$ to $r(t)$ for all values of k ranging from $2m-2$ down to m.

Squaring: Squaring in GF(2^m) is a linear operation and much faster than conventional multiplication in GF(2^m) [27]. In the formula for squaring a binomial, $(a+b)^2 = a^2 + 2ab + b^2$, the cross-term vanishes modulo 2 and the square reduces to $a^2 + b^2$. To square a binary polynomial $a(t)$ of degree $m-1$, we compute

$$a(t)^2 = \left(\sum_{i=0}^{m-1} a_i \cdot t^i \right)^2 = \sum_{i=0}^{m-1} a_i \cdot t^{2i} \text{ over GF}(2) \tag{10}$$

The bit-string representation of $a(t)^2$ is obtained by inserting a 0 between consecutive bits a_i and a_{i-1} of the original bit-string $a(t)$. For example, $t^3 + t + 1$ is represented as 1011, and the square is 1000101. This can be done very quickly with the help of a pre-computed look-up table. The squared polynomial then has to be reduced modulo $p(t)$ to obtain the final result.

Inversion: Inversion is the most computationally intensive, and hence slowest arithmetic operation in GF(2^m). There are two basic techniques for inversion of an element $a(t)$ of GF(2^m). The first approach is based on Fermat's theorem and computes the inverse a^{-1} by exploiting the cyclic nature of GF(2^m).

$$a^{-1} = a^{2^m-2} \quad \forall a \in \text{GF}(2^m) \backslash \{0\} \tag{11}$$

Another option is to use the Extended Euclidean Algorithm (EEA) or one of its optimized variants [16]. The most efficient of these techniques is the Almost Inverse Algorithm (AIA) which defers the numerous single-bit shifting operations present in the EEA until the end so that they can be performed all at once in a much more efficient manner [27]. Since the output of the AIA is the "almost" inverse $t^x \cdot a^{-1}$, a Montgomery-like conversion is necessary to get the actual inverse. Note that EC cryptosystems do not depend on the costly inversion operation when projective coordinates are used.

4. Design of a Unified Multiplier Datapath

The shift-and-xor method for polynomial multiplication delivers poor performance in software because of its bit-level characteristic. Thus, the authors of references [24, 17] proposed to equip processors with a fast hardware multiplier for word-level multiplication of binary polynomials. Providing a `MULGF2` instruction to better support the multiplication in binary fields of large order can significantly improve the overall performance of EC cryptosystems. In this section we demonstrate that we can simply integrate polynomial multiplication into the datapath of an integer multiplier.

The first approach for combining the hardware of a conventional multiplier with a multiplier for finite fields GF(2^m) was introduced by W. Drescher *et al* [5]. They examined the implementation of arithmetic in GF(2^m), $m \leq 8$, on a DSP datapath for signal coding. In recent years, polynomial arithmetic was also integrated into the datapath of modulo multipliers found in cryptographic coprocessors, resulting in so-called *unified* multiplier datapaths [8, 26].

Figure 1 shows a new design of a dual-field adder (DFA) for standard cell implementation. A DFA is basically a full adder with the capability of doing addition both with and without carry [26]. It has an extra input called *fsel* (field select) that allows to switch between integer mode and polynomial mode.

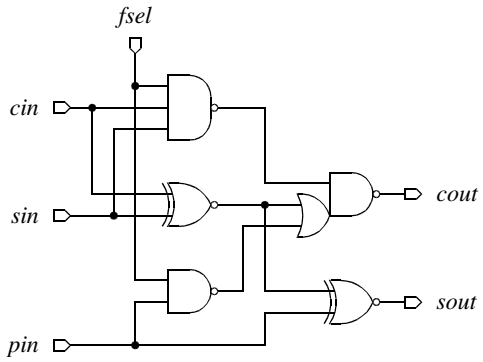

Figure 1. Dual-field adder (DFA)

When $fsel = 1$, the DFA operates like a conventional full adder, i.e. it performs a bit-wise addition with carry, which enables the multiplier to do integer arithmetic. On the other hand, when $fsel = 0$ (polynomial mode), the output *cout* is forced to 0 regardless of the values of the inputs. The output *sout* produces the result of a bit-wise XOR of the input values. Consequently, we can use exactly the same hardware (i.e. full adders) for the addition of both integers and binary polynomials.

The field selection logic of a DFA causes a slight increase in area and delay compared to a standard full adder. We point out that in polynomial mode ($fsel = 0$) the outputs of the two NAND gates are forced to 1 and the output *cout* is always 0. Hence, only the two XNOR gates are active and contribute to power consumption. Therefore, it can be expected that the a properly designed unified multiplier consumes significantly less power in polynomial mode than in integer mode. Besides full adders, also higher order counters and compressors can be adapted for dual-field arithmetic. A unified radix-4 partial-product generator for integers and binary polynomials was presented in [10]. In summary, polynomial arithmetic can be integrated into a multiplier datapath for integers without significant increase of the critical path. The overall silicon area of a unified multiplier is only marginally larger than that of a conventional multiplier.

5. New Arithmetic Algorithms for Binary Extension Fields

In this section we investigate the potential of word-level arithmetic algorithms for GF(2^m) assuming that the `MULGF2` instruction has been made available. We focus on algorithms for multiplication, squaring, and modulo reduction of binary polynomials. Word-level algorithms are generally faster than the "conventional" algorithms discussed in Section 3 because they avoid inefficient bit-level instructions like left-shifts. The performance gain results from the processing of polynomials in a word-by-word fashion instead of operating on single bits at a time.

We start with introducing the basic notation that will be used in this section. In Subsection 2.2 we mentioned that a binary polynomial of degree $m-1$ can be written as a bit string of length m, i.e. $a(t) = (a_{m-1}, \ldots, a_1, a_0)$. Here we split the bit string into s words containing w bits each, whereby w denotes the native word-size of the processor (e.g. 8, 16, 32 or 64 bits) and $s = \lceil m/w \rceil$. These word-polynomials are represented by $\tilde{a}_k$ with $0 \leq k < s$ and we can write $a(t)$ in terms of an array of s single-precision (w-bit) words: $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_1, \tilde{a}_0)$. Equation (12) shows the relation between $a(t)$ and the word-polynomials $\tilde{a}_k$. The most significant word $\tilde{a}_{s-1}$ is padded with zeros if necessary.

$$a(t) = \sum_{k=0}^{s-1} \tilde{a}_k \cdot t^{k \cdot w} = \tilde{a}_{s-1} \cdot t^{(s-1) \cdot w} + \cdots + \tilde{a}_1 \cdot t^w + \tilde{a}_0 \quad \text{with} \quad \tilde{a}_k = \sum_{i=0}^{w-1} a_{i+kw} \cdot t^i \tag{12}$$

Algorithm 1. Pencil-and-paper method

Input: Two binary polynomials $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_0)$, $b(t) = (\tilde{b}_{s-1}, \ldots, \tilde{b}_0)$ consisting of s words each.
Output: Product $r(t) = a(t) \cdot b(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_0)$.

1: $r(t) \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $\quad \tilde{u} \leftarrow 0$
4: $\quad$ **for** j from 0 by 1 to $s-1$ **do**
5: $\quad\quad (\tilde{u}, \tilde{v}) \leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$
6: $\quad\quad \tilde{r}_{i+j} \leftarrow \tilde{v}$
7: $\quad$ **end for**
8: $\quad \tilde{r}_{s+i} \leftarrow \tilde{u}$
9: **end for**

Algorithm 2. Comba's method for bin. polynomials

Input: Two binary polynomials $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_0)$, $b(t) = (\tilde{b}_{s-1}, \ldots, \tilde{b}_0)$ consisting of s words each.
Output: Product $r(t) = a(t) \cdot b(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_0)$.

1: $(\tilde{u}, \tilde{v}) \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $\quad$ **for** j from 0 by 1 to i **do**
4: $\quad\quad (\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
5: $\quad$ **end for**
6: $\quad \tilde{r}_i \leftarrow \tilde{v}$
7: $\quad \tilde{v} \leftarrow \tilde{u}$, $\tilde{u} \leftarrow 0$
8: **end for**
9: **for** i from s by 1 to $2s-2$ **do**
10: $\quad$ **for** j from $i-s+1$ by 1 to $s-1$ **do**
11: $\quad\quad (\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
12: $\quad$ **end for**
13: $\quad \tilde{r}_i \leftarrow \tilde{v}$
14: $\quad \tilde{v} \leftarrow \tilde{u}$, $\tilde{u} \leftarrow 0$
15: **end for**
16: $\tilde{r}_{2s-1} \leftarrow \tilde{v}$

Algorithm 3. Word-level squaring

Input: Binary polynomial $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_0)$ consisting of s words.
Output: Square $r(t) = a(t) \cdot a(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_0)$.

1: **for** i from 0 by 1 to $s-1$ **do**
2: $\quad (\tilde{u}, \tilde{v}) \leftarrow \tilde{a}_i \otimes \tilde{a}_i$
3: $\quad \tilde{r}_{2i} \leftarrow \tilde{v}$
4: $\quad \tilde{r}_{2i+1} \leftarrow \tilde{u}$
5: **end for**

Algorithm 4. Left-to-right reduction modulo $t^d \cdot p(t)$

Input: $r(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_0)$ consisting of $2s$ words, and $q(t) = t^d \cdot p_0(t) = (\tilde{q}_{s-2}, \ldots, \tilde{q}_0)$ consisting of $s-1$ words.
Output: Reduced polynomial $r(t)$ mod $(t^d \cdot p(t))$.

1: **for** i from $2s-1$ by 1 down to s **do**
2: $\quad \tilde{u} \leftarrow 0$
3: $\quad$ **for** j from 0 by 1 to $s-2$ **do**
4: $\quad\quad (\tilde{u}, \tilde{v}) \leftarrow \tilde{r}_{(i-s)+j} \oplus (\tilde{r}_i \otimes \tilde{q}_j) \oplus \tilde{u}$
5: $\quad\quad \tilde{r}_{(i-s)+j} \leftarrow \tilde{v}$
6: $\quad$ **end for**
7: $\quad \tilde{r}_{i-1} \leftarrow \tilde{r}_{i-1} \oplus \tilde{u}$
8: **end for**
9: **return** $(\tilde{r}_{s-1}, \tilde{r}_{s-2}, \ldots, \tilde{r}_0)$

Algorithm 5. Fast reduction modulo $t \cdot (t^{191} + t^9 + 1)$

Input: $r(t) = (\tilde{r}_{23}, \tilde{r}_{22}, \ldots, \tilde{r}_0)$ consisting of 24 words, and $q(t) = t \cdot (t^9 + 1) = t^{10} + t = 0\text{x}0402, w = 16$.
Output: Reduced result $r(t)$ mod $(t \cdot (t^{191} + t^9 + 1))$.

1: **for** i from 23 by 1 down to 12 **do**
2: $\quad (\tilde{u}, \tilde{v}) \leftarrow \tilde{r}_i \otimes 0\text{x}0402$
3: $\quad \tilde{r}_{i-12} \leftarrow \tilde{r}_{i-12} \oplus \tilde{v}$
4: $\quad \tilde{r}_{i-11} \leftarrow \tilde{r}_{i-11} \oplus \tilde{u}$
5: **end for**
6: **return** $(\tilde{r}_{11}, \tilde{r}_{10}, \ldots, \tilde{r}_0)$

5.1. Polynomial Multiplication

Polynomial multiplication is essentially identical to integer multiplication, except that all carries are suppressed [16]. Thus, the well-known algorithms multiple-precision multiplication are applicable to binary polynomials as well. The efficiency of polynomial multiplication is the dominant performance constraint of an EC cryptosystem, especially in projective representation [28]. Any effort spent optimizing this operation is well spent.

Algorithm 1 illustrates the well-known pencil-and-paper method for computing a multiple-precision multiplication over GF(2). The algorithm is taken from [16] and adapted for the multiplication of binary polynomials. EC cryptography involves arithmetic on operands that may be some hundreds of bits long. Since no conventional processor supports calculations of this size, we are forced to represent the operands as multi-word data structures (i.e. arrays of w-bit polynomials). Arithmetic operations on these arrays are performed by means of software routines which manipulate the individual words using instructions like `MULGF2` and `XOR`. Algorithm 1 consists of an outer loop and a relatively simple

inner loop which does the bulk of computation. Any iteration of the inner loop carries out an operation of the form $(\tilde{u},\tilde{v}) \leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$. Characters $\otimes$ and $\oplus$ denote the `MULGF2` and `XOR` instruction, respectively. The tuple $(\tilde{u},\tilde{v})$ represents a double-precision quantity of the form $u(t) \cdot t^w + v(t)$, i.e. a polynomial of degree $2w-1$. Both $\tilde{u}$ and $\tilde{v}$ have a degree of $w-1$, whereby $\tilde{u}$ contains the w most significant coefficients of $(\tilde{u},\tilde{v})$ and $\tilde{v}$ the w least significant coefficients. The multiple-precision multiplication is entirely performed with word-level multiplications (`MULGF2`) and additions (`XOR`). No other arithmetic/logical operations like bit manipulations or left-shifts are required apart from address increments for $\tilde{a}_j$ and $\tilde{b}_i$.

Comba's method (Algorithm 2) forms the product $r(t)$ by computing each word $\tilde{r}_i$ of the result at a time, starting with the least significant word $\tilde{r}_0$ [4]. The partial-products $\tilde{a}_j \otimes \tilde{b}_i$ are processed in a "column-by-column" fashion instead of the "row-by-row" approach which is used in the pencil-and-paper method. Comba's method is generally applied to implement long integer multiplication on processors with a multiply/accumulate unit (e.g. digital signal processors) [7]. Algorithm 2 shows an adapted variant for polynomial multiplication. By keeping the running sum $(\tilde{u},\tilde{v})$ in a register pair, Comba's method minimizes the number of `STORE` instructions. Each iteration of the inner loop of Algorithm 1 requires three memory accesses in order to load the values of $\tilde{a}_j$ and $\tilde{r}_{i+j}$, and write-back the result to $\tilde{r}_{i+j}$. Comba's method eliminates the write-back operation by changing the order of partial-product generation/accumulation such that each word of $r(t)$ is computed in its entirety sequentially. The drawbacks of Comba's method are the reversed addressing of the words of $a(t)$ and $b(t)$ along with the more complicated loop control. However, both methods execute s^2 `MULGF2` instructions and $2s^2$ `XOR`s. It is very difficult to state with confidence which method is better since their actual running times depend on architectural details like addressing modes. For instance, Comba's method is likely to perform better than the conventional pencil-and-paper method when the processor supports an auto-increment/decrement addressing mode. In such case the computation of the addresses of $\tilde{a}_j$, $\tilde{b}_{i-j}$ comes for free.

5.2. Polynomial Squaring

Squaring a binary polynomial is a linear operation and hence requires only s `MULGF2` instructions assuming that the polynomial consists of s words. Algorithm 3 shows the word-level squaring of a binary polynomial. The result consists of $2s$ words and needs to be reduced modulo the irreducible polynomial $p(t)$. In a "conventional" software implementation, the `MULGF2` instruction is emulated by table look-up. However, storage of a look-up table with 8-bit entries would require 512 bytes in memory, which is quite a lot for smart cards.

5.3. Polynomial Modular Reduction

In this section we discuss the reduction of a polynomial of degree $2m-2$ ($2s$ words), such as obtained from the product of two polynomials of degree $m-1$ (s words), modulo an irreducible polynomial $p(t)$. We introduce a novel word-oriented reduction technique based on the availability of a `MULGF2` instruction. Instead of working on one bit at a time, our approach is to work on w-bit words at a time, reducing the degree of $r(t)$ by w in each step. An inspection of polynomial reduction techniques proposed in the literature reveals an evident categorization depending on the computational direction in which the reduction proceeds. Right-to-left methods (LSB first) like Montgomery reduction [17] have some advantages for fast hardware implementation and can be mapped to systolic arrays. On the other hand, the left-to-right methods (MSB first) do not incur a scaling factor and require no operand conversion. Algorithm 4 shows a novel method to perform a word-oriented reduction of binary poly-

nomials. Unlike to Montgomery reduction, the words of the polynomial to be reduced are considered from left to right, i.e. from more to less significant words. The algorithm basically implements a word-level variant of the reduction method reported in [20] combined with the principle of "partial reduction" introduced in [11]. Note that Algorithm 4 imposes some restrictions on the form of the irreducible polynomial $p(t)$, i.e. it does not work for any irreducible polynomial but for those which are of interest in EC cryptography. More precisely, the algorithm requires an irreducible polynomial of form $p(t) = t^m + p_0(t)$ where the degree of polynomial $p_0(t)$ is less than $m-w$ (see [20]). This means nothing else than that the coefficients p_{m-1} to p_{m-w} must be all zero, whereby w denotes the word-size of the processor. Thus, we have

$$p(t) = t^m + p_0(t) = t^m + \sum_{i=0}^{(m-w)-1} p_i \cdot t^i \tag{13}$$

with polynomial $p_0(t)$ representing the least significant $m-w$ coefficients of $p(t)$. The restriction on the form of $p(t)$ causes no problems in practice given that most standards recommend to use polynomials which have as few terms as possible and that those terms are of the smallest possible degree. Since the reduction proceeds from left to right (i.e. MSB first) it is advantageous to left-align the polynomial $p_0(t)$ to register boundaries. We can simply achieve this through multiplying p_0 by the constant factor t^d, $d = s \cdot w - m$. For instance, when $m = 191$ and we work on a 16-bit machine (i.e. $w = 16$) then we get $s = \lceil m/w \rceil = 12$ and consequently $d = 1$. Algorithm 4 denotes the aligned representation of $p_0(t)$ as $q(t) = \{\tilde{q}_{s-2}, \ldots, \tilde{q}_0\}$.

$$q(t) = t^d \cdot p_0(t) = \sum_{i=0}^{(m-w)-1} p_i \cdot t^{i+d},\ d = s \cdot w - m \tag{14}$$

Polynomial $q(t)$ consists of (at most) $s-1$ words and is a constant through the point multiplication. Algorithm 4 works by zeroing out the most significant word of $r(t)$ in each iteration of the outer loop. A chosen multiple of polynomial $q(t)$ is added to $r(t)$ which lowers the degree of $r(t)$ by w. This is possible because the degree of $p_0(t)$ is less than $m-w$. The algorithm operates on $r(t)$ "in place", and therefore no extra storage for the reduced result is needed.

Another feature of Algorithm 4 is that it employs the principle of "incomplete" reduction [11] in order to avoid bit-level operations. Instead of reducing $r(t)$ to a polynomial of degree less than m, $r(t)$ is only partially reduced until it fits into s words, resulting in a congruent polynomial of degree up to $s \cdot w - 1$. Note that any element of GF(2^m) represents a residue class modulo $p(t)$, i.e. for any $a(t) \in$ GF(2^m) there exist congruent polynomials over GF(2) of a degree $\geq m$. Performing addition and multiplication with incompletely reduced but congruent polynomials again results in a congruent polynomial. That way, we avoid unnecessary bit-level reductions until the actual output of a point multiplication is produced. The "final" reduction of an incompletely reduced polynomial of degree $s \cdot w - 1$ to a congruent one of degree less than m can be performed with the bit-level reduction.

Left-to-right reduction is especially efficient when $p(t)$ is a polynomial of low weight (e.g. a trinomial or pentanomial) where all non-zero coefficients of $q(t)$ are located within the least significant word. An example for such case is $p(t) = t^{191} + t^9 + 1$, the irreducible polynomial for the field GF(2^{191}) as specified in [1]. Given a word-size of $w = 16$, the aligned representation of $p_0(t) = t^9 + 1$ is $q(t) = t \cdot (t^9 + 1) = t^{10} + t$, a polynomial which fits into a single word. Algorithm 5 illustrates the pseudo-code for performing a left-to-right reduction modulo $p(t) = t^{191} + t^9 + 1$. The product polynomial $r(t)$ consists of 12 words containing 16 bits each ($w = 16$). We point out that a properly selected irreducible polynomial reduces the complexity of left-to-right reduction from $\mathcal{O}(s^2)$ to $\mathcal{O}(s)$. In other

Arithmetic operation	# LOAD	# XOR	# MULGF2	# STORE	# cycles ($s = 12$)
Addition	$2s$	s	–	s	110
Pencil-and-paper multiplication	$2s^2 + s$	$2s^2$	s^2	s^2	–
Comba multiplication	$2s^2$	$2s^2$	s^2	$2s$	2100
Word-level squaring	s	–	s	$2s$	120
Left-to-right reduction	$2s^2$	$2s^2 - s$	$s^2 - s$	s^2	–
Fast reduction (e.g. trinomial)	$2s$	$2s$	s	$2s$	200

Table 1. Workload characteristics of the word-oriented arithmetic algorithms

words, polynomial reduction is a linear operation (i.e. requires only s multiplications) when the non-zero coefficients of $q(t)$ have a degree of less than w. Fortunately, most standards for EC cryptography recommend to use either a trinomial or a pentanomial as irreducible polynomial whereby the degrees of the non-zero terms should be as small as possible.

6. Experimental Results and Discussion

The most important feature of the arithmetic algorithms presented in the previous section is that they avoid bit-level operations but employ word-level operations which are intrinsically more efficient, particularly when the word-size w is large. All algorithms are entirely performed with single-precision multiplications and additions over GF(2), i.e. `MULGF2` and `XOR` instructions. No other arithmetic or logical operations are required except of increment/decrement of addresses and loop count variables. In the following we evaluate the execution time of a 191-bit ECDSA signature on a 16-bit RISC-like processor. We start by providing a workload characterization of the word-level algorithms.

6.1. Workload Characteristics

Table 1 summarizes the frequency of basic instructions like `MULGF2` and `XOR` depending on the number of words s. Public-key cryptosystems are generally characterized by a significant number of memory accesses (`LOAD` and `STORE` instructions) since the long operands can not be kept in the register file [6]. A first-order approximation of the workload characteristics can be made by simply counting the total number of `MULGF2`, `XOR`, `LOAD`, and `STORE` instructions in terms of the input size s. The entries in Table 1 do not represent an exact workload characterization since the effects of address calculation, increment of loop count variables, branch instructions, and the like are not considered. However, these effects depend on details of the actual implementation. For instance, the address calculation comes for free if the processor supports an auto-increment/decrement addressing mode. The loop overhead can be eliminated by loop unrolling.

The inner loop of the pencil-and-paper multiplication (Algorithm 1) is iterated s^2 times. In any iteration, two `XOR` instructions and one `MULGF2` are executed. Operand $\tilde{b}_i$ does not change during the iterations of the inner loop and can be kept in a general-purpose register. It is also advantageous to hold the quantities $\tilde{u}$ and $\tilde{v}$ in a register pair. That way, the number of memory accesses is reduced to two `LOAD` operations (for operands $\tilde{r}_{i+j}$, $\tilde{a}_j$) and one `STORE` (for $\tilde{r}_{i+j}$). Comba's method (Algorithm 2) performs the storage of the result in the outer loop and hence requires only $2s$ `STORE` operations instead of s^2. However, the number of `XOR` and `MULGF2` instructions is the same as with the pencil-and-paper method. The other arithmetic operations (addition, squaring, and fast reduction) can be performed in linear time; the corresponding algorithms consist of one loop which is iterated s times. Multiplication is therefore by far the most computation-intensive arithmetic operation in GF(2^m).

6.2. Simulation Results

We implemented a functional model of a 16-bit RISC-like processor with a simple single-issue pipeline. The processor can execute the usual arithmetic/logical instructions in one clock cycle. We count two clock cycles for memory accesses (`LOAD`/`STORE` instructions) and four cycles for the `MULGF2` instruction. Our model does not implement a specific instruction set architecture but has many similarities with "real-world" RISC cores. The processor has separate caches for data and instructions (which is the case for most modern processors). Furthermore, we assume that the data cache is sufficiently large to avoid cache miss penalties during the execution of a point multiplication. The processor shall support indexed addressing as well as auto-increment/decrement addressing modes[3].

Note that a concrete implementation of the `MULGF2` instruction may differ from architecture to architecture. Some architectures, like the ARM architecture, allow the output of multiply instructions to be directed to any general-purpose registers. Other architectures, such as MIPS32, specify that the output of a multiplication has to be placed into two result-accumulation registers (referenced by the names `HI` and `LO` in MIPS32). There are two special instructions to transfer the contents of `HI` and `LO` to general-purpose registers. However, without going into detail, we stipulate that the processor can execute a $w \times w$-bit `MULGF2` operation and that the result is available in two general-purpose registers after four cycles. On a MIPS-like architecture this would necessitate a $w \times (w/2)$-bit multiplier.

Now we look at the execution time of a point multiplication $k \cdot P$ on an EC over the field GF(2^{191}). In order to take advantage of the fast left-to-right reduction, we selected $p(t) = t^{191} + t^9 + 1$ as irreducible polynomial. Any element of GF(2^{191}) can be stored in $s = 12$ words when $w = 16$. Simulations performed on our 16-bit processor model showed that a single iteration of the inner loop of Algorithm 2 requires 12 clock cycles. The inner loop contains two `LOAD` instructions (two cycles each), a `MULGF2` (four cycles), and two `XOR`s (one cycle each). In addition, the increment of the loop count variable together with the branch instruction takes another two clock cycles[4]. Address calculation requires no extra cycles since the processor supports an auto-increment/decrement addressing mode. In summary, Comba multiplication of two 12-word polynomials requires about 2100 clock cycles. The execution times of the other arithmetic operations are presented in the last column of Table 1. A multiplication in GF(2^{191}) (i.e. polynomial multiplication including fast reduction) can be performed in 2300 clock cycles, whereas a squaring together with fast reduction requires only 320 cycles.

In order to estimate the overall execution time of $k \cdot P$, let us take an integer k of approximately 190 bits. We use projective coordinates and Montgomery's point multiplication algorithm [19]. Thus, we have to perform $6 \cdot \log_2(k) \approx 1100$ multiplications, $5 \cdot \log_2(k) \approx 950$ squarings, and $3 \cdot \log_2(k) \approx 570$ additions in the underlying finite field GF(2^{191}), which amounts to roughly $3 \cdot 10^6$ clock cycles. The corresponding execution time is about 600 msec when the processor is clocked at 5 MHz. Note that point multiplication is the dominant computational step in EC signature schemes[5]. Therefore, we can estimate that an entire ECDSA signature generation requires no more than 650 msec. This result is comparable to the performance of commercial 16-bit smart cards with a crypto co-processor. Let us consider, for example, the Infineon SLE 66CL160S, a 16-bit smart card processor with an integrated accelerator for EC cryptography [15]. According to [15], the SLE's so-called EC2 Accelerator requires 436 msec to generate an ECDSA signature over the finite field GF(2^{163}) when the clock frequency is 5 MHz. Of course, a fair comparison of our timings with the results of previous work is extremely difficult due to the variety of implementation options and design goals. We point out that the presented

[3] For example, the ARM and PA-RISC architectures support auto-increment addressing modes.

[4] Some processors have "delayed" loads and branches. In many cases, a careful ordering of the instructions may help to fill such delay slots (e.g. the increment of a loop count variable can be used to fill a delay slot).

[5] Typical ECDSA implementations spend more than 95% of the overall execution time for point multiplication.

concepts also confirm the advantages of EC cryptography over RSA and discrete logarithm systems. For instance, it takes approximately $20 \cdot 10^6$ clock cycles to compute a 1024-bit modulo exponentiation on an extended 32-bit RISC core [9].

7. Concluding Remarks

We presented a new approach for efficient implementation of EC cryptography on constrained devices like smart cards. The main idea is to replace the "micro-controller + crypto co-processor" combination found in today's smart cards by a simple RISC processor with architectural enhancements for arithmetic in finite fields GF(2^m). These enhancements include an instruction for single-precision multiplication of binary polynomials (`MULGF2`) and auto-increment/decrement addressing modes, both of which are easy to integrate into common RISC architectures.

The availability of a `MULGF2` instruction allows very efficient implementation of arithmetic in finite fields GF(2^m). We presented new word-level algorithms for multiplication, squaring, and reduction of binary polynomials. These algorithms are substantially faster than the straightforward algorithms based on shift and XOR operations. In order to support our claims, we have implemented a functional model of a 16-bit RISC-like processor with the proposed architectural enhancements. This model was used to evaluate the execution time of the presented algorithms and to verify their correctness. Our simulations indicate that a "conventional" polynomial multiplication without the `MULGF2` instruction is about 6-10 times slower (depending on the window-size of the shift-and-xor multiplication).

The most significant result is that an enhanced 16-bit RISC core provides enough performance to implement EC cryptography in software. We demonstrated that a cryptographically strong 191-bit ECDSA signature can be generated in less than 650 msec when the processor is clocked at 5 MHz. This execution time can be considered as a reference value for the cryptographic performance of an enhanced 16-bit RISC processor. There are a variety of both hardware-related and software-related methods to further increase the performance. An example for the former is a multiply/accumulate (MAC) unit that combines `MULGF2` and `XOR` into one instruction. Software optimization techniques include loop unrolling and the full use of available registers to store intermediate values.

The timings reported in Subsection 6.2 were achieved with the help of only one custom instruction, namely `MULGF2`. Note that the `MULGF2` instruction does not even need an extra functional unit (FU) since polynomial arithmetic can be easily integrated into the datapath of an integer multiplier. The hardware cost of the proposed enhancements is moderate compared to the cost of a cryptographic co-processor; a fully parallel (single-cycle) 16×16-bit multiplier requires just 256 dual-field adders. Another advantage of the presented approach is the high degree of flexibility. Fundamental advances in cryptanalysis (or its associated mathematics) could, hypothetically speaking, endanger a 191-bit EC cryptosystem or make it considerably less secure than expected today. Software-based cryptography can be easily adapted to meet new security requirements. We conclude that the presented architectural enhancements facilitate fast yet flexible implementations of EC cryptography on smart cards, thereby eliminating the need for a cryptographic co-processor.

References

[1] American National Standards Institute (ANSI). X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA), Jan. 1999.

[2] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[3] Certicom Corporation. The elliptic curve cryptosystem for smart cards. White Paper, available for download at `http://www.certicom.com/resources/download/ECC_SC.pdf`, Apr. 1998.

[4] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Oct. 1990.

[5] W. Drescher, K. Bachmann, and G. Fettweis. VLSI architecture for datapath integration of arithmetic over GF(2^m) on digital signal processors. In *Proceedings of the 22nd IEEE Int. Conference on Acoustics, Speech, and Signal Processing (ICASSP '97)*, pp. 631–634. IEEE, 1997.

[6] A. M. Fiskiran and R. B. Lee. Workload characterization of elliptic curve cryptography and other network security algorithms for constrained environments. In *Proceedings of the 5th Annual IEEE Workshop on Workload Characterization (WWC-5)*, pp. 127–137. IEEE, 2002.

[7] J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.

[8] J. R. Goodman and A. P. Chandrakasan. An energy efficient reconfigurable public-key cryptography processor architecture. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, pp. 175–190. Springer Verlag, 2000.

[9] J. Großschädl. Instruction set extension for long integer modulo arithmetic on RISC-based smart cards. In *Proceedings of the 14th Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2002)*, pp. 13–19. IEEE Computer Society Press, 2002.

[10] J. Großschädl. A unified radix-4 partial product generator for integers and binary polynomials. In *Proceedings of the 35th IEEE Int. Symposium on Circuits and Systems (ISCAS 2002)*, pp. 567–570. IEEE, 2002.

[11] N. Gura, H. Eberle, and S. Chang Shantz. Generic implementations of elliptic curve cryptography using partial reduction. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, pp. 108–116. ACM Press, 2002.

[12] Y. Han, P.-C. Leong, P.-C. Tan, and J. Zhang. Fast algorithms for elliptic curve cryptosystems over binary finite field. In *Advances in Cryptology — ASIACRYPT '99*, pp. 75–85. Springer Verlag, 1999.

[13] D. R. Hankerson, J. C. López Hernandez, and A. J. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, pp. 1–24. Springer Verlag, 2000.

[14] M. A. Hasan. Look-up table-based large finite field multiplication in memory constrained cryptosystems. *IEEE Transactions on Computers*, 49(7):749–758, July 2000.

[15] Infineon Technologies AG. SLE 66CL160S short product information. Available for download at `http://www.infineon.com`, Dec. 2001.

[16] D. E. Knuth. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Addison-Wesley, 1998.

[17] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.

[18] R. Lidl and H. Niederreiter. *Finite Fields*. Cambridge University Press, 1996.

[19] J. C. López Hernandez and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems — CHES '99*, pp. 316–327. Springer Verlag, 1999.

[20] J. C. López Hernandez and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. In *Progress in Cryptology — INDOCRYPT 2000*, pp. 203–212. Springer Verlag, 2000.

[21] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[23] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.

[24] E. M. Nahum, S. W. O'Malley, H. K. Orman, and R. C. Schroeppel. Towards high performance cryptographic software. In *Proceedings of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS '95)*, pp. 69–72. IEEE, 1995.

[25] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards (FIPS) Publication 186-2, Feb. 2000.

[26] E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, pp. 277–292. Springer Verlag, 2000.

[27] R. C. Schroeppel, H. K. Orman, S. W. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology — CRYPTO '95*, pp. 43–56. Springer Verlag, 1995.

[28] N. P. Smart. A comparison of different finite fields for elliptic curve cryptosystems. *Computers and Mathematics with Applications*, 42(1-2):91–100, July 2001.