[chapter]

KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Public Key Cryptography in 32-bit for Ultra Low-Power Microcontrollers

P.J. de Clercq

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Embedded Systems and Multimedia

Thesis supervisor:
Prof. Dr. Ir. I. Verbauwhede

Assessor:
Dr. Ir. F. Vercauteren
Prof. Dr. Ir. W. Dehaene

Mentors:
Dipl. Ing. L. Uhsadel
Ir. A. Van Herrewege

Academic year 2013 – 2014

© Copyright KU Leuven

Without written permission of the thesis supervisor and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to Departement Elektrotechniek, Kasteelpark Arenberg 10 postbus 2440, B-3001 Heverlee, +32-16-321130 or by email `info@esat.kuleuven.be`.

A written permission of the thesis supervisor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

List of Figures

2.1 The three layers of an ECC system. 7
2.2 A point addition is performed on an elliptic curve using the tangent-and-chord method. 11
2.3 The two-stage pipeline inside the ARM Cortex-M0+. The decode stage is split over two cycles, which allows the processor to complete an instruction in two cycles. This image is based on an image found in [1]. 29
2.4 Energy usage per cycle for different microcontrollers. 30

3.1 The input parameters are split into three equal parts and the following multiplication methods are used: a) Schoolbook multiplication b) Operand-scanning c) Optimized operand-scanning. 38
3.2 Splitting the input parameters a) Symmetrically into two parts of equal length b) Asymmetrically on the word size of the processor. The word size for this example is 32-bit. 39
3.3 An asymmetric split causes the most significant polynomial to be completely populated with bits and will therefore require a larger lookup table. 40
3.4 Cost estimation for different values of w. 54

4.1 Energy consumption of the ADD instruction on the target platform. . . 76
4.2 Energy consumption of a fixed point multiplication on the target platform. Each peak includes 10 identical fixed point multiplications on identical keys. 77
4.3 A breakdown of the execution time for a random point multiplication. . 84
4.4 A breakdown of the execution time for a fixed point multiplication. . . . 84
4.5 Determining the optimal values for the window parameter w for the field multiplication algorithms on $\mathbb{F}_{2^{233}}$. 89
4.6 Point multiplications on different word sizes for $\mathbb{F}_{2^{233}}$. 90

List of Tables

2.1 Security levels reported in ECRYPT [2]. 6
2.2 The key strength of various key length of Elliptic Curves are compared to RSA and symmetric key cryptography. 6
2.3 The number of operations required to perform a Point Addition and a Point Doubling, for the Affine (A), Projective (P), Jacobian (J) and López-Dahab (LD) coordinate systems on prime fields. The field multiplication operation is indicated with M, and a field inversion is indicated with I. 16
2.4 The number of operations required to perform a Point Addition and a Point Doubling, for the Affine (A), Projective (P), Jacobian (J) and Chudnovsky (C) coordinate systems on prime fields. The field multiplication operation is indicated with M, the polynomial square operation is indicated with S and a field inversion is indicated with I. . 21

3.1 Prime field simulation. * ECE is an abbreviation for Efficiently computable endomorphism. 34
3.2 Binary field simulation. 35
3.3 Analysis of the read instructions required to perform different field multiplications. 52
3.4 Analysis of the write instructions required to perform different field multiplications. 52
3.5 Analysis of the XOR instructions required to perform different field multiplications. 52
3.6 Analysis of the shift instructions required to perform different field multiplications. 53
3.7 Analysis of the costs associated with different field multiplications on the target platform. This assumes that $n = 8$, $W = 32$ and $w = 4$. The sum column represents the weighted sum of all the instructions with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts. 53
3.8 The memory requirements and cycle counts associated with different values of the parameter w for the `López-Dahab with fixed registers` algorithm. The cycle count includes precomputation and multiplication. 53

3.9 Analysis of the read instructions required to perform different field multiplications. . . . 59
3.10 Analysis of the write instructions required to perform different field multiplications. . . . 59
3.11 Analysis of the XOR instructions required to perform different field multiplications. . . . 59
3.12 Analysis of the shift instructions required to perform different field multiplications. . . . 60
3.13 Analysis of the instruction count required to perform different field multiplications. This assumes that $n = 8, W = 32, w = 4$. The sum column represents the estimated cycles and is calculated with the weighted sum of all the instructions, with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts. . 60
3.14 A summary of the memory and cycle count costs associated with all the different analysed methods for $n = 8, W = 32, w = 4$. The sum column represents the weighted sum of all the instructions with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts. . . . 61
3.15 The cycle counts of the different functions during optimization. . . . 69
3.16 The number of times the different functions were executed during optimization. . . . 70
3.17 Simulation vs Implementation. . . . 70
3.18 Lookup table memory consumption for different algorithms. . . . 71

4.1 The energy used per cycle for different instructions. The clock frequency is 48MHz. . . . 75
4.2 Energy measurements of point multiplications. The clock frequency is 48MHz. The *measured energy* is column indicates the average measured energy, and the *energy* column indicates the consumed energy per operation. . . . 76
4.3 Timings for point multiplications. All timings are given in milliseconds and energy is given in millijoules (mJ). The ATMega128L runs at 7.37MHz except when indicated with [a], the MSP430 runs at 8.192MHz, the ARM7TDMI runs at 80MHz, and the ARM Cortex-M0+ runs at 48MHz . . . 78
4.4 Timings in cycles for modular multiplication and modular squaring on different platforms. . . . 79
4.5 Timings in cycles for field arithmetic algorithms in $\mathbb{F}_{2^{233}}$. . . . 82
4.6 Total accumulated timings per operation for random and fixed-point multiplication. . . . 83
4.7 Optimal values for w on $\mathbb{F}_{2^{233}}$. . . . 90
4.8 Memory operations on the lookup tables required for the Asymmetrical Hybrid, Symmetrical Hybrid and Symmetrical Karatsuba methods by using a factor two split in $\mathbb{F}_{2^{233}}$. . . . 91

4.9 Memory operations on the lookup tables required for the Symmetrical Hybrid vs Symmetrical Karatsuba methods by using a factor two split in $\mathbb{F}_{2^{233}}$. 91
4.10 Memory overhead for Symmetrical and Asymmetrical methods in $\mathbb{F}_{2^{233}}$. Symmetrical overhead is the cost of performing the pre and post shift operations (Eq. 3.7). Asymmetrical overhead is calculated from the additional memory accesses due to the larger lookup table. 91
4.11 Memory operations to the intermediate values stored inside memory in $\mathbb{F}_{2^{233}}$. The `Symm/Asymm` column is used to indicate that both the `Asymmetrical Hybrid`, and the `Symmetrical Hybrid` methods do not use any intermediate values stored in memory. 92

List of Algorithms

1 ECDSA signature generation. 9
2 ECDSA signature verification. 9
3 Double-and-add method for point multiplication in $\mathbb{F}_q$. 12
4 Non Adjacent Form (NAF) of a positive integer. 13
5 Binary NAF method for point multiplication. 13
6 Window NAF method for point multiplication. 14
7 Right-to-left shift-and-add polynomial multiplication. 18
8 Left-to-right comb method for polynomial multiplication. 19
9 Basic modular reduction. 19
10 Inversion in $\mathbb{F}_{2^m}$. 20
11 Addition in $\mathbb{F}_p$. 23
12 Subtraction in $\mathbb{F}_p$. 23
13 Integer multiplication (operand-scanning form). 24
14 Binary algorithm for inversion in $\mathbb{F}_p$. 25
15 López-Dahab multiplication in $\mathbb{F}_{2^m}$. 42
16 López-Dahab with rotating registers multiplication in $\mathbb{F}_{2^m}$. 43
17 López-Dahab multiplication with fixed registers in $\mathbb{F}_{2^m}$ for $n = 8$. . . 44
18 López-Dahab multiplication with fixed registers in $\mathbb{F}_{2^m}$ for $n = 8$ and $s = 8$. 45
19 Fast reduction modulo $f(z) = z^{233} + z^{74} + 1$ (with W=32). 46
20 Polynomial squaring (with word length W=32). 47
21 Polynomial squaring with reduction (with word length W=32). . . . 48
22 Inversion in $\mathbb{F}_{2^m}$ using no swap operation. 49

Abbreviations, parameters, and mathematical symbols

$\&$	Logical AND
$\gg$	Logical shift right
$\leftarrow$	Multi-precision assignment
$\ll$	Logical shift left
$\oplus$	Bitwise XOR
m	Degree of the field parameter
n	Number of words in field parameter
s	Scanning size parameter
W	Processor word size parameter
w	Window parameter
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Helman
ECDL	Elliptic Curve Discrete Logarithm
ECDSA	Elliptic Curve Digital Signature Algorithm
Eq.	Equation
Fig.	Figure
LD	Field multiplication algorithm by López and Dahab
PKC	Public-Key Cryptography

Preface

I am grateful to my promoter, Ingrid Verbauwhede for providing me with the opportunity of doing this thesis under her supervision. I would further like to thank my daily supervisors Anthony Van Herrewege, and Leif Uhsadel for their continuous support and advice. They provided me with guidance to find great algorithmic solutions, their critical eyes in proof-reading helped shape this manuscript to something I am proud of, their sound practical advice on a variety of technical topics, and their speedy responses to all my questions via email, are what made this thesis possible.

My sincere gratitude goes to my girlfriend Ansie Yssel for her continuous support, and for bearing with the fact that I was sacrificing all my time on my thesis and spending very little time on her. I would further like to thank her for all her advice on technical writing.

I am very grateful to *ema2sa* for providing me with the financial means for studying towards a Masters degree in Electrical Engineering at KU Leuven. Without this scholarship I would never have been able to study at such a fine educational institution. My time here has provided me with a wealth of knowledge which I will carry with me to the end of my days.

Last, but not least, I would like to thank my family and parents-in-law for their support, and for regrettably not being able to see them more often.

P.J. de Clercq

Contents

List of Figures i

List of Tables ii

Preface vii

Abstract ix

1 Introduction 1
- 1.1 An introduction to Public-Key Cryptography . . . 1
- 1.2 PKC on wireless sensor networks . . . 2
- 1.3 Goals of the thesis . . . 3
- 1.4 Thesis outline . . . 4

2 Theory and Background 5
- 2.1 Introduction . . . 5
- 2.2 Elliptic Curve Protocols . . . 8
- 2.3 Elliptic Curve Arithmetic . . . 10
- 2.4 Field Arithmetic . . . 14
- 2.5 Low-Power Processors and Microcontrollers . . . 25
- 2.6 Conclusion . . . 29

3 Implementation 31
- 3.1 Matching a curve to the architecture . . . 31
- 3.2 Actual algorithms used . . . 36
- 3.3 Analysis of algorithms . . . 47
- 3.4 Optimizations . . . 63
- 3.5 Conclusion . . . 71

4 Results 73
- 4.1 Measurement setup . . . 73
- 4.2 Measurement results . . . 74
- 4.3 Comparison with other libraries . . . 75
- 4.4 Analysis of the effect of different processor word sizes . . . 84
- 4.5 Conclusion . . . 88

5 General Conclusion 93
- 5.1 Future work . . . 94

Bibliography 97

Abstract

In this work, Elliptic Curve Cryptography (ECC) is used to make an efficient implementation of a public key cryptography algorithm on the ARM Cortex-M0+. A system is introduced to make energy consumption measurements of the ARM Cortex-M0+ microcontroller. Measurements show that there is a large variation (up to 22.5%) in the energy consumption of different instructions. When comparing a simulation of a point multiplication on a binary curve to that on a prime curve, the former will have a slightly faster execution time with a lower power consumption. We show that binary curve arithmetic uses instructions which requires less energy than prime curve arithmetic on the target platform. A new field multiplication algorithm is proposed called `López-Dahab with fixed registers`, which is an optimization of the `López-Dahab` (`LD`) algorithm. The proposed algorithm has a performance improvement of 20.24% over the `LD with rotating registers` algorithm. Our implementation of the `LD with fixed registers` field multiplication algorithm uses a binary Koblitz curve in $\mathbb{F}_{2^{233}}$, the wTNAF point multiplication method with $w = 4$ for random point multiplication, and $w = 6$ for fixed point multiplication. The RELIC toolkit is used to perform the wTNAF precomputation, and wTNAF transformation of the scalar k. All the elliptic curve arithmetic, and field arithmetic are implemented in C and assembly. On average, our implementation of a random point multiplication requires 2814827 cycles, and 34.16 µJ, whereas our fixed point multiplication requires 1864470 cycles, and 20.63 µJ. The proposed `LD with fixed registers` algorithm requires 3672 cycles to perform a field multiplication. A number of different field multiplication methods are developed with the `LD with fixed registers` algorithm, and a simulation is made of these field multiplication methods for different processor word sizes. Findings show that the `LD with rotating registers` algorithm will always outperform the other algorithms, regardless of the word size. The energy consumption of our implementation beats all known software implementations, of a point multiplication, on the same equivalent security level on other platforms by a factor of 7.4. Our cycle count for a field squaring operation also compare favourably to that of other implementations.

Chapter 1

Introduction

The purpose of this chapter is to serve as a general introduction to the thesis topic. The research field of the topic is described in a broad context with an introduction to public-key cryptography and a discussion on the goals of the thesis.

1.1 An introduction to Public-Key Cryptography

Cryptography is the art of providing secure communications over an insecure communication channel. We will briefly introduce symmetric key cryptography and public-key cryptography; followed by an introduction to two Public-Key Cryptography (PKC) algorithms.

Symmetric-key cryptosystems use the same keys to perform both the encryption and decryption of messages. They are very efficient at performing these cryptographic operations but they require that all parties have agreed upon the same key. A problem associated with symmetric-key cryptography is called the key management problem. In a network of n users, each user can communicate with $n-1$ other users. Each pair of users require a different key, which leads to a total of $n \cdot (n-1)/2$ unique keys that are needed for all users to be able to communicate with each other. This also means that $n-1$ keys need to be stored by each user and storing the keys can become a problem if a large number of users are present on the network. This also leads to the problem of storing all the keys securely by each user.

In a Public-Key Cryptographic (PKC) system, a key pair is selected so that one key is public and the other key is private. The two keys are mathematically linked in a way so that the one key can be used to encrypt a message while the other key can be used to decrypt the encrypted message. It is easy to generate the key pair but the problem of deriving the private key from the public is computationally infeasible or believed to be very hard. To send a message from Alice to a Bob, the message is encrypted using Bob's public key, but only Bob can decrypt the message using his private key. The public key and all parameters of the cryptosystem may be published without compromising the security of the cryptosystem, while the private key needs to be kept secret.

PKC systems can be used to perform the following three protocols: key establish-

ment, digital signatures, and encryption. Key establishment protocols are used to generate a secret key between two parties over an insecure channel. Digital signatures provide authentication, which means that the two communicating parties can guarantee that they are the ones generating the messages between each other. Encryption provides secure communications between parties. PKC is the only type of system that can achieve non-repudiation. Non-repudiation is the ability to provide prove the integrity and origin of the data. While PKC systems can be used for many different cryptographic tasks, their biggest drawback are that they are computationally very demanding. The best symmetric key schemes are always several orders of magnitude faster than a properly implemented PKC system. This gives rise to the use of Hybrid cryptosystems where the key exchange is done using a public key system and the computationally less demanding symmetric key cryptosystem is used to efficiently perform the encryption and decryption operations. This solves the problem of each user needing to store the key to every other user as the two communicating parties can just exchange their public keys over the insecure channel, generate a secret key by means of a key establishment protocol, and use this key to encipher communications between each other.

The first public key cryptographic system that was ever published in literature is due to Rivest, Adleman and Shamir (RSA) in 1978 [3]. It is based on the difficulty of factoring a large integer which is composed of two large prime factors.

Miller [4] and Koblitz [5] independently introduced the idea of Elliptic Curve Cryptography (ECC) in 1986. It is based on the ability to perform a multiplication of a scalar k with a point P, and the difficulty of obtaining the scalar k given the original point P and product kP. This is called the Elliptic Curve Discrete Logarithm problem. ECC is mathematically more complex than RSA, but its key size is smaller and it's operations can be performed faster for the same level of security.

1.2 PKC on wireless sensor networks

A wireless sensor network (WSN) is an ad-hoc wireless network that consists of a number of nodes and one or more base stations. The nodes are designed so that they can be deployed unattended for months or even years. Each node consists of a microcontroller, a radio unit, one or more sensors and an energy source like a battery or a mechanism to perform energy harvesting. Such a sensor network can consist of a large amount of nodes which pass data through the network to the base stations for data analysis. The nodes route the data through the network by transmitting it from node to node and the topology of the networks range from a star topology to a multi-hop mesh network. Typical applications for such a network are environmental monitoring, agricultural control, health monitoring and smart home monitoring and military applications like battlefield surveilance and reconnaisance. Security mechanisms are required in some of these applications to guarantee the confidentiality, integrity and authenticity of the data.

WSNs require security because they communicate through an insecure communication medium and they often operate unattended. The nodes cannot be made

tamper proof because they are often made to be as cheap as possible with no tamper-resistance. As these devices are made to be economically viable, they have a limited amount of energy, computation power, memory and communication abilities. A node's lifetime is also directly influenced by the amount of energy that it uses to perform computations and is therefore also directly influenced by the efficiency of its algorithms.

Due to their expensive computational requirements, RSA and DSA are considered to be impractical for use in WSNs where the devices are very constrained in processing power and energy. ECC is an attractive alternative due to its low computational and memory requirements, and is particularly attractive for use in Hybrid cryptosystems where PKC is used for key exchange, and symmetric cryptography is used for the efficient encryption of data. Digital signature can also be useful for WSNs as they can be used to guarantee the authenticity and integrity of the data. To perform a key exchange the Elliptic Curve Diffie-Helman exchange (ECDH) can be used and for generating digital signatures the Elliptic Curve Digital Signature Algorithm (ECDSA) can be used.

1.3 Goals of the thesis

The goal is of the thesis is to implement a Public-Key Cryptographic algorithm on the ARM Cortex-M0+ [6]. Efficient long number arithmetic should be performed on the architecture using assembly optimizations for the processor's instruction set. The results should be compared to the existing solutions in the ultra low-power domain, as well as to standard libraries which have been compiled on ARM.

The ARM Cortex-M0+ is designed for a broad range of embedded applications. It's a low cost, ultra low-power mircocontroller that provides great performance due to its 32-bit architecture and it features a small but powerful instruction set. Since this processor has only been available since 2012, we do not know of any other PKC implementations optimized for this architecture. By being the first to make an implementation on this architecture we are setting a benchmark to which the performance of future implementations can be measured. Freescale is using the Cortex-M0+ in a wireless sensor node called the MKW01Z128 [7].

In the past a lot of research has gone towards doing software implementations on WSNs like the ATMega128L based MICAx, the MSP430 based TelosB and only a small number of implementations can be found in literature for ARM processors like the IMote2.

We chose to make an ECC implementation due to its low processing and storage requirements. To allow for efficient long number arithmetic, we will analyse the architecture of the device and will select the ECC parameters that will lead to the best performance for the architecture. We will further analyse many different algorithms and methods to try to find an efficient method of implementing the ECC operations.

1.4 Thesis outline

In Chapter 2 we discuss the theory of elliptic curve cryptography and provide some background to the processors which we chose as the target platform as well as the processors which are often used in the ultra low-power domain.

In Chapter 3 we describe how our approach to select the appropriate elliptic curve parameters in order to allow for an efficient implementation on the target platform. We will develop a model to identify the curve which will lead to a good match on the target architecture. Various methods to perform the underlying field operations are analysed and discussed in order to determine which methods will lead to the fastest execution time. The optimizations which were done in the C and assembly implementations will also be discussed.

In Chapter 4 we provide the timings and energy estimates/measurements of our proposed implementation, the implementations found in literature, as well as the implementations from software libraries. A description of the implementations are provided and all the results are compared. The models developed for the different field multiplication methods in Chapter 3 are used to analyse the algorithms for various different processor word lengths.

In Chapter 5 we will provide a general conclusion as well as some ideas for future work that could be done.

The Appendix will contain a discussion about the build toolchain which was used.

Chapter 2

Theory and Background

This chapter will serve as a general introduction to the theory of Elliptic Curve Cryptography (ECC), and provide some background on processors which will be discussed in the following chapters, including the target processor, the ARM Cortex-M0+.

First, the introduction section discusses the relative key strength of ECC systems compared to other cryptographic systems, a brief motivation for the choice of ECC, followed by a discussion on elliptic curve parameters, including the three-level hierarchy of an ECC. Next, the Elliptic Curve Discrete Logarithm (ECDL) is introduced, followed by a description of two ECC protocols: (1) Elliptic Curve Diffie-Hellman (ECDH), and (2) Elliptic Curve Digital Signature Algorithm (ECDSA). Subsequent, Elliptic Curves are defined, Point Addition and the Point Doubling operations are discussed, followed by a brief introduction to a number of different point multiplication algorithms. Following, finite fields are introduced with a brief mathematical overview of group laws, coordinate systems, and field arithmetic including: field multiplications, additions, squaring, reduction and inversion over binary fields, as well as prime fields. Finally, we introduce several processors including the target platform, and processors for which implementations were found in literature and software libraries.

2.1 Introduction

This section will be used to discusses the relative key strength of ECC systems compared to other cryptographic systems, motivate the choice of ECC, as well as discuss elliptic curve parameters, including the three-level hierarchy of an ECC.

2.1.1 Key strength

The *security level* of an algorithm is directly related to the amount of operations required to perform an exhaustive search of all the keys. Here we will discuss the relative key strengths of different cryptosystems, including the amount of protection that they provide.

Table 2.1: Security levels reported in ECRYPT [2].

Symmetric key	Protection against agencies
80 bit	Very short term protection against agencies, long-term protection against small organizations.
96 bit	Legacy standard. Provides short term (± 10 years) protection
112 bit	Medium-term protection. Provides protection for ± 20 years
128 bit	Long-term protection. Provides protection for ± 30 years
256 bit	Foreseeable future. Good protection against quantum computers

ECRYPT [2] performed an analysis of the key lengths of various different cryptographic methods. The authors classified a number of different threats according to the threats' capabilities. A small organization is defined as having a $10k budget and an FPGA, a medium organisation has a $300k budget and FPGA/ASIC, a large organization has a $10M budget and FPGA/ASIC, and an intelligence agency has a $300M budget and ASIC. Table 2.1 shows the duration that a key strength will provide protection against different threats. We can see that a symmetric key size of 80 bits will only provide very short term protection against agencies. The minimum recommended symmetric key size that will provide medium term protection against agencies is 112 bits.

Table 2.2 shows a key strength comparison between symmetric cryptography, ECC, and RSA [3]. The results from [8, 9, 10, 2, 11] are used to make a comparison of the recommended key lengths for ECC, RSA and symmetric key cryptography. The reason why the number of bits for ECC and RSA are displayed as a range of values is due to the fact that different authors used different methods to determine the strength of each algorithm. An online tool to easily compare the recommendations for key lengths from the mentioned publications can be found at [12].

Table 2.2: The key strength of various key length of Elliptic Curves are compared to RSA and symmetric key cryptography.

Symmetric (bit)	Elliptic Curve (bit)	RSA (bit)
80	160	1024-1513
96	172-192	1776-2768
112	224	2048-4509
128	242-256	3072-6790
256	384-512	15360-49979

ECRYPT II [2] notes that the elliptic curve key length recommendations are only applicable for prime fields. Cryptosystems based on binary curves are required to have a field length of approximately 10 bits more than prime curves to have the same relative key strength.

From the above comparisons it is clear that the key size required by ECC is always significantly smaller than that of RSA and this makes ECC particularly

appealing for use in constrained devices. There are alternatives but that is beyond the focus of this paper.

2.1.2 The choice of ECC

Elliptic Curves allow for a much smaller key size than that of other public-key cryptography algorithms like e.g. RSA while still providing the same security. The size of the key has a large impact on constrained devices because a smaller key size leads to a reduced number of memory accesses, and memory accesses are usually a very expensive operation on constrained devices. It is for this reason that ECC usually outperforms RSA on constrained devices.

Another type cryptosystem based on a different type of algebraic curve, called Hyper-Elliptic Curve Cryptography (HECC), requires an even shorter key length than ECC to provide the same level of security. Therefore, HECC has the potential to lead to even faster timings, and subsequently an even lower energy consumption for the same security level as ECC. However, ECC was chosen because a much larger amount of literature and software libraries are available, which eases the implementation process greatly.

2.1.3 Elliptic Curve Parameters

The implementation of an ECC system requires the careful selection of many parameters which depend on the architecture, and the required application. The parameters of the system can be divided into a three-layer hierarchy (illustrated in Fig. 2.1), with the protocol at the highest layer, elliptic curve arithmetic in the middle, and field arithmetic at the foundation.

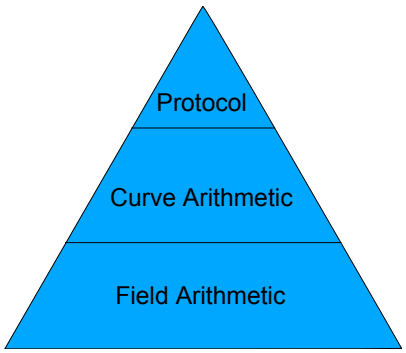


FIGURE 2.1: The three layers of an ECC system.

The protocol layer can be seen as a type of applications layer as it depends on the type of elliptic curve cryptography operation being performed. The parameters for this layer consists of the elliptic curve protocol. The curve arithmetic layer depends on this layer.

The elliptic curve arithmetic layer consists of all the operations which are performed on the curve. The parameters for this layer includes the choice of coordinate

system (which depends on the underlying field type), algorithm selection for the point addition, and point doubling operations (which depend on the projective coordinate system used), as well as the algorithm for point multiplication. The point multiplication algorithm typically depends on the protocol. e.g. when performing the Elliptic-Curve Diffie-Helman (ECDH) key exchange protocol, one random point multiplication and one fixed point multiplication is required. However, when performing the Elliptic-Curve Digital Signature Algorithm (ECDSA) protocol, the signature generation is done with a random point multiplication and the signature verification is typically done by performing two simultaneous point multiplications. By implementing the point multiplication according to the application (protocol) it is possible to get a more efficient implementation.

Field arithmetic forms the foundation of an ECC system as most of the execution time is spent in this layer. The parameters for this layer include the field type ($\mathbb{F}_p, \mathbb{F}_{2^m}$ or $\mathbb{F}_{p^m}$), the field order, and the field arithmetic algorithms for field addition, multiplication, reduction and inversion. Optimizations in the field arithmetic layer will lead to a speed increase of the whole cryptosystem. The field multiplication routine is typically where the most time is spent inside an elliptic curve cryptosystem, and therefore a fast implementation of this algorithm is crucial for an efficient implementation.

The following sections will provide some background related to the three layers in order to aid in the process of selecting the correct parameters for the target platform and application.

2.2 Elliptic Curve Protocols

Elliptic Curve Cryptosystems are based on the difficulty of solving the Elliptic Curve Discrete Logarithm (ECDL) problem. The problem statement is as follows: Consider a prime curve of order n. To generate a private key, an integer between the range $[1, n-1]$ is selected uniformly at random. A public key is generated with $Q = dP$, where P is a point on the curve. The problem of obtaining d from the known parameters: P, Q, p and n is considered hard and is known as the ECDL problem. ECDL is used as the basis of many cryptographic algorithms including Elliptic Curve Diffie Hellman (ECDH), Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve El Gamal. The methods shown in this section are provided for reference only.

2.2.1 Elliptic Curve Diffie-Hellman

The Elliptic curve Diffie-Hellman (ECDH) is a key agreement protocol that allows parties A and party B to establish a shared secret over an insecure channel. The curve parameters, and base point G need to be agreed upon before the exchange. Each party randomly generates a private: Alice generates d_A, and Bob generates d_B. Thereafter, they generate the corresponding public key pairs: $P_A = d_A \cdot G$, and $P_B = d_B \cdot G$. Next, the public keys are then exchanged over the insecure channel.

LOA 1: ECDSA signature generation.

Input: Curve parameters (a, b, P, m), private key d and message msg
Output: Signature r, s
1: Select $k \in [1, m-1]$
2: Compute $kP = (x_1, y_1)$ and convert x_1 to an integer $\bar{x}_1$.
3: Compute $r = \bar{x}_1 \bmod m$. If $r = 0$ then go to step 1
4: Compute e =Hash(msg)
5: Compute $s = k^{-1}(e + dr) \bmod m$. If $s = 0$ then go to step 1.
6: **return** (r, s)

LOA 2: ECDSA signature verification.

Input: Curve parameters (a, b, P, m), private key d, message msg and signature (r, s)
Output: Accept or Reject signature
1: Verify if $r, s \in [1, m-1]$
2: Compute e =Hash(msg)
3: Compute $w = s^{-1} \bmod m$
4: Compute $u_1 = ew \bmod m$ and $u_2 = rw \bmod m$
5: Compute $X = u_1 P + u_2 Q$
6: If $X = \infty$ then return("Reject signature")
7: Convert the x-coordinate x_1 of X to an integer $\bar{x}_1$.
8: Compute $v = \bar{x}_1 \bmod m$.
9: If $v = r$ then return("Accept signature") else return("Reject signature")

After receiving the public keys, both parties can calculate the shared secret by:

$$d_B \cdot P_A = d_A \cdot P_B = d_A \cdot d_B \cdot G.$$

An attacker cannot determine the shared secret from the shared curve parameters and the base point G due to the difficulty of solving the Elliptic Curve Discrete Logarithm problem.

2.2.2 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the Elliptic Curve analogue of the Digital Signature Algorithm (DSA). The signature generation is shown in Alg. 1 and the signature verification is shown in Alg. 2. The two point multiplications on line 4 in Alg. 2 are usually performed as a simultaneous point multiplication and is known as "Shamir's" trick.

As can be seen from both the ECDSA and ECDH protocols, point multiplication forms the core of the computations of this protocol. As such, this work will focus on creating an implementation that performs a random and fixed point multiplication, as these operations can be used to perform a number of different types of ECC protocols. This is also a common strategy which is followed in literature.

2.3 Elliptic Curve Arithmetic

An elliptic curve E over a field $\mathbb{F}$ is the collection of points which satisfies the equation:

$$Y^2Z + a_1XYZ + a_2YZ^2 = X^3 + a_3X^2Z + a_4XZ^2 + a_5Z^3, \tag{2.1}$$

where $a_1, a_2, a_3, a_4, a_5 \in \mathbb{F}$ and $\Delta \neq 0$, together with a point at infinity denoted by $\mathcal{O}$. The discriminant Δ is defined by:

$$\begin{aligned}
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_7 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2.
\end{aligned}$$

The equation can be rewritten in the affine Weierstrass form if $x = X/Z$ and $y = Y/Z$:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_5, \tag{2.2}$$

where $a_1, a_2, a_3, a_4, a_5 \in \mathbb{F}$. This equation can further be simplified considerably depending on the characteristic of the underlying field.

A valid point on the elliptic curve is any point which satisfies Eq. 2.1. The collection of all valid points including the point at infinity is denoted by $E(\mathbb{F}) \cup \mathcal{O}$ and this set forms an Abelian group together with the addition operation.

A curve with char($\mathbb{F}$)$\neq 2, 3$, and p a prime number, is called a prime curve, and is denoted with E($\mathbb{F}_p$). The Weierstrass equation can then be simplified to

$$y^2 = x^3 + ax + b, \tag{2.3}$$

where $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \not\equiv 0 \mod p$.

A curve with char($\mathbb{F}$) $= 2$, is called a binary curve, and is denoted with E($\mathbb{F}_{2^m}$). There are two cases to consider: supersingular and non-supersingular curves. If $a_1 \neq 0$ then the curve is said to be non-supersingular and the Weierstrass equation can be reduced to:

$$y^2 + xy = x^3 + ax^2 + b, \tag{2.4}$$

where $a, b \in \mathbb{F}_{2^m}$ and $\Delta = b$. A binary Koblitz curve is a non-supersingular curve with $a \in \{0, 1\}$ and $b = 1$.

If $a_1 = 0$ then the curve is said to be supersingular and the Weierstrass equation can be reduced to:

$$y^2 + cy = x^3 + ax + b, \tag{2.5}$$

where $a, b \in \mathbb{F}_{2^m}$ and $\Delta = c^4$.

2.3.1 Point Addition and Doubling

Adding two points on the curve can be done with a geometric method called the *tangent-and-chord* method. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ denote two points on the curve, and $R = (x_3, y_3)$ be the result. If $P \neq \pm Q$, then the following method can be used: draw a line is through point P and Q and extend it in both directions. The line will intersect the elliptic curve at a third point R'. The result R is found by reflecting R' around the x-axis.

A similar method exists for performing point addition when $P = Q$. Find the tangent of the the curve at point Q. Draw a line from the point using the tangent and extend it in both directions towards infinity. The line will intersect the curve at point R'. The result R is found by reflecting R' around the x-axis. Fig. 2.2 shows a graphical representation of this operation.

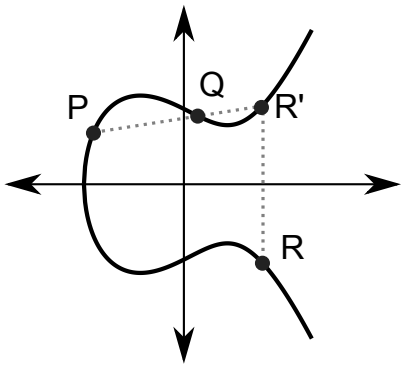


FIGURE 2.2: A point addition is performed on an elliptic curve using the tangent-and-chord method.

2.3.2 Point Multiplication

The following text will discuss a number of different point multiplication methods. These methods are all applicable to both binary curves and prime curves, with the exception of the τ-adic Non-Adjacent Form method which is only applicable to binary Koblitz curves.

Point multiplication is the operation of multiplying a scalar k with a point P on an elliptic curve, and it is responsible for the majority of the execution time of an ECC system. It is defined by the repeated addition of the point P with itself, $k - 1$ times:

$$kP = \underbrace{P + P + \cdots + P}_{k-1 \text{ additions}}.$$

Double-and-add method for point multiplication in $\mathbb{F}_q$

The most basic random point multiplication is the double-and-add algorithm which is listed as Alg. 3. This algorithm scans through all the values of the scalar k from

LOA 3: Double-and-add method for point multiplication in $\mathbb{F}_q$.

Input: $k = (k_{m-1}, \cdots, k_1, k_0)_2, P \in \mathrm{E}(\mathbb{F}_q)$
Output: kP
1: $Q \leftarrow \infty$
2: **for** i from 0 to $m-1$ **do**
3: if $k_i = 1$ then $Q \leftarrow Q + P$
4: $P \leftarrow 2P$
5: **end for**
6: Return(Q)

left-to-right. For each bit inside k, a point addition is performed when k_i is equal to 1, followed by a point doubling operation which is always performed. This is the slowest point multiplication algorithm and it has an expected running time of $m/2A + mD$ since a random k will have an average of $m/2$ bits set equal to one. (This is a reasonable assumption because each of the bits inside the scalar can be seen as a random variable, each with an expected value of 0.5). Therefore, on average half the bits are set to one and the other half of the bits are set to zero.

Non Adjacent Form (NAF)

The Non Adjacent Form (NAF) uses a unique signed-digit transformation of the scalar k where no two consecutive digits are non-zero. The NAF can be computed with Alg. 4, and Alg. 5 can be used to perform the point multiplication. NAF(k) will on average lead to $m/3$ terms that are non-zero; therefore, the expected running time is $m/3A + mD$. The NAF point multiplication can be accelerated considerably by using a window method that processes w digits of the scalar k at a time. The standard NAF(k) operation is equivalent to obtaining the NAF representation with $w = 2$ (i.e. $\mathrm{NAF}_2(k)$). The $\mathrm{NAF}_w(k)$ operation is performed with Alg. 4, and replacing line 4 with $k_i \leftarrow 2 - (k \bmod 2^w)$, $k \leftarrow k - k_i$. Alg. 6 shows the wNAF point multiplication method.

This method is used for simulation in section 3.1.1.

Endomorphisms on elliptic curves

An endomorphism ϕ on an elliptic curve E over $\mathbb{F}$ is a map: $\phi : E \rightarrow E$ such that $\phi(\mathcal{O}) \rightarrow \mathcal{O}$, and $\phi(P) = (g(P), h(P))$ for all $P \in E$, where g and h are rational functions whose coefficients lie in $\mathbb{F}$.

The q^{th} power map $\phi : E \rightarrow E$ is defined by $(x, y) \rightarrow (x^q, y^q), \phi : (O) \rightarrow (O)$ is an endomorphism of E defined over $\mathbb{F}$, called the Frobenius endomorphism.

Point multiplication ($kP, k \in [0, m-1]$) can be accelerated by first writing:

$$k = k_1 + k_2 \lambda \mathrm{mod} n$$

where k_1 and k_2 are approximately half the bit-length of k, and is called a balanced

LOA 4: Non Adjacent Form (NAF) of a positive integer.

Input: A positive integer k
Output: NAF(k)
1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: **if** k is odd **then**
4: $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$
5: **else**
6: $k_i \leftarrow 0$
7: **end if**
8: $k \leftarrow k/2, i \leftarrow i + 1$
9: **end while**
10: Return $(k_{i-1}, k_{i-2}, \cdots, k_1, k_0)$.

LOA 5: Binary NAF method for point multiplication.

Input: $k = (k_{n-1}, \cdots, k_1, k_0)_2, P \in \mathrm{E}(\mathbb{F}_q)$
Output: kP
1: Compute NAF(k) $= \sum_{i=0}^{l-1} k_i 2^i$
{Using algorithm 4}
2: $Q \leftarrow \infty$
3: **for** $i = l - 1$ down to 0 **do**
4: $Q \leftarrow 2Q$
5: if $k_i = 1$ then $Q \leftarrow Q + P$
6: if $k_i = -1$ then $Q \leftarrow Q - P$
7: **end for**
8: Return (Q).

length-two expression of k. Since

$$kP = k_1 P + k_2 \lambda P = k_1 P + k_2 \phi(P). \tag{2.6}$$

kP is obtained by first computing $\phi(P)$, and then using a simultaneous multiplication method, or an interleaved multiplication method. The bit-length of k_1 and k_2 are half the bit-length of k, and therefore the amount of point doubling operations are also halved. This strategy works if the decomposition in Alg. 2.6 and $\phi(P)$ can be computed efficiently.

This method is described in more detail in [13].

τ-adic non-adjacent form (TNAF)

If a binary Koblitz curve is used, this special form of the curve can exploited with Solinas's TNAF algorithm [14], which accelerates the computation of the random point multiplication kP by swapping point doublings for squarings by means of the efficiently computable endomorphism based on the Frobenius map $\tau(x, y) = (x^2, y^2)$.

LOA 6: Window NAF method for point multiplication.

Input: Window of w bits, positive integer k, point $P \in \mathrm{E}(\mathbb{F}_q)$
Output: kP
1: Compute $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$
{Using algorithm 4}
2: Compute $P_i = iP$ for $i = 1, 3, 5, \cdots, 2^{w-1} - 1$
3: $Q \leftarrow \infty$
4: **for** $i = l - 1$ down to 0 **do**
5: $Q \leftarrow 2Q$
6: **if** $k_i \neq 0$ **then**
7: **if** $k_i > 0$ **then**
8: $Q \leftarrow Q + P_{k_i}$
9: **else**
10: $Q \leftarrow Q - P_{-k_i}$
11: **end if**
12: **end if**
13: **end for**
14: Return (Q).

The TNAF method can be seen as the τ-adic analogue of the NAF method. When the TNAF method is used in a binary point multiplication scheme, then the running time is approximately $m/3A$, because the number of non-zero elements are on average $m/3$. A squaring operation can be computed very quickly on binary curves. Just like the NAF method, this method can be accelerated considerably by a employing windowing method. This method is used for simulation in section 3.1.2, as well as in an implementation in section 3.4.2.

2.4 Field Arithmetic

This section will introduce finite fields, provide a brief mathematical overview of group laws, coordinate systems, and field arithmetic including: field multiplications, additions, squaring, reduction and inversion over binary fields, as well as prime fields. A number of different algorithms will be provided to perform field arithmetic. However, the set of algorithms which were implemented is discussed in section 3.2. The complete definition of algorithms is listed here for the sake of completeness.

A *field* $\mathbb{F}$ is an abstraction of a number system. It consists of a set of numbers, together with two mathematical operations: addition and multiplication. If the set of numbers are finite, then the field is also finite. Subtraction is done by making use of addition: $a - b = a + (-b)$ where $a, b \in \mathbb{F}$ and b is the element which satisfies $b + (-b) = 0$.

Let $\mathbb{F}_{q^m}$ define the field, then the characteristic of the field is defined by q. If $q > 2, m = 1$ and q a prime number then it's referred to as a prime field which is sometimes denoted with $\mathrm{char}(\mathbb{F}) \neq 2, 3$. If $q = 2$ then $\mathbb{F}_q$ is a binary field which

is denoted with $\text{char}(\mathbb{F}) = 2$. Some cryptographic methods do exist for fields of characteristic three, but these will not be discussed in this text.

2.4.1 Elliptic Curve Arithmetic over Binary Fields

This section will provide a mathematical introduction to binary curves. The non-supersingular form of this curve (Eq. 2.4) is the only form of the binary curve which is considered in this text.

First we discuss the polynomial representation for a binary field, followed by a discussion on the different coordinate systems, subsequent we present the group laws, and finally we discuss a number of different field arithmetic operations.

Polynomial representation

A useful way to represent a binary field is with the polynomial basis representation:

$$\mathbb{F}_{2^m} = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2z^2 + a_1z + a_0, a_i \in 0, 1$$

Here we represent the elements of $\mathbb{F}_{2^m}$ inside a polynomial, with each element having a value of either 0 or 1. The degree of the polynomial is at most $m - 1$. An irreducible polynomial $f(z)$ of degree m is used for reduction after a field multiplication, or squaring operation.

Coordinate systems

If the standard affine coordinate system is used, then both the point addition, and point doubling operations require multiple field multiplications, as well as a field inversion. It is common to find that the inversion operation is significantly more expensive than the multiply operation in an actual implementation. If this is the case then it is often more efficient to represent the point in another coordinate system, because in a non-affine system no inversion is required. However, inversion is required to change the system, but only once.

Affine coordinates The affine coordinate (x, y) represent a point on the curve and must satisfy Eq. 2.3.

Standard projective coordinates The standard projective point (X, Y, Z) , $Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z)$. The point $\mathcal{O}$ corresponds to the coordinate $(0, 1, 0)$, and the negative of (X, Y, Z) is $(X, X + Y, Z)$. This leads to the equation:

$$Y^2Z + XYZ = X^3 + aXZ^2 + bZ^3$$

Jacobian projective coordinates The Jacobian projected point (X, Y, Z) , $Z \neq 0$ corresponds to the affine point $(X/Z^2, Y/Z^3)$. The point $\mathcal{O}$ corresponds to the coordinate $(1, 1, 0)$, and the negative of (X, Y, Z) is $(X, X + Y, Z)$. This leads to the equation:

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$$

López-Dahab coordinates The López-Dahab(LD) point (X, Y, Z), $Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z^2)$. The point $\mathcal{O}$ corresponds to the coordinate $(1, 0, 0)$, and the negative of (X, Y, Z) is $(X, X + Y, Z)$. This leads to the equation:

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$$

Both the point doubling operation, and the point addition operation can be performed in any one of the above mentioned coordinate systems, which will eliminate the need for a field inversion. Furthermore, a mixed-coordinate point addition is also possible, where the two input points use different coordinate systems. By doing so the amount of multiplication operations can be reduced to an even lower number than would be possible if input both points use the same coordinate system.

Table 2.3: The number of operations required to perform a Point Addition and a Point Doubling, for the Affine (A), Projective (P), Jacobian (J) and López-Dahab (LD) coordinate systems on prime fields. The field multiplication operation is indicated with M, and a field inversion is indicated with I.

Coordinate system over binary fields	**Doubling**	**General Addition**	**Mixed Addition**
Affine	1I + 1M	1I + 1M	-
Projective	7M	13M	12M (P + A)
Jacobian	5M	14M	10M (J + A)
López-Dahab	4M	14M	8M (LD + A)

Table 2.3 shows the amount of field operations which are required for a point doubling, general addition and mixed-coordinate addition. The results in this table only show the multiply and inversion operations because it was assumed that field additions and squarings are cheap in comparison with multiplications and inversions. The operation count for the different coordinate systems: Affine (A), Projective (P), Jacobian (J) and López-Dahab (LD) are written in terms of the number of field operations (multiplication (M) and inversion (I)) which are required to perform the point operation. It can be seen that the mixed-coordinate addition of a López-Dahab, and an Affine point provides the fastest point addition with 8 field multiplication operations. The fastest point doubling operation is for a point on the López-Dahab coordinate system with 4 field multiplications.

For a mixed Point addition in $\mathbb{F}_{2^m}$, with $P(X_1, Y_1, Z_1)$ in López-Dahab coordinates, and $Q(X_2, Y_2)$ in affine coordinates, the following operations need to be performed:

$$A \leftarrow Y_2 \cdot Z_1^2,\ B \leftarrow X_2 \cdot Z_1 + X_1,\ C \leftarrow Z1 \cdot B,\ D \leftarrow B^2 \cdot (C + aZ_1^2),\ Z_3 \leftarrow C^2,$$
$$E \leftarrow A{\cdot}C,\ X_3 \leftarrow A^2{+}D{+}E,\ F \leftarrow X_3{+}X_2{\cdot}Z_3, G \leftarrow (X_2{+}Y_2){\cdot}Z_3^2,\ Y_3 \leftarrow (E{+}Z_3){\cdot}F{+}G.$$

For a point doubling with $P(X_1, Y_1, Z_1)$ in López-Dahab coordinates, the following operations need to be performed:

$$Z_3 \leftarrow X_1^2 \cdot Z_1^2, X_3 \leftarrow X_1^4 + b \cdot Z_1^4, Y_3 \leftarrow bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4).$$

For an algorithmic listing of these methods please see [15] page 94.

Group Laws

The group laws for the non-supersingular form are as follows:

Identity $P + \mathcal{O} = P$ for all $P \in E(\mathbb{F}_{2^m})$

Negation If $P = (x, y) \in E(\mathbb{F}_{2^m})$, then $(x, y) + (x, y + y) = \mathcal{O}$. The point $(x, y + y)$ is denoted with $-P$. This implies that $P + (-P) = \mathcal{O}$.

Point addition Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m}), Q = (x_2, y_2) \in E(\mathbb{F}_{2^m})$, and $P \neq \pm Q$ then:

$$P + Q = (x_3, y_3) = (\lambda^2 + \lambda + x_1 + x_2 + a \ , \ \lambda(x_1 + x_3) + x_3 + y_1). \tag{2.7}$$

$$\lambda = (y_1 + y_2)/(x_1 + x_2)$$

where $P = (x_1, y_1), Q = (x_2, y_2), P \neq Q$

Point doubling Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$ where $P \neq -P$ then:

$$P + P = 2P = (x_3, y_3) = (x_1^2 + b/x_1^2 \ , \ x_1^2 + \lambda x_3 + x_3) \tag{2.8}$$

$$\lambda = x_1 + y_1/x_1.$$

In the following text we discuss a number of different basic field arithmetic algorithms.

Right-to-left shift-and-add polynomial multiplication

The right-to-left shift-and-add polynomial multiplication method (Alg. 7) is a very basic field multiplication algorithm. During each iteration of the main loop the polynomial y is shifted left one bit. Next, the parameter x is scanned from right to left, where each bit is tested to see if it's equal to 1. If $x_i = 1$ then the shifted y is added to the partial product c. This method is not suitable for a software implementation because of the large number of multi-precision shift operations. In total $(m - 1)$ multi-precision shift operations are required. Recall that m is the degree of the field $(\mathbb{F}_{2^m})$. Shifting a polynomial which is stored in multiple words requires many accesses to memory and is a slow process. This method is provided for reference only.

LOA 7: Right-to-left shift-and-add polynomial multiplication.

Input: Binary polynomials $x(z)$ and $y(z)$ of degree at most $m-1$
Output: $c(z) = c[0 \cdots n-1] = x(z)y(z) \bmod f(z)$
1: **if** $x_0 = 1$ **then**
2: $c(z) \leftarrow y(z)$
3: **else**
4: $c(z) \leftarrow 0$
5: **end if**
6: **for** $k \leftarrow 1$ to m **do**
7: $y(z) \leftarrow y(z) \cdot z \bmod f(z)$
8: **if** x_i=1 **then**
9: $c(z) \leftarrow c(z) + y(z)$
10: **end if**
11: **end for**
12: **return** $c(z)$

Left-to-right comb method for polynomial multiplication

The left-to-right comb method (Alg. 8) is an improvement to the right-to-left shift-and-add multiplication algorithm, when implemented in software. It reduces the number of shift operations to $W-1$, by observing that when $x(z) \cdot z^k$ has been computed, $x(z) \cdot z^{jW+k}$ can easily be obtained by just appending jW (W is the word size of the processor) zeros to the least significant part of the polynomial. This is essentially the same as filling the j words of the least significant part of the vector representation of the polynomial with zeros. Instead of testing all the bits of x sequentially, it tests the bits that are W bits away from each other. The first tests are done on bit $W-1$ of each of the words in the polynomial (i.e. $x_0, x_W, x_{2W}, \cdots, x_{nW}$) then it tests bit $W-2$ of each of the words in the polynomial (i.e. $x_1, x_{1+W}, x_{1+2W}, \cdots, x_{1+nW}$) etc.

This method is used for simulation in section 3.1.2. More advanced methods for field multiplication are discussed in Chapter 3.

Field addition

The addition of two polynomials x and y on a binary field are done by performing the bit-wise exclusive or (XOR) operation on x and y. This method is used extensively in many of the algorithms listed in this document.

Square

The squaring operation is obtained by inserting a 0 between each of the bits of the input polynomial $c(z)$. If $c(z) = c_{m-1}z^{m-1} + \cdots + c_2z^2 + c_1z^1 + c_0$ then $c(z)^2 = c_{m-1}z^{2m-2} + \cdots + c_2z^4 + c_1z^2 + c_0$.

This process can be accelerated considerately by means of a lookup table. This is discussed in section 3.2.3.

LOA 8: Left-to-right comb method for polynomial multiplication.

Input: Binary polynomials $x(z)$ and $y(z)$ of degree at most $m-1$, Processor word size W.
Output: $c(z) = c[0 \cdots 2n-1] = x(z)y(z)$

```
1: c[0 ··· 2n − 1] ← 0
2: for k ← W − 1 downto 0 do
3:    for j ← 0 to n − 1 do
4:       if bit k of x[j] = 1 then
5:          for l ← 0 to n do
6:             c[l + j] ← c[l + j] ⊕ y[l]
7:          end for
8:       end if
9:    end for
10:   if (k ≠ 0) then
11:      c(z) = c(z) · z^w
12:   end if
13: end for
14: return c
```

LOA 9: Basic modular reduction.

Input: Binary polynomials $c(z)$ of degree at most $2m-2$, reduction polynomial $f(z)$
Output: $c(z) = c(z) \bmod f(z)$

```
1: Compute u_k(z) = f(z) · z^k, for 0 ≤ k ≤ w − 1
2: for i ← 2m − 2 downto m do
3:    if c_i = 1 then
4:       j = ⌊(i − m)/W⌋
5:       k = (i − m) − Wj
6:       c(z) ← c(z) + u_k(z) · z^{jw}
7:    end if
8: end for
9: return (c[n − 1], ··· , c[1], c[0])
```

Reduction

Reduction is used to reduce a polynomial of degree $2m-2$ to $m-1$. This operation is required after each squaring and multiply operation. The most basic method to perform modular reduction algorithm is by means of polynomial long division with the reduction polynomial $f(z)$. The degree of $c(z)$ is lowered by reducing $c(z)$ one bit at a time, starting with the most significant bit. Algorithm 9 shows the algorithm for the basic modular polynomial reduction algorithm. This method is provided for reference only.

LOA 10: Inversion in $\mathbb{F}_{2^m}$.

Input: A non-zero polynomial $a(z)$ of degree at most $m-1$, reduction polynomial $f(z)$
Output: $c(z) = a(z)^{-1} \bmod f(z)$
1: $u(z) \leftarrow a(z)$, $v(z) \leftarrow f(z)$
2: $g_1 \leftarrow 1$, $g_2 \leftarrow 0$
3: **while** $u \neq 1$ **do**
4: $j \leftarrow \deg(u)$ - $\deg(v)$
5: **if** $j < 0$ **then**
6: swap(u,v), swap(g_1, g_2), $j \leftarrow -j$
7: **end if**
8: $u \leftarrow u + z^j v$
9: $g_1 \leftarrow g_1 + z^j g_2$
10: **end while**
11: **return** c

Inversion

The Extended Euclidean Algorithm for polynomials is used to perform inversion and is shown in algorithm 10. This method is used in an implementation in section 3.4.2.

2.4.2 Elliptic Curve Arithmetic over Prime Fields

This section will provide a mathematical introduction to prime curves. First we discuss the different coordinate systems, followed by a discussion on the group laws, and finally we present a number of different field arithmetic operations.

Coordinate systems

If the standard affine coordinate system is used, then both the point addition, and point doubling operations require multiple field multiplications, as well as a field inversion. Recall that the inversion operation is significantly more expensive than the multiply operation. Hence it is often more efficient to represent the point in another coordinate system:

Affine coordinates The affine coordinate (x, y) represent a point on the curve and must satisfy Eq. 2.4.

Standard projective coordinates The standard projective point (X, Y, Z) , $Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z)$. The point $\mathcal{O}$ corresponds to the coordinate $(0, 1, 0)$, and the negative of (X, Y, Z) is $(X, -Y, Z)$. This leads to the equation:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

Jacobian projective coordinates The Jacobian projected point (X, Y, Z) , $Z \neq 0$ corresponds to the affine point $(X/Z^2, Y/Z^3)$. The point $\mathcal{O}$ corresponds to the coordinate $(1, 1, 0)$, and the negative of (X, Y, Z) is $(X, -Y, Z)$. This leads to the equation:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

Chudnovsky coordinates The Jacobian project point (X, Y, Z) is represented as (X, Y, Z, Z^2, Z^3). Here the Z-coordinate is represented multiple times. This can lead to fewer addition operations when point multiplications are performed in projective coordinates. This gain is offset by the increase in the amount of memory required to store each point.

Both the point doubling operation, and the pont addition operation can be performed in any one of the above mentioned coordinate systems, which will eliminate the need for a field inversion. Furthermore, a mixed-coordinate point addition is also possible, where the two input points use different coordinate systems. By doing so the amount of multiplication operations can be reduced to an even lower number than would be possible if input both points use the same coordinate system.

TABLE 2.4: The number of operations required to perform a Point Addition and a Point Doubling, for the Affine (A), Projective (P), Jacobian (J) and Chudnovsky (C) coordinate systems on prime fields. The field multiplication operation is indicated with M, the polynomial square operation is indicated with S and a field inversion is indicated with I.

Coordinate system	**Doubling**	**General Addition**	**Mixed Addition**
Affine	1I, 2M, 2S	1I, 2M, 1S	(J + A) 8M, 3S
Projective	7M, 3S	12M, 2S	-
Jacobian	4M, 4S	12M, 4S	(J + C) 11M, 3S
Chudnovsky	5M, 4S	11M, 3S	(C + A) 8M, 3S

Table 2.4 shows the amount of field operations which are required for a point doubling, general addition, and mixed-coordinate addition. The results in this table only show the multiply, square and inversion operations because it was assumed that field additions are cheap in comparison with multiplications, squarings and inversions. The operation count for the different coordinate systems: Affine (A), Projective (P), Jacobian (J) and Chudnovsky (C) are written in terms of the number of field operations (multiplication (M), squaring (S) and inversion (I)) which are required to perform the point operation. It can be seen that the mixed-coordinate addition of a Jacobian, and an Affine point provides the fastest point addition with 8 field multiplications and 3 polynomial square operations. The fastest point doubling operation is for a point on a Jacobian coordinate system with 4 field multiplications and 4 polynomial square operations.

The fastest coordinate system to perform a point addition for $\mathbb{F}_p$ is a mixed-coordinate Jacobian-affine addition and can be realized with:

$$A \leftarrow Z_1^2,\ B \leftarrow Z_1 \cdot A,\ C \leftarrow X_2 \cdot A,\ D \leftarrow Y_2 \cdot B,$$

$$E \leftarrow C - X_1,\ F \leftarrow D - Y_1,\ G \leftarrow E^2,\ H \leftarrow G \cdot E, I \leftarrow X_1 \cdot G,$$

$$X_3 \leftarrow F^2 - (H + 2I),\ Y_3 \leftarrow F \cdot (I - X_3) - Y_1 \cdot H,\ Z_3 \leftarrow Z_1 \cdot E$$

For the complete algorithmic listing please see [15, p. 89].

Group Laws

The group laws for the prime form are as follows:

Identity $P + \mathcal{O} = P$ for all $P \in E(\mathbb{F}_p)$.

Negation If $P = (x, y) \in E(\mathbb{F}_p)$, then $(x, y) + (x, -y) = \mathcal{O}$. The point $(x, -y)$ is denoted with $-P$. Thus, $P + (-P) = \mathcal{O}$.

Point addition Let $P = (x_1, y_1) \in E(\mathbb{F}_p), Q = (x_2, y_2) \in E(\mathbb{F}_p)$, and $P \neq \pm Q$ then:

$$P + Q = (x_3, y_3) = (\lambda^2 - x_1 - x_2\ ,\ \lambda(x_1 - x_3) - y_1), \tag{2.9}$$
$$\lambda = (y_2 - y_1)/x_2 - x_1)$$

.

Point doubling Let $P = (x_1, y_1) \in E(\mathbb{F}_p)$ where $P \neq -P$ then:

$$P + P = 2P = (x_3, y_3) = (\lambda^2 - 2x_1\ ,\ \lambda(x_1 - x_3) - y_1), \tag{2.10}$$
$$\lambda = (3x_1^2 + a)/(2y_1)$$

.

The fastest coordinate system to perform a point doubling for $\mathbb{F}_p$ is Jacobian coordinates and it can be realized with:

$$A \leftarrow Y_1^2,\ B \leftarrow 4X_1 \cdot A,\ C \leftarrow 8A^2,\ D \leftarrow 3X_1^2 + a \cdot Z_1^4,$$

$$X_3 \leftarrow D^2 - 2B,\ Y_3 \leftarrow D \cdot (B - X_3) - C,\ Z_3 \leftarrow 2Y_1 \cdot Z_1$$

For the complete algorithmic listing please see [15, p. 91].

In the following text we discuss a number of different basic algorithms to perform prime field arithmetic.

Field addition

Addition in $\mathbb{F}_p$ is done by first performing multi-precision integer addition on the two input parameters x and y followed by a reduction modulo p. Algorithm 11 shows the multiprecision integer addition. This method is provided for reference only.

LOA 11: Addition in $\mathbb{F}_p$.

Input: Integers $x, y \in \mathbb{F}_p$)
Output: $c = x + y \bmod p$
{Multi-precision integer addition}
1: $c \leftarrow x + y$.
2: **if** $c \geq p$ **then**
3: $c \leftarrow c - p$.
4: **end if**
5: **return** c

LOA 12: Subtraction in $\mathbb{F}_p$.

Input: Integers $x, y \in \mathbb{F}_p$)
Output: $c = x - y \bmod p$
{Multi-precision integer subtraction}
1: $c \leftarrow x - y$.
2: **if** $c < 0$ **then**
3: $c \leftarrow c + p$.
4: **end if**
5: **return** c

Field subtraction

Addition in $\mathbb{F}_p$ is done by first performing multi-precision integer subtraction on the two input parameters x and y followed by a reduction modulo p. Algorithm 12 shows the multiprecision integer subtraction. This method is provided for reference only.

Field multiplication

Field multiplication is accomplished by first performing a multiplication with the two operands x and y followed by a reduction modulo p. A number of different methods for performing the multiplication exists including the operand-scanning form, product-scanning or Comba [16], the Hybrid method [17], operand caching [18] and the Karatsuba-Ofman [19] method. Each one of these methods has its advantages and disadvantages and the appropriate method needs to be selected based on the underlying architecture.

The operand-scanning form is the most basic way to perform multiplication and requires n^2 multiplications. Both the product scanning form and the Hybrid method reduces the number of memory accesses when compared to the operand-scanning form. The product-caching method reduces the number of memory load instructions by performing a novel way of caching the operands. The Karatsuba-Ofman method reduces the number of multiplications at the cost of a more complex way of adding the multiplication results.

Algorithm 13 shows the operand-scanning form of a field multiplication in $\mathbb{F}_p$. (UV) denotes a vector of $2W$ words. This is a way to carry it out without handling

LOA 13: Integer multiplication (operand-scanning form).

Input: Integers $x, y \in [0, p-1]$
Output: $c = x \cdot y$
1: $c \leftarrow 0$
2: **for** i from 0 to $n-1$ **do**
3: $U \leftarrow 0.$
4: **for** j from 0 to $n-1$ **do**
5: $(U, V) \leftarrow C[i+j] + X[i] \cdot Y[j] + U$
6: $C[i+j] \leftarrow V$
7: **end for**
8: $C[i+t] \leftarrow U$
9: **end for**
10: **return** c

carry bits, but the underlying platform needs to provide a 2-word register to make efficient use of it.

All the methods listed here, except for the Hybrid method and the Operand-scanning form are provided for reference purposes. The Hybrid method and operand-scanning forms are used for simulation in section 3.1.1.

Square

Squaring in $\mathbb{F}_p$ can be accomplished by first performing the square operation, followed by a reduction modulo p. This operation is essentially a field multiplication with $x = y$. Because the two operands are equal to each other, the number of multiplications are reduced to $m \cdot (m+1)/2$. This method is provided for reference only.

Reduction

If the curve is not of a special form then `Barrett` reduction [20] and `Montgomery` reduction [21] can be employed to perform the reduction operation. Barret reduction works by using an estimation technique to make an estimate of the quotient $\lfloor z/p \rfloor$, followed by a subtraction of the quotient multiplied with p. Thereafter p is subtracted from the remainder until the remainder is smaller than p.

`Pseudo-Mersenne` primes have a special form which make them very suitable for fast modular reduction techniques which are not available to general primes. The set of NIST prime curves all have this property so that they allow for a very fast reduction operation. For a thorough description of this method please see [15]. This method is provided for reference only.

Inversion

The extended Euclidean method can be used to perform inversion but the drawback with it is that it contains a large number of expensive divisions. The extended binary Euclidean (Algorithm 14) method can be used to replace the expensive division

LOA 14: Binary algorithm for inversion in $\mathbb{F}_p$.

Input: Integer $x \in [0, p-1]$ and prime p
Output: $c = x^{-1} \bmod p$

$u \leftarrow x, v \leftarrow p$
$x_1 \leftarrow 1, x_2 \leftarrow 0$
while $u \neq 1$ and $v \neq 1$ **do**
  while u is even **do**
    $u \leftarrow u/2.$
    if x_1 is even then $x_1 \leftarrow x_1/2$ else $x_1 \leftarrow (x_1 + p)/2$
  end while
  while v is even **do**
    $v \leftarrow v/2.$
    if x_2 is even then $x_2 \leftarrow x_2/2$ else $x_2 \leftarrow (x_2 + p)/2$
  end while
  if $u \geq v$ **then**
    $u \leftarrow u - v, x_1 \leftarrow x_1 - x_2$
  else
    $v \leftarrow v - u, x_2 \leftarrow x_2 - x_1$
  end if
end while
if $u = 1$ **then**
  return $x_1 \bmod p$
else
  return $x_2 \bmod p$
end if

operations with cheaper shift and subtraction operations. This method is provided for reference only.

2.5 Low-Power Processors and Microcontrollers

In this section a number of different processors and microcontrollers which are found in the ultra low-power domain are discussed and compared. The majority of these processors are currently found on Wireless Sensor Nodes, with the exception of the ARM7TDMI and the ARM Cortex-M3, which to the best of our knowledge are not applied WSNs.

As one of the goals of the thesis is to compare the implementation of this thesis work with different software implementations on different architectures, this section aims to serve as an introduction to the architectures in literature, as well as in software libraries. The energy consumption estimates for each processor are also provided, and at the end of the section the energy efficiency of the processors are compared.

The power consumption of a microcontroller is given by:

$$P = \alpha C V^2 f + V I_{leak} = P_{dynamic} + P_{static}$$

where α is the fraction of gates actively switching, and C is the total capacitive load of all gates. The first term is the dynamic power lost from charging and discharging the processor's capacitive loads. The second term is the static power lost due to leakage current. A term which is not included in this equation is the relatively small amount of leakage due to a short-circuit which momentarily occurs every time a gate switches. Many of the modern microcontroller's data sheets give the static power as a constant value, and the dynamic power consumption in the unit W/MHz. Since there is sometimes a difference between the manufacturer's stated power consumption and the actual power consumption, a literature study was made to find actual energy consumption measurements of the different microcontrollers. The dynamic power increases linearly with frequency, whereas the static power consumption remains constant regardless of the frequency. Therefore, a processor runs more efficiently at a higher frequency because the static power consumption contributes a smaller percentage of the overall power consumption.

ARM Microcontrollers

There are generally two instruction sets that ARM microcontrollers can use: (1) ARM code with 32-bit instructions and (2) Thumb code with 16-bit instructions.

In ARM mode all instruction are 32-bits long, and all 16 registers are addressable from the instruction set. Many instructions support conditional execution. Many of the data-processing instructions can apply a shift on one of its parameters at no extra cost in cycles. e.g. The instruction `add r0, r1, r1, lsl #3` is equivalent to `r0 = r1 + (r1 << 3)`. Most data processing instructions use one destination register and two operands.

In Thumb mode all instructions are 16 bits long, and most of the instructions can only address the 8 low registers (r0 to r7). In this mode the only conditional instruction is the branch instruction, no shift operation can be embedded with a data-processing instruction, and most data instructions take only two operands, where the destination register is also one of the two sources.

The benefits of working in Thumb mode are that the code size is much smaller, and it costs less energy to fetch an instruction because moving data from memory to the processor costs energy; however, the drawbacks are that the Thumb code executes slower than the ARM code because Thumb has a limited instruction set, with access to less of the registers, and no embedding of shifts into the data-processing instructions.

ATMega128L

The Atmel AVR ATMega128L [22] is found on a number of different Wireless Sensor Nodes like the MICA2, MICAz and MICA2DOT. It has a RISC processor with a modified Harvard architecture, a maximum clock speed of 8MHz, a word size of 8

bits, 4 kB SRAM, 128 kB Flash, and 32 general purpose registers. It can perform 1 MIPS per MHz.

A power analysis done by [23] showed that this microcontroller consumes 30mW when the processor is clocked at 7.37 MHz, and 12mW when the processor is clocked at 4MHz. The energy per cycle when the processor is clocked at 4 MHz is therefore 12 mW/4 MHz = 4.125 nWs. The energy per cycle when the processor is clocked at 7.37 MHz is therefore 30 mW/7.37 MHz = 4.07 nWs. Note that the energy consumption per cycle is lower when the processor is clocked at a higher frequency, as expected.

MSP430F1611

The Texas Instruments MSP430F1611 [24] is found in a number of Wireless Sensor Nodes like the TelosB[25], and the Tmote Sky[26]. It has a RISC processor that uses a Von Neumann architecture with a maximum clock speed of 25 MHz, a word size of 16 bits, 10 kB SRAM, 48 kB Flash, and 12 general purpose registers. It can perform 1 MIPS per MHz.

A power analysis done by [23] showed that this microcontroller has an average power consumption of 12 mW when clocked at 8 MHz. The energy per cycle when the processor is clocked at 8 MHz is therefore 12 mW/8 MHz = 1.5 nWs.

ARM7TDMI

The ARM7TDMI [27] is not found on any WSNs that we are aware of. It has a RISC processor with a Von Neumann architecture, a three-stage pipeline, a maximum clock speed of 133 MHz, a word size of 32 bits, 32 kB SRAM, 512 kB Flash, supports ARM and Thumb instructions, and has support for a long multiply instruction which produces a result of 64-bits. It has 31 general-purpose 32-bit registers, and 6 status registers. It can perform 60 MIPS at 59.8 MHz.

ARM claims that the typical power consumption in in 0.18 µm technology is 0.25 mW/MHz at 1.8 V, and in 0.13 µm technology is 60 µW/MHz at 1.2 V [28]. The energy per cycle in 0.13 µm is therefore 60 µW/1 MHz = 0.06 nWs.

To make a fair energy consumption comparison with the two other ARM chips, we scale the energy values so that they are representative for a technology size of 90 nm. Let's assume that the ARM microcontroller runs at 1 V on 90 nm technology. We use Eq. 2.11 to scale the power consumption of the 0.13 µm technology to 90 nm. This leads to a power consumption of 0.029mW/MHz, and an energy per cycle of 0.029 nWs. This estimated value is just given for reference and is not used in the actual energy estimations in Chapter 4.

$$scale_{atob} = (size_b/size_a) \cdot (V_b/V_a)^2 \tag{2.11}$$

PXA271

The PXA271 microcontroller uses the ARMv5TE [29] architecture, and is found on the Imote2 [30] WSN. It has a RISC processor, with a word size of 32 bits, 32 kB

SRAM, 512 kB Flash, supports ARM and Thumb instructions, and has support for a long multiply instruction which produces a result of 64-bits. The PXA271 uses dynamic voltage scaling and frequency scaling in order to provide different modes of energy efficient operation. It has a Harvard architecture, and supports clock speeds between 13MHz and 540MHz.

Some models also have a Wireless MMX coprocessor which supports 43 SIMD instructions and also provides 16 extra 64-bit registers.

The Imote2 data sheet [30] states that at a clock frequency of 13 MHz and an input voltage of 0.9 V, the current consumption is 33 mA. The power consumption is therefore 29.7 mW, which leads to an energy per cycle of 29.7 mW/13 MHz = 2.285 nWs.

ARM Cortex-M3

The ARM Cortex-M3 [31] has a 32-bit word size with 13 general purpose registers. It has a RISC processor with a Von Neumann architecture, 3-stage pipeline, a maximum clock speed of 150 MHz, a word size of 32 bits, up to 200 kB SRAM, up to 1 MB Flash, supports only Thumb instructions, and has support for a long multiply instruction which produces a result of 64-bits.

It can perform 1.25 MIPS/MHz and has a dynamic power consumption of 149 µW/MHz for 0.180um, 32 µW/MHz for 90 nm and 7 µW/MHz for 40nm. Therefore, on the 90 nm technology it has energy consumption per cycle of 32 µW/1 MHz = 0.032 nWs.

ARM Cortex-M0+

The ARM Cortex-M0+ [6] has a 32-bit word size that uses the ARMv6-M architecture (which is a subset of the ARMv7-M with fewer instructions), 13 general purpose registers, and always runs in Thumb mode. It has a RISC processor with a Von Neumann architecture with a two-stage pipeline in which the fetch, decode and execute operations are split over two cycles. It has a dynamic power consumption of 11 µW/MHz and a static power requirement of 0.95 µW in 90 nm technology. It has 16 kB of SRAM, and the amount of flash memory is manufacturer dependent. It has a short 32-bit hardware multiply instruction which executes in a single cycle, but only outputs the low 32 bits. It is capable of performing 1.08 DMIPS/MHz. Freescale is using the Cortex-M0+ in a wireless sensor node called the MKW01Z128 [7].

The two-stage pipeline is illustrated in Fig. 2.3. By splitting the decode stage over two cycles, the processor is able to complete an instruction in two cycles.

Section 4.2 describes a measurement of the power consumption of the ARM Cortex-M0+. The microcontroller consumes 600 µW at a clock speed of 48 MHz. The energy consumption per cycle is therefore 600µW/48MHz = 12.5pWs.

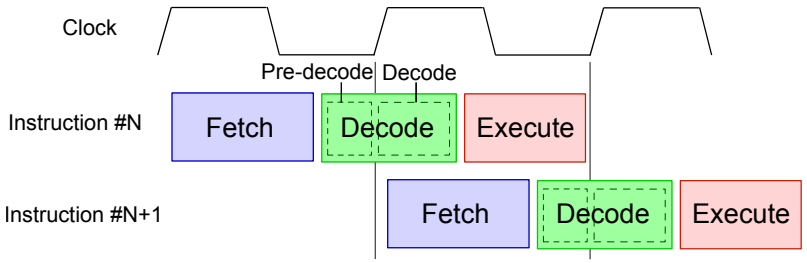


FIGURE 2.3: The two-stage pipeline inside the ARM Cortex-M0+. The decode stage is split over two cycles, which allows the processor to complete an instruction in two cycles. This image is based on an image found in [1].

Discussion

A comparison of the energy consumption per cycle is shown in Fig. 2.4. It needs to be mentioned that some of these microcontrollers have a different technology size. The technology size of the three microcontrollers found on the WSNs (ATMega128L, MSP430, PXA271) are unknown, and therefore the energy consumption was determined from measurements. The energy consumption values for the ARM Cortex-M3, and the ARM Cortex-M0+ were taken from their data sheets. The ARM Cortex-M3 and the ARM Cortex-M0+ both use a technology size of 90 nm, whereas the ARM7TDMI uses a technology size of 0.13µm.

The ATMega128L is the least efficient microcontroller with 4.07nWs per cycle. The MSP430 is the second least efficient with 1.5nWs per cycle. The PXA271 is closely following with 1.25 nWs per cycle. The ARM7TDMI is the third most efficient microcontroller with 0.06nWs per cycle. The ARM Cortex-M3 is the second most efficient with a 0.032nWs per. The ARM Cortex-M0+ is the most efficient per cycle, with an improvement of 125% over the ARM7TDMI.

As the technology size decreases, the dynamic power consumption decrease due to a lowered amount of gate capacitance together with a lowered input voltage. However, with a decrease in technology size, comes an increase in leakage current [32]. Generally speaking the microcontrollers with a smallest technology size have the lowest power consumption.

2.6 Conclusion

We introduced finite fields and described how they are used in elliptic curves. Elliptic curves were introduced on binary fields, as well as prime fields together with some mathematical background for field arithmetic, and elliptic curve arithmetic, and the basic workings of the point multiplication operation. Different key strengths were evaluated to RSA and symmetric key cryptography. Elliptic Curve Cryptography was introduced with two different protocols which use it. The target microcontroller was described together with microcontrollers which are used in implementations found in literature as well as software libraries. The choice of ECC was motivated.

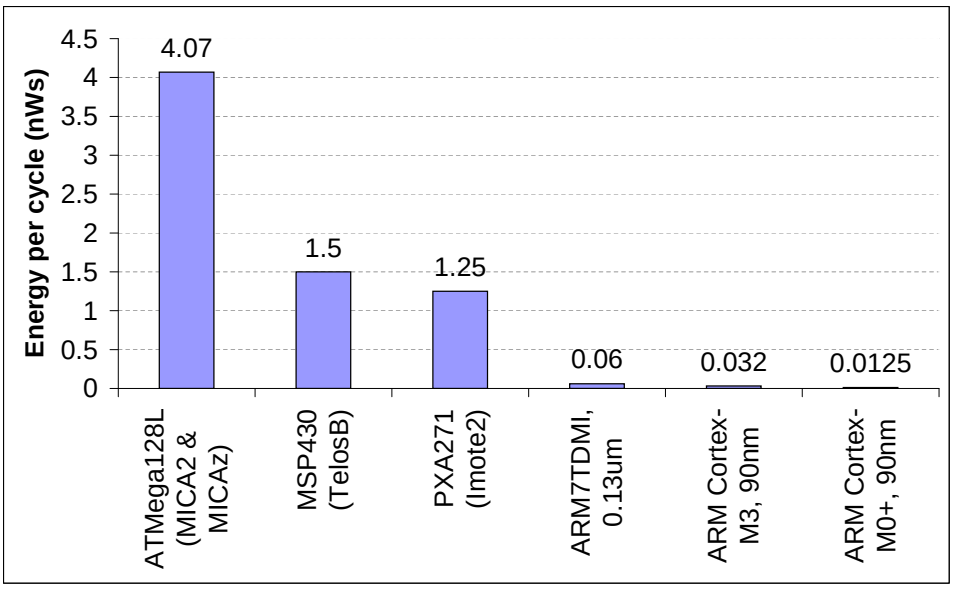


Figure 2.4: Energy usage per cycle for different microcontrollers.

Chapter 3

Implementation

Point multiplication is the core operation of any ECC protocol. Two major choices exist and both have a major impact on the operations carried out in the ECC protocol. As such, we will first compare binary curves and prime curves with each other in order to determine which curve will lead to the faster implementation of a point multiplication. Next, we will provide the algorithms which were actually used inside the implementation and (in some cases) compare their performance with each other for the target platform. Subsequent, the algorithms are analysed to determine the optimal algorithms, and the optimal values for certain parameters. Finally, we will present the initial implementation and discuss the optimizations which were done in order to increase the performance.

3.1 Matching a curve to the architecture

Obtaining fast elliptic curve operations on a specific architecture requires a careful selection of curve, field and algorithms. A fast implementation will lead to a low energy consumption which is a highly desirable property. In this section we will make some estimates for two different curves in order to determine which curve will lead to the least number of cycles to perform a point multiplication on the target architecture. Only binary curves and prime curves are considered.

A model was built to estimate the number of cycles required to perform a point multiplication on both a binary curve and a prime curve. In both cases an in-depth literature research was performed to reveal a set of suitable algorithms, and we consider curves with special properties for our the analysis. Since the majority of the execution time is spent inside the modular multiplication routine for both binary curves and prime curves, models were built for field multiplication routine and all the other field operations were estimated. Since we would like to provide protection for up to 20 years, we considered curves with an equivalent symmetric-key security of 112-bits. Curves with special properties (which will allow for an even faster point multiplication) for binary field and prime fields were considered:

- A pseudo-Mersenne prime curve is evaluated because it allows for fast reduction, and also has the property of being an efficiently computable endomorphism.

This leads to a reduced number of point additions and doublings. The bit-length for the underlying field is 224 because this provides 112-bit security.

- A binary Koblitz curve is evaluated because it allow all doublings to be replaced by squarings by use of a Frobenius map. The polynomial squaring operations are very cheap for binary curves and a fast reduction method is available if a sparse reduction polynomial is used. The bit-length of the underlying field is 233 because this provides 112-bit security.

A model is created to estimate the number of cycles that is required to perform a 224-bit point multiplication on a prime curve and a 233-bit point multiplication on a binary curve. The point multiplication cycles are found by estimating the number of point additions and doublings required to perform the multiplication from the algorithm which is used to perform the point multiplication. The number of additions and doubling are determined by the scalar k. Next, a suitable coordinate system is selected which will determine the amount of field multiplications and squaring operations required to perform the point addition and doubling operations. Next, an efficient point multiplication and squaring algorithm was selected, and an estimate was made of the number of read, write, XOR and shift instructions required to perform the field multiplication, and squaring operations. The instruction counts were used to make an estimate of the amount of cycles required to perform a point multiplication on the target platform.

Excel spreadsheets were constructed to ease the process of building the models. The estimated cycle counts for field multiplication, squaring, elliptic curve operations and point multiplications were calculated using spreadsheets.

3.1.1 Prime Fields

The underlying field's bit-length is 224 ($m = 224$), and will therefore fit in 7 32-bit words or registers ($n = \lceil m/W \rceil$ with W being the processor word size) on the target platform.

The total number of field operations required for a point multiplication using the Non Adjacent Form (NAF) (section 2.3.2) is $m/3A + mD$ where m is the field length, A is the point addition operation and D is the point doubling operation. This assumes that the reduction algorithm is much faster than multiplication (which is the case if the underlying field is based on a pseudo-Mersenne prime which makes use of a fast reduction algorithm). By using the efficiently computable endomorphism, the point multiplication time is reduced by approximately 33% due to the number of point doubling operations being reduced by approximately 50% [15, p. 124] when compared to the NAF method. This is valid if the endomorphism computation doesn't take more execution time than a small number of point additions.

Table 2.4 shows a number of different projective coordinate systems which are available for prime fields. The most efficient coordinate system for a point doubling is the Jacobian projective coordinate system with 4 multiplications and 4 squarings; the fastest system for a point addition is a mixed coordinate Jacobian-Affine addition with 8 multiplications and 3 squarings.

To perform a field multiplication use the Hybrid multiplication method. A row-wise approach is combined with a column-wise approach which multiplies two 32-bit words with each other. On the target platform we have 14 available registers if the link register (LR) is pushed onto the stack, but we also need to consider that the target platform only has a 32-bit output for a multiplication instruction. This means that the multiplier is essentially a 16-bit multiplier which uses two 32-bit words as its inputs. The column-wise approach requires $2n^2 + 2n$ memory operations, n^2 multiplications and requires $4 + \lceil log_2(n)/k \rceil$ registers.

Each 32×32-bit row-wise multiplication will require 4 multiplication operations and 5 addition operations. The overhead of each multiplication is estimated to be 3 MOV instructions to move the values into the correct registers before and after the multiplication instruction. The overhead for each addition to the output vector (which includes the carry-handling) is estimated to be three MOV instruction. The total cost for a 32×32-bit multiplication is therefore 36 cycles which consists of 4 multiply, 5 ADD, and 27 MOV instructions.

The column-wise multiplication requires a total of 49 32×32-bit multiplications. Therefore, to perform a 224-bit scalar multiplication a total of 245 multiply, 245 ADD, and 1323 MOV instructions are required. This leads to a total cost of 1988 cycles. From the literature it was observed that the squaring operation requires approximately 85% of the cost of a field multiplication. It is therefore estimated that the field squaring operation will require approximately 1690 cycles.

For the NAF approach we estimate that 4.86M cycles are required to perform a point multiplication. If the efficiently computable endomorphism is used then we estimate that 3.25M cycles are required to perform a point multiplication. Table 3.1 shows the estimates for the field multiplication, field squaring, NAF and efficiently computable endomorphism methods.

3.1.2 Binary Fields

The underlying field's order is 2^{233} ($m = 233$), which fits inside 8 32-bit words or registers ($n = \lceil m/W \rceil$ with W being the processor word size) on the target platform.

The TNAF method replaces all the point doublings by a point squaring and the number of operations required to perform the point multiplication is roughly $m/3$A [15] where m is the field length and A is field additions.

Table 2.3 shows the number of field operations which are required for different coordinate systems. From this table we can see that the most efficient projective coordinate system for both point additions and doublings is the López-Dahab coordinate system with 8 multiplications for a mixed coordinate LD-affine addition and 4 multiplications for a point doubling.

The field multiplication method was chosen to be the Left-to-Right Comb method (Algorithm 8). In this analysis the parameter W is used to indicate the processors' word size ($W = 32$ on the target platform). The inner loop of this algorithm requires $W(n^2/2)$ reads from y, assuming that the branch in the if-statement is taken half of the time. We further require $(n + 1)(W - 1)$ reads for the shift operation, $W(n^2/2)$ writes to C, $(n + 1)W$ writes during the shift, $(W/2)n^2$ add operations and $(W - 1)$

Table 3.1: Prime field simulation. * ECE is an abbreviation for Efficiently computable endomorphism.

Row-wise (32×32) MUL		
Operation	**Amount**	**Note**
Number of additions	5	
Number of 16-bit MULs	4	
Overhead per 16-bit MUL	4	3×MOV
Overhead per addition	4	3×MOV
Total	36	

Hybrid MUL		
Operation	**Amount**	**Note**
Reads	112	$2n + 2n^2$
Writes	49	n^2
32×32 MULs	49	n^2
Total	1988	

Squaring		
Operation	**Amount**	**Note**
SQR Cost	1670	85% of Hybrid MUL

Elliptic Curve Operations		
Operation	**Amount**	**Note**
Point Addition (A)	8 MUL + 3 SQR	Jacobian-affine
Point Double (D)	4 MUL + 4 SQR	Jacobian

Totals		
Operation	**Amount**	**Note**
NAF	4.86 M	$(m/3)$A+mD
ECE *	*3.25 M*	NAF×77%

shift operations. Each shift operation requires $2n$ shift and n add operations. We estimate that each 233-bit field multiplication requires 1208 read, 752 write, 745 XOR and 315 shift instructions, and further requires a total of 11350 cycles to perform a point multiplication. From the results in [33] it was observed that the amount of cycles are reduced by a factor of 2.3 if the López-Dahab (LD) method with a window of size 4 is used. Therefore with a factor 2.3 reduction in cost it was estimated that the LD, $w = 4$ method will require 4935 cycles for a field multiplication.

Using the TNAF approach we estimate that a point multiplication will require approximately 3.07M cycles. Table 3.2 shows the estimates for the field multiplication

and point multiplication using TNAF.

Table 3.2: Binary field simulation.

Left-to-Right Comb		
Operation	**Amount**	**Note**
Read	2967	$(n^2/2)W + (n+1)(W-1)$
Write	1312	$(n^2/2)W + (n+1)W$
n-word additions	2048	$(n^2/2)W$
$n+1$ word shifts	744	$W-1$
Left-to-Right Comb Total	11350	
LD, $w = 4$	4935	(Left-to-Right Comb Total)/2.3

Elliptic Curve operations		
Operation	**Amount**	**Note**
Point Addition (A)	8 M + 7 S	LD, $w = 4$

Totals		
Operation	**Amount**	**Note**
TNAF	*3.07 M*	m/3A

3.1.3 Conclusion

In order to determine which curve will lead to the fastest execution time efficient algorithms were selected to make an estimate on the cost of performing a point multiplication on curves with different underlying fields. Even though we tried to make the estimate as accurate as possible, it is very difficult to make an accurate estimate without actually writing the algorithms. The estimates show that the cycle counts for both binary field and prime fields will lead to similar execution times with binary fields requiring 3.07 M cycles and prime fields requiring 3.25 M cycles for a point multiplication. This is consistent with results found in literature [15, 34, 35] where the execution times for a point multiplication on either a prime or a binary field will lead to similar timings.

A disadvantage of using prime fields is that a large amount of multiply instructions are required due to the target having only a 16-bit multiplication instruction. This has the further disadvantage of having a large number of carry handling operations.

Prime fields require a large amount of multiply and addition instructions whereas binary fields require a large amount of shift and XOR instructions. If we look at the energy measurement results in Chapter 4 we can see that both the shift and XOR instructions require approximately 8% less energy than the multiply and addition instructions. Therefore an implementation which is based on a binary field will require less energy than an implementation based on a prime field with the same

cycle count because the instructions required to perform the arithmetic on the binary field requires less energy.

We chose to use the standard SEC [36] binary Koblitz curve of order 2^{233} called *sect233k1*. This curve allows all doublings to be replaced by squarings by use of a Frobenius map. The polynomial squaring operations are very cheap for binary curves and a fast reduction method is available because the reduction polynomial is a trinomial. The irreducible polynomial is

$$f(x) = x^{233} + x^{74} + 1$$

The curve

$$E : y^2 + xy = x^3 + ax^2 + b$$

over $\mathbb{F}_{2^m}$ is defined by

$$a = 0, b = 1$$

The base point in compressed form is:
$G =$ 04 017232BA 853A7E73 1AF129F2 2FF41495 63A419C2 6BF50A4C
9D6EEFAD 612601DB 537DECE8 19B7F70F 555A67C4 27A8CD9B
F18AEB9B 56E0C110 56FAE6A3

3.2 Actual algorithms used

This section discusses the algorithms which are analysed for the target architecture in the following section. Since a large amount of algorithms exist for performing operations on the field and on elliptic curves, a number of algorithms were analysed and adapted to suit the architecture of the target platform.

3.2.1 Field multiplication

Recall that field multiplication is the most time-consuming operation of ECC, hence a lot of effort was invested to find the best method to perform this operation on the target architecture. There are generally two approaches which are followed to perform field multiplication. The first is approach performs a single operations with large input parameters, whereas the second approach performs a number of operations with smaller input parameters and is called the divide-and-conquer approach. Both approaches are explained in the following:

The divide-and-conquer approach attempts to reduce the memory requirements of the multiplication operation by shortening the input parameters of the multiplication routine. This is done by splitting the input parameters into two or more chunks, performing multiple multiplications on the smaller input parameters, and then combining the results. By reducing the size of the input parameters, the number of intermediate values that need to be stored inside registers and memory are reduced, which reduces the number of expensive accesses to memory. However, overhead is added by this approach because of splitting the input parameters and combining the multiplication results. A trade-off exists between the overhead from the splitting and

combination steps, and the reduction of memory accesses inside the multiplication routine. By increasing the number of splits, the overhead (and number of cycles) from the splitting and recombining steps increase; whereas the number of memory accesses inside the multiplication operation reduces. One needs to find the sweet spot which will lead to the best results for the target architecture.

Processing the input parameters as a whole has the disadvantage of requiring a larger amount of intermediate values which leads to a larger number of memory accesses inside the multiplication routine. The advantage of this method is that there is no overhead for splitting and recombination as the whole input block is processed in a single function.

In the remainder of this section we describe the different approaches to perform field multiplications in more detail. First the basis of the divide-and-conquer method is introduced, followed by two potential algorithmic enhancements: The operand-scanning method and the Karatsuba-Ofman method. Thereafter the effects of combining the divide-and-conquer approach with a windowed-multiplication method is discussed. Subsequent three different methods are discussed to process the input parameters as a whole.

Divide-and-conquer

The divide-and-conquer method is based on the principle of splitting the input parameters, performing the multiplications on the smaller input parameters, and then combining the results of the multiplications. Suppose that $n = 2l$, $x = x_1 2^l + x_0$ and $y = y_1 2^l + y_0$. Then

$$xy = x_0 \cdot y_0 + [x_1 \cdot y_0 + x_0 \cdot y_1]2^l + x_1 \cdot y_1 2^{2l}$$

with $n = 2l$, $x = x_1 2^l + x_0$, $y = y_1 2^l + y_0$, 2^l and 2^{2l} are shift operations. $x \cdot y$ is therefore computed by splitting x and y each into two parts, performing four multiplications and then recombining the results of the multiplications using shift and add operations. The amount of memory required to perform each multiplication operations is reduced, but the number of multiplication operations have increased.

Operand-Scanning

The order in which the multiplication operations are performed can have an impact on the overall performance of the multiplication. Fig. 3.1 shows three different methods for performing field multiplications when the input parameters are split into three equal parts. The first method is the classic *row-wise multiplication* method in which the summands are sorted from left-to-right. The second method is the *operand-scanning* method in which the summands are sorted in a manner where the algorithm moves through the multiplier x_i, starting with x_0, and multiplies x_i with any multiplicand y_j. The third method is a type of *optimized operand-scanning* method in which the same approach is followed as for the operand-scanning method where the algorithm moves through the multiplier x_i, starting with x_0 and multiplying x_i with any multiplicand of y_j; however, instead of performing the multiplication

operations each inside its own function, the multiplication operations for all terms with the same x_i are interleaved inside a single function. In other words if a split of 3 is performed on the input parameters, then the three terms $(x_2 \cdot y_2 + x_2 \cdot y_1 2^l + x_2 \cdot y_0 2^{2l})$, $(x_1 \cdot y_0 2^l + x_1 \cdot y_1 2^{2l} + x_1 \cdot y_2 2^{3l})$, and $(x_0 \cdot y_2 2^{2l} + x_0 . y_1 2^{3l} + x_0 . y_0 2^{4l})$ are each multiplied inside a single function. In Fig. 3.1 the three multiplication functions are indicated with *P1*, *P2*, and *P3*. Inside each function the multiplication of x_i with y_j are interleaved in a manner in which the accesses to x_i are minimized. This has the advantage of minimizing the number of accesses to read x_i at the cost of an increase in code size because the multiplication function is larger.

If combined with a windowed multiplication algorithm, the total number of loops inside the multiplication routine is reduced at the cost of storing all the precomputation table simultaneously in memory. All three precomputation tables need to be available at the same time because the multiplicand now has three forms: y_0, y_1, and y_2.

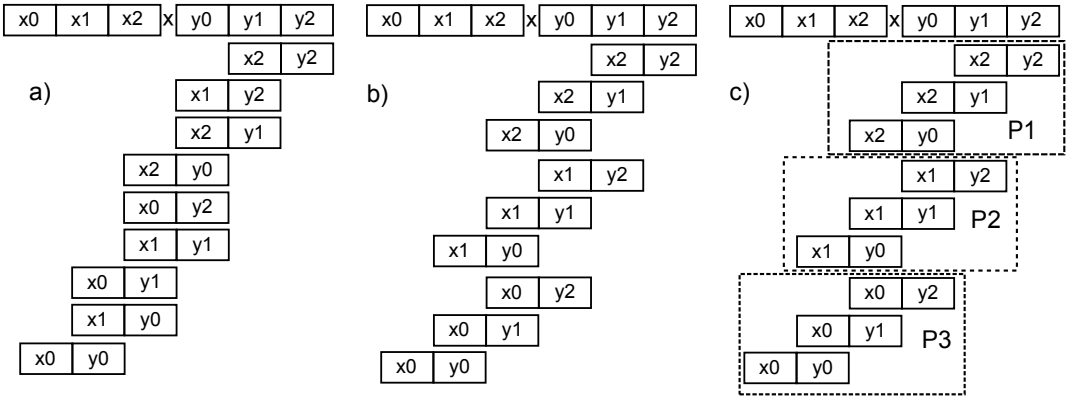


Figure 3.1: The input parameters are split into three equal parts and the following multiplication methods are used: a) Schoolbook multiplication b) Operand-scanning c) Optimized operand-scanning.

In the remainder of this text when referring to the operand-scanning form we intend to actually refer to the optimized operand-scanning form. This method was used in a simulation in section 3.3.

Karatsuba-Ofman

The Karatsuba-Ofman method reduces the complexity of the operand-scanning form from $O(n^2)$ down to $O(n^{log_2 3})$. Assume we have the input parameters x and y which are both n bits long. Then

$$x \cdot y = x_1 \cdot y_1 2^{2l} + [(x_0 + x_1) \cdot (y_0 + y_1) - x_1 \cdot y_1 - x_0 \cdot y_0]2^l + x_0 \cdot y_0 \qquad (3.1)$$

with $n = 2l$, $x = x_1 2^l + x_0$ and $y = y_1 2^l + y_0$. The above equation performs the multiplication using three multiplications of l-bit polynomials (instead of 4

multiplications as with the other divide-and-conquer methods), two additions, two subtractions and two shift operations (denoted by 2^l and 2^{2l}). The algorithm only requires three multiplications but the recombination process is more complex due to having an added number of shift operations and further requiring all the results of the multiplications to be kept in memory for simultaneous recombination. If combined with a windowed multiplication algorithm, the number of loops inside the multiplication routine is reduced at the cost of some storage for a precomputation table as well as some overhead for calculating the precomputation table. Three precomputation tables are required (instead of two for the other divide-and-conquer methods with a split of two) for this method because the multiplicand now has three different forms: y_0, y_1 and $(y_0 + y_1)$.

This method is used in a simulation in section 3.3.

Divide-and-conquer methods combined with windowed multiplication

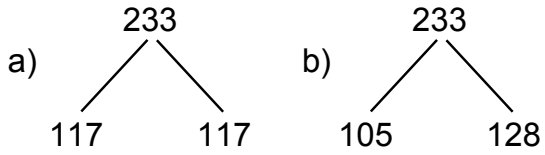


FIGURE 3.2: Splitting the input parameters a) Symmetrically into two parts of equal length b) Asymmetrically on the word size of the processor. The word size for this example is 32-bit.

When combining a divide-and-conquer method with a windowed multiplication algorithm, the input parameters are split into two or more parts before being processed by the multiplication function. The input parameters can either be split into equal-sized blocks (symmetrically), or it could be split in an asymmetrical fashion according to the word-size of the processor. Fig. 3.2 shows a graphical representation of the splitting process.

The symmetric approach splits the parameters into parts which all have exactly the same length. The advantage of this method is that it is possible to re-use the code for the multiplication routine because all of the input parameters are the same size. In the case of a windowed field multiplication algorithm we are quite likely to need only small lookup tables. The disadvantage of this method is that the input parameters need to be shifted before the multiplications in order to get word-aligned and the output of the multiplications need to be shifted back to their original positions. This method was used in a simulation in section 3.3, and in an implementation in section 3.4.3.

The asymetrical approach splits the input parameters into parts which are word-aligned. There is therefore no need to perform any shifting of the input parameters or multiplication results. The disadvantages of this approach are that that the multiplication algorithm cannot be re-used due to the varying length of input parameters, and a larger lookup table is required for the most significant polynomial.

The storage requirements for the lookup table of the windowed field multiplication methods is given in Eq. 3.6. While computing the lookup table the input polynomial is shifted left by $w-1$ bits. If the entire most significant word of the input polynomial is used (i.e. it contains a number which is large enough to occupy the entire word), a left shift of the most significant polynomial will cause an overflow into the next word, requiring an extra word to store the result of the left shift operation. The most significant input polynomial will always be completely populated with bits and will always require a larger lookup table for an asymmetrical split. Fig. 3.3 illustrates this process. The possibility exists that the least significant polynomial would not require a large lookup table because its least significant word contains a number which occupies less than, or equal to $W-(n-1)$ bits. A large lookup table requires more computations to generate the lookup table, more memory, and additional table lookups while performing the actual multiplication. This method is used in a simulation in section 3.3, and in an implementation in section 3.4.3.

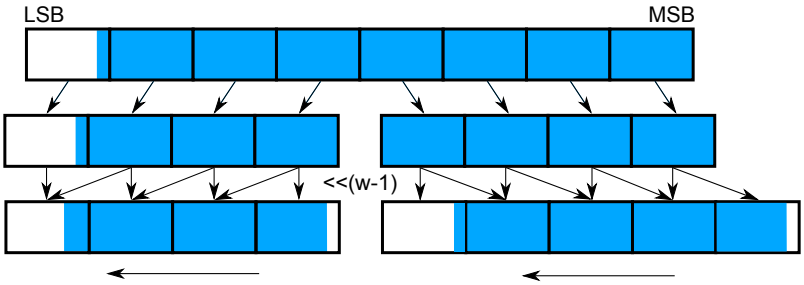


Figure 3.3: An asymmetric split causes the most significant polynomial to be completely populated with bits and will therefore require a larger lookup table.

For an asymmetrical split, the total size of the lookup tables is given by

$$\begin{cases} (2^w(n/d) + (d-1)2^w((n/d)+1)) & \text{, if } (W - m \bmod W) \geq w-1. \\ 2^w(n+1) & \text{, if } (W - m \bmod W) < w-1. \end{cases} \tag{3.2}$$

where d is the number of splits, w is the window size and n is the number of words in the input parameters. The term $(W - m \bmod W)$ indicates the number of bits in the most significant word which are not currently being used. Here we can see that the amount of memory required to store the lookup tables also increase as the parameter d increases.

For an asymmetrical split, the total number of accesses to the lookup table is given by

$$\begin{cases} \lceil W/w \rceil (n^2 + n(d-1)) & \text{, if } (W - m \bmod W) \geq w-1. \\ \lceil W/w \rceil (n^2 + nd) & \text{, if } (W - m \bmod W) < w-1. \end{cases} \tag{3.3}$$

where d is the number of splits, w is the window size and n is the number of words in the input parameters. Here we can see that the amount of memory accesses increase with the parameter d.

For a symmetrical split, the total size of the lookup tables is given by

$$\begin{cases} 2^w(n) & \text{, if } ((W - m \bmod W)/d) \geq w - 1. \\ 2^w(n+1) & \text{, if } ((W - m \bmod W)/d) < w - 1. \end{cases} \tag{3.4}$$

where d is the number of splits, w is the window size and n is the number of words in the input parameters.

For an symmetrical split, the total number of accesses to the lookup table is given by

$$\begin{cases} \lceil W/w \rceil (n^2) & \text{, if } ((W - m \text{mod} W)/d) \geq w - 1. \\ \lceil W/w \rceil ((n+1)^2) & \text{, if } ((W - m \text{mod} W)/d) < w - 1. \end{cases} \tag{3.5}$$

where d is the number of splits, w is the window size and n is the number of words in the input parameters.

For the remainder of this text, assume that the conditions for a small lookup table in Eq. 3.2, 3.3, 3.4, and 3.5 are satisfied. This is a valid assumption given that $m = 233$, $W = 32$, and $d \leq 2$.

A split of more than two was considered but was found to be impractical due to the following reasons:

- The symmetrical split method has the benefit of reducing the number of memory operations in the multiplication, at the expense of having to split the input parameters, and having to shift the results of the multiplications before recombination. Each additional split leads to an increasingly larger number of shift operations due to the increased number of splits inside the input polynomial, as well as an increased number of shifting operation on the multiplication results. For a split of more than two the benefits that come from reducing the amount of memory accesses inside the multiplication is outweighed by the disadvantages of having to split the input parameters and recombine multiplication results.

- The asymmetrical split method has the benefit of reducing the number of memory operations in the multiplication, at the expense of requiring a larger lookup table for the most significant input parameters. Each additional split leads to an increased number of large lookup tables (Eq. 3.2), more computations to generate the lookup tables, as well as significantly more accesses to the larger lookup tables (Eq. 3.3) during the multiplication. When performing a first-level split by a factor of two, the number of intermediate values inside the multiplication routine are already reduced to an amount which can be kept completely inside the target platform's registers. Due to the fact that each extra split leads to a significant extra overhead due to the larger lookup tables, it is therefore less advantageous to perform a split by a factor of more that two.

López-Dahab Left-to-right comb method with windows of width w

The López-Dahab (LD) field multiplication algorithm is a windowed multiplication algorithm for $\mathbb{F}_{2^m}$. Its goal is to reduce the number of multi-precision shift operations

LOA 15: López-Dahab multiplication in $\mathbb{F}_{2^m}$.

Input: $x(z) = a[0 \cdots n-1], b(z) = b[0 \cdots n-1]$
Output: $c(z) = c[0 \cdots 2n-1] = a(z)b(z)$
1: Compute $T(u) \leftarrow u(z)y(z)$ for all polynomials $u(z)$ of degree lower than w
2: $c[0 \cdots 2n-1] \leftarrow 0$
3: **for** $j \leftarrow \lceil W/w \rceil - 1$ downto 0 **do**
4: **for** $k \leftarrow 0$ to $n-1$ **do**
5: $u = (x[k] \gg j \cdot w)$ & 0xF
6: **for** $l \leftarrow 0$ to n **do**
7: $c[l+k] \leftarrow c[l+k] \oplus T[u][l]$
8: **end for**
9: **end for**
10: **if** $(k \neq 0)$ **then**
11: $c(z) = c(z).z^w$
12: **end if**
13: **end for**
14: **return** c

by scanning x with w bits at a time and performing a table lookup, thereby reducing the number of outer loop iterations to only $\lceil W/w \rceil$. The lookup table is computed with $T(u) \leftarrow u(z)y(z)$ for all polynomials $u(z)$ of degree lower than w. The number of words required to store the lookup table is given by:

$$\begin{cases} 2^w(n+1) & \text{, if degree}(y) > nW - (w-1). \\ 2^w(n) & \text{, if degree}(y) \leq nW - (w-1). \end{cases} \tag{3.6}$$

If the most significant word of y is smaller or equal than $W-(n-1)$ then the lookup table will fit into $2^w n$ words. This is due to the fact that while generating the lookup table, y gets shifted by $w-1$ which causes the polynomial to overflow into the next word.

Algorithm 15 shows the López-Dahab field multiplication algorithm. For $\mathbb{F}_{2^m}$ the two inputs x and y are both n words long and the result is $2n$ words long. This method is used in a simulation in section 3.3, and in an section 3.4.2.

López-Dahab with rotating registers

The López-Dahab with rotating registers method is an improvement on the original López-Dahab method and was introduced by Aranha et al. [37]. This method makes use of a rotating register scheme to accumulate results inside the multiplication. The original algorithm was published as a customization for the ATMega128L. The algorithm is designed with the aim of reducing the number of memory accesses while accumulating the results that occurs inside the outer loop inside a rotating register window instead of writing it back to the memory. Algorithm 16 shows the algorithm with a rotating register window of $n+1$. This algorithm assumes that there are $n+1$

LOA 16: López-Dahab with rotating registers multiplication in $\mathbb{F}_{2^m}$.

Input: $x(z) = x[0 \cdots n-1]$, $y(z) = y[0 \cdots n-1]$.
Output: $c(z) = c[0 \cdots 2n-1] = x(z)y(z)$.
Note: $v_i \leftarrow (r_i, \cdots, r_0, r_n, \cdots, r_i)$, v_i is a vector of $n+1$ registers.

1: Compute $T(u) \leftarrow u(z)y(z)$ for all polynomials $u(z)$ of degree lower than w
2: $c[0 \cdots 2n-1] \leftarrow 0$
3: **for** $j \leftarrow \lceil W/w \rceil - 1$ downto 0 **do**
4: **for** $k \leftarrow 0$ to $n-1$ **do**
5: $u = (x[k] \gg j.w)$ & 0xF
6: $v_i[0] \leftarrow T[u][0]$
7: **for** $l \leftarrow 1$ to n **do**
8: $v_i[l] \leftarrow v_i[l] \oplus T[u][l]$
9: **end for**
10: $c[k] \leftarrow c[k] \oplus r_k$
11: **end for**
12: $c[n \cdots 2n-1] \leftarrow c[n \cdots 2n-1] \oplus (r_{n-2}, \cdots, r_0, r_n)$
13: **if** $(k \neq 0)$ **then**
14: $c(z) \leftarrow c(z) \ll w$
15: **end if**
16: **end for**
17: **return** c

registers available to accumulate the results. This method is used in a simulation in section 3.3, and in an implementation in section 3.4.3.

López-Dahab with fixed registers

We propose a new optimization to the field multiplication algorithm by López and Dahab and call it the López-Dahab with fixed registers method. This algorithm aims to reduce the number of memory operations by keeping as many of the words of c as possible inside registers. The most frequently used words are stored inside fixed register positions and the least frequently used words are stored inside memory. On the target platform it is feasible to store a maximum of nine words inside registers. If we assume that $n = 8$ (e.g. $\mathbb{F}_{2^{233}}$) then we can store the $n+1$ most frequently used words inside registers and the remaining $n-1$ words inside memory. By analysing the standard López-Dahab algorithm for $w = 4$ it can be determined that $c[3 \cdots 12]$ are the most frequently used $n+1$ elements and these are therefore stored inside the registers. $c[0 \cdots 2]$ and $c[13 \cdots 15]$ are the least frequently used $n-1$ elements inside c and are therefore stored inside memory. Algorithm 17 shows the López-Dahab multiplication with fixed registers for $n = 8$ and a register count of $n+1$. The vector v denotes the internal state vector of the intermediate results. The vector v contains all the intermediate results and stores these inside memory as well as fixed register positions. This method is used in a simulation in section 3.3, and in an

LOA 17: López-Dahab multiplication with fixed registers in $\mathbb{F}_{2^m}$ for $n = 8$.

Input: $x(z) = x[0 \cdots n-1], y(z) = y[0 \cdots n-1]$
Output: $c(z) = c[0 \cdots 2n-1] = x(z)y(z)$
Note: v denotes the internal state vector composed of $n-1$ memory addresses and $n+1$ registers $v \leftarrow (c[0], c[1], c[2], r_0, r_1, \cdots, r_n, c[12], c[13], c[14], c[15])$

1: Compute $T(u) \leftarrow u(z)y(z)$ for all polynomials $u(z)$ of degree lower than w
2: $v[0 \cdots 2n-1] \leftarrow 0$
3: **for** $j \leftarrow \lceil W/w \rceil - 1$ downto 0 **do**
4: **for** $k \leftarrow 0$ to $n-1$ **do**
5: $u = (x[k] \gg j.W)$ & 0xF
6: **for** $l \leftarrow 0$ to $n-1$ **do**
7: $v[l+k] \leftarrow v[l+k] \oplus T[u][l]$
8: **end for**
9: **end for**
10: **if** $(k \neq 0)$ **then**
11: $v(z) = v(z).z^w$
12: **end if**
13: **end for**
14: **return** v

implementation in section 3.4.3.

If $n = 4$ (as would be required by the divide-and-conquer methods on the target platform for $\mathbb{F}_{2^{233}}$) then the algorithm needs to be modified slightly. Since it's feasible to keep 9 intermediate values inside registers on the target platform, all of the intermediate values can be kept inside the registers and the internal state vector will then become $v \leftarrow (r_0, r_1, \cdots, r_n)$, so that all the intermediate values are kept inside registers.

López-Dahab with fixed registers, $w \neq$ scanning size s

Each precomputation table requires $2^w n$ words to be precomputed for an input polynomial which is shorter than $n - w + 2$ bits. If the parameter $w = 8$ then the memory requirements and time spent generating the lookup table is very large. An alternative method could be to generate two precomputation tables with $w = 4$ and then parsing the input polynomial 8-bits at a time. Considering the multiplication $x \cdot y$, the first precomputation table is generated from y and the second precomputation table is generated from $y \ll 3$. During the multiplication x is read in 8 bits at a time and is split symmetrically into two 4-bit sections. The lookup is done from both tables and the result is added together. This is called scanning $s = 8$ with a window of $w = 4$. This method was used in a simulation in section 3.3.

This method is attractive because the number of outer loops iterations is reduced to $\lceil W/s \rceil$; which means we will have a large reduction of multi-precision shifts on $c(z)$. The disadvantage of this method is that the inner loop is more complex with

LOA 18: López-Dahab multiplication with fixed registers in $\mathbb{F}_{2^m}$ for $n = 8$ and $s = 8$.

Input: $x(z) = x[0 \cdots n-1], y(z) = y[0 \cdots n-1]$
Output: $c(z) = c[0 \cdots 2n-1] = x(z)y(z)$
Note: v denotes a vector of $n-1$ memory addresses and $n+1$ registers $(c[0], c[1], c[2], r_0, r_1, \cdots, r_n, c[12], c[13], c[14], c[15])$

1: Compute $T1(u) \leftarrow u(z)y(z)$ for all polynomials $u(z)$ of degree lower than w
2: Compute $T2(u) \leftarrow u(z)(y(z) \ll 3)$ for all polynomials $u(z)$ of degree lower than w
3: $c[0 \cdots 2n-1] \leftarrow 0$
4: **for** $j \leftarrow \lceil W/s \rceil - 1$ downto 0 **do**
5: **for** $k \leftarrow 0$ to $n-1$ **do**
6: $u = (x[k] \gg 2.j.W)$ & 0xF
7: **for** $l \leftarrow 0$ to $n-1$ **do**
8: $v[l+k] \leftarrow v[l+k] \oplus T1[u][l]$
9: **end for**
10: $u = (x[k] \gg 2.(j+1).W)$ & 0xF
11: **for** $l \leftarrow 0$ to $n-1$ **do**
12: $v[l+k] \leftarrow v[l+k] \oplus T2[u][l]$
13: **end for**
14: **end for**
15: **if** $(k \neq 0)$ **then**
16: $c(z) = c(z).z^w$
17: **end if**
18: **end for**
19: **return** c

more table lookup operations as well as needing to store two lookup tables in memory. If we assume that $n = 8$ (e.g. $\mathbb{F}_{2^{233}}$) then we can store the $n+1$ most frequently used words inside registers and the remaining $n-1$ words are stored inside memory. By analysing the standard López-Dahab algorithm for $w = 4$ it can be determined that $c[3 \cdots 12]$ are the most frequently used $n+1$ elements and are therefore stored inside the registers. $c[0 \cdots 2]$ and $c[13 \cdots 15]$ are the least frequently used $n-1$ elements inside c and are therefore be stored inside memory. This method is used in a simulation in section 3.3.

The algorithm is listed as algorithm 18.

3.2.2 Modular Reduction

Since the curve we are using has a sparse reduction polynomial, the reduction can be computed efficiently one word at a time. The reduction polynomial is $f(z) = z^{233} + z^{74} + 1$. Algorithm 19 shows the fast modular reduction algorithm adapted for a 32-bit processor. The algorithm reduces the polynomial of $2n$ words to a polynomial of only 233 bits. This method is used in a simulation in section 3.3, and an implementation in section 3.4.2

LOA 19: Fast reduction modulo $f(z) = z^{233} + z^{74} + 1$ (with W=32).

Input: A polynomial $c(z)$ of degree at most 464
Output: $c(z)$ mod $f(z)$
1: **for** $i \leftarrow 15$ downto 8 **do**
    {Reduce $C[i]z^{32i}$ modulo $f(z)$}
2: $T \leftarrow C[i]$
3: $C[i-8] \leftarrow C[i-8] \oplus (T \ll 23)$
4: $C[i-7] \leftarrow C[i-7] \oplus (T \gg 9)$
5: $C[i-5] \leftarrow C[i-5] \oplus (T \ll 1)$
6: $C[i-4] \leftarrow C[i-4] \oplus (T \ll 31)$
7: **end for**
8: $T \leftarrow C[7] \gg 9$ {Extract bits 9-31 of $C[7]$}
9: $C[0] \leftarrow C[0] \oplus T$
10: $C[2] \leftarrow C[2] \oplus (T \ll 10)$
11: $C[3] \leftarrow C[3] \oplus (T \gg 22)$
12: $C[7] \leftarrow C[7]$ & 0x1FF {Clear the reduced bits of $C[7]$}
13: **return** $(C[7], C[6], C[5], C[4], C[3], C[2], C[1], C[0])$

It is possible to reduce the number of memory operations by keeping as many of the elements of C as possible inside registers. If $C[0 \cdots 7]$ is stored inside registers, algorithm 19 can be executed and all the updates to C can be kept inside the registers. As soon as an element of C has obtained it's final value then that element is written to memory to free up the register which it's occupying. By doing this it is possible to have no redundant read or write operations.

3.2.3 Squaring

Squaring is done by using a 16-bit lookup table which contains all the precomputed square values for an 8-bit input and a 16-bit output. The lookup table requires 256, 16-bit entries which requires a total of 4 kB of memory. Algorithm 20 shows this method. This method is used in an implementation in section 3.4.2

It can be observed that the squaring operation is always followed by modular reduction which leads to redundant memory operations. These redundant memory operations could be reduced by interleaving the modular reduction and the squaring functions. The lower half of the output of the squaring operation is kept inside the registers and the upper half is expanded and then immediately reduced. The upper half of the elements are therefore not required to be stored first and reduced later. This leads to at least 8 less write, and 8 less read operations from memory. This method is shown in algorithm 21. This method is used in a simulation in section 3.3, and an implementation in section 3.4.2

Input: $x(z) = x[0 \cdots n-1], y(z) = y[0 \cdots n-1]$
Output: $c(z) = c[0 \cdots 2n-1] = x(z)y(z)$
Note: v_i denotes the vector of $n+1$ registers $(r_i, \cdots, r_0, r_n, \cdots, r_i)$

LOA 20: Polynomial squaring (with word length W=32).

Input: A polynomial $a(z)$ of degree at most $m-1$
Output: $c(z) = a(z)^2$
1: Precomputation. For each byte $d = (d_7, \cdots, d_1, d_0)$, compute the 16-bit quantity $T(d) = (0, d_7, \cdots, d_1, 0, d_0)$
2: **for** $i \leftarrow 0$ to $n-1$ **do**
3: $A[i] \leftarrow (u_3, u_2, u_1, u_0)$ where u_j is a byte
4: $C[2i] \leftarrow (T(u_1), T(u_0))$
5: $C[2i+1] \leftarrow (T(u_3), T(u_2))$
6: **end for**
7: **return** c

3.2.4 Inversion

The Extended Euclidean Algorithm for polynomials is used to perform inversion and is shown in algorithm 10. This method is used in an implementation in section 3.4.2.

One optimization to this algorithm is to avoid the swap operation (algorithm 22) by means of performing code duplication and thereby eliminating a large number of expensive memory operations. A further optimization which isn't indicated in algorithm 22 is to use indices to the most significant word of u and v which contains a value larger than zero. These indices are then used to perform a fast calculation of the degree of the polynomial and is also used to reduce the number of operations inside the variable field shift function. This method is used in an implementation in section 3.4.3.

3.3 Analysis of algorithms

In this section we will analyse different algorithms and make estimates on the number of instructions which are required to execute these algorithms on the target platform. In order to verify if the estimates are correct, the algorithms were first implemented in C to obtain a thorough understanding of the algorithm.

By trial and error it was determined that a maximum of nine 32-bit words can be kept inside the registers. To simulate these nine registers inside C, nine local variables are declared and are used as if they are registers. Whenever more than nine intermediate values needs to be stored, a variable is written to memory and the local variables (registers) are then re-used.

In many cases we were faced with the option to use either the ADD or the XOR instruction. In all of these cases the XOR instruction was used because it requires less energy than the ADD instruction.

3.3.1 Field Multiplication

Several field multiplication algorithms and combinations of these algorithms are analyzed to determine which combination of algorithms will lead to the fastest

loa 21: Polynomial squaring with reduction (with word length W=32).

Input: A polynomial $a(z)$ of degree at most $m-1$
Output: $c(z) = a(z)^2$
Note: $A[i] \leftarrow (B_{i,3}, B_{i,2}, B_{i,2}, B_{i,1}, B_{i,0})$ where $B_{i,*}$ is a byte
$R(r_7, r_8, r_{10}, r_9, r_{11}) = \{r_7 \oplus r_{11} << 23, r_8 = r_8 \oplus r_{11} >> 9, r_{10} = r_{10} \oplus r_9 >> 31\}$

1: Precomputation. For each byte $d = (d_7, \cdots, d_1, d_0)$, compute the 16-bit quantity $T(d) = (0, d_7, \cdots, d_1, 0, d_0)$
2: $r_2 \leftarrow (T(B_{1,1}), T(B_{1,0}))$ $\{C[2]\}$
3: $r_3 \leftarrow (T(B_{1,3}), T(B_{1,2}))$ $\{C[3]\}$
4: $r_4 \leftarrow (T(B_{2,1}), T(B_{2,0}))$ $\{C[4]\}$
5: $r_5 \leftarrow (T(B_{2,3}), T(B_{2,2}))$ $\{C[5]\}$
6: $r_6 \leftarrow (T(B_{3,1}), T(B_{3,0}))$ $\{C[6]\}$
7: $r_7 \leftarrow (T(B_{3,3}), T(B_{3,2}))$ $\{C[7]\}$
8: $r_8 \leftarrow (T(B_{4,1}), T(B_{4,0}))$ $\{C[8]\}$
9: $r_{10} \leftarrow (T(B_{5,1}), T(B_{5,0}))$ $\{C[10]\}$
10: $r_9 \leftarrow (T(B_{5,3}), T(B_{5,2}))$ $\{C[11]\}$
11: $r_{11} \leftarrow (T(B_{7,3}), T(B_{7,2}))$ $\{C[15]\}$
12: $R(r_7, r_8, r_{10}, r_9, r_{11})$ {Reduce $r_{11}/C[15], r_{11}$ free}
13: $R(r_3, r_4, r_6, r_7, r_9)$ {Reduce $r_9/C[11], r_9$ free}
14: $r_9 \leftarrow (T(B_{4,3}), T(B_{4,2}))$ $\{C[9]\}$
15: $r_{11} \leftarrow (T(B_{7,1}), T(B_{7,0}))$ $\{C[14]\}$
16: $R(r_6, r_7, r_9, r_{10}, r_{11})$ {Reduce $r_{11}/C[14], r_9$ free}
17: $R(r_2, r_4, r_5, r_6, r_{10})$ {Reduce $r_{10}, C[10], r_{10}$ free}
18: $r_{10} \leftarrow (T(B_{6,3}), T(B_{6,2}))$ $\{C[13]\}$
19: $R(r_5, r_6, r_8, r_9, r_{10})$ {Reduce $r_{10}/C[13]$}
20: $r_2 = r_2 \oplus (r_9 \gg 9)$, $r_4 = r_4 \oplus (r_9 \ll 1)$, $r_5 = r_5 \oplus (r_9 \gg 31)$ {$r_9/C[11]$ free}
21: $r_{10} \leftarrow (T(B_{6,1}), T(B_{6,0}))$ $\{C[12]\}$
22: $R(r_4, r_5, r_7, r_8, r_{10})$ {Reduce $r_{10}/C[12]$}
23: $r_3 = r_3 \oplus (r_8 \ll 1)$, $r_4 = r_4 \oplus (r_8 \gg 31)$
24: $C[4] = r_4$, $C[5] = r_5$, $C[6] = r_6$
25: $r_4 \leftarrow (T(B_{0,1}), T(B_{0,0}))$ $\{C[0]\}$
26: $r_5 \leftarrow (T(B_{0,3}), T(B_{0,2}))$ $\{C[1]\}$
27: $r_5 = r_5 \oplus (r_9 \ll 23)$, $r_4 = r_4 \oplus (r_8 \ll 23)$, $r_5 = r_5 \oplus (r_8 \gg 9)$
28: $C[1] = r_5$
29: $r_{10} = (r_7 \gg 9)$
30: $r_4 = r_4 \oplus r_{10}$
31: $C[0] = r_4$
32: $r_2 = r_2 \oplus (r_{10} \ll 10)$
33: $C[2] = r_2$
34: $r_3 = r_3 \oplus (r_{10} \gg 22)$
35: $C[3] = r_3$
36: $r_7 = r_7$ & 0x1FF
37: $C[7] = r_7$
38: **return** c

LOA 22: Inversion in $\mathbb{F}_{2^m}$ using no swap operation.

Input: A non-zero polynomial $a(z)$ of degree at most $m-1$
Output: $c(z) = a(z)^{-1} \bmod f(z)$

```
1: u(z) ← a(z), v(z) ← f(z)
2: g1 ← 1, g2 ← 0
3: while true do
4:    j ← deg(u) - deg(v)
5:    if j < 0 then
6:       v ← v + z^{-j} u
7:       g2 ← g2 + z^{-j} g2
8:       if v = 1 then
9:          break
10:      end if
11:   else
12:      u ← u + z^j v
13:      g1 ← g1 + z^j g2
14:      if u = 1 then
15:         break
16:      end if
17:   end if
18: end while
19: return c
```

execution time.

First the López-Dahab algorithm and its derivatives are analyzed. Next, our proposed algorithm, López-Dahab with fixed registers algorithm is analyzed with different values for the window size parameter (w) in order to determine the optimal value for this parameter on the target platform. Next, we analyze some divide-and-conquer approaches in combination with the López-Dahab with fixed registers algorithm.

To quantify how well any given algorithm or method performs, a cycle count estimation or cost is calculated for each method. This is done with a weighted sum of all the instruction counts. Since the memory read and write instructions both take 2 cycles to complete, and all the other instructions take only 1 cycle, we assign a weight of two to all the memory read and memory write instructions and a weight of 1 to all other instructions. The cost can be seen as the theoretical lower limit of the number of cycles which is required to perform the operation on the target platform.

It needs to be mentioned that for the sake of simplicity the AND instruction was not added to the cost analysis. In the analysis of the `LD` algorithm the effect of the AND operation on the cycle count is estimated to be 2.6% and will therefore have a relatively small impact on the total cycle count for this estimation.

The actual execution time will typical be higher than the estimated cost due to having to move values from the high registers to the low registers because the target

platform can only perform computations on the low registers.

López-Dahab Left-to-right comb method with windows of width w

The `LD` method (Algorithm 15) uses a single precomputation table of $2^w n$ words (assuming y is short). To generate the precomputation table we need n reads to obtain the values of y, $2^w n$ writes (assuming y is short), $2^w n$ XOR instructions and $n(2w-2)$ shifts to generate the table. The inner loop of the multiplication requires n reads from $c[l+k]$, n reads from $T[u][l]$, n writes to $c[l+k]$ and n XOR instructions, for each of all $n\lceil W/w \rceil$ loop iterations. The outer loop requires 1 read from $x[k]$ and 1 shift instruction, for each of all $n\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $2n$ reads, $2n$ writes, $4n-3$ shifts and $2n-1$ xor instructions, for each of all $n\lceil W/w \rceil - 1$ loop iterations. This leads to a total of $\lceil W/w \rceil(2n^2+3n) - n$ reads, $\lceil W/w \rceil(n^2+2n) - 2n + 2^w n$ writes, $\lceil W/w \rceil(n^2+2n-1) + 2^w n - 2n + 1$ XOR and $\lceil W/w \rceil(5n-3) + 2wn - 6n + 3$ shift instructions. A summary of the instruction estimates are shown in Table 3.3, 3.4, 3.5 and 3.6.

If $W = 32, w = 4, n = 8$ we obtain 1208 read, 752 write, 745 XOR and 315 shift instructions. The total cost is 4980 cycles and the total memory consumption is 4 kB for the lookup table. A summary of the estimated execution times is shown in Table 3.7.

The AND instruction on line 5 is executed $(W/w)n$ times. For $n = 8, w = 4$ and $W = 32$ this is equal to 64 instructions. One can say that every AND instruction can be performed using two shift operations, therefore the total estimated cost for the AND operation is 128 cycles and account for 2.6% of the total cycle count of the algorithm. The effect of the AND operation is therefore relatively small and will not be included in further analysis.

López-Dahab with rotating registers

The `LD with rotating registers` (Algorithm 16) method uses a single precomputation table of $2^w n$ words (assuming y is short). To perform the analysis we assume that $n+1$ registers are available for use in the rotating register scheme. To generate the precomputation table we need n reads to obtain the values of y, $2^w n$ writes (assuming y is short), $2^w n$ XOR instructions and $n(2w-2)$ shifts to generate the table. The inner loop requires 1 read from $c[k]$, 1 read for $x[k]$, n reads for $T[u][j]$, 1 write to $c[k]$, n xor for $T[u][j]^r eg$, 1 xor for c[k] and 1 shift instruction, for each of all $n\lceil W/w \rceil$ loop iterations. The outer loop requires n reads from $C[n \cdots 2n-1]$, n writes to $C[n \cdots 2n-1]$, n xor on $C[n \cdots 2n-1]$, for each of all $n\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $2n$ reads, $2n$ writes, $4n-3$ shifts and $2n-1$ xor instructions, for each of all $\lceil W/w \rceil - 1$ loop iterations. This leads to a total of $\lceil W/w \rceil(n^2+5n-1) - n$ reads, $\lceil W/w \rceil(4n) - 2n + 2^w n$ writes, $\lceil W/w \rceil(n^2+3n-1) + 2^w n - 2n + 1$ XORs and $\lceil W/w \rceil(5n-3) + 2wn - 6n + 3$ shift instructions. A summary of the instruction estimates are shown in Table 3.3, 3.4, 3.5 and 3.6.

If $W = 32, w = 4, n = 8$ we obtain 816 read, 368 write, 809 XOR and 315 shift instructions. The total cost is 3492 cycles and the memory consumption is 4 kB for the precomputation table. Compared to the standard `LD` we can see a drastic reduction in the number of memory operations due to the implementation of the rotating register scheme which minimizes the storing of intermediate values in memory. We can also see a slight rise in the number of XOR instructions due to the extra XOR instruction on line 10 of algorithm 16. A summary of the estimated execution times is shown in Table 3.7.

López-Dahab with fixed registers

The proposed `LD with fixed registers` (Algorithm 17) method uses a single precomputation table of $2^w n$ words (assuming y is short). To perform the analysis of the `LD with fixed registers` algorithm we need to make some assumptions with regards to the number of available registers. Assume that $(n + 1)$ registers are available and are used to store the most frequently accessed $n + 1$ words of $C[0 \cdots 2n]$ inside the registers. The remaining items are stored inside memory. The precomputation requires n reads to obtain the values of y, $2^w n$ writes (assuming y is short), $2^w n$ XOR instructions and $n(2w - 2)$ shifts to generate the table. The inner loop requires n reads from $T[u][l]$, 1 read from $x[k]$, n xor and 1 shift instruction, for each of all $n\lceil W/w \rceil$ loop iterations. The outer loop requires $n - 1$ reads and writes to and from both $C[0 \cdots 2]$ and $C[12 \cdots 15]$, for each of all $n\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $n - 1$ reads, $n - 1$ writes, $4n - 3$ shifts and $2n - 1$ xor instructions, for each of all $\lceil W/w \rceil - 1$ loop iterations. This leads to a total of $\lceil W/w \rceil (n^2 + 3n - 2) + 1$ reads, $\lceil W/w \rceil (2n - 2) + 2^w n - n + 1$ writes, $\lceil W/w \rceil (n^2 + 2n - 1) + 2^w n - 2n + 1$ XOR and $\lceil W/w \rceil (5n - 3) + 2wn - 6n + 3$ shift instructions.

If $W = 32, w = 4, n = 8$ we obtain 689 read, 233 write, 745 xor and 315 instruction. The total cost is 2904 cycles and the memory consumption is 4 kB for the precomputation table. Compared to the `LD with rotating registers` algorithm we can see an even further reduction in memory accesses with an XOR instruction count which is the same as the `LD` method's.

Table 3.3, 3.4, 3.5 and 3.6 shows a number of models of the instructions required to perform a field multiplication. Table 3.7 shows an estimate of the number of instructions required to perform the multiplication. This assumes that $n = 8$, $W = 32$ and $w = 4$. The cycle count estimate assumes that a memory operation will take 2 cycles and all other operations takes only 1 cycle to complete. From this we can see that the `LD with fixed registers` has a performance increase of 20.24% over the `LD with rotating registers` method and a performance increase of 71.5% over the standard `LD` method.

Optimal window length parameter for `LD with fixed registers`

A trade-off exists between the time spent inside the precomputation generation and the time spent doing the actual multiplication. When increasing the value of w the

Table 3.3: Analysis of the read instructions required to perform different field multiplications.

Algorithm	Number of reads	Count
`LD`	$\lceil W/w \rceil (2n^2 + 3n) - n$	1208
`LD rotating registers`	$\lceil W/w \rceil (n^2 + 5n - 1) - n$	816
`LD fixed registers`	$\lceil W/w \rceil (n^2 + 3n - 2) + 1$	689
`LD fixed registers, 8-bit scanning`	$\lceil W/s \rceil (\lceil s/w \rceil n^2 + 5n - 3) + n(\lceil s/w \rceil - 1) + 1$	725

Table 3.4: Analysis of the write instructions required to perform different field multiplications.

Algorithm	Number of writes	Count
`LD`	$\lceil W/w \rceil (n^2 + 2n) - 2n + 2^w n$	752
`LD rotating registers`	$\lceil W/w \rceil (4n) - 2n + 2^w n$	368
`LD fixed registers`	$\lceil W/w \rceil (2n - 2) + 2^w n - n + 1$	233
`LD fixed registers`, 8-bit scanning	$\lceil W/s \rceil (3n - 3) + n(\lceil s/w \rceil . 2^w - 1) + 1$	333

Table 3.5: Analysis of the XOR instructions required to perform different field multiplications.

Algorithm	Number of XORs	Count
`LD`	$\lceil W/w \rceil (n^2 + 2n - 1) + 2^w n - 2n + 1$	745
`LD rotating registers`	$\lceil W/w \rceil (n^2 + 3n - 1) + 2^w n - 2n + 1$	809
`LD fixed registers`	$\lceil W/w \rceil (n^2 + 2n - 1) + 2^w n - 2n + 1$	745
`LD fixed registers, 8-bit scanning`	$\lceil W/s \rceil (\lceil s/w \rceil n^2 + 2n - 1)$ $+ n(\lceil s/w \rceil . 2^w - 2) + 1$	813

time spent generating the lookup table goes up while the time spent performing the multiplication goes down. The memory requirements for the lookup table also increases as the parameter w increases. The standard López-Dahab with rotating registers requires a precomputation table of size $2^w p$ (if the input polynomial is shorter than $n - w + 2$ bits).

Fig. 3.4 and Table 3.8 show the cycle counts for a field multiplication using the López-Dahab method with fixed registers for different values of the parameter w. We can see that the fastest cycle count of 2904 cycles is obtained for $w = 4$ with a memory consumption of 4 kB. We will therefore use $w = 4$.

TABLE 3.6: Analysis of the shift instructions required to perform different field multiplications.

Algorithm	Number of shifts	Count
`LD`	$\lceil W/w \rceil (5n-3) + 2wn - 6n + 3$	315
`LD rotating registers`	$\lceil W/w \rceil (5n-3) + 2wn - 6n + 3$	315
`LD fixed registers`	$\lceil W/w \rceil (5n-3) + 2wn - 6n + 3$	315
`LD fixed registers, 8-bit scanning`	$\lceil W/s \rceil (6n-3) + n(\lceil s/w \rceil \cdot (2w-2) - 4) + 3$	247

TABLE 3.7: Analysis of the costs associated with different field multiplications on the target platform. This assumes that $n = 8$, $W = 32$ and $w = 4$. The sum column represents the weighted sum of all the instructions with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts.

Algorithm	Read	Write	XOR	Shift	Sum	Memory
`LD`	1208	752	745	315	4980	4 kB
`LD rotating registers`	816	368	809	315	3492	4 kB
`LD fixed registers`	689	233	745	315	2904	4 kB
`LD fixed registers, 8-bit scanning`	725	333	813	247	3176	8 kB

TABLE 3.8: The memory requirements and cycle counts associated with different values of the parameter w for the `López-Dahab with fixed registers` algorithm. The cycle count includes precomputation and multiplication.

w	Memory (kB)	Costs (cycles)
1	0.5	10104
2	1	5112
3	2	3644
4	4	2904
5	8	2988
6	16	3456
7	32	4692
8	64	7464
9	128	13624

`López-Dahab with fixed registers and` s`-bit scanning`

The `LD with fixed registers` algorithm uses two precomputation tables of $2^w n$ words each (assuming y is short). The advantage of this method is the large reduction in outer iterations which includes a decrease in the number of multi-precision shift operations on $c(z)$. The disadvantages of this method are that two lookup tables are

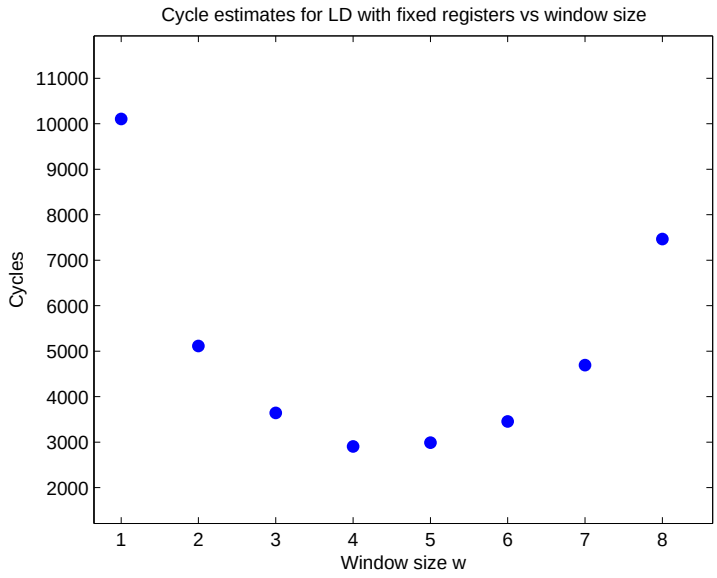


Figure 3.4: Cost estimation for different values of w.

required which leads to increased storage requirements and the inner loop is more complex with lookups from multiple lookup tables. It is assumed that the parameter s is chosen so that s/w will lead to a real number. e.g. If $w = 4$ then s could be 4,8,16,32 etc.

Assume that $n = 8$ and that $n + 1$ of the $2n$ intermediate values ($C[0 \cdots 2n]$) are stored inside registers. We require $\lceil s/w \rceil$ precomputation tables to be generated, each requiring n reads to obtain the values of y, $2^w n$ writes (assuming y is short), $2^w n$ XOR instructions and $n(2w - 2)$ shifts to generate the table. The inner loop requires $\lceil s/w \rceil n$ reads from the lookup tables T, $\lceil s/w \rceil$ reads from $x[k]$, $\lceil s/w \rceil n$ xor and $\lceil s/w \rceil$ shift instructions, for each of all $n\lceil W/s \rceil$ loop iterations. The outer loop requires $\lceil s/w \rceil 2(n-1)$ read and $\lceil s/w \rceil 2(n-1)$ write instructions to and from $C[0 \cdots 2]$) and $C[12 \cdots 15]$, for each of all $\lceil W/s \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $n - 1$ reads, $n - 1$ writes, $4n - 3$ shifts and $2n - 1$ xor instructions, for each of all $\lceil W/s \rceil - 1$ loop iterations. This leads to a total of $\lceil W/s \rceil (\lceil s/w \rceil (n^2 + 3n - 2)) + \lceil W/s \rceil (n-1) + \lceil s/w \rceil n - n + 1$ reads, $\lceil W/s \rceil (3n - 3) + n(\lceil s/w \rceil . 2^w - 1) + 1$ writes, $\lceil s/w \rceil . 2^w n + \lceil W/s \rceil (\lceil s/w \rceil n^2 + 2n - 1) - 2n + 1$ xor and $\lceil W/s \rceil (6n - 3) + 2n(2w - 2) + 3$ shift instructions. A summary of the instruction estimates are shown in Table 3.3, 3.4, 3.5 and 3.6.

If $W = 32, w = 4, n = 8$ we obtain 725 reads, 333 writes, 813 xor and 247 shift operations which lead to a total estimated cost of 3176 cycles and a precomputation table of 8 kB. This method is slightly slower than the `LD with fixed registers` and $w = 4, s = 4$ and it requires twice the amount of memory for storing the precomputation table. A summary of the estimated execution times is shown in Table 3.7.

From the previous four analyses we determined that $w = 4, s = 4$ is indeed a very good choice of parameters.

Divide-and-Conquer methods

The divide-and-conquer methods are analyzed in combination with the `LD with fixed registers` algorithm. The divde and conquer methods all split the input parameters into smaller parts and then perform the multiplications using these smaller parts. This leads to a reduced requirement for storing the intermediate values inside the multiplication algorithm as well as a potentially reduced storage requirement for the precomputation table because the precomputation table now only needs to be computed on a polynomial which is a fraction of the original length.

In order to get an accurate estimation of the number of instructions which is required for the divide-and-conquer approach, a model is developed for four different variants of the divide-and-conquer algorithms. These are:

- Hybrid with López-Dahab ($w = 4$), asymmetrical split (128-bit, 105-bit)
- Hybrid with López-Dahab ($w = 4$), symmetrical split (117-bit, 117-bit)
- Karatsuba with López-Dahab ($w = 4$), symmetrical split (117-bit, 117-bit)
- Operand-scanning with López-Dahab with fixed registers ($w = 4$), asymmetrical split (128-bit, 105-bit)

`Symmetrical Hybrid`

A symmetrical split is performed on the two input parameters x and y to divide them into two equal parts of 117 bits each to fit inside 4 words. The `LD with fixed registers` algorithm with $w = 4$ is used to perform the multiplications. If the input polynomial is split into two equal-sized parts then the number of intermediary results inside the multiplication algorithm is reduced from $2n$ (For the full-sized multiplication) to n. All the intermediates are therefore stored in registers which leads to a reduced number of accesses to the memory.

A pre-shift operation is required in order to prepare the input data for input into to the precomputation and the multiplication functions. A post-shift operation is required to shift the results from the multiplication to the correct bit position for combination into the final result. The pre-shift requires $n/2 + 1$ reads, $n/2$ writes, n XOR and $2n$ shifts instructions. The post-shift requires $2n + 3$ reads, 4n+3 writes, $5n$ XOR and $6n$ shift instructions. Table generation is performed twice to generate two tables, requiring $n/2$ reads, $2^w n/2$ writes (assuming y is short), $2^w n/2$ XOR instructions and $n/2(2w-2)$ shifts, for each of all 2 table generations. The inner loop requires $n/2$ reads from $T[u][l]$, 1 read from $x[k]$, $n/2$ XOR and 1 shift instruction, for each of all $2\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $n - 1$ XOR and $2n - 1$ shift instructions, for each of all $4\lceil W/w-1 \rceil$ loop iterations. This leads to a total number of $(3.5n+4)+\lceil W/w \rceil(n^2+2)$

reads, $(4.5n + 3) + 2^w n$ writes, $2n + 2^w n + \lceil W/w \rceil (n^2 + 4n - 4) + 4$ XOR and $\lceil W/w \rceil (10n - 4) + 4$ shift instructions.

The number of memory accesses associated with the pre and post-shift operations is given by:

$$7n + 7 \tag{3.7}$$

If $W = 32, w = 4, n = 8$ we obtain 672 read, 167 write, 884 XOR and 612 shift instructions. This leads to a total estiated cost of 3174 cycles. The memory instructions have been reduced from the original `López-Dahab with fixed registers` algorithm at the cost of an increased number of XOR instructions. The advantage of this method is that all the precomputation tables do not need to be stored simultaneously inside the memory. This leads to the precomputation tables only requiring a modest 2 kB of memory. A summary of the estimated execution times is shown in Table 3.7.

`Asymmetrical Hybrid`

An asymmetrical split on the two input parameters x and y followed by an `LD with fixed registers` multiplication on the smaller input parameters. The split into two is done by considering the word-size of the processor and therefore both x_0 and y_0 are 128-bits long whereas x_1 and y_1 are both 105-bits long and each fits into 4 words. Two lookup tables needs to be computed; one for y_0 and the other for y_1. When y_0 is shifted by $w - 1$ it cannot fit inside 4 words anymore and another word needs to be assigned to hold the values for the lookup table. Therefore the lookup table of y_0 contains $2^w(n + 1)$ words and the lookup table for y_1 contains only $2^w n$ words. Four multiplications are required with precomputation tables with two different lengths. In total for this method we require the following operations: 2 precomputations, 2 field multiplications with a large lookup table and 2 field multiplications with a smaller lookup table. The biggest disadvantage of this method comes from the larger precomputation table which requires 2^w extra words to be computed during the precomputation phase, stored inside memory, and accessed again during the multiplication.

In order to perform the asymmetrical split the input parameters x and y are both split into two; one part is 128 bits (or four words) long and the other part is 105 bits long. The precomputation has to be executed on both these polynomials but the 128-bit polynomial needs a lookup table of size $(n + 1)2^w$ whereas the shorter polynomial only needs a lookup table of size $n.2^w$. Therefore, two different multiplication algorithms are needed to efficiently access the two precomputation tables of different lengths.

To generate the short precomputation table requires $n/2$ reads, $2^w n/2$ writes, $2^w n/2$ XOR and $n/2(2w-2)$ shift instructions. To generate the long precomputation table requires $n/2+1$ reads, $2^w(n/2+1)$ writes, $2^w(n/2+1)$ XOR and $(n/2+1)(2w-2)$ shift instructions. The multiplication with the short precomputation table requires for the inner loop $n/2$ reads from $T[u][l]$, 1 read for $x[k]$, $n/2$ XOR and 1 shift instruction, for each of all $n\lceil W/w \rceil$ loop iterations; for the logical shift of $c(z)$ computed at the

intermediary stage requires $n-1$ XOR and $2n-1$ shift instructions, for each of all $2\lceil W/w-1\rceil$ loop iterations; and the output of the multiplication requires $2n$ writes.

The multiplication with the long precomputation table requires for the inner loop $n/2+1$ reads from $T[u][l]$, 1 read for $x[k]$, $n/2+1$ XOR and 1 shift instructions, for each of all $n\lceil W/w\rceil$ loop iterations; for the logical shift of $c(z)$ computed at the intermediary stage requires n XOR and $2n-1$ shift instructions, for each of all $\lceil W/w-1\rceil$ loop iterations; at the output of the multiplication $2n$ writes. A summary of the instruction estimates are shown in Table 3.9, 3.10, 3.11 and 3.12.

The recombination of the results from the four multiplications requires $2n$ read and $2n$ XOR instructions. In total this method requires $\lceil W/w\rceil(n^2+3n)+3n+1$ reads, $(n+1)2^w+4n$ writes, $(n+1)2^w+\lceil W/w\rceil(n^2+5n-2)-4n+2$ XOR and $\lceil W/w\rceil(8n)+(n)(2w-10)+2w+2$ shift instructions. If we set $W=32, w=4, n=8$ this method requires 729 reads, 176 writes, 930 XOR and 506 shift instructions. This leads to a total estimated cost of 3246 cycles. A summary of the estimated execution times is shown in Table 3.13.

`Symmetrical Karatsuba with LD`

The Karatsuba-Ofman method (algorithm 3.1) with a symmetrical split on the two input parameters x and y are analyzed here. The input parameters are split into two equal parts of 117 bits which fit into 4 words each. The three multiplications are performed using the `LD with fixed registers` algorithm. To compute the final result requires 1 pre-shift, 3 pre-computations, 3 multiplications, 3 post-shifts, 4 n-word adds, 2 $(n/4)$-word add and 2 n-word shift operations. The results of all three multipilcations need to be kept in memory and then recombined. This means that a total of $3n$ words are required to keep the results from the three multiplications.

The advantage of the Karatsuba-Ofman method is that it requires only three multiplications. The disadvantage of this method is that three precomputation tables are needed, more memory is needed to store the results of the multiplications and recombining the results of the multiplications are also expensive. The precomputation table needs to be calculated three times, once for each unique multiplicand. Each precomputation table will consume only $(n/2)2^w$ words (assuming a short multiplicand). It is not necessary that all three precomputation tables need to be accessed simultaneously and therefore only a single table needs to be kept inside the memory at a given time. Thus, the total memory usage for this algorithm is $3n+(n/2)2^w$ words.

Assume that the `LD with fixed registers` algorithm can store all the intermediate values inside the registers when $l=n/2$. The short precomputation table generation requires $n/2$ read, $2^w n/2$ write, $2^w n/2$ XOR instructions and $n/2(2w-2)$ shift instructions. The $n/2$-word addition requires n read, $n/2$ write and $n/2$ XOR instructions. The n-word addition requires $2n$ read, n write and n XOR instructions. The n-word shift requires n read, n write, $n-1$ XOR and $2n-1$ shift instructions. The pre-shift requires $n/2+1$ read, $n/2$ write, n XOR and $2n$ shift instructions. The post-shift requires $2n+3$ read, 4n+3 write, 6n+1 XOR and $6n-3$ shift instructions. The multiplication requires for the inner loop $n/2$ reads from $T[u][l]$, 1 read for $x[k]$,

$n/2$ XOR and 1 shift instructions, for each of all $(3)(n/2)\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires $n-1$ XOR and $2n-1$ shift instructions, for each of all $(3)\lceil W/w-1 \rceil$ loop iterations. This leads to a total of $16n+4+(n)\lceil W/w \rceil((3/4)n+3/2)$ reads, $(3/2)2^w n+7.5n$ writes, $(3)2^w n/2+10n-3+\lceil W/w \rceil(n^2+4n-4)$ XOR and $3wn+1n-1+\lceil W/w \rceil(10n-4)$ shift instructions. A summary of the instruction estimates are shown in Table 3.9, 3.10, 3.11 and 3.12.

If $W=32, w=4, n=8$ this leads to 660 read, 252 write, 1005 XOR and 711 shift instructions with a cost of 3540 cycles and a memory consumption of 2.75 kB. A summary of the estimated execution times is shown in Table 3.13.

`Asymmetrical Hybrid with Operand Scanning`

The hybrid method with an asymetrical split is combined with the operand scanning method and the `López-Dahab with fixed registers` with the window parameter $w=4$. An asymmetrical split is performed on the input parameters x and y which results in a 128-bit polynomial and a 105-bit polynomial. This means that two precomputation tables of different sizes will need to be generated. The operand-scanning method is used to reduce the number of memory reads for x_0 and x_1. The two multiplications which would both access x_0 are grouped together in the same function and the two multiplications which both access x_1 are grouped together in the same function. The `López-Dahab with fixed registers` method is used but only two intermediate values are stored inside the memory. The two intermediate values are each read and modified only twice per loop. The large precomputation table requires 2,5 kB of memory and the smaller one only 2 kB. These two tables will both need to be kept inside the memory simultaneously which leads to a total memory consumption of 4.5 kB. In total the following operations are required: 1 short (4-word) precomputation, 1 long (5-word) precomputation, 2 operand scanning multiplications. The operand scanning multiplication essentially consists of two multiplications which both use the same multiplicand but different precomputation tables. The biggest disadvantage of this method comes from the larger precomputation table which requires 2^w extra words to be computed during the precomputation phase, stored inside memory, and accessed again during the multiplication.

Assume that $n+1$ registers are available for storing the intermediate values. The `LD with fixed registers` algorithm will then need to store the two least frequently used intermediate values ($C[0]$ and $C[10]$) inside the memory. The short precomputation table generation requires $n/2$ read, $2^w n/2$ write, $2^w n/2$ XOR instructions and $n/2(2w-2)$ shift instructions to generate the table. The long precomputation table generation requires $n/2+1$ read, $2^w(n/2+1)$ write, $2^w(n/2+1)$ XOR instructions and $(n/2+1)(2w-2)$ shift instructions to generate the table. The multiplication requires for the inner loop $n/2$ read from $T1[u][j]$, $n/2+1$ read from $T2[u][j]$, 1 read from $A[k]$, n XOR and 1 shift, for each of all $n\lceil W/w \rceil$ loop iterations; the outer loop requires 1 read from $C[0]$ and 1 write to $C[0]$, for each of all $2\lceil W/w \rceil$ loop iterations. The logical shift of $c(z)$ computed at the intermediary stage requires 2 read from $C[0]$ and $C[10]$ and 2 write to $C[0]$ and $C[10]$, $n+1$ XOR

and $2(n+2)+1$ shift instructions, for each of all $2\lceil W/w-1\rceil$ loop iterations. The write back and recombine step requires $(n-2)$ read, $(2n+4)$ write, $(n-2)$ XOR instructions. A summary of the instruction estimates are shown in Table 3.9, 3.10, 3.11 and 3.12.

This leads to a total of $\lceil W/w\rceil(n^2+2n+6)+2n-5$ read, $2^w(n+1)+\lceil W/w\rceil(6)+2n$ write, $2^w(n+1)+\lceil W/w\rceil(n^2+2n+2)-n-4$ XOR and $n2w+2w+\lceil W/w\rceil(5n+10)-6n-12$ shift instructions. If $W=32, w=4, n=8$ this leads to 699 read, 208 write, 788 XOR and 412 shift instructions and a cost of 3014 cycles. A summary of the estimated execution times is shown in Table 3.13.

TABLE 3.9: Analysis of the read instructions required to perform different field multiplications.

Algorithm	**Number of reads**	**Count**
`Symmetrical Hybrid`	$\lceil W/w\rceil(n^2+2n)+(3.5n+4)$	672
`Asymmetrical Hybrid`	$\lceil W/w\rceil(n^2+3n)+3n+1$	729
`Symmetrical Karatsuba + LD`	$\lceil W/w\rceil(n^2+2)+(16n)+4$	660
`Asymmetric Op. Scanning`	$\lceil W/w\rceil(n^2+2n+6)+2n-5$	699

TABLE 3.10: Analysis of the write instructions required to perform different field multiplications.

Algorithm	**Number of writes**	**Count**
`Symmetrical Hybrid`	$(4.5n+3)+2^w n$	167
`Asymmetrical Hybrid`	$(n+1)2^w+4n$	176
`Symmetrical Karatsuba + LD`	$(3/2)2^w n+7.5n$	252
`Asymmetrical Op. Scanning`	$\lceil W/w\rceil(6)+2^w(n+1)+2n$	208

TABLE 3.11: Analysis of the XOR instructions required to perform different field multiplications.

Algorithm	**Number of XORs**	**Count**
`Symmetrical Hybrid`	$\lceil W/w\rceil(n^2+4n-4)+2^w n+2n+4$	884
`Asymmetrical Hybrid`	$\lceil W/w\rceil(n^2+5n-2)+(n+1)2^w+-4n+2$	930
`Symm. Karatsuba + LD`	$\lceil W/w\rceil(n^2+4n-4)+(3)2^w n/2+10n-3$	1005
`Asymm. Op. Scanning`	$\lceil W/w\rceil(n^2+2n+2)+2^w(n+1)-n-4$	788

Conclusion

Table 3.14 shows a summary of all the different field multiplication approaches which were analysed.

Table 3.12: Analysis of the shift instructions required to perform different field multiplications.

Algorithm	Number of shifts	Count
`Symmetrical Hybrid`	$\lceil W/w \rceil (10n - 4) + 4$	612
`Asymmetrical Hybrid`	$\lceil W/w \rceil (8n) + (n)(2w - 10) + 2w + 2$	506
`Symmetrical Karatsuba + LD`	$\lceil W/w \rceil (10n - 4) + 3wn + 1n - 1$	711
`Asymmetrical Op. Scanning`	$\lceil W/w \rceil (5n + 10) + 2w(n + 1) - 6n - 12$	412

Table 3.13: Analysis of the instruction count required to perform different field multiplications. This assumes that $n = 8, W = 32, w = 4$. The sum column represents the estimated cycles and is calculated with the weighted sum of all the instructions, with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts.

Operation	Read	Write	XOR	Shift	Sum	Memory
`Symmetrical Hybrid`	672	167	884	612	3174	2 kB
`Asymmetrical Hybrid`	729	176	930	506	3246	2.75 kB
`Symm. Karatsuba + LD`	660	252	1005	711	3540	2.75 kB
`Asymm. Op. Scanning`	699	208	788	412	3014	4.5 kB

The standard `LD` method was found to be the slowest algorithm due to very frequent memory accesses. The `LD with rotating registers` is an improvement to the standard `LD` method with a cycle count of 3492 cycles. The proposed method (`LD with fixed registers`) leads to the fastest execution time on the target platform with 2904 cycles and a modest requirement of 4 kB for the lookup table. From this we can see that the `LD with fixed registers` has a performance increase of 20.24% over the `LD with rotating registers` method.

For the divide-and-conquer methods, we stated that we will only consider a split by a factor of two. The `Symmetrical Hybrid` and `Asymmetrical Hybrid` methods both have very similar run-times but the asymmetric method has a smaller code size because the code for the multiplication can be re-used for all four multiplications and the memory requirements for its lookup tables are low because both y_0 and y_1 are short polynomials. The `Symmetrical Karatsuba with LD` method was found the slowest of all the divide-and-conquer methods. This method is often cited in literature but it lead to very poor results in our analysis due to the complexity of recombining the results of the multiplications and having to compute three precomputation tables instead of two. The `Asymmetrical Hybrid with Operand Scanning` method was found to be the fastest divide-and-conquer method. It saves on memory reads by grouping the same multiplicands together and thereby reducing the number of memory accesses inside the main loop. It's only drawback is that it requires one small lookup table and one large lookup table which leads to a total memory consumption of 4.5 kB.

If it is desired to have a small memory footprint then the `Symmetrical Hybrid` method uses the least amount with an estimated cost of 3174 cycles and a memory footprint of 2 kB. If the execution time is the most important factor then the `LD with fixed registers` method is the fastest with an estimated cost of 2904 cycles and a memory footprint of 4 kB. As we would like to make an implementation that will lead to the fastest possible execution time and will therefore continue with the `LD with fixed registers` algorithm.

TABLE 3.14: A summary of the memory and cycle count costs associated with all the different analysed methods for $n = 8, W = 32, w = 4$. The sum column represents the weighted sum of all the instructions with a weight of two assigned to the reads and writes, and a weight of one assigned to the XORs and shifts.

Algorithm	**Sum (cycles)**	**Memory**
LD	4980	4 kB
`LD with rotating registers`	3492	4 kB
`LD with fixed registers`	2968	4 kB
`LD with fixed registers, 8-bit scanning`	3176	8 kB
`Symmetrical Hybrid`	3174	2 kB
`Asymmetrical Hybrid`	3246	2.75 kB
`Symmetrical Karatsuba + LD`	3540	2.75 kB
`Asymmetrical Hybrid with Op. scanning`	3014	4.5 kB

3.3.2 Modular Reduction

Algorithm 19 performs many redundant memory accesses. In each loop iteration it performs 5 read, 4 write, 4 XOR and 4 shift instructions, for each of all 8 loop iterations. Outside the loop it performs 5 read, 4 write, 3 XOR and 3 shift instructions. This leads to a total of 45 read, 36 write, 35 XOR and 35 shift instructions. This leads to a total estimated cost of 232 cycles. We can see from this analysis that there is a large number of redundant memory read and write instructions which are redundant because in total only $2n$ read and n write operations are truly needed to perform the reduction.

The modular reduction algorithm can be interleaved with the field multiplication algorithm. After a field multiplication all the elements of $C[0 \cdots 2n]$ can either be found on the stack or inside the registers. Algorithm 19 is then executed but with the exception of only writing a value to the memory when we know that the value will not be changed. By using this method it's possible to reduce the instruction counts to a total of 16 read, 8 write, 35 XOR and 35 shift instructions. This leads to a total estimated cost of 118 cycles. Using this method saves a total of 96.6% in cycles when compared to the standard fast reduction algorithm.

The modular reduction algorithm can also implemented in an interleaved fashion with the square function. Here a few more read and write operations are required

because we do not have so many free registers due to the required table lookups for the squaring operation.

3.3.3 Squaring

The polynomial squaring operation is always followed by reduction and which leads to a large amount of memory overhead if the two operations are called as two separate functions.

The standard squaring (Algorithm 20) operation followed by a standard fast reduction (Algorithm 19) is analyzed. The squaring operation requires n reads from the input parameter $a[i]$, $2n$ reads for the lookups in table T, $2n$ writes to the result vector c, $3n$ shifts to obtain u_1, u_2 and u_3. For $n = 8$ the squaring operation requires a total of 24 read, 16 write and 24 shift instructions. The reduction operation requires in each loop iteration 4 read, 4 write, 4 XOR and 4 shift instructions, for each of all 8 loop iterations; outside the loop it performs 5 read, 4 write, 3 XOR and 3 shift instructions. The reduction requires in total 37 read, 36 write, 35 XOR and 35 shift instructions. The cost of performing the squaring and reduction operations non-interleaved is 61 read, 52 write, 35 XOR and 59 shift operations. This leads to an estimated cost of 288 cycles.

For the interleaved squaring and reduction algorithm (Algorithm 21) the number of read operations from the input parameter $A[i]$ is counted as follows: Only a single read is required from A when a read from A is performed in two consecutive lines which both access the same element in A. e.g. line 3 and line 4 both access a[1]; otherwise, two read operations are required to access the elements of A (e.g. line 9 and 15 both access $A[5]$ and therefore requires two reads). This algorithm requires a total of 11 reads from the input parameter a, 16 reads for table lookups from T, 8 write, 35 XOR and 59 shift instructions. This leads to a total of 27 read, 8 write, 35 XOR and 59 shift operations and a total estimated cost of 164 cycles, and is a 75.6% improvement in the runtime from the combined (non-interleaved) squaring and reduction operations.

3.3.4 Inversion

The extended Euclidean inversion algorithm (Algorithm 10) has a non-deterministic run-time. Therefore we will only analyse the number of instructions inside the inner loop. Let's assume that the function which determines the degree is implemented and optimized to a degree that it will only require 1 read. One loop iteration will requires 2 reads for the two degree functions, n reads and n writes for each of the two swap functions, $2n$ reads, n writes, n XOR and n shifts for line 8, and line 9 separately. This leads to a total of $6n + 2$ read, $4n$ writes, $2n$ XOR and $2n$ shift for a loop in the worst case scenario where the if-statement is taken.

Algorithm 22 reduces the memory operations by performing some code-duplication in order to avoid doing the two swap operations in Algorithm 10. One loop iteration will then require 2 reads for the two deg functions, $2n$ reads, n writes, n XOR and n shifts for line 6, and line 7 or line 12 and line 13 (depending on which branch was

taken by the if-statement) separately. This leads to a total of $4n + 2$ read, $2n$ writes, $2n$ XOR and $2n$ shift instructions for each loop iteration.

3.4 Optimizations

In the following we discuss all the optimizations which were performed, from the base implementation to the final optimized implementation. First, we will describe the setup which was used to profile the code; second, we discuss the base implementation; third, we will discuss the effect of each of the optimizations which were done.

3.4.1 Profiling

In order to be able to measure the performance of a given algorithm a profiling tool is used to measure the execution time of a routine. The profiling was done by means of the SysTick timer which is a 24-bit timer that runs at the same clock speed as the ARM Cortex-M0+ and counts down from a specified value to zero. To profile a function, the timer is read at the start and the end of a function and the execution time of the function is found by calculating the difference between the two values. The duration spent inside each function is added to a global variable and another global variable is used to record the number of times that the function was called. The execution time for each function is printed to the debugging console after the profiled code has finished executing.

Before starting the optimization process, random test vectors were generated and the same vectors were used throughout the optimization process. The parameter k can therefore be seen as a constant throughout the optimization process. All the profiling information which is reported here is done by performing a single point multiplication with the same k for both the fixed and random point multiplication functions. One problem with this method is that certain functions like the inversion and point multiplication have a variable run-time which depends on the scalar k. To get a more accurate estimate multiple inversion operations should be performed with randomly generated input vectors and the average executed time should be reported.

Another problem with this method is that the profiling code adds a small amount of overhead every time it is executed. No attempt was made to compensate for this effect. The actual runtime is a little bit faster when the profiling is removed.

Yet another problem with this method is that some variation exists in the number of reported cycles. It sometimes happens that if one block of code (called Block A) is followed by another block of code (called Block B) which is being profiled, and the code in Block A is altered without making any alterations to Block B, the number of cycles reported for Block B by the profiler changes. This probably happens due to compiler optimizations which leads to the profiling instructions not being executed in the intended position (in this case immediately before and after block B). This is probably the reason that some functions have fluctuating values even though there were no algorithmic changes made to them (The reduction optimization values in Table 3.15 is a good example of this).

3.4.2 Base Implementation

The base implementation is called version 1.00 and had the following configuration: Point multiplication is done by means of the wTNAF point multiplication method for Koblitz curves (algorithm 3.70 in [15]). The RELIC toolkit was used to calculate the lookup table as well as transform the input scalar k into its width-w TNAF representation. All the underlying field arithmetic and elliptic curve arithmetic was implemented in C. For random point multiplication the TNAF window parameter was set to $w = 4$.

Polynomial squaring is done by means of a 256 entry 16-bit lookup table (Algorithm 20). Field multiplication is done using the López-Dahab left-to-right comb method (Algorithm 15) with a window of $w = 4$. Reduction is done with the Fast reduction method (Algorithm 19). Field additions are done in a separate function and are performed on $2n$ words. Inversion is done with the Extended Euclidean algorithm (Algorithm 10). Mixed point addition is done in LD coordinates. For multiplying both a random point and a fixed point by a scalar, we use Solinas's tau-adic non-adjacent form (TNAF)[14] for Koblitz curves together with the sliding window method for point multiplication. For random point we chose the parameter $w = 4$ which means the precomputation table contains 8 points which are precomputed at the start of each point multiplication. For fixed point multiplication we chose the parameter $w = 6$ which requires 16 precomputed points which are calculated when the platform starts up.

All vectors which contain the polynomials were initially declared to be $2n$ words long and all of the field operations and elliptic curve operations were implemented in their own separate functions (i.e non-interleaved).

3.4.3 Optimisations

The general approach followed during optimization was to first create a base implementation in C, and then profile the code to identify potential bottlenecks. Next, the number of operations inside the routines with the highest cycle counts were reduced as much as possible by reducing the number of elements which are being processed as much as possible. The number of function calls were reduced by interleaving the functions as much as possible. Each time an optimization attempt was made, the code was profiled in order to determine if the optimization was successful. Subsequent, we analysed a number of different field multiplication algorithms in order to identify the fastest one we can find for the target platform. Once we identified the fastest algorithm, we implemented it in C and then implemented it in assembly. Finally, we implemented many of the field arithmetic functions in assembly.

In the following we discuss the effect of all optimizations which were done. Table 3.15 shows the cycle counts of individual functions during the optimization process. Please note that the squaring and multiply includes a field reduction operation. Table 3.16 shows the number of times the individual functions were executed the optimization process. These tables should be used together in order to get a better understanding of what is happening inside the algorithms. A saving of

one cycle in a function that gets called many times will lead to a large total savings.

Profiling info for **version 1.00** revealed that the squaring operation consumed 9236 cycles per operation, or 48% of the total execution time. The literature states that the field multiplication should take approximately 70%-95% of the total execution time, so we concluded that there is clearly a room for optimization in the squaring function. The total runtime for a random point multiplication was found to be 17.6 M cycles.

In version 1.01 an improvement was made to the polynomial squaring because the squaring table was wrongly being allocated on the stack and was populated inside the polynomial squaring function which caused a huge overhead. To fix this problem the lookup table was declared as a global constant with fixed values. This reduced the runtime of the squaring to only 500 cycles (an improvement of 1847%). Field multiplication was reduced from 6843 cycles to 4337 cycles per operation (an improvement of 57.8%) by using shorter vectors for the precomputation table and by also performing less add operations because of the shorter vector size. A large amount of initializations was also removed from many of the function which further improved runtime. The total runtime for a random point multiplication was decreased to 7.72 M cycles, which is an improvement of 127.9% from the previous version.

In version 1.02 improvements were made to the overflow handling in the shift functions. A more efficient shifting method was used which doesn't use any variables. The method used for testing if a point is at infinity was improved by declaring the point at infinity as a global constant and then comparing it with the point being tested. The old method declared a new point and then assigned the values of the point at infinity to this point, whereafter it gets compared the test point. The new method leads to a more efficient test and the global constant allows for a more efficient way to assign a point to infinity. Checking if a point is at infinity is performed inside the Point Addition operation. The precomputation table generation was improved by implementing a shift which operates on n words instead of $2n$. The total runtime for a random point multiplication was decreased to 6.62 M cycles, which is an improvement of 16.6% from the previous version.

In version 1.03 the inverse function was improved by by reducing the number of 'if' statements inside the loop and by improving the variable shift function to allow it to take a different input and output vector. The old variable shift function required the input and output parameters to be the same which lead to many n-word copy operations inside the main loop. A more efficient way to calculate the degree of the field was also made for the inverse function. Two indices are introduced to keep track of the position of the most significant positive word of the polynomials u and v. This means that every time the degree function is called we do not need to check all the elements from the most significant to the least significant element to find the first item which is positive. The indices point to the correct word so now all we need to do is just find the most significant bit inside that word. By performing all these optimizations the inverse function was improved by 232%. The indices are updated at the end of each loop. The total runtime for a random point multiplication was decreased to 6.62 M cycles, which is an improvement of 11.07% from the previous version.

In version 1.04 a squaring function was introduced which requires a single parameter as an input and an output. The old squaring function required the input and output parameters to point to different vectors which lead to a large amount of n-word copy operations inside the point squaring function which is called in the main loop of the point multiplication. The total runtime for a random point multiplication was decreased to 5.17 M cycles, which is an improvement of 15.28% from the previous version.

In version 1.05 all the add and copy operations which operated on vectors of $2n$-words were replaced by add and copy operations which operate on only n words. This leads to a reduction in cycles of 38.9% in the multiplication precalculation, and 12.2% in the inverse. The test to see if a point is at infinity also now only compares n elements instead of the original $2n$ elements. The total runtime for a random point multiplication was decreased to 4.75 M cycles, which is an improvement of 8.84% from the previous version.

In version 1.06 the overall speed of the point multiplication is improved by introducing a global constant which contains the point which is used as input for the point multiplication. Now the input point's asignment isn't included in the runtime measurement for a point multiplication. The pre-calculation for the field multiplication is improved with 71.98% by reducing the number of reads, writes and function calls required to calculate the 16 points of the precomputation table by eliminating all function calls. The function calls to perform shifting and adding on the 16 points were eliminated by introducing a loop which reads in a single element from y and then performs all the required operations inline. The field multiplication was improved by 20.9% by adding a check to only perform the table lookup and addition if $u > 0$, thereby eliminating the accesses to $T[0]$ which only contains zeros. The total runtime for a random point multiplication was decreased to 4.14 M cycles, which is an improvement of 14.7% from the previous version.

In version 1.07 the `LD with rotating registers` (Algorithm 16) is implementated in C. Further analysis is required to determine why the cycle count for the multiplication precomputation has increased. The total runtime for a random point multiplication was decreased to 4.05 M cycles, which is an improvement of 2.22% from the previous version.

In version 1.08 the multiplication routine's main loop was partially unrolled to remove the if-statement before the shift of $c(z)$. The inverse function's calls to the add function was removed and replaced by inlining the add operations inside the loop which lead to a 20.4% improvement. Some optimizations were done inside the multiplication routine to take advantage of the fact that the variable x is only 233 bits long which means that the most significant word is not fully populated. A large amount of cycles can be saved by using a switch statement inside the outer loop to only access the most significant word of x if it contains a value (See below for a further explanation). The multiplication routine's runtime was improved by 10.16%. The total runtime for a random point multiplication was decreased to 3.85 M cycles, which is an improvement of 5.19% from the previous version.

In version 1.09 the polynomial squaring operation is performed in assembly and was actually a little slower than the C version, but it was kept in the source-code

as a test to see if it behaves well inside the project. The reduction is still done in a separate function. The total runtime for a random point multiplication was decreased to 3.71 M cycles, which is an improvement of 3.77% from the previous version.

In version 1.10 the `Symmetrical Hybrid` method is implemented in C. The cycle counts given inside the table is for the complete multiplication process including the overhead of splitting the input parameters, performing four multiplications, recombining the results from the multiplication and for generating two precomputation tables. Some optimizations were done inside the multiplication routine to take advantage of the fact that the variable y is only 117 bits long (See below for a further explanation). A large amount of cycles can be saved by using a switch statement inside the outer loop to only access the most significant word of y if it contains a value*. The runtime of the multiplication routine was improved by 0.5% when compared to the previous version. The total runtime for a random point multiplication was decreased to 3.66 M cycles, which is an improvement of 3.01% from the previous version.

In version 1.11 the `Asymmetrical Hybrid` method is implemented in C and optimizations were also done inside the multiplication routine to take advantage of the fact that the variable y is only 117 bits long. A large amount of cycles can be saved by using a switch statement inside the outer loop to only access the most significant word of y if it contains a value (See below for a further explanation). The cycle counts in Table 3.15 includes the time needed to calculate two precomputation tables and four multiplications. We can see that the precomputation table takes more cycles to be generated because the asymmetrical input polynomial after the split leads to one large $(2^w(n/2+1))$ precomputation table and one smaller $(2^w(n/2))$ precomputation table. The runtime of the multiplication routine was increased by 2.16% when compared to the previous version. The total runtime for a random point multiplication was increased to 3.70 M cycles, which is an increase of 1.08% from the previous version.

In version 1.12 an implementation of the `LD with fixed registers` (Algorithm 17) was done in C. Local variables were used try to simulate the registers of the CPU. We can see that there is a slightly larger cycle count for the multiplication compared to many of the other versions. The goal of this implementation was to provide a reference to compare the intermediate results of the assembly implementation. The total runtime for a random point multiplication was increased to 3.81 M cycles, which is an increase of 2.88% from the previous version.

In version 1.13 the `LD with fixed registers` is implemented in assembly. The reduction algorithm is also interleaved with the multiplication routine in order to take advantage of the results which were still stored inside the registers. This leads to a significant decrease in the number of redundant memory operations. Since the reduction operation and field multiplication is interleaved, the reduction was removed from the profiling info and is noted in in Table 3.16. Some modifications were done to the precomputation table generation which leads to less reads and a faster cycle count which lead to a 42.7% improvement in cycles. The value for the link register (LR) is pushed onto the stack in order to increase the number of free registers to 14. The runtime of the multiplication routine was decreased by

55.47% when compared to the previous version. The total runtime for a random point multiplication was decreased to 3.01 M cycles, which is an improvement of 26.6% from the previous version.

In version 1.14 the squaring is interleaved with the reduction algorithm to reduce the number of redundant memory accesses (Algorithm 21). Note that field reduction is now interleaved with both the multiplication algorithm as well as the squaring function. The cycle count, and count for the reduction field is therefore a zero inside Table 3.15 and 3.16. We didn't bother to try to calculate the execution time for the reduction anymore because it wasn't considered useful to include it in the profiling information. The squaring function's cycle count was improved by 17.4%. The total runtime for a random point multiplication was decreased to 2.97 M cycles, which is an improvement of 1.35% from the previous version.

In version 1.15 the squaring function from version 1.14 is implemented in assembly. We can see from the results that there is only a 5% improvement in the cycle count for writing this routine in assembly. The value for the link register (LR) is pushed onto the stack in order to increase the number of free registers. The total runtime for a random point multiplication was decreased to 2.91 M cycles, which is an improvement of 2.06% from the previous version.

In version 1.16 the precomputation tables are generated inside the same assembly function which performs the multiplication. This leads to a 3.6% improvement in the multiplication function's execution time. The total runtime for a random point multiplication was decreased to 2.81 M cycles, which is an improvement of 3.55% from the previous version.

In version 1.17 we show the cycle count for a fixed-point multiplication. Here a window parameter of $w = 6$ is used and a precomputation table of 16 points are generated just after the microcontroller has started up. This leads to a greatly improved execution time for multiplying a scalar p with the generator G. The total runtime for a fixed point multiplication is 1.96 M cycles, which is an improvement of 43.4% from the random point multiplication timing in version 1.16. We can see in Table 3.16 that the number of point additions and point subtractions differ from that of the random point multiplication. This is due to the different window size parameter of the fixed point multiplication.

In the following discussion the optimizations which performed better or worse than expected is analysed. The `LD with fixed registers` C implementation (version 1.12) performs worse than the `LD with rotating registers` C implementation. This is probably partly due to the C compiler having a difficulty with keeping all the intermediate values inside the registers. There is still some room for optimization of the `LD with fixed registers` C implementation because it still performs a number of initializations.

The assembly implementation of the square function had no performance improvement over the C implementation. However, the assembly implementation of the squaring function interleaved with reduction has a performance improvement of 5% over the C implementation.

In the `LD` method a large amount of cycles can be saved by using a switch statement inside the outer loop to only access the most significant word of x if it

TABLE 3.15: The cycle counts of the different functions during optimization.

Ver.	Total	Mul Pre	Mul	Add	Sqr	Inv	Red	P. Add
1.00[a]	17.6M	6843	16009	165	9236	847k	182	162460
1.01	7.72M	4337	11244	103	500	847k	182	123168
1.02	6.62M	2387	8796	100	500	886k	182	79906
1.03	5.96M	2387	8796	94	500	267k	182	79906
1.04	5.17M	2387	8796	94	510	267k	182	79906
1.05	4.75M	1718	8127	69	510	238k	182	70631
1.06	4.14M	999	6722	60	456	230k	182	59358
1.07[b]	4.05M	1069	6485	61	456	230k	182	57487
1.08	3.85M	1090	5887	59	473	191k	201	52725
1.09	3.71M	1054	5592	57	492	189k	194	50311
1.10[c]	3.66M	800	5563	57	492	189k	194	50103
1.11[d]	3.70M	870	5683	57	492	189k	194	50977
1.12[e]	3.81M	1106	5964	57	492	189k	197	53343
1.13[f]	3.01M	775	3836	57	492	189k	197	36206
1.14[g]	2.97M	775	3835	57	419	139k	0	35877
1.15[h]	2.91M	762	3805	57	399	139k	0	35456
1.16	2.81M	675	3672	68	395	139k	0	34426
1.17[i]	1.96M	675	3672	68	395	139k	0	34426

[a] Base implementation: `LD`
[b] Algorithmic change: `LD with rotating registers`
[c] Algorithmic change: `Symmetrical Hybrid`
[d] Algorithmic change: `Asymmetrical Hybrid`
[e] Algorithmic change: `LD with fixed registers` in C
[f] Algorithmic change: `LD with fixed registers` in assembly
[g] Algorithmic change: Interleaved squaring with reduction in C
[h] Algorithmic change: Interleaved squaring with reduction in
[i] Fixed-point multiplication

contains a value. Assume that x contains a 117-bit value. This means that the input parameter fits inside 3 words and 21 bits. In line 5 of algorithm 15 the parameter u is calculated by taking w bits of parameter x, from the most significant part to the least significant. If $W = 32, w = 4$, then the first two times $x[3]$ is read, the value is zero because $x[3]$ only contains 21 useful bits. The table lookup only needs to be performed when $u > 0$ because $T[0]$ only contains zeros. This optimization is applicable to all the variants of the `LD` method.

Table 3.17 shows the cycle counts of the implemented field multiplication routines, compared with the results from the simulations. The cycle counts for the C and assembly implementations include a field reduction (which account for approximately 180 cycles), whereas the results from the simulation doesn't include a reduction step. The simulated values are generally much faster than the C implementations. However, the assembly implementation of the `LD with fixed registers` has a cycle

Table 3.16: The number of times the different functions were executed during optimization.

Ver.	Mul Pre	Mul	Add	Sqr	Reduce	Point Add	Point Sub
1.00	370	370	4911	921	1291	46	42
1.01	370	370	4911	921	1291	46	42
1.02	370	370	4911	921	1291	46	42
1.03	370	370	4911	921	1291	46	42
1.04	370	370	4911	921	1291	46	42
1.05	370	370	4911	921	1291	46	42
1.06	370	370	841	921	1291	46	42
1.07	370	370	841	921	1291	46	42
1.08	370	370	389	921	1291	46	42
1.09	370	370	389	921	1291	46	42
1.10	740	1480	389	921	1291	46	42
1.11	740	1480	389	921	1291	46	42
1.12	370	370	389	921	1291	46	42
1.13	370	370	389	921	921	46	42
1.14	370	370	389	921	0	46	42
1.15	370	370	389	921	0	46	42
1.16	370	370	389	921	0	46	42
1.17	274	274	290	867	0	34	60

count that is very close to the simulated value. The rows inside the table are sorted according to their simulated speeds. We can see that the C implementations all have roughly the same execution times, except for the `LD with fixed registers` algorithm, which is one of the second slowest C implemented algorithm. This is because there is still some room for optimization in the C implementation of the `LD with fixed registers` algorithm.

Compared to version 1.00, version 1.16 is 526% faster. Compared to version 1.00, version 1.17 is 978% faster.

Table 3.17: Simulation vs Implementation.

Algorithm	Simulation	C	Assembly
`LD`	4980	6722	-
`Asymmetrical Hybrid`	3246	5563	-
`LD with rotating registers`	3238	5592	-
`Symmetrical Hybrid`	3174	5683	-
`LD with fixed registers`	2904	5964	3672

3.4.4 Memory

Table 3.18 gives a summary of all the different items which contribute to the memory consumption of the implementation. The generator point is stored in memory which requires 768 bytes. The reduction polynomial requires 256 bytes. The field multiplication precomputation table requires 4 kB of memory. The polynomial squaring precomputation table requires 256 16-bit entries which totals 4 kB of memory. The TNAF table for random point multiplication requires 8 precomputed points to be stored in memory. This requires a total of 192 words or 6 kB. The TNAF table for fixed point multiplication requires 16 precomputed points to be stored in memory. This requires a total of 384 words or 12 kB. Point Addition uses seven 16-element 32-bit vectors with a total memory usage of 3.5 kB. Point Double uses two 16-element 32-bit vectors with a total memory usage of 1 kB.

TABLE 3.18: Lookup table memory consumption for different algorithms.

Operation	**Memory**
Generator point	0.768 kB
Reduction polynomial	0.256 kB
Field multiplication precomputation	4 kB
Polynomial squaring lookup	4 kB
TNAF random point multiplication table	6 kB
TNAF fixed point multiplication table	12 kB
Point Addition	3.5 kB
Point Double	1 kB
Total during Point Addition	30.5 kB
Total during Point Double	28 kB

3.5 Conclusion

We described the model which was used to select a curve which will be a good match on the architecture of the target platform. The model showed that binary Koblitz curves will lead to a slightly faster implementation with a lower power consumption than prime curves, due to the fact that binary curve arithmetic uses instructions which requires less energy on the target platform.

We discussed a number of different existing field multiplication algorithms, and attempted to improve these algorithms to find an even better matching algorithm for our architecture. In general two different types of algorithms were explored: (1) algorithms based on splitting the input parameters, and (2) algorithms which process the input parameters as a whole. For the split input parameter algorithms we combined the `LD` algorithm with a symmetrical split, an asymmetrical split, an operand-scanning form, and the Karatsuba-Ofman approach. For the whole input parameter processing algorithms we tried to find optimizations to the `LD`, and `LD with rotating register` algorithms. A new field multiplication algorithm was introduced,

called the `López-Dahab with fixed registers` which is an optimization of the `LD` algorithm.

We performed an instruction count analysis of these algorithms, and we found that if it is desired to have a small memory footprint then the `Symmetrical Hybrid` method has the lowest memory usage of 2 kB, and a cycle count of 3174 cycles, which improves upon the `LD with rotating registers` algorithm with 10%. If execution time is the most important factor then the `LD with fixed registers` method is the fastest method with a memory consumption of 4 kB, and an estimated cost of 2904 cycles, which is an improvement of 20.24% over the `LD with rotating registers` algorithm. As we would like to make an implementation that will lead to the fastest possible execution time and will therefore continue with the `LD with fixed registers` algorithm.

The general approach to the optimization process was described, followed by a detailed description of each optimization step, including its impact on performance.

Chapter 4

Results

This chapter is dedicated to comparing the results that were obtained from the proposed implementation with the results from software libraries and literature. First we will describe the system that we designed to measure the power consumption of the target as well as give the results from our energy measurements. Next, we will provide a description of our results, and compare it with the results from implementations found in literature and in software libraries. Next, we will use the cycle estimation models from Chapter 3 to make an estimate of how well the different field multiplication methods perform for different processor word lengths.

4.1 Measurement setup

In order to determine the power efficiency of different instructions as well as cryptographic software implementations, a system was designed to measure the power consumption of the target platform. The Freedom KL25Z is a low-cost development board built for the ARM Cortex-M0+ with a debugging interface, some LEDs, a capacitive slider and an accelerometer. It has a dedicated header which allows the power consumption of the Cortex-M0+ microcontroller to be measured. A sense resistor of 5.6 Ω is inserted between two pins on the header, and a sense amplifier is used to amplify the voltage drop across the resistor by a factor of twenty in order for it to be in the measurable range of the ADC. As temperature influences the value of a resistor, we chose to use a resistor with a low temperature coefficient of $\pm$50ppm/°C to minimize this effect. The amplified voltage is measured by an Arduino board which performs sampling at 21kHz. A low-pass anti-aliasing filter is used to compensate for the aliasing effect. The Arduino board is connected to a PC running a Matlab script which monitors the serial port output from the Arduino board. A wire is connected from the development board to the Arduino board which acts as a sampling trigger to notify the Arduino when to sample. Only when the output of this wire is high the Arduino samples and writes a non-zero value to its serial port.

Two different sets of power measurements were performed:

- The first set measures the power consumption of a number of different instruc-

tions on the target platform. The aim is to determine the energy consumption of each individual instruction in order to make decisions about which instructions and algorithms will lead to a more energy efficient implementation.

- The second set measures the power consumption of different implementations including our own implementation, and also from other cryptographic libraries that we successfully ran on the target platform.

In a typical sampling scenario the test code is executed multiple times. Just before the test code starts to run the trigger wire goes high. The moment after the test code has terminated the trigger wire goes low. A timer is then started which waits 100 ms before executing the next test code.

The instruction measurements are performed by placing 112 copies of the same instruction performed on different registers (to avoid a pipeline stall) in a function which is then executed a 10k times before making the trigger go low and waiting a 100 ms by means of the timer. The reason for repeating the instruction so many times is to be able to take a measurement with the Arduino which is running at the slow sampling frequency of 21kHz.

4.2 Measurement results

All measurements were made at room temperature. A variation in the value of the energy measurements was observed and it is believed that this is due to the variation in the temperature of the room.

The power consumption for a number of different instructions were measured in order to determine how the selection of instructions affect the overall power consumption. Table 4.1 shows the results of these measurements. A variation in energy consumption of up to 22.5% can be observed between different instructions, with the ADD instruction being the most energy hungry, requiring 6.9% more energy than any other instruction measured. This means that not only the number of cycles but also the choice of instruction has an impact on the overall energy efficiency of an algorithm. This is important for the choice of the underlying field because binary fields will require a large amount of bitwise shift (LSL and LSR) and XOR instructions, whereas prime fields require a large amount of MUL and ADD instructions. If we compare the energy consumption of the ADD and the XOR instructions then we can see that the ADD instruction requires 8.2% more energy than the XOR instruction. Inside the binary field arithmetic there are many cases where either both the ADD or the XOR instruction would be able to perform the job. In all of these cases the XOR instruction should be used because it requires less energy than the ADD instruction.

An implementation was done with a software library called the RELIC toolkit, which was compiled with the following configuration: A binary Koblitz curve of order 2^{233} is used. The point multiplication algorithm uses the left-to-right wTNAF method with the window parameter $w = 4$. Point addition is done using mixed LD-affine coordinates and the fast reduction modulo a trinomial. Inversion is performed

TABLE 4.1: The energy used per cycle for different instructions. The clock frequency is 48MHz.

Instruction	Energy [pJ]
LDR	10.98
NOP	11.56
LSR	12.05
MUL	12.14
LSL	12.21
XOR	12.43
STR	12.58
ADD	13.45

with the Extended Euclidean algorithm and squaring is done using the table-based method. The easy-to-understand arithmetic option is selected but this could be replaced with calls to the GMP library. However, we were unable to get the GMP library cross-compiled for the ARM Cortex-M0+ and were therefore unable to test this feature.

The RELIC implementation was measured to have an average power consumption of 600 µW while performing a random point multiplication, and 600.5 µW while performing a fixed point multiplication. A random point multiplication takes 117.1 ms, and therefore consumes 70.26 µJ per operation. A fixed point multiplication takes 115.7 ms, and therefore consumes 69.48 µJ per operation. These results are listed in Table 4.3, and Table 4.2.

Our proposed implementation was measured to have an average power consumption of 577.2 µW for a random point multiplication, and 519.6 µW for a fixed-point multiplication. A random point multiplication takes 59.18 ms, and therefore consumes 34.16 µJ per operation. A fixed point multiplication takes 39.70 ms, and therefore consumes 20.63 µJ per operation. These results are listed in Table 4.3, and Table 4.2.

Fig. 4.1 shows a measurement of the energy consumption of the ADD instruction on the target platform. Fig. 4.2 shows a measurement of the energy consumption of ten fixed point multiplication operations performed using the `LD with fixed registers` on the ARM Cortex-M0+. Ten identical multiplications are performed with identical keys. Notice the large variation of the peaks due to a number of different instructions which are used by the point multiplication routine.

4.3 Comparison with other libraries

This section aims to compare the proposed implementation with implementations found in literature, as well as in software libraries. The focus goes toward software implementations on low power constrained devices which include the ATMega128L, MSP430F1611, ARM7TDMI, PXA271, ARM Cortex-M3 and ARM Cortex-M0+.

Table 4.2: Energy measurements of point multiplications. The clock frequency is 48MHz. The *measured energy* is column indicates the average measured energy, and the *energy* column indicates the consumed energy per operation.

Operation	**Time [ms]**	**Measured Energy [µW]**	**Energy [µJ]**
This work kP	59.18	577.2	34.16
This work kG	39.70	519.6	20.63
RELIC kP	117.1	600	70.26
RELIC kG	115.7	600.5	69.48

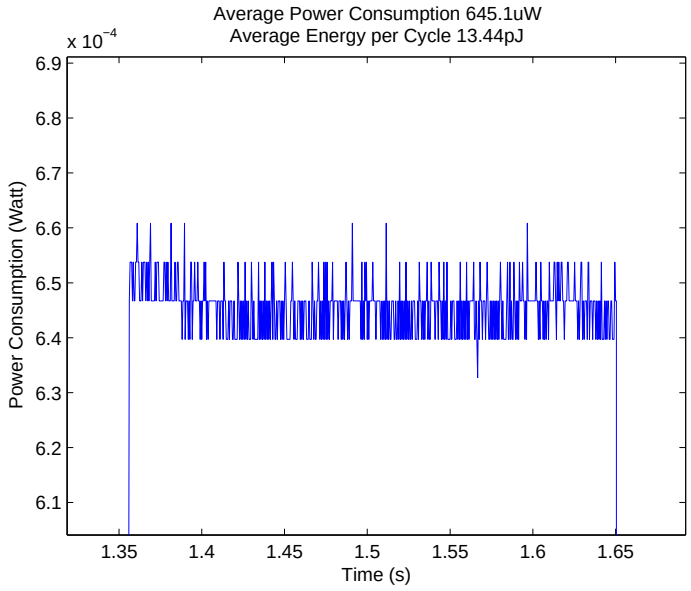


Figure 4.1: Energy consumption of the ADD instruction on the target platform.

First we will give a short overview of several implementations that were found in literature, followed by a number of implementations from software libraries; finally we will present the results of the proposed implementation and make a number of comparisons to the previously mentioned implementations.

4.3.1 Implementations found in literature

This section aims to describe all the state-of-the-art implementations which were found in the literature as well as in software libraries. The implementations are compared based on the execution times and energy consumptions on their respective platforms.

A number of low-power implementations exist in the literature; however, in the past a lot of the focus has gone towards doing software implementations on existing Wireless Sensor Nodes like the MICA2 and MICAz (which both contain the

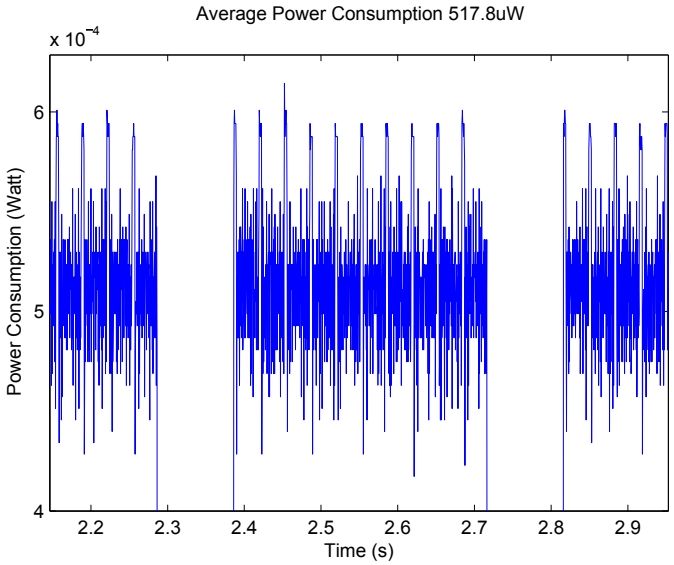


FIGURE 4.2: Energy consumption of a fixed point multiplication on the target platform. Each peak includes 10 identical fixed point multiplications on identical keys.

ATMega128L) and the TelosB (which contains the MSP430). Only a small number of implementations were found in the literature for ARM microcontrollers like the IMote2 (which contains the ARMv5TE based PXA271), ARM7TDMI and the ARM Cortex-M3. As far as we know there are no known implementations for the ARM Cortex-M0+.

Both the ARM7TDMI and the PXA271 are more powerful platforms than the ARMv6-M based ARM Cortex-M0+. The ARMv6-M only supports execution of Thumb instructions and can therefore never enter ARM mode (See section 2.5 for an explanation of ARM mode and Thumb mode) in which shifts instructions can be embedded with data-processing instructions. Both the PXA271 and the ARM7TDMI support instructions in ARM mode. The ARMv6-M also has a much smaller instruction set than both the ARM7TDMI and the PXA271. The PXA271 wireless MMX (wMMX) has has a special set of SIMD instructions which operates on the registers of the processor.

Table 4.3 shows the execution time of point multiplications in milliseconds. Some of the execution times are for fixed-point multiplications and others are for random-point multiplications. The amount of energy consumed to perform the point multiplication is also indicated. In the cases where the amount of energy consumed was not given in the author's results, the amount of energy is estimated by using the typical energy consumption values for the target platform from section 2.5 to make an estimate. Since there is a large variation in energy consumption depending on the type of instruction used, the energy values should be used as a rough guide

only. Table 4.4 shows the execution times for modular multiplication and squaring operations in cycles. The field multiplication and squaring operations are the two most important operations for elliptic curve cryptography.

Table 4.3: Timings for point multiplications. All timings are given in milliseconds and energy is given in millijoules (mJ). The ATMega128L runs at 7.37MHz except when indicated with [a], the MSP430 runs at 8.192MHz, the ARM7TDMI runs at 80MHz, and the ARM Cortex-M0+ runs at 48MHz

Platform	**Author**	**Curve**	**Security**	**Mul [ms]**	**[mJ]**
ARM7TDMI	MIRACL [38]	P-192	96	38 [r]	0.1824 [e]
ARM7TDMI	MIRACL [38]	P-224	112	53 [r]	0.2544 [e]
ATMega128L	Aranha et al. [37]	K-163	80	320 [r]	9.6 [e]
ATMega128L	Aranha et al. [37]	B-163	80	740 [r]	22.2 [e]
ATMega128L	Großschädl et al. [39]	160-bit [p]	80	745 [r]	22.3 [e]
ATMega128L[a]	Kargl et al. [34]	160-bit [p]	80	1261 [r]	41.05 [e]
ATMega128L[a]	Kargl et al. [34]	168-bit [p]	80	1317 [r]	42.88 [e]
ATMega128L[a]	Kargl et al. [34]	167-bit [b]	80	763 [r]	24.84 [e]
ATMega128L[d]	Seo et al. [40]	sect163k1	80	990 [f]	29.7 [e]
ATMega128L	Szczechowiak et al. [41]	P-160	80	1270 [f]	30.02 [c]
ATMega128L	Szczechowiak et al. [41]	K-163	80	2160 [f]	50.93 [c]
ATMega128L	Uhsadel et al. [42]	secp160r1	80	760 [r]	24.84 [e]
ATMega128L	Aranha et al. [37]	K-233	112	730 [r]	21.9 [e]
ATMega128L	Aranha et al. [37]	B-233	112	1890 [r]	56.7 [e]
Cortex-M0+	*This work kG*	*sect233k1*	112	*39.70* [f]	*0.02063* [c]
Cortex-M0+	*This work kP*	*sect233k1*	112	*59.18* [r]	*0.03416* [c]
Cortex-M0+	Relic kP	K-233	112	115.7 [r]	0.06948 [c]
Cortex-M0+	Relic kG	K-233	112	117.1 [f]	0.07026 [c]
MSP430F1611	Szczechowiak et al. [41]	P-160	80	720 [f]	8.847 [c]
MSP430F1611	Szczechowiak et al. [41]	K-163	80	1040 [f]	12.78 [c]

[a] Runs at 8MHz.
[b] A custom binary curve is used.
[c] Energy values obtained by measurement
[d] Assumed to run at 7.37MHz.
[e] Energy values obtained by estimation
[f] Fixed-point multiplication.
[p] A custom prime curve is used.
[r] Random point multiplication.

TABLE 4.4: Timings in cycles for modular multiplication and modular squaring on different platforms.

Author	**Platform**	**Word size (bit)**	**Sqr**	**Mul**	**Field**
S. Erdem [43]	ARM7TDMI	32	348	4359	$\mathbb{F}_{2^{228}}$
S. Erdem [43]	ARM7TDMI	32	389	5398	$\mathbb{F}_{2^{256}}$
S. Erdem [43]	ARM7TDMI	32	430	6551	$\mathbb{F}_{2^{288}}$
Aranha et al. [37]	ATMega128L	8	570	4508	$\mathbb{F}_{2^{163}}$
Kargl et al.[34]	ATMega128L	8	-	2593	$\mathbb{F}_{160}$
Kargl et al.[34]	ATMega128L	8	663	5490	$\mathbb{F}_{2^{167}}$
Uhsadel et al. [42]	ATMega128L	8	-	2881	$\mathbb{F}_{160}$
Aranha et al. [37]	ATMega128L	8	956	8314	$\mathbb{F}_{2^{233}}$
P. Szczechowiak [44]	ATMega128L	8	1581	13557	$\mathbb{F}_{2^{271}}$
Gouvea [35]	MSP430X [a]	16	630	741	$\mathbb{F}_{160}$
Gouvea [35]	MSP430X [a]	16	199	3585	$\mathbb{F}_{2^{163}}$
Gouvea [35]	MSP430X [a]	16	1369	1620	$\mathbb{F}_{256}$
Gouvea [35]	MSP430X [a]	16	325	8166	$\mathbb{F}_{2^{283}}$
This work	*Cortex-M0+*	32	*395*	*3672*	$\mathbb{F}_{2^{233}}$
P. Szczechowiak [44]	PXA271	32	499	4926	$\mathbb{F}_{2^{271}}$
P. Szczechowiak [44]	PXA271	32	1363	10147	$\mathbb{F}_{2^{271}}$
TinyPBC [45]	PXA271	32	187	2025	$\mathbb{F}_{2^{271}}$
TinyPBC [45]	PXA271 wMMX	32	187	1411	$\mathbb{F}_{2^{271}}$

[a] This model has a long 32-bit multiplier

Gouvea et al. [35] made a number of different implementations on prime, binary and binary Koblitz curves for a number of different microcontrollers of the MSP430 family. We only list the results for the version of the MSP430X with the 32-bit multiplier. For the prime field they used Comba [16] multiplication for the 160-bit underlying field and Karatsuba multiplication for the 256-bit underlying field. Montgomery reduction of sparse primes are done because it doesn't require a division which isn't available on the MSP430. For binary fields they use the `LD` multiplication [46] for the 163-bit underlying field and Karatsuba in combination with `LD` for the 283-bit underlying field. The square operation is computed with a lookup table as described in [47]. A fast reduction method is used which is due to the special form of the reduction polynomials [15]. For fixed point multiplication they used the Comb method [48] in the prime case and wTNAF [14] in the binary case. It is interesting to note that they obtained better ECDSA results for their two prime curves than for the Koblitz binary curves.

Szczechowiak et al. [41] made a number of binary field and prime field implementations for the Tmote Sky and the MICA2. Their implementations are based on the MIRACL library with optimizations for different platforms made in assembly. They performed all the field arithmetic for prime curves in assembly and all field arithmetic for binary curves in C. They found that performing binary polynomial multiplication in assembly leads to only a very small improvement in performance

because they do not have a variable-length shift instruction on their target device. Their optimal configuration for prime curves uses Hybrid multiplication and for binary polynomial multiplication they used the Karatsuba-Ofman multiplication algorithm. For binary fields they used binary Koblitz curves as no expensive doubling operations are required. For their fixed point multiplication in prime fields they did some pre-computation with the Comb method and $w = 4$. Their point multiplications in prime fields was found to be faster than in binary.

Kargl et al. made some software implementations on the ATMega128L for prime, binary and optimal extension fields [34]. For multiplication in the binary field they use the `LD` method with $w = 4$. They use a sparse reduction polynomial for fast reduction in binary. They found that the field arithmetic of prime fields are faster than that of binary fields. For prime fields they use the Jacobian coordinate system and for binary fields they use the López-Dahab coordinate system. To secure against a simple timing-based side-channel attacks they use the Montgomery-ladder algorithm which executes faster on binary fields than on prime fields.

Uhsadel et al. made a software implementation on the ATMega128L for the prime secp160r1 curve in [42]. They implemented the hybrid multiplication and found an optimized way to perform carry-bit handling to achieve a fast field multiplication.

Seo et al. was able to make a sub 1-second implementation for the binary Koblitz sect163k1 curve on the ATMega128L on [40]. Their implementation is based upon the TinyECC library which was optimized for Koblitz curves. Their fastest implementation used windowed wTNAF with $w = 5$. They used the left-to-right comb method with window width $w = 4$ for the field multiplication. They adapted the algorithm somewhat to reduce the number of redundant memory accesses. They used fast reduction due to a sparse reduction polynomial. They use the *LD* coordinate system and they store their precomputed points in affine coordinates.

Aranha et al. describes the fastest known point multiplication on the ATMega128L in [37]. For a field multiplication they propose an optimization to the `LD` algorithm called `LD with rotating registers`. Redundant memory accesses are reduced by interleaving the multiplication with the reduction operation. For modular squaring they use a 16-byte lookup table and the reduction is interleaved with the squaring so that the upper half of the words which are produced by squaring doesn't need to be written to memory. Inversion is done with the Extended Euclidean algorithm and they use a fast reduction algorithm due to the sparse reduction polynomial. For multiplication with a random point they used the 4-TNAF method for Koblitz curves and the López-Dahab [49] method for binary curves. This point multiplication method by López and Dahab does not make use of precomputation and has a constant cycle count. For multiplying the generator, they employ the same 4-TNAF method for Koblitz curves with 16 precomputed points. For generic curves they employ the Comb method. These results were put into a software library called the RELIC toolkit [50]. Their implementations were optimized in assembly.

S. Erdem [43] made several implementations on binary fields of different orders for the ARM7TDMI using the operand-scanning method combined with `LD` with a window of width $w = 4$ by López and Dahab. They achieved a polynomial multiplication in 4359 cycles (224 bit), 5398 cycles (256-bit) and 6551 cycles (288-bit).

Reduction was done in a separate step and this they achieved in 348 cycles (224-bit), 389 cycles (256-bit) and 430 cycles (288-bit).

P. Szczechowiak et al. [44, 51] made an implementation on a binary curve which uses an underlying field of order 2^{271}. Field multiplication is done using the `LD` with a window of $w = 8$ and a 2-bit scanning. By doing this they drastically decreased the number of loop iterations and further unrolled all the loops. Shifting $c(z)$ by 8 in the intermediary stage was not necessary because 2 pointers were used that allowed the load and store operations to access the appropriate bytes. This is possible because the architecture supports non-word aligned read and write operations. The authors also exploited the multiple load and store operations of the ARM microcontroller.

Großschädl et al. made a software implementation on the ATMega128L on a 160-bit prime curve in [39]. They used an optimal prime field which leads to even faster reductions than NIST prime fields. Another advantage of their curve is that it is possible to compute efficiently computable endomorphisms which leads to a reduced number of addition and doubling operations.

B. Oliveira et al. [45] made an implementation for the PXA27x with an underlying binary field of order 2^{271}. For multiplication they used the `LD with a rotating registers` and $w = 4$. For squaring they used a 16-byte lookup table which stores the square of all 4-bit polynomials. They also performed immediate reduction of the upper half of the words instead of writing them to memory for later reduction. All their finite field arithmetic was implemented in assembly and since a bitwise shift is free for a data-processing instruction, they performed all bitwise shifts inside the XOR instructions. They achieved modular squaring in 187 cycles, multiplication in 2025 cycles and multiplication using the wireless MMX (wMMX) instructions in 1411 cycles.

4.3.2 Software Libraries

This section will discuss a number of software libraries that can be used for elliptic curve cryptography. These libraries include OpenECC, MIRACL, CompactECC and the RELIC toolkit.

The MIRACL Crypto SDK [52] is an open-source Elliptic Curve Crypto SDK for embedded, mobile and applications. It supports many different platforms including Intel, Atmel, Sun, ARM, IBM, Texas Instruments, MIPS and Analog Devices. It is a C library with some field arithmetic in assembly for many of its supported the platforms. It supports RSA, DSS, Diffie-Helman, ECDSA, ECDH and pairings. It also has a big number arithmetic library. It contains many algorithmic optimizations which can be found in literature and supports many curves including the standard SEC and NIST curves and has support for curve-specific optimizations. Some timings for this library can be found here [38] and are also listed in Table 4.3.

OpenECC [53] is an open-source library with support for the two binary Koblitz curves: K-163 and K-233. It is a C only library that is optimized for the ARM Cortex-M3. Some timings for the ECDSA, ECQV and ECMQV protocols can be found on their website. Unfortunately this library only has timings for ECDSA.

CompactECC is a commercial ECC library by Ubisys which supports many of the

SEC curves. It has support for ARM and is a C-only library with no optimizations in assembly. It supports the ECDH and ECDSA protocols and some of its timings are listed in Table 4.3.

The RELIC toolkit [50] is an open-source cryptographic library. It currently forms the basis for both TinyPBC as well as the current speed record holder for the ATMega128L (Aranha et al.) It supports many different architectures including ARM, ATMega128L, MSP430, x86 and x86_64. It has support for elliptic curves and pairings over binary fields, prime fields, and further has support for hyperelliptic curves. Unfortunately this library has no published timings, but we were able to get it compiled for the ARM Cortex-M0+ and we made some timing and energy measurements for point multiplication which are shown in Table 4.3.

4.3.3 Proposed implementation

The execution times for the C implementation were obtained by compiling with the "-O2" flag because it gave faster results than compiling with "-O3". It is interesting to note that the C version of the `LD with rotating registers` performs better that `LD with fixed registers`. The assembly implementation of the `LD with fixed registers` algorithm performs 62.4% faster than the C implementation.

The execution time for the inversion routine can be found by performing the inversion routine on 20 randomly generated polynomials and taking the average. The execution time for the random point multiplication kP were found by taking the average of 20 operations where a randomly generated scalar k is multiplied with a randomly generated point. The execution time for the fixed point multiplication kG were found by obtaining the average of 20 operations where the randomly generated scalar k is multiplied with the base point G. Table 4.5 shows the execution times of some field arithmetic functions and point multiplications.

Table 4.5: Timings in cycles for field arithmetic algorithms in $\mathbb{F}_{2^{233}}$.

Operation	C language	Assembly
Modular Squaring	419	395
Inversion	141 916	-
`LD with rotating registers`	5 592	-
`LD with fixed registers`	5 964	3 672
kP	3 516 295	2 761 640
kG	2 494 757	1 864 470

The total accumulated timings for kP and kG were found by taking a randomly generated 233-bit scalar and multiplying it with the base point G. Fig. 4.3 and 4.4, and Table 4.6 shows a breakdown of the running times for both a random and a fixed point multiplication. The *TNAF representation* operation is the process of converting an input scalar into its TNAF representation. The square operation includes both the field squaring and point squaring operations, followed by the reduction operation. The inverse operation is only called once when the LD coordinates are converted into

affine coordinates. The *Support functions* term is used to describe all the functions which were not profiled in the code which includes: testing if a point is at infinity, obtaining a copy of the point at infinity, checking if a field contains only zeros, copying the values from one field to another, initialization of a field and adding two polynomials.

There are two important differences between the processes inside the fixed-point multiplication and random point multiplication: (1) The TNAF based fixed-point multiplication's precomputation table is generated from the base point G just after the microcontroller has started up. We can see from Fig. 4.3 that 14.2% of the total cycle count is used to perform the precomputation table generation for $w = 4$. The timing for $w = 6$ would be substantially more. (2) The window size parameter for the TNAF table is set to $w = 6$ for kG and $w = 4$ for kP. The larger window size for kG will lead to a TNAF representation which contains less elements because each element now refers to a table lookup which is 4 bits long instead of the 3 bits value for kP. This leads to a lower number of multiplication operations which explains the lower number of accumulated cycles for the multiply and multiply precomputation operations. The data inside Table 4.6 can be used to determine the percentage of execution time performed by the RELIC toolkit. The RELIC toolkit is used to perform all the TNAF transformations; for kP this accounts for 20.5% of the execution time and for kG it accounts for 10% of the execution time.

TABLE 4.6: Total accumulated timings per operation for random and fixed-point multiplication.

Operation	***kP***	***kG***
TNAF Representation	178 135	185 926
TNAF Precomputation	398 387	0
Multiply	1 108 890	821 178
Multiply Precomputation	249 750	184 950
Square	362 379	342 294
Inversion	139 936	139 656
Support functions	377 350	376 392
Total	2 814 827	1 864 470

The most energy-efficient 112-bit security point multiplication is our proposed implementation with a random point multiplication taking 57.54 µJ, and a fixed point multiplication taking 38.84 µJ. This result beats all the other known implementations on other platforms by a factor of 7.4. The most energy efficient 80-bit security random point multiplication is by the MIRACL library on the P-192 curve with 851mJ.

The fastest point multiplication for 112-bit security is the MIRACL library with a random point multiplication in 53 ms on the P-224 curve. The fastest point multiplication for 80-bit security is the implementation from the MIRACL library on the P-192 curve for the ARM7TDMI with 38 ms.

The implementation that requires the lowest number of cycles to perform both

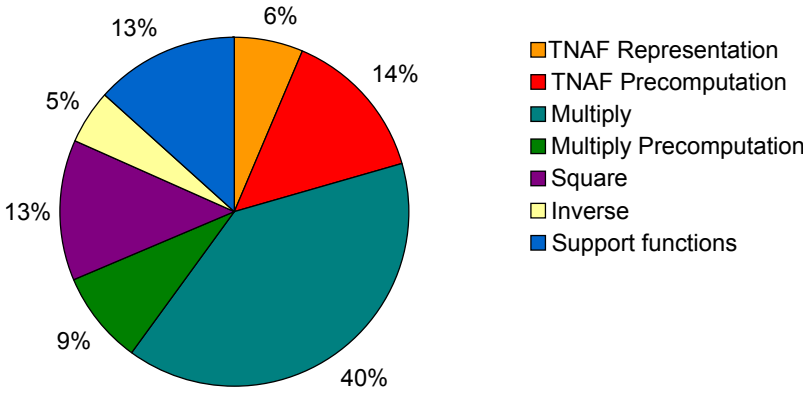


Figure 4.3: A breakdown of the execution time for a random point multiplication.

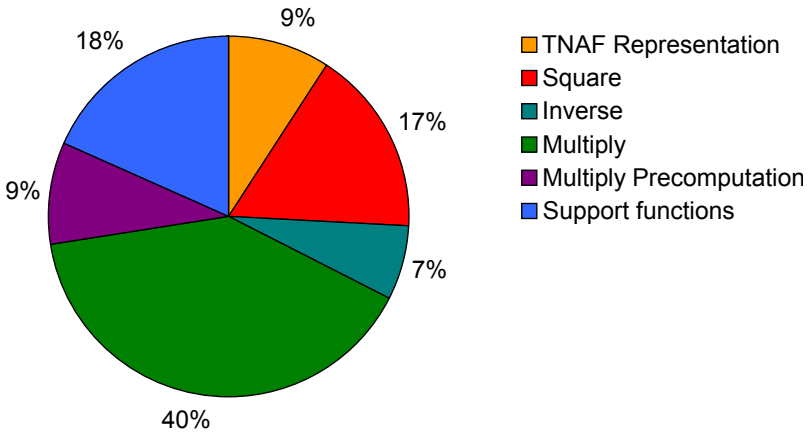


Figure 4.4: A breakdown of the execution time for a fixed point multiplication.

a modular squaring and a field multiplication operation for 128-bit security was by TinyPBC on the PXA271. In this implementation they made heavy use of the target's ability to embed the shift instructions inside data-processing instructions, which is not possible on the ARM Cortex-M0+. The implementation that requires the lowest number of cycles to perform a modular multiplication for 112-bit security is our proposed implementation with 3672 cycles. The implementation that requires the lowest number of cycles to perform a modular multiplication for 80-bit security is by Kargl et al. with 2593 cycles.

4.4 Analysis of the effect of different processor word sizes

So far we have analysed a number of algorithms for field arithmetic on a specific platform (the ARM Cortex-M0+) and it might be beneficial to investigate how well some of these algorithms perform on a different architecture. Since field multiplication

is the most expensive algorithm with regards to execution time in an elliptic curve cryptographic system, we will focus our analysis exclusively on field multiplication. We will evaluate how well the field multiplication methods from Chapter 3 perform on processors with different word sizes. We hope that this will ease the process of algorithm selection for processors with different word size.

During the analysis the following assumptions were made:

- The target architecture doesn't allow shift operations to be embedded with data-processing instructions.
- A memory operation is twice as expensive as any other operation
- The underlying field is $\mathbb{F}_{2^{233}}$.
- The divide-and-conquer methods split the input parameters in two.
- $n + 1$ registers are available for the rotating register scheme in the `LD with rotating registers` method
- $n + 1$ registers are used in the fixed register scheme in the `LD with fixed registers` method
- $n + 1$ registers are used for the fixed register scheme in the `LD with fixed registers` with $s = 8$ method.
- All the intermediate computed values can be stored inside the available registers for the divide-and-conquer methods.
- The Operand scanning method stores only $C[0]$ and $C[10]$ inside memory and all other intermediate values are stored inside the registers.

First we will determine the optimal value for the window size parameter w in relation to word size W for each method in order to achieve the fastest possible run-time for each multiplication method. The word lengths which are analysed are 8, 16, 32, 64 and 128-bit. The number of words required to store the underlying field is found with $n = \lceil m/W \rceil$ which leads to a different value of n for each word size. Next, we will use the optimal values for w to make an estimate of the cycle counts for the different word lengths. The cycle count estimates can then be used to determine which method will provide the fastest execution time for a given word length.

4.4.1 Optimal values for window parameter w

The optimal values for the window parameter w are found by cycling through a number of different values for w and performing an estimate on the number of cycles to perform a field multiplication for each word length. Fig. 4.5 shows the figures which were used in determining the optimal values for the window parameter w for the field multiplication algorithms on $\mathbb{F}_{2^{233}}$. Each figure represents a different

method for which the optimal parameter of w is being determined. Each curve on each figure represents a different word size. We found that there is a variation in the optimal value for w for the different methods. Table 4.7 shows the optimal values for the window size parameter w for each algorithm and word size.

4.4.2 Analysis

The relative performance of the multiplication methods developed in Chapter 3 are analysed for different processor word sizes. For each processor word size, we will use the optimal values of window size parameter w as determined above. Fig. 4.6 shows the results of these estimations.

Asymmetric Methods

The methods which make use of an asymmetrical split are all quite effective for a small word size, but their efficiency relative to the symmetrical methods reduces as the word size is increased. The reason for this performance decrease is due to the overhead associated with the larger precomputation table for the most significant polynomial. Consider equations 3.3 and 3.5; when the word size W increases, the parameter n decreases; as parameter n is decreased, the number of memory accesses relative to the symmetrical split method will also increase. Table 4.8 shows the number of memory accesses to the lookup tables for both the asymmetrical, and symmetrical methods for $\mathbb{F}_{2^{233}}$. This table clearly shows that as the word size W increases, the overhead due to memory accesses to the lookup table for the asymmetric split increases. For the asymmetric methods the number of instructions for read, write and XOR decreases as W increase but the number of shift instructions increase with W. This effect is especially noticeable for the Asymmetric Operand-scanning with LD method where the shift instructions increase at a much higher rate than for the `Asymmetrical Hybrid` method.

Symmetric Methods

The performance of the methods that make use of a symmetrical split increase (relative to the asymmetrical methods) as the processors' word size is increased. Eq. 3.7 shows that the overhead due to the pre and post-shifting operations reduces as the number of words is decreased, while the number of read, write, shift and XOR instructions decrease as W increases.

The `Symmetrical Hybrid` method performs well above average for most word sizes and it is second best for $W = 128$ and $W = 64$, and third best for W=32.

The `Symmetrical Karatsuba with LD` method doesn't perform very well for a small word size, but it shines for a large word size. The reduced value of n leads to a lowered impact from the overhead encountered due to the third lookup table as well as the overhead due to the pre and post-processing. This method performs below average when $W \leq 32$, average for $W = 64$ and above average for $W = 128$. Even though it performs better for a larger word size, the `Symmetrical Hybrid` method always outperforms it.

Table 4.9 shows the number of memory operations on the lookup table for the Symmetrical Hybrid and the Symmetrical Karatsuba methods. Recall that the Karatsuba method only requires three multiplications at the cost of having to compute three lookup tables. We can see that the Karatsuba method always requires between 9% and 10% less accesses to the lookup table than the Symmetrical Hybrid method.

4.10 shows the overhead which is associated with the Asymmetrical and Symmetrical Hybrid methods for different word sizes. The Asymmetrical overhead field is calculated with the difference between the asymmetrical table accesses and the symmetrical table accesses. The Symmetrical overhead field is calculated with Eq. 3.7

Methods with no splitting

The `LD` method has the worst performance of all the methods for $W \leq 64$ but performs better than the `Asymmetrical Hybrid with Operand Scanning` and `Asymmetrical Hybrid` methods for $W = 128$. This method is crippled by a large number of memory accesses inside the inner loop because all its intermediate results are stored inside memory.

The `LD with rotating registers` method provides a decent run-time for $W = 8$ but its relative performance decreases as the word size is increased. The problem with this method is that the number of memory accesses for the intermediate values are directly related to the number of outer loop iterations $\lceil W/w \rceil$ inside the multiplication. For a small word size the number of loops and memory accesses are relatively low, but both increase with the word size.

The `LD with fixed registers` and 8-bit scanning provides a below average performance for $W = [8, 16, 64]$ and provides an above average performance for $W = [32, 128]$. The biggest drawback of this method comes from having to generate two precomputation tables which depend on the term $\lceil s/w \rceil n \cdot 2^w$. By increasing the word length, n is decreased and the amount of words used to store the table decreases and therefore the number of cycles required to generate and access the table also decreases. This is balanced by the fact that the number of loop iterations $\lceil W/s \rceil$ inside the multiplication increase with the word size.

The `LD with fixed registers` method achieved the best performance for all word sizes. It uses the least amount of memory accesses to perform a multiplication and it shows a remarkably low cycle count for all word sizes.

Table 4.11 shows the number memory operations required by different methods to access the intermediate values stored inside the memory. The `LD` method has a large amount of memory accesses due to storing all the intermediate values inside memory. The `LD with rotating registers` method reduces the number of memory accesses when compared to the `LD` method, by storing the intermediate results inside a rotating register scheme. The `LD with fixed registers` has the least amount of memory accesses of all the non-splitting methods due to an effective register usage. The model for the `Asymmetrical Operand Scanning` method assumes that 2 of the intermediate values are stored inside memory, which is based on the assumption that $n+1$ registers are available for intermediate storage. The number of accesses increase

when the word size is increased due to the $\lceil W/w \rceil$ term. Table 4.11 also shows that the number of memory accesses to intermediate values for the `Symmetrical Hybrid`, `Asymmetrical Hybrid` and `Symmetrical Karatsuba` methods are always zero because all the intermediate values are stored inside registers.

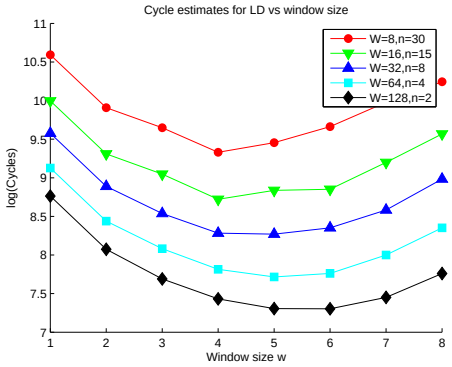
The reader is reminded that the model for the different methods were constructed under the assumption that $n + 1$ registers are available to store intermediate results. This assumption is valid for $W = 8$ [37], and $W = 32$, but will quite likely be an invalid assumption for processors with $W \geq 64$. As W increases, the number of words required to store a large number decreases, and therefore the number of words required to store the large number also decreases. For a processor with a word size of $W = 64$, only 4 registers are required to store a 233-bit number; if the processor has a total of 13 working registers, then it might be possible to store all the intermediate values of the field multiplication inside the processors' registers.

Fig. 4.5 shows that there is an improvement of approximately 50% in the number of cycles when doubling the word size. The reason for this might be due to the following: upon doubling the word length, the number of iterations in the inner loop of the `LD` based algorithms is effectively doubled; this is weighed to a smaller amount of overhead due to precomputation and splitting of the input parameters, as well as fewer table lookups.

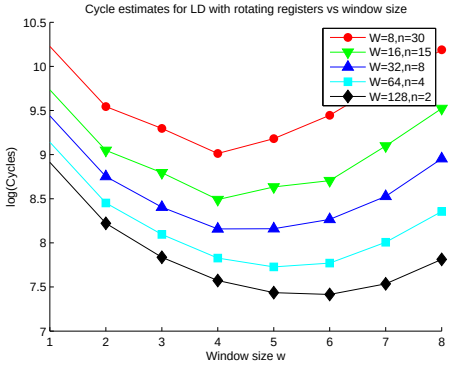
4.5 Conclusion

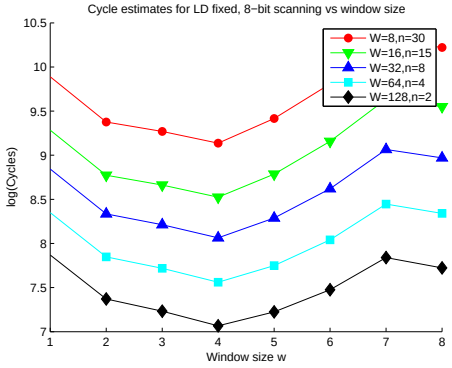
We introduced a system to make energy consumption measurements of the ARM Cortex-M0+ on the KL25Z development board. Results from our measurements shows that there is a large variation (up to 22.5%) in the energy consumption of different types of instructions. We compared the energy consumption of our implementation and found that our implementations' energy consumption beats all the other known implementations on other platforms by a factor of 7.4. Our timings for field multiplication, field squaring and point multiplication also compare favourably to that of other implementations.

We made an analysis of the effect of different word lengths on the field multiplication methods discussed in Chapter 3, and we determined that `LD with fixed registers` will provide the best performance on all word sizes for $\mathbb{F}_{2^{233}}$. If it is desired to reduce memory consumption for the precomputation tables then the `Asymmetrical Hybrid` method is recommended for $W \leq 16$, and the `Symmetrical Hybrid` method for $W \geq 32$. In a multi-processing architecture with 2, 4, or more available cores the `Symmetrical Hybrid` and `Asymmetrical Hybrid` methods will provide the best performance; however, if only three processor cores are available, then the `Symmetrical Karatsuba with LD` method should be considered because each of the three multiplication operations can be assigned to a different core, which could lead to a faster runtime than performing four multiplications (as would be the case with the other splitting methods) in groups of two.

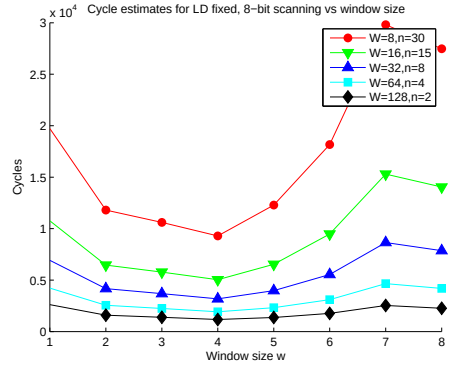


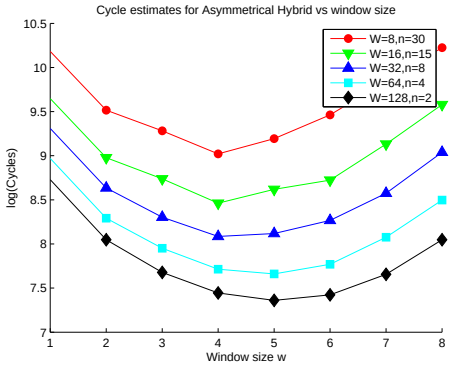
[b]0.49 [b]0.49





[b]0.49 [b]0.49





[b]0.49 [b]0.49

Table 4.7: Optimal values for w on $\mathbb{F}_{2^{233}}$.

Method	Processor word size 8-bit	16-bit	32-bit	64-bit	128-bit
LD	4	4	5	5	6
LD with rotating registers	4	4	4	5	6
LD with fixed registers	4	4	4	5	5
LD with fixed registers, s=8	4	4	4	5	5
Symmetrical Hybrid	4	4	4	5	5
Asymmetrical Hybrid	4	4	4	5	5
Symmetrical Karatsuba + LD	4	4	4	4	5
Asymmetrical Op. Scanning	4	4	5	5	6

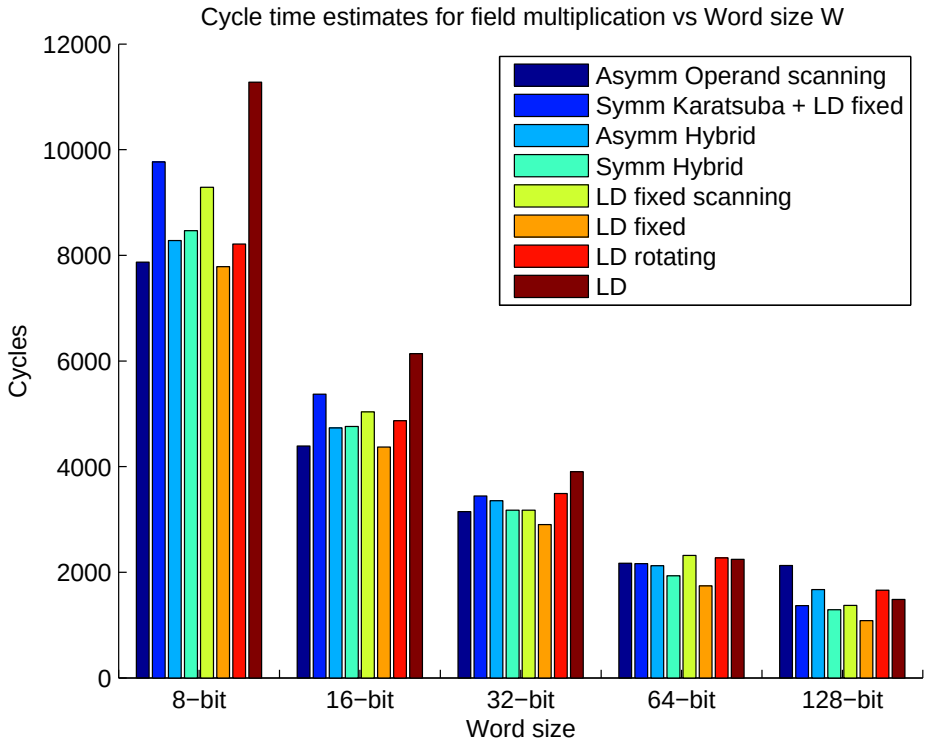


Figure 4.6: Point multiplications on different word sizes for $\mathbb{F}_{2^{233}}$.

TABLE 4.8: Memory operations on the lookup tables required for the Asymmetrical Hybrid, Symmetrical Hybrid and Symmetrical Karatsuba methods by using a factor two split in $\mathbb{F}_{2^{233}}$.

		Symmetric Hybrid		**Asymmetric Hybrid**		
W	***n***	**Read**	**Write**	**Read**	**Write**	**Difference**
8	30	1800	480	1860	496	3.23%
16	15	900	240	960	256	6.25%
32	8	512	128	576	144	11.11%
64	4	256	64	320	80	20.00%
128	2	128	32	192	48	33.33%

TABLE 4.9: Memory operations on the lookup tables required for the Symmetrical Hybrid vs Symmetrical Karatsuba methods by using a factor two split in $\mathbb{F}_{2^{233}}$.

		Symmetric Hybrid			**Symmetric Karatsuba**			
W	***n***	**Read**	**Write**	**Sum**	**Read**	**Write**	**Sum**	**Difference**
8	30	1800	480	2280	1350	720	2070	9.21%
16	15	900	240	1140	675	360	1035	9.21%
32	8	512	128	640	384	192	576	10%
64	4	256	64	320	192	96	288	10%
128	2	128	32	160	96	48	144	10%

TABLE 4.10: Memory overhead for Symmetrical and Asymmetrical methods in $\mathbb{F}_{2^{233}}$. Symmetrical overhead is the cost of performing the pre and post shift operations (Eq. 3.7). Asymmetrical overhead is calculated from the additional memory accesses due to the larger lookup table.

W	***n***	**Symmetric overhead**	**Asymmetric overhead**	**Difference**
8	30	217	256	15.23%
16	15	112	136	17.65%
32	8	63	80	21.25%
64	4	35	48	27.08%
128	2	21	32	34.38%

Table 4.11: Memory operations to the intermediate values stored inside memory in $\mathbb{F}_{2^{233}}$. The `Symm/Asymm` column is used to indicate that both the `Asymmetrical Hybrid`, and the `Symmetrical Hybrid` methods do not use any intermediate values stored in memory.

W	n	`LD`	`LD rotating`	`LD fixed`	`Symm/Asymm`	`Operand Scanning`
8	30	3660	360	174	0	16
16	15	1890	420	196	0	40
32	8	1136	480	210	0	88
64	4	632	496	186	0	184
128	2	380	504	126	0	376

Chapter 5

General Conclusion

In this dissertation we were faced with the task of implementing a PKC algorithm on the ARM Cortex-M0+, which is an ultra low-power microcontroller which has been on the market for less than a year. We chose to make an ECC implementation, and we performed a careful selection of parameters in order to make an implementation which has a good match with the architecture of our platform. We designed a system to perform energy measurements of the ARM Cortex-M0+ which we used to measure our implementations' energy consumption, as well as the energy consumption of different instructions on the processor. We found that there is a variation of up to 22.5% in the energy consumption of different instructions. We decided to use a curve which has security equivalent to a 112-bit symmetric encryption algorithm. In order to determine which curve will match our architecture the best, we created two models (one for binary curves and one for prime curves) to perform a simulation of a point multiplication on each curve. We selected binary Koblitz curves in $\mathbb{F}_{2^{233}}$ because we estimated that these curves will lead to a faster implementation due to replacing the point doubling operations with squaring using the wTNAF method, and they also do not require an `ADD` instruction which requires a minimum of 6.9% more energy than any other instruction on the ARM Cortex-M0+. The target platform only has a short multiply instruction which would be used frequently for a prime curve implementation, and would have caused a lot of overhead.

We discussed a number of different existing field multiplication algorithms, and attempted to improve these algorithms to find an even better matching algorithm for our architecture. In general two different types of algorithms were explored: (1) algorithms based on splitting the input parameters, and (2) algorithms which process the input parameters as a whole. For the split input parameter algorithms we combined the `LD` algorithm with a symmetrical split, an asymmetrical split, an operand-scanning form, and the Karatsuba-Ofman approach. For the whole input parameter processing algorithms we tried to find optimizations for the `LD`, and `LD with rotating register` algorithms. A new field multiplication algorithm was introduced called the `López-Dahab with fixed registers`, which is an optimization of the `LD` algorithm. This algorithm leads to a performance improvement of 20.24% over the `LD with rotating registers` algorithm. We created instruction count

models for all the different methods in order to select the fastest algorithm.

We made an implementation with our `LD with fixed registers` field multiplication algorithm that uses the wTNAF point multiplication method, with $w = 4$ for random point multiplication, and $w = 6$ for fixed point multiplication. The RELIC toolkit was used to perform the TNAF precomputation, and TNAF transformation of the scalar k. All the elliptic curve arithmetic, and field arithmetic was implemented in C and assembly. On average our implementation of the random point multiplication requires 2814827 cycles, and 36.6 µJ, whereas our fixed point multiplication requires 1864470 cycles, and 24.6 µJ. The RELIC toolkit was used to make an implementation on the ARM Cortex-M0+, which requires an average of 5621045 cycles and 72.5 µJ for a random point multiplication, while requiring 5553828 cycles, and 71.6 µJ for a fixed point multiplication. Therefore our implementation has a cycle count that is 1.99 times faster than the RELIC implementation for random point multiplication, and has a cycle count which is 2.98 times faster than the RELIC implementation for fixed point multiplication.

We used the instruction count models for all the algorithms to make a comparison of the different algorithms when the word size is changed. We found that the `LD with rotating registers` algorithm will always perform better than the other algorithms, regardless of the word size.

5.1 Future work

This section will be used to discuss some work that could still be done for this project.

The current implementation uses the RELIC toolkit to compute the wTNAF precomputation table, as well as to perform the wTNAF transformation of the scalar k before performing a point multiplication. If this routine is implemented in C or assembly this will lead to an improvement in the overall execution time and memory requirements of the implementation. The code size and memory requirements will be reduced because it will not be necessary to store all the RELIC routines, as well as some if its precomputation tables. The execution time will be improved of the current implementation because (1) The point coordinates and scalar k will not need to be converted into a format that RELIC can recognize, and (2) The RELIC library is written in a generic way to support many different curves and platforms. Therefore, it is quite likely that a platform-specific, and curve-specific implementation will be faster.

The `LD with fixed registers` algorithm can be implemented on another architecture (e.g. the ARM7TDMI or the ATMega128L) in order to be able to make a comparison of the cycle counts with other implementations found in literature. Before starting an implementation on another platform, the instruction set should be examined, and afterwards an energy measurement should be performed on the individual instructions which could lead to more insight into writing energy efficient algorithms for the target platform.

Since we only focused on the single-point multiplication algorithm, there might be potential to apply the same methodology followed in the `LD with fixed registers`

algorithm for the simultaneous point multiplication routine used by the ECDSA signature verification.

The RELIC toolkit can be used in conjunction with the GMP library, which allows for even more efficient calculations. It is possible to compile the GMP library for the ARM Cortex-M0+, followed by recompiling the RELIC toolkit with the option to use the GMP library, which should lead to faster execution times.

Implementing a constant-time point multiplication algorithm based on the `LD with fixed registers` method would lead to an implementation which is resistant to a simple power-consumption based side-channel attack.

More Comparisons with other libraries can be made for the ARM Cortex-M0+. It should be possible to make an implementation on the target platform using the MIRACL library, as well as the OpenECC library.

All the formulas for the instruction counts in Chapter 3 can be altered to include a parameter for the amount of available registers on the target processor. Currently, some of the formulas assume that $n + 1$ registers are available, which might be an invalid assumption for processors with $W \geq 64$. If all the formulas were developed with a parameter for the register count, this would further have allowed us to make an estimate of the performance of these algorithms for different security parameters.

Bibliography

[1] M. Dunnett, "Freescale ARM Cortex-M0+ Core - Technical Introduction." Accessed: 2013-06-06, 2012.

[2] ECRYPT, II, "Yearly Report on Algorithms and Keysizes (2011-2012), D.SPA.20 Rev. 1.0," tech. rep., ICT-2007-216676, 2011.

[3] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[4] V. S. Miller, "Use of elliptic curves in cryptography," in *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, (New York, NY, USA), pp. 417–426, Springer-Verlag New York, Inc., 1986.

[5] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, Jan. 1987.

[6] ARM, "Cortex-M0+ Technical Reference Manual, Revision: r0p1." Accessed: 2013-06-06, 2012.

[7] I. Freescale Semiconductor, "MKW01Z128, Highly-integrated, cost-effective single-package solution for sub-1 GHz applications, Rev. 3, 5/7/2013." Accessed: 2013-06-06, 2013.

[8] A. K. Lenstra and E. R. Verheul, "Selecting Cryptographic Key Sizes," *Journal of Cryptology*, vol. 14, pp. 255–293, 1999.

[9] A. K. Lenstra, "Key Lengths," *Handbook of Information Security*, vol. 2, pp. 617–635, 2004.

[10] N. I. of Standards and Technology, "Recommendation for Key Management," 2012. Special Publication 800-57 Part 1 Rev. 3.

[11] H. Orman and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys, RFC 3766." http://www.ietf.org/rfc/rfc3766.txt, 2004. Accessed: 2013-06-06.

[12] D. Giry, "BlueKrypt Cryptographic Key Length Recommendation." http://keylength.com, May 2013. Accessed: 2013-05-29.

[13] R. P. Gallant, R. J. Lambert, and S. A. Vanstone, "Faster point multiplication on elliptic curves with efficient endomorphisms," in *Advances in Cryptology-CRYPTO 2001*, pp. 190–200, Springer, 2001.

[14] J. A. Solinas, "Efficient arithmetic on Koblitz curves," in *Towards a Quarter-Century of Public Key Cryptography*, pp. 125–179, Springer, 2000.

[15] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[16] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM systems journal*, vol. 29, no. 4, pp. 526–538, 1990.

[17] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Cryptographic Hardware and Embedded Systems-CHES 2004*, pp. 119–132, Springer, 2004.

[18] M. Hutter and E. Wenger, "Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessors," in *CHES*, vol. 6917 of *Lecture Notes in Computer Science*, pp. 459–474, Springer, 2011.

[19] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, p. 595, 1963.

[20] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology–CRYPTO*, vol. 86, pp. 311–323, Springer, 1987.

[21] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[22] ATMEL Corporation, "ATmega128(L) - 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash." http://www.atmel.com/Images/doc2467.pdf, 2006. Accessed: 2013-06-06.

[23] K. Piotrowski, P. Langendoerfer, and S. Peter, "How public key cryptography influences wireless sensor node lifetime," in *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, SASN '06, (New York, NY, USA), pp. 169–176, ACM, 2006.

[24] Texas Instruments, "MSP430 Ultra-Low-Power Microcontrollers." http://www.ti.com/lit/sg/slab034v/slab034v.pdf, 2013. Accessed: 2013-06-06.

[25] Crossbow Technology Inc., "TelosB Mote Platform." http://www.willow.co.uk/TelosB_Datasheet.pdf. Accessed: 2013-06-06.

[26] Moteiv Corporation, "Tmote sky - ultra low power IEEE 802.15.4 compliant wireless sensor module." http://insense.cs.st-andrews.ac.uk/files/2013/04/tmote-sky-datasheet.pdf, 2006. Accessed: 2013-06-06.

[27] ARM, "ARM7TDMI-S Revision: r4p3 Technical Reference Manual," 2004.

[28] D. G. Chinnery and K. Keutzer, "Closing the power gap between ASIC and custom: an ASIC perspective," in *Proceedings of the 42nd annual Design Automation Conference*, pp. 275–280, ACM, 2005.

[29] ARM, "Architecture Reference Manual." https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf, 2005. Accessed: 2013-06-06.

[30] L. Nachman, J. Huang, J. Shahabdeen, R. Adler, and R. Kling, "IMOTE2: Serious Computation at the Edge," in *International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 1118–1123, 2008.

[31] ARM, "Cortex-M3 Revision: r1p1 Technical Reference Manual." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf, 2007. Accessed: 2013-06-06.

[32] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[33] Çetin Kaya Koç and C. Paar, eds., *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, vol. 1965 of *Lecture Notes in Computer Science*, Springer, 2000.

[34] A. Kargl, S. Pyka, and H. Seuschek, "Fast arithmetic on ATmega128 for elliptic curve cryptography," *International Association for Cryptologic Research Eprint archive*, 2008.

[35] C. P. L. Gouvêa, L. B. Oliveira, and J. López, "Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller," *J. Cryptographic Engineering*, vol. 2, no. 1, pp. 19–29, 2012.

[36] Certicom Research, "Standards for efficient cryptography, SEC 1: Elliptic Curve Cryptography." www.secg.org/download/aid-780/sec1-v2.pdf, 2009. Accessed: 2013-06-06.

[37] D. F. Aranha, L. B. Oliveira, J. López, and R. Dahab, "Efficient implementation of elliptic curve cryptography in wireless sensors," *Advances in Mathematics of Communications*, vol. 4, no. 2, pp. 169–187, 2010.

[38] Certivox Ltd., "Benchmarks and Subs." https://wiki.certivox.com/display/EXT/Benchmarks+and+Subs?sortBy=createddate, Oct. 2012. Accessed: 2013-05-28.

[39] J. Großschädl, "TinySA: a security architecture for wireless sensor networks," in *Proceedings of the 2006 ACM CoNEXT conference*, CoNEXT '06, (New York, NY, USA), pp. 55:1–55:2, ACM, 2006.

[40] S. C. Seo, D.-G. Han, H. C. Kim, and S. Hong, "TinyECCK: Efficient Elliptic Curve Cryptography Implementation over GF(2m) on 8-Bit Micaz Mote," *IEICE transactions on information and systems*, vol. E91-D, pp. 1338–1347, May 2008.

[41] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks.," in *EWSN* (R. Verdone, ed.), vol. 4913 of *Lecture Notes in Computer Science*, pp. 305–320, Springer, 2008.

[42] L. Uhsadel, A. Poschmann, and C. Paar, "Enabling full-size public-key algorithms on 8-bit sensor nodes," in *Proceedings of the 4th European conference on Security and privacy in ad-hoc and sensor networks*, ESAS'07, (Berlin, Heidelberg), pp. 73–86, Springer-Verlag, 2007.

[43] S. S. Erdem, "Fast software multiplication in F2 [x] for embedded processors," *Turkish Journal of Electrical Engineering & Computer Sciences*, 2012.

[44] P. Szczechowiak, A. Kargl, M. Scott, and M. Collier, "On the application of pairing based cryptography to wireless sensor networks," in *Proceedings of the second ACM conference on Wireless network security*, pp. 1–12, ACM, 2009.

[45] L. B. Oliveira, M. Scott, J. López, and R. Dahab, "TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks," in *In Networked Sensing Systems, 2008. INSS 2008. 5th International Conference on*, pp. 173–180, 2008.

[46] J. López and R. Dahab, "High-speed software multiplication in F2m," in *Progress in Cryptology-INDOCRYPT 2000*, pp. 203–212, Springer, 2000.

[47] D. F. Aranha, J. López, L. B. Oliveira, and R. Dahab, "Efficient implementation of elliptic curves on sensor nodes," in *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc (CHiLE 2009)*, 2009.

[48] C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation," in *Advances in Cryptology-CRYPTO'94*, pp. 95–107, Springer, 1994.

[49] J. López and R. Dahab, "Fast multiplication on elliptic curves over GF (2m) without precomputation," in *Cryptographic Hardware and Embedded Systems*, pp. 316–327, Springer, 1999.

[50] D. Aranha and C. P. L. Gouvêa, "RELIC Cryptographic Toolkit." https://code.google.com/p/relic-toolkit/, May 2013. Accessed: 2013-05-28.

[51] P. Szczechowiak, *Cryptographic Key Distribution In Wireless Sensor Networks Using Bilinear Pairings*. PhD thesis, Dublin City University, 2010.

[52] Certivox Ltd., "MIRACL Cryptographic Library." https://certivox.com/solutions/miracl-crypto-sdk/, 2013. Accessed: 2013-05-28.

[53] X. Fan, "An Open Source Library for Elliptic Curve ." http://openecc.org/, 2013. Accessed: 2013-05-28.

Master thesis filing card

Student: P.J. de Clercq

Title: Public Key Cryptography in 32-bit for Ultra Low-Power Microcontrollers

UDC: 621.3

Abstract:

In this work, Elliptic Curve Cryptography (ECC) is used to make an efficient implementation of a public key cryptography algorithm on the ARM Cortex-M0+. A system is introduced to make energy consumption measurements of the ARM Cortex-M0+ microcontroller. Measurements show that there is a large variation (up to 22.5%) in the energy consumption of different instructions. When comparing a simulation of a point multiplication on a binary curve to that on a prime curve, the former will have a slightly faster execution time with a lower power consumption. We show that binary curve arithmetic uses instructions which requires less energy than prime curve arithmetic on the target platform. A new field multiplication algorithm is proposed called `López-Dahab with fixed registers`, which is an optimization of the `López-Dahab` (`LD`) algorithm. The proposed algorithm has a performance improvement of 20.24% over the `LD with rotating registers` algorithm. Our implementation of the `LD with fixed registers` field multiplication algorithm uses a binary Koblitz curve in $\mathbb{F}_{2^{233}}$, the wTNAF point multiplication method with $w = 4$ for random point multiplication, and $w = 6$ for fixed point multiplication. The RELIC toolkit is used to perform the wTNAF precomputation, and wTNAF transformation of the scalar k. All the elliptic curve arithmetic, and field arithmetic are implemented in C and assembly. On average, our implementation of a random point multiplication requires 2814827 cycles, and 34.16 µJ, whereas our fixed point multiplication requires 1864470 cycles, and 20.63 µJ. The proposed `LD with fixed registers` algorithm requires 3672 cycles to perform a field multiplication. A number of different field multiplication methods are developed with the `LD with fixed registers` algorithm, and a simulation is made of these field multiplication methods for different processor word sizes. Findings show that the `LD with rotating registers` algorithm will always outperform the other algorithms, regardless of the word size. The energy consumption of our implementation beats all known software implementations, of a point multiplication, on the same equivalent security level on other platforms by a factor of 7.4. Our cycle count for a field squaring operation also compare favourably to that of other implementations.

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Embedded Systems and Multimedia

Thesis supervisor: Prof. Dr. Ir. I. Verbauwhede

Assessor: Dr. Ir. F. Vercauteren
Prof. Dr. Ir. W. Dehaene

Mentors: Dipl. Ing. L. Uhsadel
Ir. A. Van Herrewege