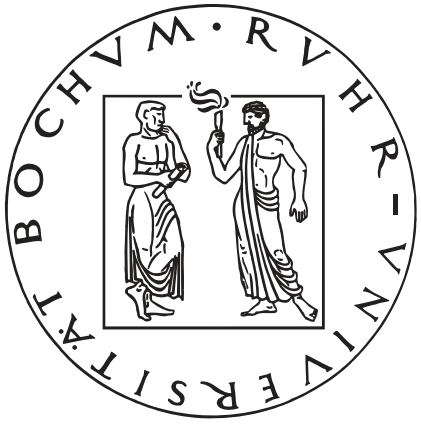
# Implementing Public Key Algorithms on Embedded Systems

Daniel Hamburg

04.11.2002

Diplomarbeit
Ruhr-Universität Bochum




Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

In this thesis we describe the implementation of fast cryptographic algorithms on a Palm OS device. We focus on the DLP problem on $GF(p^m)$ and use Optimal Extension Fields (OEF) and several optimized algorithms for subfield and extension field arithmetic to obtain a reasonable performance.

We use a combination of the Baby-Window, Giant-Window Algorithm for exponentiation and recursive Karatsuba polynomial multiplication to compute exponentiation on the extension field. We also introduce a new method to compute the first iteration of the Frobenius Map on arbitrary OEFs.

The implementation of exponentiation with exponents of 1024-bit (as used for DSA or DH with key size of 1024-bit) results in a timing of 4.55 seconds for an exponentiation with fixed base and 10.43 seconds for an exponentiation with random base. These timings will not be tolerated by most users.

A comparison of the complexity shows that the DLP based cryptography over $GF(2^k)$ and especially over ECC offer a much better performance on Palm OS devices.

# Contents

# List of Tables

# List of Algorithms

# 1 Introduction

## 1.1 Motivation

"While the Internet creates a new cyberspace separate from our physical world, technological advances will enable ubiquitous networked computing in our day-to-day life. The power of this ubiquity will follow from the *embedding* of computation and communications in the physical world that is, embedded devices with sensing and communication capabilities that enable distributed computation. D. Estrin, R. Govindan, and J. Heidemann, 2000 [9]."

Embedded systems based on different operating systems (OS) like Microsoft Pocket PC, Palm OS or Symbian OS have gained a large popularity in the last couple of years. These devices are often used as an additional device to a desktop PC because they are portable, small, light weight, offer a large variety of features like calendar, task manager, etc. and can easily be synchronized with the desktop PC. The small size and light weight of the devices induces rather slow processors (compared to a desktop PC), small memory size (2-64 MB) and small displays (sometimes only monochrome). Because of the wide spreading of embedded systems the need for security for these devices grows vastly. Most of the OS used on these devices offer standard security features based on Password Authentication, which offers a low level of security. A main aspect of implementing

security solutions on embedded systems is the efficiency of the used algorithms. Several seconds for a Diffie-Hellmann 1024-bit Key Exchange is not tolerated by most users.

In this diploma thesis we want to focus on the exponentiation in finite fields which is used for various public-key algorithms, e.g., Diffie Hellmann Key Exchange (DH), El Gamal, Digital Signature Algorithm (DSA) just to name a few. The DH requires two exponentiations to perform a key exchange between the communication partners. The public-key algorithms mentioned above are based on the fact that solving the discrete logarithm problem is believed to be a mathematical hard problem. A key length of about 1000 bits (normally 1024 bits are chosen) for a DH ensures that it is computationally infeasible for an intruder to obtain the secret key. On the other hand the exponentiation needs plenty of time using standard algorithms on constrained devices like embedded systems. In this thesis we will try to find fast algorithms to perform exponentiation without loss of security but fast enough to be accepted by users for standard security operations.

We will focus on using Optimal Extension Fields (OEF) as introduced in [2]. This special finite fields offer various possibilities to improve the performance of computations on the subfield as well as on the extension field without a loss of security. For optimal performance results while working with OEFs, the prime $p$ should be chosen to be close to the word size of the processor to take advantage of the processor's fast integer arithmetic. In this paper we concentrate on Palm OS devices which use a Motorola Dragonball Processor with a word size of 16 bit.

## 1.2 Thesis Outline

This thesis is organized as follows. Chapter 2 gives an overview of Cryptography based on Finite Fields and a short introduction to Processor Adequate Fields and Optimal Extension Fields. Chapter 3 describes the algorithms we use to perform operations on

the subfield and extension field including the ones for exponentiation on the extension field. In chapter 4 we will introduce the details of our implementation on a Palm OS device of the algorithms introduced in chapter 3 and the timings we have obtained. Chapter 5 contains the discussion of our implementation results and a comparison to ECC on Palm devices and to DLP on $GF(2^k)$.

# 2 Previous Work

## 2.1 Cryptography Based on Finite Fields

This section gives a quick survey of public-key systems based on the discrete logarithm problem (DLP). A more detailed description can be found in [6], [17] and [22].

We assume that $G$ is a finite abelian group with $\#G$ elements which is cyclic, i.e., generated by an element $g$ such that $G = \langle g \rangle$. To derive a cyclic group $G$ from an arbitrary one $O$ we select an element $g \in O$ of large order, where the order of an element $g$ is the smallest positive number $e$ such that $g^e = 1$. We use the subgroup which is generated by $g$, $G = \langle g \rangle$, as the cyclic group for the cryptographic system. The number of elements in the group $G$ is called the order of $G$ and it is equal to $e$.

A polynomial ring $R[x]$ is the set of all polynomials in the indeterminate $x$ which have coefficients from a commutative ring $R$. The number of elements of $R[x]$, $\#R$ is called the order of $R$. The two operations $+, \cdot$ are performed using standard polynomial addition and multiplication with coefficient arithmetic performed in the ring $R$. All elements of the polynomial ring $R[x]$ can be represented as polynomials of the form $A(x) = \sum_{i=0}^{m} a_i \cdot x^i$ with $a_i \in R$, $a_m \neq 0$ and $m \geq 0$ the degree of $A$. $A \in R[x]$ is called irreducible if the relation $A = G \cdot H$, with $G, H \in R[x]$, implies that either $G$ or $H$ is a constant.

A field is a commutative ring $(R, +, \cdot)$ in which all non-zero elements have multiplicative inverses. Finite fields are also known as Galois Fields (GF). The order $q$ of a Galois Field is always a prime power $q = p^m$ with $m \in \mathbb{N}$, $p$ prime. Two GFs having the same order are isomorphic. For each $p$ prime and $m$ integer, there exists up to an isomorphism exactly one GF with order $q = p^m$. We write this field as $GF(p^m)$ and call $p$ the characteristic of the field.

$GF(p)$ is a subfield of $GF(p^m)$. $GF(p)$ can be represented as $\mathbb{Z}_p := \mathbb{Z}/p\mathbb{Z}$, the field of integers modulo $p$. All operations in the subfield, like addition and multiplication are performed modulo $p$, e.g., the addition of 2 and 4 in the subfield $GF(5)$ is performed as follows: $(2 + 4) \bmod 5 = 1$.

For $m \geq 2$ we first consider the polynomial ring $R[x]$ with $R = \mathbb{Z}_p$ and choose an irreducible polynomial $f(x) \in R[x]$ of degree $m$. $f(x)$ is called the field polynomial. For any $p, m$ there exists at least one such polynomial. For example $f(x) = x^3 + x + 1$ is irreducible in $\mathbb{Z}_2[x]$ (that is $m = 3$, $p = 2$). The elements of $GF(p^m)$ can be represented as those of $R[x]$ modulo $f$: $GF(p^m) = \mathbb{Z}_p[x]/f$. Any polynomial in $\mathbb{Z}_p[x]$ is congruent modulo $f$ to a unique polynomial of degree at most $m-1$. So $GF(2^3)$ can be represented as $\{0, 1, x, x^2, x+1, x^2+x, x^2+x+1, x^2+1\}$ with the usual addition and multiplication modulo $f(x) = x^3 + x + 1$. For further reading regarding finite fields we recommend [7], [12] and [17].

For our purpose it is important that the field operations can be efficiently implemented, while computing discrete logarithms in this field is computationally impossible with current technology and algorithms. The DLP is as follows: Given $g$ and $b \in GF(p^m)$ find the smallest positive integer $x$ such that $g^x = b$. Someone who can solve the DLP can also break the system. As an example of how to use the DLP for a public-key system, we will give a short description of the Diffie-Hellman key exchange protocol. There are also digital signature methods based on the DLP, e.g., the Digital Signature Algorithm

(DSA).

Assume Alice and Bob want to share a secret integer, but their only way to communicate is over an insecure connection. This integer can be used as a secret key for a conventional cryptosystem in subsequent communication sessions. Alice and Bob (and any intruder) know the field $GF(p^m)$ and an element $g \in GF(p^m)$ of large known order. Now Alice and Bob do the following:

**Algorithm 1** Diffie-Hellman key exchange

1: Alice generates a random integer $x_A \in \{1, \ldots, \#G - 1\}$ and sends Bob the element $g^{x_A}$.
2: Bob generates a random integer $x_B \in \{1, \ldots, \#G - 1\}$ and sends Alice the element $g^{x_B}$.
3: Alice computes $(g^{x_B})^{x_A} = g^{x_A x_B}$.
4: Likewise, Bob computes $(g^{x_A})^{x_B} = g^{x_A x_B}$.

It can easily be seen that a third person, who can eavesdrop on the channel, knows $GF(p^m)$, $g$, $g^{x_A}$, $g^{x_B}$ and who can solve the DLP, can also recover $g^{x_A x_B}$. Again, solving the DLP means computing $x_A$ or $x_B$ from $GF(p^m), g, g^{x_A}$ and $g^{x_B}$. It is believed for most finite fields in use in cryptography recovering the Diffie-Hellman key and solving the DLP are equivalent. However, if there is a fast way to solve the DLP in $GF(p^m)$, then the cryptosystems based on it are not secure. Today, discrete logarithms in finite fields can be found in sub-exponential time, using the index-calculus method as described in [17].

## 2.2 Processor Adequate Fields and Optimal Extension Fields

Processor Adequate Finite Fields (PAFF) as introduced in [18] are a special type of finite fields $GF(p^m)$ where the prime $p$ of the subfield $GF(p)$ is chosen to be about the size of the word length of the processor of the targeted system. If $w$ is the word size of

the processor, then the following inequation must be true:

$$w/2 \leq \lfloor log_2(p) \rfloor < w \tag{2.1}$$

The basic idea is that integer operations for integers not larger than the word size of the processor (CPU) can be performed fast. While $p = 2$ is often chosen for hardware based implementations (e.g. on smart cards) the word size of many systems is between 16 and 64 bit. When CPUs can perform fast operations with word sizes other than the one declared by the manufacturer one can choose this "effective" word size as $w$.

Optimal Extension Fields (OEF) are a special type of finite fields introduced by Bailey and Paar in [2] and [3]. They are finite fields $GF(p^m)$ with the following properties:

1. $p$ is a pseudo-Mersenne prime. That means that $p$ is a prime number of the form $p = 2^n \pm c$, with c a positive integer and $log_2(c) \leq \lfloor n \rfloor$

2. There exists an irreducible binomial $P(x) = x^m - \omega$ over $GF(p)$

For optimal performance results while working with OEFs, the prime $p$ should be chosen to be close to the word size of the processor to take advantage of the processor's fast integer arithmetic. In addition to the general definition of OEFs given above there are two special types of OEFs

1. Type I OEF: $p = 2^n - 1$

2. Type II OEF: the irreducible binomial has the form: $P(x) = x^m - 2$

While Type I OEFs allow fast subfield modular reduction with very low complexity, Type II OEFs allow a reduction in the complexity of extension field modular reduction because multiplication by 2 can be performed as left shift. Instead of $\omega = 2$ we can also choose $\omega = 2^k$, $k \in \mathbb{N}$ where the multiplication with $\omega$ can also be realized as left

shift. The choice of $\omega = 2^k$ is though of no big practical interest because whenever $f(x) = x^m - 2^k$ is irreducible also $f(x) = x^m - 2$ is, which offers the better performance.

# 3 Arithmetic

In this chapter we discuss the algorithms needed to perform exponentiations on a finite field $GF(p^m)$. We will present a method called "Baby-Window Giant-Window Exponentiation" introduced in [1] which helps us to improve the speed of exponentiations. This algorithm requires both operations in the extension field $GF(p^m)$ and in the subfield $GF(p)$.

## 3.1 Subfield Arithmetic

Subfield arithmetic is performed using integers as elements of the subfield $GF(p)$ and modulo arithmetic. Improvements in the subfield help us to get a speed up for the extension field exponentiation because subfield operations are repeated several time to perform operations in the extension field. We need the following subfield operations: addition, multiplication, squaring, inversion and modular reduction.

Addition and multiplication of two elements of the subfield is performed using standard methods. Because the elements of the subfield $GF(p)$ are not larger than the processor's word size and fit into a register, these operations are done by the hardware support.

In the next section we will introduce an algorithm called binary inversion to perform

inversion for elements of $GF(p)$. Furthermore we will consider the Barrett modular reduction to perform reductions on arbitrary finite subfields $GF(p)$, the mixed modular reduction introduced in [1] to perform reductions on PAFFs and an algorithm to perform fast modular reduction on OEFs.

### 3.1.1 Binary Inversion

The binary inversion as introduced in [5] computes the inverse of a non-zero element $a \in$ [1,p-1] of the subfield $GF(p)$ using a variant of the Extended Euclidean Algorithm. Algorithm 2 shows a possible implementation of the binary inversion.

### 3.1.2 Barrett Modular Reduction

The Barrett modular reduction improves the performance of subfield reduction compared to a division with remainder. It computes $x$ mod $m$, $x, m \in \mathbb{N}$ where $x, m$ are given in base $b$ representation with $x = (x_{2k-1} \ldots x_1 x_0)_b$ and $m = (m_{k-1} \ldots m_1 m_0)_b$ with $m_{k-1} \neq 0$. So, $x$ is at most twice as long as $m$. This restriction does not affect us because we will use the Barrett modular reduction to compute the reduction mod $m$ of a multiplication $x = y \cdot z$ with $y, z \leq m$ so $x$ will satisfy the given condition. The algorithm requires the precomputation of $\mu = \lfloor b^{2k}/m \rfloor$ which needs to be done only once for each $m$. We present Algorithm 3 as described in [17]. If the base $b$ of the integer $x$ and the modulus $m$ is of the form $b = 2^n$ with $n$ an integer, the divisions performed in Algorithm 3 can be performed as right shifts to improve the running time.

### 3.1.3 Mixed Modular Reduction

The so called mixed modular reduction was introduced in [1] to speed up the modular reduction on Processor Adequate Finite Fields (PAFF) which do not offer the same simplifications as OEFs. It is a modified version of the Barrett modular reduction. The

**Algorithm 2** Binary inversion algorithm

**INPUT:** Prime $p$, $a \in$ [1,p-1]
**OUTPUT:** $a^{-1}$ mod p

1: $u \leftarrow a$, $v \leftarrow p$, $A \leftarrow 1$, $C \leftarrow 0$.
2: **while** $u \neq 0$ **do**
3:  **while** $u$ is even **do**
4:  $u \leftarrow u/2$.
5:  **if** $A$ is even **then**
6:  $A \leftarrow A/2$.
7:  **else**
8:  $A \leftarrow (A + p)/2$.
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:  $u \leftarrow u - v$, $A \leftarrow A - C$.
21:  **else**
22:  $v \leftarrow v - u$, $C \leftarrow C - A$.
23:  **end if**
24: **end while**
25: **Return** ($C$ mod $p$).

**Algorithm 3** Barrett Modular Reduction

**INPUT:** positive integers $x = (x_{2k-1} \dots x_1 x_0)_b, m = (m_{k-1} \dots m_1 m_0)_b$ (with $m_{k-1} \neq 0$), and $\mu = \lfloor b^{2k}/m \rfloor$.
**OUTPUT:** $r = x \bmod m$

1: $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor, q_2 \leftarrow q_1 \cdot \mu, q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$.
2: $r_1 \leftarrow x \bmod b^{k+1}, r_2 \leftarrow q_3 \cdot m \bmod b^{k+1}, r \leftarrow r_1 - r_2$.
3: **if** $r < 0$ **then**
4:  $r \leftarrow r + b^{k+1}$.
5: **end if**
6: **while** $r \geq m$ **do**
7:  $r \leftarrow r - m$
8: **end while**
9: **Return** ($r$)

prime $p$ must be less or equal to the processors word size $w$ (if not we would not work on a PAFF), and the integer $x$ to be reduced must be less or equal to $p \cdot 2^w$. Algorithm 4 summarizes this method. The while loop in Algorithm 4 is repeated at most three time. Depending on the prime $p$ the while loop is repeated about 0.5-1.5 times.

**Algorithm 4** Mixed Modular Reduction

**INPUT:** $p$ prime $\leq 2^w$ and $2^u > p > 2^{u-1}$, integer $x \leq p \cdot 2^w$
**OUTPUT:** $r = x \bmod p$

1: $\mu_0 \leftarrow \left\lfloor \frac{2^w(2^u - p)}{p} \right\rfloor$
2: $x_i \leftarrow \lfloor x/2^w \rfloor$
3: $q \leftarrow x_1 + \left\lfloor \frac{x_1 \cdot \mu_0}{2^w} \right\rfloor$
4: $r \leftarrow x - q \cdot$
5: **while** $r \geq p$ **do**
6: $\quad r \leftarrow r - p$
7: **end while**
8: **Return** $(r)$

### 3.1.4 Fast subfield modular reduction using OEFs

OEFs allow for very fast subfield modular reduction since it does not require divisions. The algorithm we give here can be found in [2]. The operator << indicates a left shift and >> a right shift. Algorithm 5 represents an implementation of the fast subfield modular reduction for OEFs and works with primes $p$ of the form $p = 2^n - c$ but can easily be adopted for primes $p$ of the form $p = 2^n + c$. It terminates after a maximum of two iterations of the while loop, so we require at most two multiplications by c, six shifts by n, and six additions and subtractions.

For a Type I OEF ($p = 2^n - 1$) the algorithm can be improved because no multiplications by $c$ ($c = 1$) need to be performed. The first while-loop will then be executed only once and the entire operation can be performed with 2 shifts and 2 additions if the intermediate result is contained in a single word.

**Algorithm 5** Fast subfield modular reduction using OEFs

**INPUT:** $p = 2^n - c, log_2 c \leq \frac{1}{2}n, x < p^2$ is the integer to reduce
**OUTPUT:** $r \equiv x \bmod p$

1: $q_0 \leftarrow x >> n$
2: $r_0 \leftarrow x - q_0 2^n$
3: $r \leftarrow r_0$
4: $i \leftarrow 0$
5: **while** $q_i > 0$ **do**
6: $\quad q_{i+1} \leftarrow q_i c >> n$
7: $\quad r_{i+1} \leftarrow q_i c - (q_{i+1} >> n)$
8: $\quad i \leftarrow i + 1$
9: $\quad r \leftarrow r + r_i$
10: **end while**
11: **while** $r \geq p$ **do**
12: $\quad r \leftarrow r - p$
13: **end while**
14: **Return** $(r)$

## 3.2 Extension Field Arithmetic

When working in the extension field $GF(p^m)$ we use polynomials as representatives of the field elements. As mentioned before, all operations are performed modulo the field polynomial $f(x)$, which in our case of OEFs to increase the speed of our algorithms, is a binomial of the form $f(x) = x^m - \omega$ with $\omega$ an element of the subfield $GF(p)$. An arbitrary element $A(x)$ of $GF(p^m)$ is represented as

$$A(x) = \sum_{i=0}^{m-1} a_i x^i, \ a_i \in GF(p) \tag{3.1}$$

The operations in the extension field are, compared to the ones in the subfield, more complicated so we will discuss them in detail. An operation in the extension field requires multiple operations on the subfield. For instance if we want to add to polynomials of degree $m-1$ we have to add each two coefficients of the polynomials having the same index. If $A(x)$ and $B(x)$ are two polynomials of degree $m-1$, $A(x), B(x) \in GF(p^m)$, we need $\#ADD = m$ additions in the subfield to perform the addition in the extension field.

The multiplication of two polynomials of degree at most $m-1$ results in a polynomial of degree at most $2m-2$. After performing a modular reduction with the field binomial $f(x)$ we receive a result polynomial of degree at most $m-1$. The multiplication can be performed by several algorithms. In this thesis we will introduce three algorithms, the schoolbook method, the iterative and the recursive Karatsuba algorithm (KA) and analyze the number of subfield multiplications and additions needed by these algorithms.

To perform polynomial squarings we will use the multiplication algorithms because there is no algorithm to improve the efficiency of squaring significantly.

Modular reduction is needed for squaring and multiplication of polynomials in $GF(p^m)$ and will be treated separately. In addition to the standard modular reduction where an explicit division by the field polynomial $f(x)$ is performed, we will show a faster method for performing modular reduction based on OEFs.

The fourth operation needed to be performed in the extension field is the Frobenius map for which we will give two algorithms.

### 3.2.1 Schoolbook Polynomial Multiplication

The standard algorithm for multiplying two polynomials of degree $m-1$ is called the schoolbook method. It can be implemented very easily but it does not offer an efficient performance. The schoolbook method needs $\#MUL = m^2$ subfield multiplications and $\#ADD = (m-1)^2$ subfield additions to perform a multiplication. Algorithm 6 shows how the schoolbook polynomial multiplication can be implemented.

### 3.2.2 Iterative Karatsuba

The iterative KA algorithm we use in this thesis was introduced by Weimerskirch and Paar in [24] who generalized the algorithm which was named after its inventor who presented the idea in [11] in the early 60's. The KA needs less subfield multiplications

**Algorithm 6** Schoolbook polynomial multiplication

**INPUT:** polynomials $A(x), B(x) \in GF(p^m)$
**OUTPUT:** $R(x) = A(x) \cdot B(x), R(x) = \sum_{i=0}^{2m-2} r_i x^i$

```
1: i ← 1, j ← 0
2: r_n ← 0 for 0 ≤ n ≤ m − 1
3: while i ≤ m − 1 do
4:    while j ≤ m − 1 do
5:       r_{i+j} ← r_{i+j} + a_i · b_j
6:       j ← j + 1
7:    end while
8:    i ← i + 1
9: end while
10: Return (R)
```

but more subfield additions to compute the product of two polynomials compared to the schoolbook method. It gains a performance increase on many processors where subfield additions can be performed much faster than subfield multiplications. The number of subfield multiplications needed for the iterative KA is $\#MUL = \frac{1}{2}n^2 + \frac{1}{2}n$ and the number of subfield additions $\#ADD = \frac{5}{2}n^2 - \frac{7}{2}n + 1$. The KA is considered efficient if the ratio between subfield multiplication and addition is larger than 3. Further details about the iterative KA can be found in [24]. Algorithm 7 gives a compact description of the iterative KA.

### 3.2.3 Recursive Karatsuba

In this section we introduce the recursive Karatsuba algorithm for multiplying two polynomials of degree $m - 1$. In comparison to the iterative KA the number of subfield additions needed can be reduced by using the recursive KA. The idea, also known as "divide and conquer", is to divide the given polynomials into several polynomials of degree $n$ with $n|(m-1)$. The coefficients of this polynomials are themselves polynomials and the multiplication of these coefficients results in further applications of the KA for polynomials of degree $(m-1)/n$. In this thesis we will consider only polynomials with

**Algorithm 7** Iterative Karatsuba

**INPUT:** polynomials $A(x), B(x) \in GF(p^m)$
**OUTPUT:** $R(x) = A(x) \cdot B(x), R(x) = \sum_{i=0}^{2m-2} r_i x^i$

1: $i \leftarrow 0$
2: **while** $i \leq m - 1$ **do**
3: $D_i \leftarrow a_i \cdot b_i$
4: $i \leftarrow i + 1$
5: **end while**
6: $i \leftarrow 1$
7: **while** $i \leq 2m - 3$ **do**
8: for all $s$ and $t$ with $s + t = i$ and $t > s \geq 0$
9: $D_{s,t} \leftarrow (a_s + a_t) \cdot (b_s + b_t)$
10: $i \leftarrow i + 1$
11: **end while**
12: $i \leftarrow 1$
13: $r_0 = D_0$
14: $r_{2m-2} = D_{n-1}$
15: **while** $i \leq 2n - 3$ **do**
16: **if** $i$ is odd **then**
17: $c_i = \sum_{s+t=i;t>s\geq 0} D_{s,t} - \sum_{s+t=i;m-1\geq t>s\geq 0}(D_s + D_t)$
18: **else**
19: $c_i = \sum_{s+t=i;t>s\geq 0} D_{s,t} - \sum_{s+t=i;m-1\geq t>s\geq 0}(D_s + D_t) + D_{i/2}$
20: **end if**
21: **end while**
22: **Return** $(R)$

| | #MUL | #ADD |
|---|---|---|
| $m$=2 | 3 | 4 |
| $m$=4 | 9 | 24 |
| $m$=8 | 27 | 100 |
| $m$=16 | 81 | 360 |
| $m$=32 | 243 | 1204 |
| $m$=64 | 729 | 3864 |

Table 3.1: Number of additions and multiplications needed for the recursive Karatsuba .

the number of coefficients $m$ a power of 2, that means $m = 2^n$, $n$ an integer, and the maximum degree of this polynomials $m-1 = 2^n-1$. In every recursive step of the algorithm the polynomial will be split into two new polynomials. This is repeated $n$ times until we obtain polynomials of degree 0 which consist of a constant. In [24] there is a table showing the number of subfield additions and multiplications needed to perform the recursive KA for given $m$. It can be seen that the use of polynomials which degrees are a power of 2 offer the best performance. In our case, $m$=64 is a good choice suited to DH-keys of length of about 1024 bit which is considered to be secure for most current applications. If we have polynomials with smaller degree we can use dummy coefficients (coefficients equal to zero) to fill up the polynomials to degree $m$. Table 3.1 shows the number of subfield operations needed for the recursive KA used for polynomials with the number of coefficients $m = 2^n$. The number of subfield multiplications needed converges to $n^{log_2 3}$. Algorithm 8 describes this method.

### 3.2.4 Standard Modular Reduction

The standard modular reduction can be used to reduce a given polynomial of arbitrary degree $k$, over the subfield $GF(p)$, by another arbitrary polynomial of smaller degree. Algorithm 9 is a polynomial division with remainder as the reduced polynomial. It works for arbitrary irreducible polynomials of degree $m$.

**Algorithm 8** Recursive Karatsuba for polynomials with $m = 2^n$ coefficients

**INPUT:** polynomials $A(x), B(x) \in GF(p^m)$, with degree $m = 2^n - 1$
**OUTPUT:** $R(x) = A(x) \cdot B(x), R(x) = \sum_{i=0}^{2m-1} r_i x^i$

1: $A(x) = A_h(x) \cdot x^{m/2} + A_l(x)$, with $A_l(x) = \sum_{i=0}^{m/2-1} a_i x^i$, $A_h(x) = \sum_{i=m/2}^{m-1} a_i x^{i-m/2}$
2: $B(x) = B_h(x) \cdot x^{m/2} + B_l(x)$, with $B_l(x) = \sum_{i=0}^{m/2-1} b_i x^i$, $B_h(x) = \sum_{i=m/2}^{m-1} b_i x^{i-m/2}$
3: compute $D_0 = A_l \cdot B_l$ using the recursive Karatsuba
4: compute $D_1 = A_h \cdot B_h$ using the recursive Karatsuba
5: compute $D_{0,1} = (A_l + A_h) \cdot (B_l + B_h)$ using the recursive Karatsuba
6: $R(x) = D_1 \cdot x^m + (D_{0,1} - D_0 - D_1) \cdot x^{m/2} + D_0$
7: **Return** $(R)$

**Algorithm 9** Standard Modular Reduction

**INPUT:** polynomial $A(x) = \sum_{i=0}^{k} a_i x^i$, $k \geq m$, field polynomial $f(x)$ of degree $m$
**OUTPUT:** $R(x) = A(x) \bmod f(x), R(x) = \sum_{i=0}^{m} r_i x^i$

1: $R \leftarrow A$
2: $i \leftarrow k$, $j \leftarrow 0$, $k \leftarrow 0$
3: **while** $i \geq m$ **do**
4: $\quad k \leftarrow r_i \cdot (f_m)^{-1}$
5: $\quad j \leftarrow i$
6: $\quad$ **while** $j \geq 0$ **do**
7: $\quad\quad r_{i+j-m} \leftarrow r_{i+j-m} - k \cdot f_j$
8: $\quad\quad j \leftarrow j - 1$
9: $\quad$ **end while**
10: $\quad i \leftarrow i - 1$
11: **end while**
12: **Return** $(R)$

### 3.2.5 OEF Modular Reduction

The modular reduction using OEFs introduced by Bailey and Paar in [2] and [3] performs faster than the standard modular reduction. It is based upon the fact that the field polynomial is a binomial $f(x) = x^m - \omega$ and so $x^m \equiv \omega \bmod f$, $x^{m+1} \equiv \omega x^i \bmod f$. The modular reduction in an OEF, summarized in Algorithm 10, needs at most $\#MUL = m - 1$ multiplications by $\omega$ and $\#ADD = m - 1$ additions but no inversions to compute $A(x)$ mod $f(x)$ with the degree of $A(x)$, $k \leq 2m - 2$. Type II OEFs which have a field polynomial of the form $f(x) = x^m - 2$ improve the performance. In this case the multiplications by $\omega$ can be performed as left shifts which are executed faster than multiplications by most processors.

**Algorithm 10** OEF Modular Reduction

**INPUT:** polynomial $A(x) = \sum_{i=0}^{k} a_i x^i$, $2m - 2 \geq k \geq m$, field binomial $f(x) = x^m - \omega$ of degree $m$
**OUTPUT:** $R(x) = A(x)$ mod $f(x)$, $R(x) = \sum_{i=0}^{m-1} r_i x^i$

1: $i \leftarrow m - 2$
2: $r_{m-1} \leftarrow a_{m-1}$
3: **while** $i \geq 0$ **do**
4: $\quad r_i \leftarrow \omega \cdot a_{m+i} + a_i$
5: $\quad i \leftarrow i - 1$
6: **end while**
7: **Return** $(R)$

### 3.2.6 Frobenius Map on an OEF

The Frobenius map is an automorphic mapping defined as $\phi : GF(p) \rightarrow GF(p)$, $\phi(\alpha) = \alpha^p$, with $p$ prime, named after its inventor Ferdinand Georg Frobenius. The $i$th iterate of the Frobenius map $\phi^i : \phi^i(\alpha) = \alpha^{p^i}$ also is an automorphism. We want to analyze the action of an arbitrary $i$th iterate of the Frobenius map on an arbitrary element of $GF(p^m)$ as described in [3].

Let once more $A(x) = \sum_{j=0}^{m-1} a_j x^j$, for $a_j \in GF(p)$. Fermat's Little Theorem indicates

that $a_j^p \equiv a_j \bmod p$, so the coefficients $a_j$ are fixed points of Frobenius map iteration, so:

$$A^{p^i}(x) \equiv a_{m-1}x^{(m-1)p^i} + \ldots + a_1 x^{p^i} + a_0 \bmod f(x) \tag{3.2}$$

The next step is to analyze the elements $(x^j)^{p^i} = x^{jp^i}, 0 < j < m$ which are not kept fixed by the action of the Frobenius map. Assuming that we use a binomial $f(x) = x^m - \omega$ as field polynomial we get the following results [3].

$$(x^j)^{p^i} \equiv \omega^q x^j \bmod f(x), \text{ with } q = \frac{j(p^i - 1)}{m} \tag{3.3}$$

All $x^{jp^i}, 1 \leq j, i \leq m-1$ can be precomputed for a given field binomial $f(x)$, so we need only a single subfield multiplication to compute $a_j \cdot x^{jp^i}$.

Algorithm 11 computes the $i$th iteration of the Frobenius map on a polynomial $A(x) \in GF(p^m)$ assuming precomputed values for $x^{jp^i}, 1 \leq j, i \leq m-1$. The precomputations of $x^{jp^i}, 1 \leq j, i \leq m-1$ have to be done only once, so it suffices to execute $\#MUL = m-1$ subfield multiplications.

**Algorithm 11** Frobenius Map on an OEF

**INPUT:** polynomial $A(x) \in GF(p^m)$, integer $i$, precomputed values $\omega^{q_{ij}}, q_{ij} = \frac{j(p^i-1)}{m}, 1 \leq j, i \leq m-1$, $m|(p-1)$

**OUTPUT:** $R(x) = A^{p^i}(x) \bmod f(x), R(x) = \sum_{i=0}^{m-1} r_i x^i$

1: $r_0 \leftarrow a_0$
2: $j \leftarrow 1$
3: **while** $j < m$ **do**
4: $\quad r_j \leftarrow a_j \cdot \omega^{q_{ij}}$
5: $\quad j \leftarrow j + 1$
6: **end while**
7: **Return**$(R)$

To compute the Frobenius map as given in Algorithm 11 the condition that $m|(p-1)$ has to be fulfilled which is not true for all OEFs. We will now give an algorithm to compute the first iteration of the Frobenius map which can be applied for arbitrary OEFs.

As mentioned before, the coefficients $a_j$ are fixed points of Frobenius map iteration and we only have to analyze the elements $(x^j)^p = x^{jp}$, $0 < j < m$.

We can write:

$$j \cdot p = l \cdot m + s, \text{ with } s = j \cdot p \bmod m \text{ and } l = \frac{j \cdot p - s}{m} \tag{3.4}$$

From this follows

$$x^{jp} = x^{lm} \cdot x^s = \omega^l \cdot x^s \tag{3.5}$$

which results in

$$A^p(x) = a_0 + \sum_{j=1}^{m-1} a_j \cdot \omega^{l_j} \cdot x^{s_j} \text{ with } s_j = j \cdot p \bmod m \text{ and } l_j = \frac{j \cdot p - s}{m} \tag{3.6}$$

The values $\omega^{l_j}$, $x^{s_j}$, $0 < j < m$ do not depend on $A(x)$ and can therefore be precomputed. Algorithm 12 computes the first iteration of the Frobenius map for an arbitrary OEF. If we compare it to Algorithm 11 with $i = 1$ we see that in addition to the $\#MUL = m - 1$ subfield multiplications we have to perform not more than $\#ADD = m - 1$ subfield additions.

**Algorithm 12** First Iteration of the Frobenius Map on an OEF

**INPUT:** polynomial $A(x) \in GF(p^m)$, precomputed values $\omega^{l_j}, s_j$ with $s_j = j \cdot p \bmod m, l_j = \frac{j \cdot p - s_j}{m}, 1 \leq j \leq m-1$

**OUTPUT:** $R(x) = A^p(x) \bmod f(x), R(x) = \sum_{i=0}^{m-1} r_i x^i$

1: $R \leftarrow a_0$
2: $j \leftarrow 1$
3: **while** $j < m$ **do**
4: $\quad r_{s_j} \leftarrow r_{s_j} + a_j \cdot \omega^{l_j}$
5: $\quad j \leftarrow j + 1$
6: **end while**
7: **Return**$(R)$

## 3.3 Exponentiation in $GF(p^m)$

In the previous sections we have introduced several algorithms to perform fast arithmetic operations as well in the subfield as in the extension field. Now we want to look at the operation needed to perform public key operations, and that is the exponentiation in $GF(p^m)$. We will introduce three different algorithms to perform the exponentiation. The first one is the Binary-Exponentiation which uses the binary representation of the exponent. The second one is an adapted version of the Montgomery Exponentiation which uses a combination of Montgomery Multiplication and Binary Exponentiation to perform the exponentiation. The third method is called Baby-Window, Giant-Window and was introduced by Avanzi and Mihăilescu in [1].

### 3.3.1 Binary Exponentiation

The Binary Exponentiation is a standard algorithm used to exponentiate integers. It can easily be adapted to polynomials in $GF(p^m)$. Algorithm 13 is an implementation of the left-to-right binary exponentiation for polynomials. If $t$ is the bit length of the exponent $n$, and $wt(n)$ the number of 1's in this representation then Algorithm 13 needs $\#SQR = t - 1$ polynomial squarings and $\#MUL = wt(n) - 1$ polynomial multiplications to compute $A^n(x)$. In our case, polynomial squarings and multiplications take the same amount of time, thus the Binary Exponentiation needs $\#MUL = t + wt(n) - 2$ multiplications in the extension field.

### 3.3.2 Montgomery Exponentiation for Polynomials

The Montgomery Exponentiation for integers is a modified version of the binary exponentiation using the Montgomery multiplication which instead of computing $x \cdot y \bmod p$ computes $x \cdot y \cdot r^{-1} \bmod p$ with $r$ an arbitrary fixed element. Details regarding this algorithm can be found in [17] and [19].

**Algorithm 13** Binary Exponentiation

**INPUT:** polynomial $A(x) \in GF(p^m)$, $n = (n_{t-1} \dots n_1 n_0)_2 \in \mathbb{N}$
**OUTPUT:** $R(x) = A^n(x)$

1: $R \leftarrow 1$
2: $i \leftarrow t - 1$
3: **while** $i \geq 0$ **do**
4: $\quad R \leftarrow R \cdot R$
5: $\quad$ **if** $n_i = 1$ **then**
6: $\quad\quad R \leftarrow R \cdot A$
7: $\quad$ **end if**
8: $\quad i \leftarrow i - 1$
9: **end while**
10: **Return**(R)

In [13] there is an adapted version of the Montgomery Exponentiation for $GF(2^k)$. We will generalize this algorithm to fit our needs when working on $GF(p^m)$. To perform a Montgomery Exponentiation we also need the Montgomery Multiplication and the Extended Euclidean Algorithm for polynomials. All three algorithms will be given in this section.

Algorithm 14 presents the Extended Euclidean Algorithm, normally used for integers as in [6] or [17], applied for polynomials in $GF(p^m)$. It computes the greatest common divisor $d(x)$ of two polynomials $f(x), r(x) \in GF(p^m)$ and also outputs two polynomials $f'(x)$ and $r^{-1}(x)$ as in the condition $d(x) = f'(x)f(x) + r^{-1}r(x)$.

We want to analyze Algorithm 14 if applied to a binomial $f(x) = x^m - \omega$ which we use while working on OEFs. It is easy to see that if we perform Algorithm 14 with $f(x) = x^m - \omega$ and $r(x) = x^m$ we get the following result after two iterations of the while-loop:

$$-\omega^{-1}f(x) + \omega^{-1}r(x) = 1 \Rightarrow r^{-1} = \omega^{-1} \wedge f' = -\omega^{-1} \tag{3.7}$$

So for computing $r^{-1}$ and $f'$ we just have to perform an inversion of $\omega$ which can be done by using Algorithm 2.

The second algorithm needed to perform a Montgomery exponentiation for polynomi-

**Algorithm 14** Extended Euclidean Algorithm for Polynomials

**INPUT:** polynomials $f(x), r(x) \in GF(p^m)$

**OUTPUT:** $d(x) = \gcd(f(x), g(x))$ and polynomials $f'(x), r^{-1}(x)$ with $d(x) = f'(x)f(x) + r^{-1}(x)r(x)$

1: **if** $r = 0$ **then**
2: $d \leftarrow f$
3: $f' \leftarrow 1$
4: $r^{-1} \leftarrow 0$
5: **Return**(d,f',$r^{-1}$)
6: **end if**
7: $s_1 \leftarrow 0$
8: $s_2 \leftarrow 1$
9: $t_1 \leftarrow 1$
10: $t_2 \leftarrow 0$
11: **while** $r \neq 0$ **do**
12: $q \leftarrow f$ div $g$
13: $r \leftarrow f - gq$
14: $f' \leftarrow s_2 - qs_1$
15: $r^{-1} \leftarrow t_2 - qt1$
16: $f \leftarrow g$
17: $r \leftarrow r$
18: $s_2 \leftarrow s_1$
19: $s_1 \leftarrow f'$
20: $t_2 \leftarrow t_1$
21: $t_1 \leftarrow r^{-1}$
22: **end while**
23: $d \leftarrow f$
24: $f' \leftarrow s_2$
25: $r^{-1} \leftarrow t_2$
26: **Return** (d,f',$r^{-1}$)

als is the Montgomery multiplication for polynomials. Koç and Acar applied the Montgomery Multiplication algorithm for polynomials $A(x) = \sum_{i=0}^{m-1} a_i \cdot x^i$ with $a_i \in GF(2)$ in [13].

Algorithm 15 was introduced in [13] and can be applied with few changes for polynomials representing elements of $GF(p^m)$ to compute the Montgomery multiplication. The Montgomery multiplication requires that $f(x)$ and $r(x)$ are relatively prime, i.e. $gcd(r(x), f(x)) = 1$. We choose $r(x) = x^m$ to obtain a fast implementation and to ensure that $r(x)$ and $f(x)$ have no common divisor except 1. $r$ is the element of the field, represented by the polynomial $r(x)$ mod $f(x) = (f_{m-1}...f_1 f_0)$. The polynomial $f'(x)$ needed in the algorithm can be computed using Algorithm 14. Since $gcd(r(x), f(x)) = 1$ the two polynomials $r^{-1}(x)$ and $n'(x)$ exist. $f'(x)$ needs to be computed only once because it depends on $r(x)$ and the field polynomial $f(x)$ only.

The multiplications needed in Steps 1-3 of Algorithm 15 can be performed using the polynomial multiplication given in Section 3.2. The modular reduction with modulus $r(x) = x^m$ in Step 2 can be performed by ignoring the terms of $t(x) \cdot f'(x)$ which have powers of $x$ larger than or equal to $m$. The division by $r(x)$ in Step 3 can be performed by shifting the polynomial $t(x) + u(x) \cdot f(x)$ to the right by $m$ positions.

The basic idea of the Montgomery Multiplication is to perform the multiplication without modular reduction with the field polynomial $f(x)$ which is a very slow operation. If we look at Algorithm 15 we see that modular reduction is performed only once by $r(x)$ in Step 2. Because of the special form of $r(x)$ it can be performed almost for free. Hence the Montgomery multiplication for polynomials can be performed faster than a polynomial multiplication followed by a modular reduction.

When working on OEFs Algorithm 10 performs a fast modular reduction. Thus Montgomery Multiplication for polynomials is not faster than polynomial multiplication followed by a modular reduction on an OEF. We will show that it is even slower. We

**Algorithm 15** Montgomery Multiplication for Polynomials

**INPUT:** polynomials $a(x), b(x), r(x), f'(x)$ with $r(x)r^{-1}(x) + f(x)f'(x) = 1$
**OUTPUT:** $c(x) = a(x) \cdot b(x) \cdot r^{-1}(x) \bmod f(x)$

1: $t \leftarrow ab$
2: $u \leftarrow tf' \bmod r$
3: $c \leftarrow [t - uf]/r$
4: **Return**(c)

can use the results we have given in equation 3.7. In Step 1 we have one multiplication on the extension field. Because $f' = -\omega^{-1}$, which is a constant, Step 2 means simply performing $m$ multiplications with $-\omega^{-1}$. In Step 3, we have $u \cdot f$ with $f = x^m - \omega$. We can write $u \cdot f = u \cdot x^m - u \cdot \omega$. $u \cdot x^m$ means just shifting the coefficients. Because we have to perform a division by $r(x) = x^m$ we don't have to perform the multiplication $u \cdot \omega$. So in Step 3 we just have to perform $m - 1$ subtractions. This means that to perform the Montgomery Multiplication on an OEF we need one extension field multiplication, $m$ subfield multiplications and $m-1$ subfield subtractions. This is one subfield subtraction more than needed for an OEF reduction as discussed in section 3.2.5.

In case of an OEF we can compute $c(x) = a(x) \cdot b(x) \cdot r^{-1} \bmod f(x)$ even directly. Because $r^{-1} = \omega^{-1}$ is a constant we have:

$$a(x) \cdot b(x) \cdot r^{-1}(x) = a(x) \cdot b(x) \cdot \omega^{-1} \tag{3.8}$$

If we use Algorithm 10 to perform the modular reduction we can simplify which leads to the following equation which can be used to compute the Montgomery Multiplication on an OEF with $s(x) = a(x) \cdot b(x) = \sum_{i=0}^{2m-2} s_i$:

$$s(x) \cdot r^{-1}(x) \bmod f(x) = \omega^{-1} s_{m-1} x^{m-1} + (s_{2m-2} + \omega^{-1} s_{m-2}) x^{m-2} + \ldots + (s_m + \omega^{-1} s_0) \tag{3.9}$$

As mentioned before we need one extension field multiplication, $m$ subfield multipli-

cations and $m-1$ subfield additions to perform the Montgomery multiplication.

The Montgomery Multiplication can be used for a fast implementation of the exponentiation over $GF(p^m)$ as shown in Algorithm 16. It is an adapted version of the Binary Exponentiation Algorithm. In Step 2, $A \cdot r$ can be computed by simply multiplying all coefficients of $A(x)$ by $r(x) = \omega$. All other multiplications are performed using Algorithm 15.

**Algorithm 16** Montgomery Exponentiation for Polynomials

**INPUT:** polynomial $A(x) \in GF(p^m)$, $n = (n_{t-1} \ldots n_1 n_0)_2 \in \mathbb{N}$
**OUTPUT:** $R(x) = A^n(x)$

1: $c \leftarrow r$
2: $a \leftarrow A \cdot r$
3: $i \leftarrow t-1$
4: **while** $i \geq 0$ **do**
5: $\quad c \leftarrow c \cdot c \cdot r^{-1}$
6: $\quad$ **if** $n_i = 1$ **then**
7: $\quad\quad c \leftarrow c \cdot a \cdot r^{-1}$
8: $\quad$ **end if**
9: $\quad i \leftarrow i-1$
10: **end while**
11: $c \leftarrow c \cdot 1 \cdot r^{-1}$
12: **Return**(c)

The Montgomery Exponentiation is supposed to be more efficient than the Binary-Exponentiation described in 3.3.1 because the Montgomery Multiplication for polynomials can be performed faster than polynomial multiplication with modular reduction. As mentioned before this isn't true for OEFs.

### 3.3.3 Baby-Window, Giant-Window Algorithm

This algorithm, introduced in [1], is based on the idea to use the Frobenius map to gain a speed-up for the extension field exponentiation. It assumes that the Frobenius automorphism can be computed efficiently.

We want to compute $A^n$, with $A \in GF(p^m)$ and $n \in \mathbb{N}$, where $n$ is

$$n = \sum_{i=0}^{d-1} m_i p^i \text{ with } 0 \leq m_i < p \tag{3.10}$$

Now, we can write $A^n$ as follows

$$A^n = A^{\sum_{i=0}^{d-1} m_i p^i} = \prod_{i=0}^{d-1} A^{m_i p^i} = \prod_{i=0}^{d-1} (A^{m_i})^{p^i} \tag{3.11}$$

The last term $(A^{m_i})^{p^i}$ is the $i$th iteration of the Frobenius map $\phi$ applied to $A^{m_i}$. So we can write

$$A^n = \prod_{i=0}^{d-1} \phi^i(A^{m_i}) = \phi(\ldots \phi(\phi(A^{m_{d-1}}) \cdot A^{m_{d-2}}) \ldots \cdot A^{m_1}) \cdot A^{m_0} \tag{3.12}$$

To evaluate Equation 3.12, we need to perform $m-1$ steps in which we have to perform one exponentiation, by exponents of size $p$, one Frobenius map and one multiplication. The process is called the subdivision of $n$ in giant windows.

The next step is to find a method for fast computation of the exponentiation $A^{m_i}, 0 \leq i \leq d-1$, with $m_i < p$. We can use any method for exponentiation in the extension field e.g. Binary Exponentiation (see Algorithm 13) or Montgomery Exponentiation (see Algorithm 16). In this section we will introduce the algorithm presented by Avanzi and Mihăilescu in [1] called Baby Window method.

First we fix a small integer $\ell$ called the width of the baby-window. Let $u$ be the bit length of $p$, so $2^u > p > 2^{u-1}$, and $u = K\ell + R$, with $0 \leq R < \ell$. A $2^u$-bit word will be subdivided in $K'$ baby-windows, with

$$K' = \begin{cases} K & \text{if R=0} \\ K+1 & \text{else} \end{cases}$$

Now we can write any number $t$, $0 \leq t < p$ in base $2^\ell$ notation

$$t = (t_{K'-1}t_{K'-2}\ldots t_1t_0)_{2^\ell} = \sum_{j=0}^{K'-1} t_j 2^{j\ell} \textbf{ with } 0 \leq t_j < 2^\ell \tag{3.13}$$

and apply it to our initial problem to compute $A^t$, for given $0 \leq t \leq d-1$

$$A^t = \prod_{0 \leq j < K'} A^{t_j 2^{j\ell}} \tag{3.14}$$

If we know $A$, we can precompute $A^{i2^{j\ell}}$ for $0 \leq j < K'$ and $1 \leq i \leq 2^\ell - 1$ and we would need only $\#MUL = K' - 1$ multiplications to compute $A^t$ according to equation 3.14. The number of operations needed to compute the baby-window results depends on $\ell$. According to [1] there is no analytic optimization for the window length $\ell$ so the best method is trial and error. Algorithm 17 is an implementation of the Baby-Window, Giant Window Exponentiation which uses the precomputations mentioned above. First we expand the exponent $n$ into the Giant-Windows and then precompute the values $A^{i2^{j\ell}}$. The first while-loop computes the value $A^{n_{m-1}}$. In the nested while-loop we compute $A^{n_k}$, with $m-1 \geq k \geq 0$. We first write $n_k$ in base $2^\ell$ and use the method given in equation 3.14. Then we apply the Frobenius map on the result. We repeat this procedure for every Giant-Window, e.g. $m$-times.

## 3.4 OEF Construction

We have seen in this chapter that computations on OEFs offer a better performance than computations on arbitrary $GF(p^m)$. Before working with OEFs we have to find a field binomial $f(x) = x^m - \omega$ and a prime $p = 2^n - c$ such that $GF(p^m)$ is an OEF. In [3] we can find an algorithm which computes $p$ and $m$ so, that $f(x) = x^m - 2$ and $p$ form a Type II OEF. Because the word length of the Motorola Dragonball processor, used in

**Algorithm 17** Baby-Window, Giant-Window Exponentiation

**INPUT:** polynomial $A(x) \in GF(p^m)$, $n \in \mathbb{N}$
**OUTPUT:** $R(x) = A^n(x)$

Expand $n$ to $n = \sum_{i=0}^{m-1} n_i p^i$ with $0 \leq n_i < p$
Precompute $B_{i,j} = A^{i2^{j\ell}}$, $0 \leq j < K'$ and $1 \leq i \leq 2^\ell - 1$
$k \leftarrow m - 1$, $q \leftarrow 0$, $r_0 = 1$
$t = n_k = (t_{K'-1} t_{K'-2} \ldots t_1 t_2)_{2^\ell}$
**while** $q < K'$ **do**
    $R \leftarrow R \cdot B_{t_q,q}$
    $q \leftarrow q + 1$
**end while**
$k \leftarrow k - 1$
**while** $k \geq 0$ **do**
    $R = \phi(R)$
    $q \leftarrow 0$
    $t = n_k = (t_{K'-1} t_{K'-2} \ldots t_1 t_2)_{2^\ell}$
    **while** $q < K'$ **do**
        $R \leftarrow R \cdot B_{t_q,q}$
        $q \leftarrow q + 1$
    **end while**
    $k \leftarrow k - 1$
**end while**

Palm OS devices, is 16 bit, we fix $n$ to be 16. To obtain a key length of 1024-bit we choose the degree of the field binomial $m = 64$. We have adapted the algorithm given in [3] for arbitrary $\omega$. Because $m = 2^6$, 2 is the only prime factor of $m$. The order $ord(\omega)$ of $\omega \in GF(p)$ is a divisor of $p-1$ so we use the factorization of $p-1$ to compute $ord(\omega)$.

**Algorithm 18** OEF Construction Procedure

**INPUT:** $n$ given

**OUTPUT:** $p$ and $f(x) = x^{64} - \omega$ define an OEF with field order between $2^{low}$ and $2^{high}$

$c \leftarrow 1$
**while** $log_2 c \leq \lfloor \frac{1}{2} n \rfloor$ **do**
  $p \leftarrow 2^n - c$
  **if** $p$ is prime **then**
    $\omega = 3$
    **while** $\omega \leq 16$ **do**
      factor $p - 1$
      $ord(\omega) \leftarrow$ the order of $\omega \in GF(p)$
      $BadMValue \leftarrow 0$
      **if** $2 \nmid ord(\omega)$ **then**
        $BadMValue \leftarrow 1$
      **end if**
      **if** $2 | \frac{p-1}{ord(\omega)}$ **then**
        $BadMValue \leftarrow 1$
      **end if**
      **if** $BadMValue = 0$ **then**
        **if** $p \equiv 1 \pmod 4$ **then**
          **Return**$(p, m)$
        **end if**
      **end if**
      $\omega \leftarrow \omega + 1$
    **end while**
  **end if**
  $c \leftarrow c + 2$
**end while**
**Return**$(R)$

Table 3.3 and Table 3.2 show the OEFs we have found using Algorithm 18 with $p = 2^n - c$ and the field binomial $f(x) = x^m - \omega$. In Table 3.2 we see that we can choose different $\omega$ for the same prime $p$ to generate an OEF. $p = 2^{16} - 1$ is no prime

| $c$ | $p$ | $\omega$ |
|---|---|---|
| 39 | 65 497 | 5, 7, 10, 14, 15 |
| 87 | 65 449 | 7, 11, 13, 14 |
| 99 | 65 437 | 5, 6, 13, 14, 15 |
| 123 | 65 413 | 5, 6, 7, 11, 15 |
| 143 | 65 393 | 3, 5, 6, 7, 10, 12, 14 |
| 155 | 65 381 | 3, 10, 11, 12, 14, 15 |
| 179 | 65 357 | 3, 5, 7, 11, 12, 13 |
| 183 | 65 353 | 5, 10, 11, 13, 15 |
| 227 | 65 309 | 3, 7, 10, 11, 12, 15 |
| 243 | 65 293 | 5, 6, 11, 13, 14, 15 |

Table 3.2: OEFs with $n = 16$ and $m = 64$

| $c$ | $p$ |
|---|---|
| 99 | 65 437 |
| 123 | 65 413 |
| 155 | 65 381 |
| 179 | 65 357 |
| 227 | 65 309 |
| 243 | 65 293 |

Table 3.3: Type II OEFs with $n = 16$ and $m = 64$

so there exists no Type I OEF we could use. We implemented the algorithm using the C language and the LCCWin32 compiler on a standard PC with a 1GHz processor and 256 MByte RAM.

# 4 Implementation on a Palm OS Handheld

In this chapter we will discuss our implementation of an extension field exponentiation based on the algorithms introduced in Chapter 3. The target system used is a Palm m500 handheld but the implementation we achieved is not restricted to this platform and can be exported to any other Palm OS platform using Palm OS 4.0 or higher and a Motorola Dragonball processor. For the implementation we used the C language and the Metrowerks Code Warrior 8.0 IDE.

## 4.1 Palm Handhelds

This devices are characterized by a combination of small dimensions and weight, long battery life time, little memory (between 2–8 MB) and slow processors (between 15 and 33 MHz). They offer many communication capabilities but almost no security features. User data, including items marked private, are stored in the clear, protected only with a standard password method. To enhance the life time of batteries an aggressive power management is used which keeps the processor in sleep mode for about 99% of the time and wakes it for user interactions with the device via the touch screen or a button. The

lack of security capabilities for Palm OS devices makes it necessary to develop security applications which do not only enable security functions like signing or encryption but also take into consideration the limitations regarding processor speed and memory.

## 4.2 Field Order and Representation

As mentioned in Section 2.2, one way to enhance the speed of algorithms using PAFFs and Optimal Extension Fields is to choose the prime $p$ of the subfield $GF(p)$ to be about the size of the word length of the processor which in case of the Motorola Dragonball processor is 16-bit. Today, key lengths of about 1000 bit (usually 1024 bit are chosen) are assumed to be secure for most applications so our implementation will use 1024 bit keys. Hence we choose the field polynomial $f(x)$ of degree 64 which generates an extension field $GF(p^{64})$ with a field length of $16 \cdot 64 = 1024$ bit.

The elements of the extension field $GF(p^m)$ are represented by arrays of 64 elements, each element of the array containing an unsigned 16-bit integer value representing a coefficient of the polynomial:

$$A \in GF(p^m),\ A = [A[63], \ldots, A[0]] \text{ and } A[i] \in GF(p) \text{ unsigned integers} \tag{4.1}$$

The array element with index $n$ for example represents the coefficient $a_n$. Because the coefficients $a_i$ of the polynomials, representing the elements of the extension field, are elements of the subfield $GF(p)$ with the prime $p \leq 2^{16}$, they can be represented by unsigned 16-bit integers.

The exponent $e$ of the exponentiation $A(x)^e$, $A \in GF(p^m)$, needs to be a 1024 bit integer to ensure key length of 1024-bit. Because there is no native representation of 1024-bit integers for Palm devices we will represent $e$ as an array of unsigned 16-bit

integers containing 64 elements.

$$e = (e_{1023}e_{1022}\dots e_1e_0)_2 = (E_{63}E_{62}\dots E_1E_0)_2 \text{ with } E_n = (e_{15+16n}e_{14+n\cdot 16}\dots e_{n\cdot 16}) \quad (4.2)$$

## 4.3 Algorithms

We used the algorithms described in Chapter 3 to implement the extension field exponentiation. In this section we will only deal with characteristics special to the implementation on a Palm OS device.

### 4.3.1 Algorithms in $GF(p)$

The representation of the subfield elements is done using unsigned 16-bit integers. All operations like addition, subtraction and multiplication are performed modulo the prime $p$ which is also represented by an unsigned 16-bit integer. Adding two 16-bit unsigned integers can result in a 17-bit integer which has to be represented as an unsigned 32-bit integer on a Palm OS device. Multiplication results in a 32-bit unsigned integer so in both cases we have to perform a cast of the result before performing the modular reduction. The subtraction of two unsigned integers can result in a negative result so here we have also to perform a casting before modular reduction.

The Palm OS platform can perform fast bit-operations like OR and AND so we adapted the fast subfield modular reduction presented in Section 3.1.4 to gain speed. Algorithm 19 shows the adapted algorithm.

The reduction presented in Algorithm 19 is used to reduce the result of subfield multiplications. The result of a subfield addition needs to be reduced at most by subtraction with p, so for subfield addition we can perform reduction faster than Algorithm 19. Algorithm 20 shows our implementation of the subfield addition including reduction. The implementation for subfield subtraction is very similar, we just have to add $p$ to the

**Algorithm 19** Adapted fast subfield modular reduction using OEFs

**INPUT:** $p = 2^n - c, log_2 c \leq \frac{1}{2}n, x < p^2$ is the integer to reduce
**OUTPUT:** $r \equiv x \bmod p$

1: $q_1 \leftarrow x >> n$
2: $q_2 \leftarrow 0$
3: $r \leftarrow x$ AND $0xFFFF$
4: $i \leftarrow 0$
5: **while** $q_1 > 0$ **do**
6: $\quad q_2 \leftarrow q_2 \cdot c$
7: $\quad r \leftarrow r + (q_2$ AND $0xFFFF)$
8: $\quad q_1 \leftarrow q_2 >> n$
9: **end while**
10: **while** $r \geq p$ **do**
11: $\quad r \leftarrow r - p$
12: **end while**
13: **Return** $(r)$

result in case it is less than 0.

**Algorithm 20** Subfield Addition including modular reduction

**INPUT:** integers $x, y$, prime $p$
**OUTPUT:** $r \equiv x + y \bmod p$

1: $r \leftarrow x + y$
2: **if** $r \geq p$ **then**
3: $\quad r \leftarrow r - p$
4: **end if**
5: **Return** $(r)$

### 4.3.2 Algorithms in $GF(p^m)$

As mentioned before we implemented the elements of the extension fields as arrays. To show how we work with such arrays we show our implementation of the Extension Field addition which simply works by adding the corresponding coefficients of two polynomials. Algorithm 21 shows the implementation. Step 3 can be performed using Algorithm 20. It is easy to see that we do not need to perform a reduction on the extension field because the resulting polynomial has a highest degree also less then $m$ and doesn't need to be

reduced on $GF(p^m)$.

**Algorithm 21** Extension Field Addition

**INPUT:** polynomials $A(x), B(x)$ represented by two 64-field arrays $A, B$
**OUTPUT:** polynomial $R(x) = A(x) + B(x)$ represented by a 64-field array

1: $i \leftarrow 1$
2: **while** $i \leq 64$ **do**
3: $\quad R[i] = A[i] + B[i] \bmod p$
4: $\quad i \leftarrow i + 1$
5: **end while**
6: **Return** $(R)$

Implementation of the recursive Karatsuba algorithm for polynomial multiplication, as introduced in Algorithm 8, leads to some problems because of the limited program stack of the Palm device. The need of several temporary variables results in long execution time and memory problems of the recursive algorithm. We implemented the recursive steps as explicit functions. For polynomials of 64 coefficients we need 6 separate functions, each computing the result of a polynomial multiplication for polynomials with 64, 32, 16, 8, 4 and 2 coefficients respectively. The algorithms used inside the functions are identical to Algorithm 8. For example the function for multiplying two polynomials with 64 coefficients uses the function for multiplying two polynomials with 32 coefficients to compute intermediate results. This function itself uses the function for multiplying two polynomials with 16 coefficients to compute intermediate results. This recursive procedure is repeated until the function for multiplying two polynomials with 2 coefficients is called which computes the coefficients according to Algorithm 22

The Frobenius Map, as introduced in Section 3.2.6, is required to perform the exponentiation by the Baby-Window, Giant-Window algorithm. We only need the first iteration of the Frobenius Map $\phi(A(x))$ with $A(x) \in GF(p^m)$. So the precomputations $x^{jp^i}, 1 \leq j, i \leq m-1$ for the Frobenius Map, as mentioned in Section 3.2.6, need to be performed only for $i = 1$. Algorithm 23 shows our implementation of the pre-

**Algorithm 22** KA Multiplication for polynomials of two coefficients

**INPUT:** polynomials $A(x) = a_1x + a_0, B(x) = b_1x + b_0$
**OUTPUT:** polynomial $R(x) = A(x) \cdot B(x) = r_2x^2 + r_1x + r_0$

1: $D_0 \leftarrow a_0 \cdot b_0 \bmod p$
2: $D_1 \leftarrow a_1 \cdot b_1 \bmod p$
3: $s_1 \leftarrow a_0 + a_1 \bmod p$
4: $s_2 \leftarrow b_0 + b_1 \bmod p$
5: $D_{0,1} \leftarrow s1 \cdot s2 \bmod p$
6: $r_2 \leftarrow D_1$
7: $r_1 \leftarrow D_{0,1} - (D_1 + D_0)$
8: $r_0 \leftarrow D_0$
9: **Return**(R)

computations needed to compute the first iteration of the Frobenius map according to Algorithm 11. The result of Step 1 is an integer. Step 2 is performed using Algorithm 20 and Step 5 uses standard integer multiplication followed by a modular reduction using Algorithm 19.

**Algorithm 23** Precomputations for the First Iteration of the Frobenius Map

**INPUT:** field binomial $f(x) = x^m - \omega$
**OUTPUT:** $R = (r_{m-1} \ldots r_2 r_1)$, $r_j = (x^j)^p$

1: $q \leftarrow (p-1)/m$
2: $r_1 \leftarrow \omega + q \bmod p$
3: $j \leftarrow 2$
4: **while** $j < m$ **do**
5: $\quad r_j \leftarrow r_{j-1} \cdot r_1 \bmod p$
6: $\quad j \leftarrow j + 1$
7: **end while**
8: **Return**(R)

Algorithm 24 shows our implementation of the first iteration of the Frobenius Map using the precomputed values of Algorithm 23.

**Algorithm 24** First Iteration of the Frobenius Map using Precomputation

**INPUT:** polynomial $A(x) \in GF(p^m)$, Precomputations $Q = (q_{m-1} \dots q_2 q_1)$, $q_j = (x^j)^p$
**OUTPUT:** $R(x) = A^p(x)$

1: $r_0 = a_0$
2: $j \leftarrow 1$
3: **while** $j < m$ **do**
4: $\quad r_j \leftarrow a_j \cdot q_j \bmod p$
5: $\quad j \leftarrow j + 1$
6: **end while**
7: **Return**(R)

### 4.3.3 Implementation of the Baby-Window, Giant-Window Algorithm

In this section we will discuss our Palm OS device specific implementation of the Baby-Window, Giant-Window exponentiation algorithm introduced in Section 3.3.3. It consists of the following functions: Frobenius Map (see Algorithm 24), the Precomputations needed for the Baby-Windows, the expansion of the exponent $n = \sum_{i=0}^{m-1} n_i p^i$ and the computation of $A(x)^n$ using precomputations and the Frobenius Map.

As mentioned in Section 3.3.3 the length $\ell$ of the Baby-Windows cannot be optimized analytically. We implemented different window length. The results for the Baby-Window, Giant-Window exponentiation with window length $\ell = 4, 6, 8$ respectively will be given in Section 4.4. Furthermore we assume $\ell = 4$. As mentioned in [1] we have $\#Pre = K(2^\ell - 1) + (2^R - 1) - 1$ precomputed elements. Because $p$ is a 16-bit integer we let $u = 16$ and $K' = K = 4$ and $R = 0$. So we need $\#Pre = 59$ precomputed elements. Algorithm 25 shows the implementation of the Baby-Windows precomputation with $\ell = 4$.

The exponent $n$ for the exponentiation is a 1024-bit unsigned integer represented as an array containing 64 elements, each a 16-bit unsigned integer. We can write $n = (n_{63} n_{62} \dots n_1 n_0)_{2^{16}}$. Algorithm 26 computes the base $p$ representation of $n$. For details see [17]. The multiplication by $2^{16}$ in Step 7 and 8 can be performed by left shifts.

**Algorithm 25** Precomputations for the Baby-Windows for $\ell = 4$

**INPUT:** polynomial $A(x) \in GF(p^m)$, field binomial $f(x) = x^m - \omega$

**OUTPUT:** precomputed elements $A(x)^{i2^{j4}}$, $0 \leq j < 4$ and $1 \leq i \leq 15$

1: $A^{1 \cdot 2^0} \leftarrow A$
2: $i \leftarrow 2$
3: **while** $i \leq 14$ **do**
4: $\quad A^{i \cdot 2^0} \leftarrow A^{(i-1) \cdot 2^0} A \bmod f$
5: $\quad i \leftarrow i + 1$
6: **end while**
7: $A^{1 \cdot 2^4} \leftarrow A^{15 \cdot 2^0} A \bmod f$
8: $i \leftarrow 2$
9: **while** $i \leq 14$ **do**
10: $\quad A^{i \cdot 2^4} \leftarrow A^{(i-1) \cdot 2^4} A^{1 \cdot 2^4} \bmod f$
11: $\quad i \leftarrow i + 1$
12: **end while**
13: $A^{1 \cdot 2^8} \leftarrow A^{15 \cdot 2^4} A^{1 \cdot 2^4} \bmod f$
14: $i \leftarrow 2$
15: **while** $i \leq 14$ **do**
16: $\quad A^{i \cdot 2^8} \leftarrow A^{(i-1) \cdot 2^8} A^{1 \cdot 2^8} \bmod f$
17: $\quad i \leftarrow i + 1$
18: **end while**
19: $A^{1 \cdot 2^{12}} \leftarrow A^{15 \cdot 2^8} A^{1 \cdot 2^8} \bmod f$
20: $i \leftarrow 2$
21: **while** $i \leq 14$ **do**
22: $\quad A^{i \cdot 2^{12}} \leftarrow A^{(i-1) \cdot 2^{12}} A^{1 \cdot 2^{12}} \bmod f$
23: $\quad i \leftarrow i + 1$
24: **end while**

**Algorithm 26** Expansion of the exponent $n$

**INPUT:** exponent $n = (n_{63}n_{62}\dots n_1n_0)_{2^{16}}$
**OUTPUT:** $R = (r_{m-1}\dots r_1r_0)$ and $n = \sum_{i=0}^{m-1} r_i p^i$

1: $q \leftarrow n$
2: $i \leftarrow m-1$
3: **while** $i \geq 0$ **do**
4: $r \leftarrow 0$
5: $j \leftarrow i$
6: **while** $j \geq 0$ **do**
7: $s_j \leftarrow (r \cdot 2^{16} + q_j) \text{ div } p$
8: $r \leftarrow (r \cdot 2^{16} + q_j) \bmod p$
9: $j \leftarrow j-1$
10: **end while**
11: $r_{m-i-1} = r$
12: $q \leftarrow s$
13: $i \leftarrow i-1$
14: **end while**
15: **Return** (R)

The computation of $A(x)^n$ is performed using Algorithm 17. Step 1 is performed using Algorithm 26 and step 2 using Algorithm 25. To perform the polynomial multiplications needed in step 6 and step 15 we can use any of the algorithms introduced in section 3.2.

## 4.4 Results

Now we take a look at the timings we have realized with our implementation of extension field exponentiation using a Palm m500 device with 8MB memory and a 33 MHz processor. For the implementation we used the CodeWarrior IDE 8.0 and the C programming language without optimizations in Assembler. All operations were performed working on OEFs as described in Chapter 2.2. A typical DH key exchange requires one exponentiation with known base and one with random base. For instance, in Algorithm 1 Alice needs one exponentiation with known base to compute $g^{x_A}$. The precomputations for this exponentiation can be performed before the key exchange starts because Alice

| | Exp. | Precomp. | Total |
|---|---|---|---|
| Binary Exponentiation | 59.21 | 0 | 59.21 |
| Montgomery Exponentiation | 60.10 | 0 | 60.10 |
| Baby-Window, Giant Window with $\ell = 4$ | 8.43 | 2.00 | 10.43 |
| Baby-Window, Giant Window with $\ell = 6$ | 6.59 | 4.55 | 11.14 |
| Baby-Window, Giant Window with $\ell = 8$ | 4.68 | 16.54 | 21.22 |

Table 4.1: Timings in seconds for one 1024-bit exponentiation on arbitrary OEF

knows the base $g$. To compute $(g^{x_B})^{x_A}$ Alice needs an exponentiation with a random base because she doesn't know the base $g^{x_B}$ before Bob sends it to her.

A signature algorithm like DSA requires one exponentiation with known base for signature generation and two exponentiations, one of these with a random base, for signature verification. Table 4.1 and Table 4.2 show the average timings of 20 test runs on a Type II OEF and on an arbitrary OEF. The first column shows the time needed to perform one exponentiation with known base, the second column shows the time needed to perform the precomputations and the last one shows the time needed to perform an exponentiation with random base. While an exponentiation with known base just takes the time for the exponentiation, working with random bases means that we also have to do precomputations.

The Baby-Window, Giant-Window algorithm offers a better performance than the Binary and the Montgomery Exponentiation, while Binary and Montgomery Exponentiation take about the same amount of time.

If we use a known base we can use a window length of $\ell = 8$ which performs an exponentiation in 4.68 seconds. To perform an exponentiation with an unknown base a window length of $\ell = 4$ offers the best performance in $2.00 + 8.43 = 10.43$ seconds to perform exponentiation. Hence DSA signature generation takes 4.6 seconds and a signature verification 15.11 seconds. Working on a Type II OEF results in 4.55 seconds for signature generation and 14.86 seconds for signature verification.

| | Exp. | Precomp. | Total |
|---|---|---|---|
| Binary Exponentiation | 57.32 | 0 | 57.32 |
| Montgomery Exponentiation | 58.45 | 0 | 58.45 |
| Baby-Window, Giant Window with $\ell = 4$ | 8.36 | 1.95 | 10.31 |
| Baby-Window, Giant Window with $\ell = 6$ | 6.38 | 4.36 | 10.74 |
| Baby-Window, Giant Window with $\ell = 8$ | 4.55 | 15.91 | 20.46 |

Table 4.2: Timings in seconds for one 1024-bit exponentiation on Type II OEF

# 5 Comparison

In this chapter we will discuss the timings we have achieved and compare them to other publications, especially to ECC on a Palm OS device.

We only found one timing of a DSA implementation on a Palm OS handheld in [4]. Brown et. al. used a Palm V with a 16 MHz Motorola 68000-type processor and Palm OS 3.0 and the DSA code from the OpenSSL and OpenPGP libraries with changes to fit the demands of the Palm platform. Because the Palm m500 we used for time measurements has a 33 MHz processor and Palm OS 4.0 we have used the benchmark comparison given in [8] to compare the speed performances of the two different devices. The m500 can perform integer operations about 1.85 times as fast as the Palm V. The conclusion is that the 1024-bit DSA implementation used in [4] for timing should take about 13.8 seconds for a DSA signature generation and 28.3 seconds for a DSA signature verification. Wee see that using OEFs and the Baby-Window, Giant-Window exponentiation results in a performance advantage of factor 3 for signature generation and of factor 1.9 for signature verification.

## 5.1 Comparison to ECC on a Palm OS Device

If we compare our results with the ones achieved by Weimerskirch, Paar and Shantz in [25] using Elliptic Curve Cryptography on a Palm OS device we see that a 1024-bit OEF DSA signature verification takes about 5.7 times the time of a 163-bit ECDSA signature verification and the ECDSA signature generation is even 7.5 times faster than OEF DSA signature generation. To understand why we have such big time differences we take a closer look at the processor functions needed to perform ECDSA as well as DSA operations.

The first step is to analyze the basic operations needed to perform a Baby-Window, Giant-Window exponentiation with window length $\ell = 4$. There are three operations, namely multiplication, reduction and application of the Frobenius map, on the extension field which are performed when using the Baby-Window, Giant-Window exponentiation.

The fastest way to perform the multiplication on the extension field is the recursive KA as given in Algorithm 8. The recursive KA performs the multiplication of two polynomials with 64 coefficients with $\#MUL = 729$ subfield multiplications and $\#ADD = 3864$ subfield additions.

To perform a reduction on the extension field we use Algorithm 10, which needs $\#MUL = 63$ subfield multiplications and $\#ADD = 63$ subfield additions. On a Type II OEF the extension field reduction can be performed faster because the multiplication with $\omega = 2$ can be performed as a 32-bit shift and, if needed, a subtraction with $p$ to reduce the result. The total cost of a Type II OEF extension field reduction is $\#SHFT = 63$ 16-bit integer shifts, $\#ADD = 32$ 16-bit integer additions and $\#ADD = 63$ subfield additions.

We can compute the first iteration of the Frobenius map as shown in Algorithm 24 which takes just $\#MUL = 63$ subfield multiplications. The precomputations performed for the computation of the Frobenius map will not be taken into consideration because

| | clock cycles |
|---|---|
| MUL | 70 |
| SHFT | 8 |
| AND | 4 |
| ADD | 4 |
| XOR | 4 |

Table 5.1: Number of clock cycles for different 16-bit processor functions

they are computed only once.

One subfield addition (see Algorithm 20) maps to one addition of two 16-bit integers and, in case that the result needs to be reduced, one 16-bit integer subtraction. In average the subtraction is performed 0.5 times per subfield addition. Thus one subfield addition needs one 16-bit integer addition and 0.5 16-bit integer subtractions. Because additions and subtractions take the same amount of time to be performed by the processor we can say that the subfield addition is performed using $\#ADD = 1.5$ 16-bit integer additions.

The subfield multiplication is performed by first multiplying two 16-bit integers and then performing an OEF subfield modular reduction as mentioned in Algorithm 19. On average each of the two while-loops is executed once. Hence one OEF subfield modular reduction needs 2 shifts, 2 ANDs, 1 multiplication and 1 addition. So one subfield multiplication including OEF reduction needs $\#MUL = 2$ multiplications, $\#SHFT = 2$ shifts, $\#AND = 2$ ANDs and $\#ADD = 1$ addition. All operations are applied to 16-bit integers.

The number of processor cycles needed to perform the different 16-bit processor functions can be found in [20] and are given in Table 5.1. Multiplications are performed very slowly compared to other operations.

Table 5.2 and Table 5.3 show the number of processor instructions and the total number of processor cycles needed to perform the different subfield and extension field

| | $\#MUL$ | $\#ADD$ | $\#SHFT$ | $\#AND$ |
|---|---|---|---|---|
| $GF(p)$ addition | 0 | 1.5 | 0 | 0 |
| integer multiplication | 1 | 0 | 0 | 0 |
| $GF(p)$ reduction | 1 | 1 | 2 | 2 |
| rec KA for $m = 64$ | 1458 | 6525 | 1458 | 1458 |
| OEF extension field reduction | 126 | 157,5 | 126 | 126 |
| OEF Type II field reduction | 0 | 127 | 63 | 0 |
| Frobenius Map | 126 | 63 | 126 | 126 |

Table 5.2: Number of processor instructions for extension field operations

| | processor cycles |
|---|---|
| $GF(p)$ addition | 6 |
| integer multiplication | 70 |
| $GF(p)$ reduction | 94 |
| rec KA for $m = 64$ | 145,656 |
| OEF extension field reduction | 10,962 |
| OEF Type II field reduction | 1,012 |
| Frobenius Map | 10,584 |

Table 5.3: Number of clock cycles for extension field operations

operations.

For exponentiation using a window size $\ell = 4$ results in 59 precomputed elements and one extension field multiplication and one extension field reduction for each of this elements. To compute the actual exponentiation we have to perform at most $K \cdot m = 4 \cdot 64 = 256$ extension field multiplications and reductions and $m - 1 = 63$ applications of the first iteration of the Frobenius map. As mentioned in [1], in Algorithm 17 one of the factors $B_{i,j} = 1$ with the probability $1/2^{\ell}$. Thus on average we have to perform just $(K - 1/2^{\ell}) \cdot m = (4 - 1/2^4) \cdot 64 = 252$ extension field multiplications. Table 5.4 and Table 5.5 show the number of clock cycles we need to compute the precomputed values and to perform the exponentiation using the precomputed values working on an arbitrary OEF and on a Type II OEF.

After we have evaluated the complexity to perform an exponentiation on $GF(p^m)$

| | processor cycles |
|---|---|
| Precomputations for $\ell = 4$ | 9,240,462 |
| Baby-Window, Giant-Window Exponentiation | 40,134,528 |

Table 5.4: Number of clock cycles for Precomputations and Exponentiation on arbitrary OEF

| | processor cycles |
|---|---|
| Precomputations for $\ell = 4$ | 8,653,412 |
| Baby-Window, Giant-Window Exponentiation | 37,627,128 |

Table 5.5: Number of clock cycles for Precomputations and Exponentiation on Type II OEF

we want to analyze the amount of processor cycles needed to perform point multiplication on random elliptic curves. For this purpose we will analyze the methods used for point multiplication presented in [25]. The most performant method to perform a point multiplication by a random point is the Montgomery point multiplication.

The Montgomery multiplication requires $6m$ field multiplications and squarings without any precomputations. Weimerskirch et. al. work with curves over $GF(2^{163})$, i.e. $m = 163$, which means that the Montgomery requires 978 field multiplications and squarings. In addition to the polynomial multiplications and squarings a field reduction needs to be performed for every polynomial operation. So there are also $2 \cdot 978 = 1956$ reductions to be taken into consideration.

The fastest way to perform a polynomial multiplication used in [25] is the Comb method with window size $w = 4$. This method needs 3 shifts over 6 32-bit words and 11 additions to perform precomputations. One addition can be computed by performing $\#XOR32 = 5$ XORs over 32-bit words and $\#XOR16 = 1$ XOR over 16-bit words. One shift over $n$ 32-bit words can be computed by performing $\#SHFT = 2 \cdot n$ 32-bit shifts and $\#XOR32 = n$ 32-bit XORs. The main loop has 44 runs, each one consisting of 1 addition. There are 3 additional shifts over 11 32-bit words. So the field multiplication

| | clock cycles |
|---|---|
| SHFT | 10 |
| AND | 6 |
| XOR | 8 |

Table 5.6: Number of clock cycles for different 32-bit processor functions

| | $\#XOR32$ | $\#XOR16$ | $\#AND$ | $\#SHFT$ |
|---|---|---|---|---|
| field multiplication | 326 | 55 | 0 | 102 |
| field squaring | — | — | — | — |
| modular reduction | 41 | 0 | 2 | 35 |
| Montgomery point mult. | 717,852 | 53,790 | 3,912 | 168,216 |

Table 5.7: Number of processor instructions for operations on elliptic curves

can be performed using 55 additions, 3 shifts over 6 32-bit words and 3 shifts over 11 32-bit words. To perform the 55 additions we need $\#XOR32 = 55 \cdot 5 = 275$ 32-bit XORs and $\#XOR16 = 55$ 16-bit XORs. To perform the 3 shifts over 6 32-bit words we have $\#SHFT = 3 \cdot 2 \cdot 6 = 36$ 32-bit shifts and $\#XOR32 = 3 \cdot 6 = 18$ 32-bit XORs and to perform the 3 shifts over 11 32-bit words $\#SHFT = 3 \cdot 2 \cdot 11 = 66$ 32-bit shifts and $\#XOR32 = 3 \cdot 11 = 33$ 32-bit XORs. The field multiplication takes $\#SHFT = 102$ 32-bit shifts, $\#XOR32 = 326$ 32-bit XORs and $\#XOR16 = 55$ 16-bit XORs.

Squaring can be performed using table lookups. If we take a look at Table 1 in [25] we see that a squaring takes about 0.21 the time of a field multiplication.

The algorithm to perform reduction given in [25] uses $\#XOR32 = 41$ XORs, $\#SHFT = 35$ shifts and $\#AND = 2$ ANDs all performed over 32-bit words.

Table 5.6 shows the number of processor cycles needed to perform processor functions over 32-bit words and Table 5.7 and Table 5.8 show the number of processor instructions and clock cycles needed to perform operations on elliptic curves.

Now we can compute the number of processor cycles used for the computation of field operations and the Montgomery point multiplication.

The average timing for a Montgomery point multiplication given in [25] is 2.73 sec,

| | processor cycles |
|---|---|
| field multiplication | 3,848 |
| field squaring | 808 |
| modular reduction | 690 |
| Montgomery point mult. | 5,903,208 |

Table 5.8: Number of clock cycles for operations on elliptic curves

measured on a Handspring Visor with a Motorola 16 MHz Dragonball EZ processor. Although the Palm m500 has a processor speed twice as fast as the Handspring Visor, the effective speed-up for integer arithmetic is just about 1.35 as given in [8]. So the timing for a Montgomery point multiplication on the Palm m500 handheld should be about 2 sec, compared to 10.26 sec we need to perform an exponentiation with unknown basis on $GF(p^m)$. The number of processor cycles needed to perform one Montgomery point multiplication and an exponentiation over $GF(p^m)$ explain the speed-up of factor more than 5. As seen in Table 5.4 the number of cycles to perform an exponentiation on $GF(p^m)$ is about 49,000,000 while the number of cycles to perform a Montgomery point multiplication is about 5,900,000. Also if we work with a Type II OEF the complexity of ECC has a big performance advantage. A main aspect why elliptic curves perform much faster than $GF(p^m)$ is that there are no integer multiplications involved.

Note that we have only analyzed the number of processor functions needed for the algorithms. We did not take into consideration the instructions for assigning values to variables etc. because these are not part of the actual algorithms.

## 5.2 Comparison to DLP in $GF(2^k)$

In Section 5.1 we have seen that elliptic curve arithmetic has a better performance on a Palm device then our arithmetic on $GF(p^m)$. In this section we want to compare the arithmetic we used to compute exponentiation on $GF(p^m)$ to arithmetic over binary

fields $GF(2^k)$, for key lengths of 1024-bit $k = 1024$. The elements of the binary field can be represented as polynomials of length $k$ written as

$$a(x) = \sum_{i=0}^{k-1} a_i x^i, \text{ with } a_i \in \{0, 1\} \tag{5.1}$$

The coefficients $a_i$ are also referred as the bits of $a$. Because the Palm device has a word length of $w = 16$ bit we will write $a$ as a $k$-bit number, consisting of $s = k/w = 64$ blocks. Thus we have $a = (A_{s-1}A_{s-2} \dots A_1A_0)$, with $A_i = (a_{i*s+15} \dots a_{i*s})$, $64 > i \geq 0$.

The first algorithm to compute exponentiation on $GF(2^k)$ we mention was proposed by Koç and Acar in [14]. They used the Montgomery Exponentiation to compute $a^e$ with $a \in GF(2^k)$ and $e$ an integer. The number of operations needed to perform the Montgomery Exponentiation can be found in [14]. Because the inversion has to be performed only once on the binary field we will not take it into consideration. Koç and Acar assume the simplification that XOR, AND and OR operations take the same amount of processor cycles which is true for the Motorola Dragonball processor if we look at Table 5.1. Thus to perform the Montgomery Exponentiation we need $s^2(\frac{3}{2}w + 7 + 4m) + s(\frac{w}{2} + 2wm + m)$ XOR/AND/OR, $s^2(2w + 2) + s(2m + 2wm + 1)$ shifts and $s^2(3 + 2m) + s(\frac{3}{2}m + 1)$ multiplications of two 16-bit polynomials. Working with a processor word length of $w = 16$, $s = 64$ blocks per field element and an exponent $e$ with bit length $m = 1024$ results in a total of 19,067,396 XOR/AND/OR, 2,303,040 shifts and 8,499,264 multiplications of two 16-bit polynomials. Koç and Acar proposed in [13] a fast way to perform word-level $GF(2)$ polynomial multiplication using table lookups. For a word length $w = 16$ these tables would need 8 Gigabytes of memory which is not realizable working on a Palm device with 8 Megabytes of memory. They also proposed a hybrid approach which uses two 8-bit tables (each of the tables is of size 64 Kilobytes). This approach performs the multiplication of two 16-bit polynomials with 4 shifts, 8 XOR/ANDs. Because precomputations for the tables have to be done

only once we will not take it into consideration. We can compute the number of 16-bit processor functions we have to perform for the Montgomery Exponentiation on $GF(2^k)$ with $k = 1024$. We have $\#XOR = 19,067,296 + 8 \cdot 8,499,264 = 87,061,408$ XORs and $\#SHFT = 2,303,040 + 4 \cdot 8,499,264 = 36,300,096$ shifts. With the help of Table 5.1 we can compute a total of 638,646,400 processor cycles to perform the exponentiation on $GF(2^k)$ using the Montgomery Exponentiation. We see that working on OEFs and using the Baby-Window, Giant-Window exponentiation gains us a speed-up of factor 13 compared to the Montgomery Exponentiation on $GF(2^k)$. We have to mention at this point that the Montgomery Exponentiation analyzed by Koç and Acar in [14] works with arbitrary field polynomials.

The second algorithm to compute exponentiation $a^e$, $a \in GF(2^k)$ we want to analyze is the Sliding Window Technique. It is an optimized version of the $m$-ary method which itself is a generalization of the binary method given in Section 3.3.1. Koç gives a detailed analysis of the Sliding Window Technique in [15] which we will use to determine the complexity of exponentiation on $GF(2^k)$ with $k = 1024$ and the bit-length of the exponent $m = 1024$. In [15] we find that working with a maximum nonzero window length $d = 6$ gives us the best performance. Hence we need 1 squaring and $2^{d-1} - 1 = 31$ multiplications for the precomputations and additional $k = 1024$ squarings and $k/(d+1) = 147$ multiplications to perform $a^e$, $a \in GF(2^{1024})$. We have a total of $1024 + 1 = 1025$ squarings and $31 + 147 = 178$ multiplications.

To perform the multiplications on $GF(2^{1024})$ we use the algorithm proposed by López and Dahab in [16]. It can be only applied for special types of field polynomials, namely trinomials and pentanomials. We can use the method introduced in Algorithm 5 in [16] to compute $c = a \cdot b \bmod f$, with $a, b \in GF(2^k)$ and $f$ an irreducible polynomial of degree $k$ over $GF(2^k)$. The algorithm works with a window length $\ell = 4$, but can be generalized to work with an arbitrary window length $\ell$. The number of precomputations

is $2^\ell - 2$ which are computed using $\ell - 1$ shifts over 64 words and $2^\ell - \ell - 1$ additions over 64 words. One addition can be computed by performing $\#XOR = 64$ XORs over 16-bit words. One multi-word shift over $n$ 16-bit words can be computed by performing $\#SHFT = 2 \cdot n$ 16-bit shifts and $\#XOR = n$ 16-bit XORs. The computation of $a \cdot b$ is then performed using $\frac{w \cdot (s-1)}{\ell}$ additions over 64 words and $\frac{w}{\ell} - 1$ shifts over 127 words.

Thus we need $\#SHFT = 128(\ell - 1)$ 16-bit shifts and $\#XOR = 2^{\ell+6} - 128$ XORs to compute the precomputations and $\#SHFT = 254(\frac{w}{\ell} - 1))$ shifts and $\#XOR = 64 \cdot \frac{w}{\ell} s + 63\frac{w}{\ell} - 127$ XORs to compute the multiplication using the precomputed values.

Squaring can be performed faster than multiplication [10]. They use 32-bit words and table lookups to compute $c = a^2$. The tables need to be computed only once. The algorithm uses $\#SHFT = \frac{k}{32} \cdot 5 = 160$ 32-bit shifts and $\#AND = \frac{k}{32} \cdot 6 = 192$ 32-bit ANDs to perform the polynomial squaring. In the last step a reduction is performed to obtain $c = a^2 \bmod f$.

Assuming that the reduction polynomial is a pentanomial the reduction can be performed with $\#XOR = 8 \cdot 127 = 1,016$ 16-bit XORs, according to [16].

The best performance for the field multiplication is achieved with window length $\ell = 6$ which results in a total of 225 additions over 64-words, 5 shifts over 64 words and 2 shifts over 127 words. In addition we have $8 \cdot 127 = 1,016$ 16-bit XORs for the reduction. This results in a total of $\#SHFT = 1,148$ 16-bit shifts and $\#XOR = 15,990$ 16-bit XORs to compute $a \cdot b \bmod f$. This is just a theoretical value. Working with a window length $\ell = 6$ results in precomputated values that have to be stored in 32-bit registers. The computation of the result will lead to performance decreases because of the need to shift between registers.

We want to compute also the complexity for the multiplication using the window length $\ell = 4$. In this case we have 263 additions over 64-words, 3 shifts over 64 words and 3 shifts over 127 words. In addition we have $8 \cdot 127 = 1,016$ 16-bit XORs for the

| | $\#XOR$ | $\#SHFT$ | processor cycles |
|---|---|---|---|
| Montgomery | 87,061,408 | 36,300,096 | 638,646,400 |
| Sliding Window | 4,320,338 | 203,988 | 21,734,056 |

Table 5.9: Number of processor instructions and clock cycles for exponentiation on $GF(2^k)$

reduction. This results in a total of $SHFT = 1146$ 16-bit shifts $\#XOR = 18421$ 16-bit XORs to compute $a \cdot b \bmod f$.

To compute $a^2 \bmod f$ we need $\#XOR = 1,016$ 16-bit XORs, $\#SHFT = 160$ 32-bit shifts and $\#AND = 192$ 32-bit ANDs.

Choosing the window length $\ell = 4$ for the multiplication we have $\#SHFT16 = 178 \cdot 1,146 = 203,988$ 16-bit shifts, $\#SHFT32 = 1025 \cdot 160 = 164,000$ 32-bit shifts, $\#XOR = 178 \cdot 18421 + 1025 \cdot 1016 = 4,320,338$ 16-bit XORs and $\#AND = 1025 \cdot 192 = 196,800$ 32-bit ANDs. With help of Table 5.1 and Table 5.6 we compute a total of 21,734,056 processor cycles needed to perform exponentiation on $GF(2^k)$ using Sliding Windows Technique and the multiplication algorithm proposed by López and Dahab. Compared to OEFs and the Baby-Window, Giant-Window exponentiation, this method gains a speed-up of factor 1.9. This speed-up may even be raised if a window length $\ell = 6$ for the multiplication is chosen and implemented without performance decreases.

Table 5.9 shows the number of 16-bit processor instructions and clock cycles used to perform exponentiation on $GF(2^k)$, $k = 1024$ using the Montgomery Exponentiation and the Sliding Window Exponentiation.

In Table 5.10 we see the performance values of the algorithms analyzed in this chapter. ECC, which offers the best performance, has the reference value 1 and we see that Sliding Window on $GF(2^k)$ performs faster than the Baby-Window, Giant-Window algorithm and especially than the Montgomery Exponentiation on $GF(2^k)$.

| | Speed Performance |
|---|---|
| Montgomery Exp. on $GF(2^k)$ | 108.1 |
| Baby-Window, Giant-Window on OEF | 6.8 |
| Sliding Window on $GF(2^k)$ | 3.78 |
| ECC on $GF(2^m)$ | 1 |

Table 5.10: Relative performance of different algorithms compared to DLP on OEFs

# 6 Conclusions and Future Research

In this thesis we analyzed the performance of exponentiation on Optimal Extension Fields (OEF) and implemented different exponentiation algorithms in software designed to fit the needs of embedded systems like the Palm OS platform. We have also introduced a new algorithm to compute the Frobenius Map on arbitrary OEFs.

The best performance has been achieved using the Baby-Window, Giant-Window algorithm which results in 4.55 seconds for an exponentiation with known base and 10.43 seconds for an exponentiation with random base. Although this result is an improvement compared to former implementations of DLP on embedded systems it probably will not be tolerated by most users.

Comparing our results to DLP on binary fields and ECC shows that ECC are the more efficient way of performing fast cryptographic algorithms on Palm devices. Also the choice of DLP on $GF(2^k)$, using the Sliding Window exponentiation results in a better performance than working on OEFs.

The next version of the Palm OS operating system, Palm OS 5.0 will also support ARM-processors which have a clock speed of more than 200 MHz. Obviously proper running times of DLP based cryptography will be possible for devices equipped with the new OS and ARM-processors.

It would be interesting to see if using the new Palm OS 5.0 and working on a PDA

based on the Intel ARM-processor has a better performance. Also the complexity when working with OEFs might improve compared to DLP on $GF(2^k)$ or even ECC if the ARM-processors support a faster integer multiplication.

Another open research problem is to analyze, if the DLP on OEFs is more secure than DLP on $GF(p)$ and DLP on $GF(2^k)$.

# Bibliography


[1] R. Avanzi and P. Mihăilescu. *Generic Performant Arithmetic Algorithms for PAFFs (Processor Adequate Finite Fields)*, Prepint.

[2] D. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public Key Algorithms. *Advances in Cryptology – Crypto '98 (LNCS 1462)*, pp. 472–485, Santa Barbara, 1998.

[3] D. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Applications in Elliptic Curve Cryptography, *Journal of Cryptology*, 2001, vol. 14, no. 3, pp. 153-176.

[4] M. Brown, F. Cheung, D. Hankerson, J. Lopez Hernandez, M. Kirkup, A. Menezes. PGP in Constrained Wireless Devices, *Proceedings of the 9th USENIX Security Symposium*, Denver, August 2000.

[5] M. Brown, D. Hankerson, J. López and A. Menezes. Software Implementation of the NIST Elliptic Curves over Prime Fields, *Topics in Cryptology - CT-RSA 2001*, Lecture Notes in Computer Science, 2001, pp. 250-265

[6] J. Buchmann. *Einführung in die Kryptographie*, Springer-Verlag Berlin Heidelberg, March 2001

[7] P. Cameron. *Introduction to Algebra*, Oxford University Press, June 1998

[8] F. Carello. *PocketMark - a free benchmark test for the PalmOS platform*, available from http://web.tiscali.it/fcarello/pocketmark.html

[9] D. Estrin, R. Govindan, and J. Heidemann. Embedding the internet: Introduction. *Communications of the ACM*, 43(5):38-42, May 2000.

[10] D. Hankerson, J. Hernandez and A. Menezes. Software Implementation of Elliptoc Curve Cryptography Over Binary Fields. *Cryptographic Hardware and Embedded Systems, CHES 2000*, LNCS 1965, Springer-Verlag, 1-24, 2000.

[11] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata, *Soviet Physics - Doklady*, 7 (1963), pp. 595-596.

[12] N. Koblitz. Algebraic Aspects of Cryptography, *Algorithms and Computation in Mathematics Vol. 3*, Springer-Verlag Berlin Heidelberg, November 2001.

[13] Ç. Koç and T. Acar. Montgomery Multiplication in $GF(2^k)$, *Designs, Codes and Cryptography*, 14(1), 57-69 (April 1998), Kluwer Academic Publishers, Boston.

[14] Ç. Koç and T. Acar. Fast Software Exponentiation in $GF(2^k)$, *13th Symposium on Computer Arithmetics*, pp. 225-231, IEEE Computer Society Press, July 1997.

[15] Ç. Koç Analysis of sliding window techniques for exponentiation, *Computers and Mathematics with Applications*, 30(10):17-24, 1995.

[16] J. López and R. Dahab. High-Speed Software Multiplication in $\mathbb{F}_{2^m}$, *IC Technical Reports*, IC-00-09, Institute of Computing, University of Campinas, May 2000, Available from http://www.dcc.unicamp.br/ic-main/publications-e.html.

[17] A. Menezes, P. van Oorschot and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, August 2001.

[18] P. Mihăilescu. Optimal Galois Field Bases which are not Normal, *Recent Results Session*, Fast Software Encryption, 1997.

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

[20] Motorola Inc. *M68000 8-/16-/32- Bit Microprocessors User's Manual*, 1993.

[21] Motorola Inc. *M68000 Family Programmer's Reference Manual*, 1992.

[22] B. Schneier. *Applied Cryptography*, John Wiley & Sons, 1996.

[23] R. Sedgewick. *Algorithmen in C*, Addison-Wesley (Deutschland) GmbH Bonn, 1992, 1. Auflage, pp. 591-601.

[24] A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Polynomial Multiplication, *Technical Report, Communication Security Group, Ruhr-University Bochum, 2002*, available at http://www.crypto.rub.de.

[25] A. Weimerskirch, C. Paar and S. Shantz. Elliptic Curve Cryptography on a Palm OS Device, *The 6th Australian Conference on Information Security and Privacy (ACISP 2001)*, July 2001.