Efficient Embedded Implementations of Security Solutions for ad-hoc Networks

Benedikt Driessen

September 28, 2007

Diploma Thesis
Ruhr-University Bochum

Faculty of Electrical Engineering
and Information Technology
Chair for Communication Security
Prof. Dr.-Ing. Christof Paar
Dipl.-Ing. Axel Poschmann

Abstract

For many foreseen applications of "wireless sensor networks" (WSN) message integrity is a crucial requirement. Usually, in the area of WSN security services, such as message integrity, are realized by symmetric cryptography only, because asymmetric cryptography is considered as too demanding for typical WSN devices. However, the proposed solutions for symmetric key establishment introduce a significant computation, storage, and – most important – communication overhead. Digital signatures and key-exchange protocols based on asymmetric algorithms would be very valuable though. In the literature usually only RSA and ECC are implemented and compared for sensor nodes, though there exist a variety of innovative asymmetric algorithms. To close this gap, we investigated the efficiency and suitability of digital signature algorithms based on innovative asymmetric primitives for WSN. We chose XTR-DSA and NTRUSign and implemented both (as well as ECDSA) for MICAz motes.

We have decomposed the schemes into layers and show where optimizations can be applied reasonably. Furthermore, we have analyzed, evaluated, and tweaked several algorithms with respect to execution time and memory requirements. We have benchmarked most of the implemented algorithms and give detailed information on precomputation overheads and required RAM and ROM memory. Finally, we have performed a comparative analysis of all three schemes with respect to their suitability for WSNs. We found that, while implemented in pure NesC code, NTRUSign is the winner for being 34% faster in signature generation and 95% faster in signature verification – compared to the de-facto standard ECDSA.

To the best of our knowledge, this thesis presents the fastest implementations of signature schemes for WSNs, while using novel modifications of well-known algorithms. Our implementation of ECDSA seems to be the fastest available for MICAz hardware and the ATMega128L micro-processor. Even our implementation of XTR-DSA performs better than comparable ECDSA implementations. We presume that we present the first detailed approach to implementing XTR-DSA and NTRUSign on constrained hardware.

Keywords:

asymmetric, encryption, signature schemes, ECC, ECDSA, XTR, XTR-DSA, NTRU, NTRUSign, wireless sensor network, finite fields, efficient implementation, NesC, TinyOS, ASM, C, micro-processor, mote, ATMega128L, MICAz

EIDESSTATTLICHE ERKLÄRUNG

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema

"*Efficient Embedded Implementations of Security Solutions for ad-hoc Networks*"

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Zitate habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bochum, den 28. September 2007

Benedikt Driessen

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aim of this thesis . . . 2
1.3 Organization . . . 2

2 Technology overview **3**
2.1 Wireless sensor networks . . . 3
2.2 MICAz motes . . . 4
2.3 TinyOS & NesC . . . 5
2.4 Tools used . . . 6

3 Introduction to cryptography **8**
3.1 Symmetric cryptography . . . 8
3.1.1 Encryption . . . 9
3.1.2 Hash and MAC functions . . . 10
3.2 Asymmetric cryptography . . . 11
3.2.1 Encryption . . . 12
3.2.2 Signatures . . . 13
3.2.3 Mathematical background . . . 14

4 XTR-DSA **15**
4.1 Introduction . . . 15
4.1.1 Related work . . . 15
4.1.2 Mathematical background . . . 16
4.2 The XTR-DSA protocol . . . 18
4.2.1 Keypair generation . . . 18
4.2.2 Signature generation . . . 19
4.2.3 Signature verification . . . 19
4.3 XTR arithmetic . . . 20
4.3.1 Double exponentiation . . . 21
4.3.2 Single exponentiation . . . 23
4.3.3 Op1, Op2, and Op3 . . . 25
4.4 Prime field arithmetic . . . 26
4.4.1 Modular addition and subtraction . . . 27
4.4.2 Multiplication . . . 29
4.4.3 Modular inversion . . . 32
4.5 Results . . . 33
4.5.1 Prime field arithmetic . . . 33
4.5.2 XTR arithmetic . . . 34
4.5.3 Overall performance of XTR-DSA . . . 35
4.6 Overview of optimizations . . . 35

5 ECDSA **36**
5.1 Introduction . . . 36
5.1.1 Related work . . . 36
5.1.2 Mathematical background . . . 37
5.2 The ECDSA Protocol . . . 40
5.2.1 Keypair generation . . . 40
5.2.2 Signature generation . . . 41

5.2.3 Signature verification . . . 41
5.3 ECC arithmetic . . . 42
5.3.1 Single scalar point multiplication . . . 42
5.3.2 Simultaneous scalar multiplication . . . 46
5.3.3 Point addition . . . 50
5.3.4 Point doubling . . . 50
5.4 Prime field arithmetic . . . 52
5.5 Results . . . 52
5.5.1 ECC arithmetic . . . 52
5.5.2 Overall performance of ECDSA . . . 53
5.6 Overview of optimizations . . . 54

6 NTRUSign 56
6.1 Introduction . . . 56
6.1.1 Related work . . . 56
6.1.2 Mathematical background . . . 56
6.2 The NTRUSign protocol . . . 59
6.2.1 Keypair generation . . . 60
6.2.2 Signature generation . . . 64
6.2.3 Signature verification . . . 65
6.3 NTRU arithmetic . . . 66
6.3.1 Single convolution . . . 66
6.3.2 Simultanous convolution . . . 71
6.4 Results . . . 73
6.4.1 NTRU arithmetic . . . 73
6.4.2 Overall performance of NTRUSign . . . 74
6.5 Overview of optimizations . . . 74

7 Evaluation 75
7.1 RAM and ROM footprint . . . 75
7.2 Keypair and signature size . . . 75
7.3 Performance . . . 76

8 Conclusion 78

9 Bibliography 79

A Appendix 83
A.1 Parameters . . . 83
A.1.1 XTR-DSA parameters . . . 83
A.1.2 ECDSA parameters . . . 83
A.1.3 NTRUSign parameters . . . 83
A.2 Double convolution with the modified Karatsuba algorithm . . . 84

List of Tables

4.1 The XTR-DSA keypair 19
4.2 Performance of modular addition/subtraction algorithms 34
4.3 Performance of Montgomery multiplication algorithms 34
4.4 Performance of single exponentiation methods 34
4.5 Performance of double exponentiation methods 35
4.6 Overall performance of our XTR-DSA implementation 35

5.1 Computational costs of point addition and doubling in different coordinate systems [28, Tab. 3.3] 39
5.2 Steps performed by the fixed-point method 46
5.3 Precomputation for Shamir's trick 46
5.5 Performance of single point multiplication algorithms 52
5.6 Performance of simultaneous point multiplication algorithms 53
5.7 Overall performance of our ECDSA implementation 53

6.1 Mapping two trinary coefficents to one byte 64
6.2 Overview of sums that form h's coefficients in the last iteration of Algorithm **59** 71
6.3 Performance of single convolution algorithms 73
6.4 Performance of simultaneous convolution algorithms 73
6.5 Overall performance of our NTRUSign implementation 74

7.1 Comparing the RAM and ROM requirements of our implementations 75
7.2 Comparing the sizes of the generated signatures and keypairs 76
7.3 Comparing the performance of our implementations 76
7.4 Performance of other ECDSA and RSA implementations 77

A.1 Parameters of our XTR-DSA implementation 83
A.2 Parameters of our ECDSA implementation 83
A.3 Parameters of our NTRUSign implementation 84

List of Figures

2.1 (**a**) Unorganized network and (**b**) 3-Level hierarchy 3
2.2 (**a**) A photo of the MICAz mote and (**b**) its components [8] 4

3.1 Cryptology and its subsets 8
3.2 Symmetric cryptography and its subsets 9
3.3 Schematic of a symmetric encryption scheme 9
3.4 Asymmetric cryptography and its subsets 11
3.5 Schematic of an asymmetric encryption scheme 12
3.6 Schematic of digital signature scheme 13

4.1 Three layers of XTR-DSA 15

5.1 Three layers of ECDSA 36
5.2 Geometric (**a**) point addition and (**b**) point doubling 38

List of Algorithms

1 exampleAlg(·): An example algorithm x
2 XTR-DSA: Keypair and prime generation 18
3 XTR-DSA: Signature generation [42, Alg. 5.41] 19
4 XTR-DSA: Signature verification [42, Alg. 5.42] 20
5 expHelper(·): Exponentiation helper algorithm [42, Alg. 2.3.7] 20
6 dblExp1(·): Matrix-less double exponentiation [60, Alg. 2.5] 21
7 dblExp2(·): Improved double exponentiation [60, Alg. 3.1] 22
8 update1(·): Update algorithm 1 for improved double exponentiation 23
9 update2(·): Update algorithm 2 for improved double exponentiation 23
10 update3(·): Update algorithm 3 for improved double exponentiation 23
11 update4(·): Update algorithm 4 for improved double exponentiation 23
12 update5(·): Update algorithm 5 for improved double exponentiation 23
13 update6(·): Update algorithm 6 for improved double exponentiation 23
14 singleExp1(·): Single exponentiation algorithm [60, Alg. 2.4] 24
15 singleExp2(·): Improved single exponentiation [60, Alg. 5.1] 24
16 Precomputation algorithm for single exponentiation [60, Alg. 4.1] 25
17 singleExp3(·): Single exponentiation with $S_{t-1}(c)$ given [60, Alg. 4.2] 25
18 Op1(·): Special operation 1 for XTR layer arithmetic 26
19 Op2(·): Special operation 2 for XTR layer arithmetic 26
20 Op3(·): Special operation 3 for XTR layer arithmetic 26
21 add(·): Multi-precision addition 27
22 sub(·): Multi-precision subtraction 28
23 modAdd(·): Modular multi-precision addition 28
24 modSub(·): Modular multi-precision subtraction 28
25 mul(·): Multi-precision multiplication with Comba's method 29
26 mod(·): Modular reduction with Barrett's algorithm [46, Alg. 14.42] 30
27 modMul(·): Modular multiplication 30
28 Montgomery multiplication (sketch) 31
29 montMul1(·): CIOS-Montgomery multiplication [39] 31
30 montMul2(·): FIPS-Montgomery multiplication [39] 32
31 modInvert(·): Binary inversion method [46, Alg. 14.62] 33
32 ECDSA: Keypair generation [28, Alg. 4.28] 40
33 ECDSA: Signature generation [28, Alg. 4.29] 41
34 ECDSA: Signature verification [28, Alg. 4.30] 41
35 singlePointMul1(·): Binary left-to-right method [28, Alg. 3.27] 42

36 Precomputation for the sliding window method 43
37 singlePointMul2(·): Sliding window method [28, Alg. 3.38] 43
38 Precomputation for the fixed-point method 45
39 singlePointMul3(·): The fixed-point method [28, Alg. 3.41] 45
40 Precomputation for Shamir's trick (and all variants) 47
41 dblPointMul1(·): Shamir's trick [28, Alg. 4.48] 48
42 dblPointMul2(·): Shamir's trick with truncated pre-computation table . . . 49
43 dblPointMul3(·): Simultaneous fixed-point method 50
44 pointAdd(·): Addition of points $\tilde{P}1, P2$ [28, Alg. 3.22] 50
45 pointDbl(·): Point doubling in Jacobian coordinates [28, Alg. 4.21] 51
46 repPointDbl(·): Repeated point doubling in Jacobian coordinates [28, Alg. 4.23] 51
47 NTRUSign: Keypair generation 60
48 compResMod(·): Resultant computation mod p [4, Alg. 2.2.7.1] 62
49 compRes(·): Resultant computation over the integers [4, Alg. 2.2.7.2] 62
50 completeBasis(·): Completing the basis [4, Alg. 3.5.1.1] 63
51 extGcdPoly(·): Extended Euclidean algorithm for polynomials 63
52 iterateInverse(·): Inversion modulo the power of a prime [58, p. 2] 63
53 invertPoly(·): Inversion of a polynomial 64
54 genMsgPoly(·): Generation of a "message point" [4, Alg. 3.6.1.1] 64
55 NTRUSign: Signature generation [4, Alg. 3.5.2.1] 65
56 NTRUSign: Signature verification [4, Alg. 3.5.3.1] 66
57 multiplyPoly1(·): The simple convolution method 67
58 highSpeedConv(·): Silverman's convolution algorithm [59] 68
59 multiplyPoly2(·): Modified Karatsuba algorithm 69
60 assemblePoly1(·): A method to assemble h 70
61 assemblePoly2(·): A method to assemble and reduce h 70
62 dblMultiplyPoly1(·): Simple simultaneous convolution 72
63 dblMultiplyPoly2(·): Simultaneous Karatsuba algorithm 72
64 assemblePolys1(·): A method to assemble polyomials $h1, h2$ 84
65 assemblePolys2(·): A method to assemble and reduce $h1, h2$ 85

List of Abbreviations

ANSI	American National Standards Institute
ASM	Assembly
CIOS	Coarsely Integrated Operand Scanning
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECSTR	Efficient and Compact Subgroup Trace Representation
EESS	Efficient Embedded Security Standards
FIPS	Finely Integrated Product Scanning
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GCD	Greatest Common Divisor
GMP	GNU Multi-Precision library
IEEE	Institute of Electrical and Electronics Engineers
MAC	Message Authentication Code
MSB	Most-Significant Bit (first)
NIST	National Institute of Standards and Technology
PRNG	Pseudo Random Number Generator
PKI	Public Key Infrastructure
RAM	Random Access Memory
ROM	Read Only Memory
SHA1	Secure Hashing Algorithm 1
TINYOS	Tiny microthreaded Operating System
TOSSIM	TinyOS Simulator
WSN	Wireless Sensor Network
XTR	Abbreviation for ECSTR

Conventions

Throughout this thesis, we refer to certain logical and arithmetical functions. The following table lists these functions together with a brief description of their output.

`AND`(a, b)	Returns the bitwise logical 'and' of two operands a and b.
`XOR`(a, b)	Returns the bitwise logical 'exclusive-or' of two operands a and b.
`GCD`(a, b)	Returns the greatest common divisor of a and b.
`SHA1`(v)	Computes the hash of v according to the SHA1 algorithm.
`swap`(a, b)	Swaps the values of the variables a and b.
`deg`(p)	Returns the degree of a polynomial.
`log2`(v)	Returns the logarithm to the base two of a value v.
`char`(r)	Returns the characterstic of a ring r.
$\|\|v\|\|$	Returns the centered norm of v.
$\lfloor v \rfloor$, `floor`(v)	Returns the next integer $\leq v$, i.e., rounds towards $-\infty$.
$\lceil v \rceil$, `ceil`(v)	Returns the next integer $\geq v$, i.e., rounds towards $+\infty$.

We also use the two constants given below.

`0xFF`	This is the hexadecimal notation of $2^8 - 1$.
`MAXVAL`	This value represents the maximum value for a program variable.

Algorithms are listed in the following form. Note that we differentiate between three types of input, the meaning of each field in the algorithm notation is explained in the following example. The algorithm will compute $a + b - d \bmod m$, where only a and b are explicitly passed as arguments when calling the algorithm from another algorithm.

Algorithm 1: `exampleAlg`$(\cdot)$: An example algorithm

Input: Integer operands a, b
These input arguments are mandatory and always passed by us when refering to this algorithm.
Opt. Input: Optional operand $c =$ `MAXVAL`
This argument is optional, i.e., sometimes we may pass a value, otherwise the variable will be set to a predefined value.
Aux. Input: Modulus m
This input argument is never passed explicitly, but must be available to the algorithm, such as pre-computation tables, domain parameters, etc.
Output: $c = a + b - d \bmod m$
The output is defined by the last "return" statement in each algorithm.

ExmplM Add $c \leftarrow a + b \bmod m$;
Subtract $c \leftarrow c - d \bmod m$;
$\vdots$
Return c;

Assigning the result of a computation step to a program variable is denoted by the '$\leftarrow$' operator. We use "marks" to highlight certain lines in algorithms. In constrast to the notation of the `ExmplM` mark, we suffix '$(\cdot)$' when refering to a function instead of a mark in a function, e.g., `exampleAlg`$(\cdot)$.

1 Introduction

In this introductory chapter we will state our motivation for this thesis, followed by a description of what we are trying to achieve. The remainder of this chapter is dedicated to a brief overview of the overall organization of this document.

1.1 Motivation

In the early days of networking, it was a challenge to connect new devices to a network: manual configuration was required and cables had to be run all over the place. This has changed with the advent of wireless network technology. Although creating networks has become trivial, managing network security has become the opposite. The reason is obvious: wireless networks do not stop at the cornerstone of a house, they can also provide direct neighbours with network access. To keep the neighbour's kids from using this network for downloading mp3s, sharing internal network data with the world, or even hacking the pentagon, the usage of encryption is imperative. In case of home networks, it does not really matter how efficient the encryption mechanism is implemented, simply because the hardware is powerful enough.

This is not necessarily the case when we envision the next evolution step in network technology: ubiquitous networks formed by autonomous sensor devices – networks, which are created "ad-hoc", i.e., totally dynamic. Participants can enter a network, communicate with other devices and consequently leave the network without any overhead due to manual configuration. These networks already exists and are used in military and civilian applications. However, the existing sensor networks do not really affect our every day life, but efforts are being made to truly integrate them into what some people perceive as their dearest child: the car. By implanting autonomous sensors into cars, a whole new way of accident avoidance is found. The "car2car" approach aims at providing cars with communication infrastructure to exchange sensor readings, thereby warning each other against accidents, critical environmental conditions and possibly more. Despite the advantages of this idea, it already illustrates the main problem: data sent across a wireless network directly influences the car and hence the driver's behaviour. In unprotected networks, this can be abused in many ways, most of them probably harmless but still annoying. The injection of false data becomes dangerous, when sensors are coupled with driver assistance systems, such as an emergency braking system. This level of automation enforces secure communication, so that an attacker is unable to manipulate sensor readings. Asymmetric cryptography is suited to provide the required security, while avoiding certain problems induced by symmetric schemes. The only drawback of asymmetric schemes is that they are slower. This becomes even more critical, as car manufacturers will try to minimize the costs for the embedded sensors by using only cheap and low-power devices.

To enable asymmetric cryptography on constrained devices, algorithms must be evaluated and and suitable schemes must be selected. Examining and implementing asymmetric cryptography on constrained devices has been done before – however, the focus usually lies on "classic" schemes such as RSA or ECDSA. The different approach of this thesis is briefly described in the following.

1.2 Aim of this thesis

This thesis aims at evaluating “exotic” signature schemes in direct comparison to a highly optimized implementation of the well-known ECDSA standard. We have chosen to implement XTR-DSA because of its compact signatures and comparable operand sizes. The third competitor, NTRUSign, does not provide short signatures, but promises a different advantage: the arithmetic required to implement NTRUSign’s signature generation and verification primitives is restricted to one simple operation, the convolution of polynomials. Our implementation and analysis specifically targets wireless sensor networks, the implied requirements and limitations due to low-power hardware. In particular, we focus on the ATMega128L micro-processor, the most powerful representative of Atmels 8-bit AVR family.

We also hope that this thesis may serve as a support document when implementing either NTRUSign or XTR-DSA on constrained hardware. Both schemes are not as well documented – with respect to an actual implementation – as ECDSA.

1.3 Organization

This document is organized in an introduction, an implementation, and a final evaluation part. Consequently, we start in Chapter **2** by giving an overiew of “wireless sensor networks” (WSN), the targeted hardware, and software tools used in this work. In the following chapter we introduce the fundamentals of cryptography. Subsequently, we give the definitions of the required mathematical background, thereby concluding the introductary part.

The implementation part consists of three chapters, one for each signature scheme. The XTR-DSA scheme is discussed in Chapter **4**, followed by the ECDSA scheme in Chapter **5**. Chapter **6** is dedicated to the implementation of the NTRUSign primitives. The inner structure of each chapter was derived from the decomposition of each scheme. We have divided all schemes into interdependent layers and discuss each layer seperately, descending from the topmost layer. The top-down approach was chosen because the comprehension of certain optimizations on lower layers of a scheme requires introducing their foregoing layer. We finish each chapter with a complete evaluation of the implemented algorithms.

In order to finally evaluate the implemented schemes in direct comparison, we have arranged their respective characteristics in Chapter **7**. Finally, this thesis is concluded in Chapter **8**.

2 Technology overview

This chapter gives an overview of a special form of ad-hoc networks, so-called "wireless sensor networks" (WSNs). Typical hardware devices for sensor networks will be presented, such as the MICAz platform. Finally, the software we used to operate and program these devices will be introduced.

2.1 Wireless sensor networks

A WSN is a network formed by distributed, autonomous sensor platforms (so-called "motes"). The motes of a WSN cooperatively form a dynamic network to monitor environmental conditions such as temperature, sound, or pressure. Although the development of WSNs was originally driven by military applications, they are now used in many civilian applications as well. A WSN typically consists of several motes and a basestation ("sink"). The sink is used by the network operator to retrieve information from all WSN motes. A sink is usually a special mote[1] that acts as gateway for wired devices (e.g., a PC).

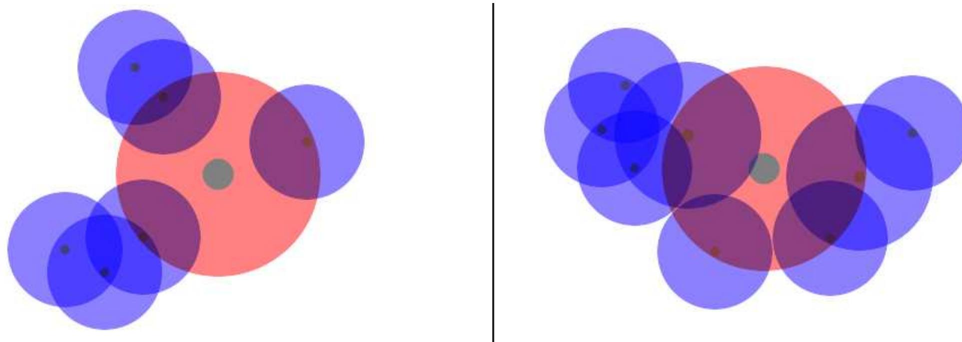

Figure 2.1: (**a**) Unorganized network and (**b**) 3-Level hierarchy

Depending on the purpose of a network, different network-topologies may be found. The simplest layout is an organization without any hierarchy known as "mesh network" (see Figure **2.1**a). This network type typically consists of motes of the same type (blue spheres) and a sink (red sphere). None of them has a special role in the topology. The motes are loosely scattered across an area and dynamically discover their direct neighbors (i.e., motes within reach of the own radio). Since motes are low-power devices, some of them may reach each other directly (the spheres overlap) – but some may not. Routing-mechanisms are employed to communicate with motes that are not directly accessible.

In a data request or status-report scenario, a query requesting data is injected into the sensor network at the sink. The query is forwarded to the other motes in the network. The simplest and least optimal query plan would require each mote to report its own readings back to the sink for processing. After receiving all data packets, the sink would aggregate the received data into a final value and report the value back to the operator. This approach, known as direct delivery, has a number of disadvantages [74]. One problem is that a large number of packets must be sent to the sink. Since each mote sends its own data to the sink, there must be at least one packet of data sent per mote. Additionally, since some network nodes may not be able to communicate directly with the host (see Figure **2.1**a), their data packets must be forwarded by other motes until they reach the

[1] Sinks often have serial interfaces, a wired current supply and stronger radio.

sink. Sending and receiving single packets is very energy intensive – when routing is required (due to the distribution of the motes) this becomes even more critical. In order to conserve both energy and bandwidth, it is useful to move the aggregation and filtering of sensor data into the network itself.

Figure **2.1**b shows a hierarchical organization of the network nodes, where the sink can only reach four motes, two of them being special motes ("aggregators") which can consolidate data. The aggregator and its directly or indirectly reachable "child nodes" form a "cluster". In this type of network, data requests are handled differently. After the request for data has reached all motes in the network, they send their sensor-readings to their corresponding[2] aggregator. The aggregator performs pre-processing and aggregation of all received values and forwards the resulting value to the sink, thereby reducing the overall bandwith usage.

WSNs can grow rather large, which illustrates the need for cheap and reliable sensor-platforms. Today, several models are commercially available, such as TelosB [18], MICA2 [16], and Imote2 [15]. We chose to use the MICAz [17] sensor platform because of its small size and flexibility. Furthermore, MICAz have become the de-facto standard platform for WSN researchers. An introduction to the capabilities and constraints of the hardware of MICAz will be given in the next section.

2.2 MICAz motes

The MICAz mote has been developed at the University of California, Berkeley [30]. The motes are produced and distributed by Crossbow Technology Inc. [17]. The MICAz consists of three main components (see Figure **2.2**b): a micro-processor, a FLASH chip, and a radio component. Additionally, the board is equipped with an interface ("Expansion

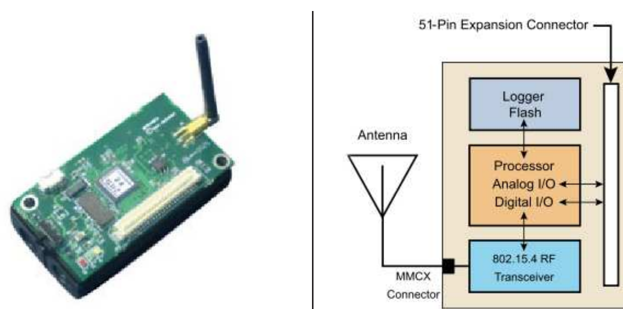

Figure 2.2: (**a**) A photo of the MICAz mote and (**b**) its components [8]

Connector") to enable the usage of various sensor-boards that are additionally available from Crossbow. These specific components can be found on current MICAz boards:

ATMega128L The ATMega128L micro-processor ("μP") is manufactured by Atmel [8]. The ATMega128L is an 8-bit μP in Harvard architecture. The μP offers 128KB program FLASH, 4KB EEPROM, and 4KB SRAM memory. The processor normally runs at 7,37MHz. A special sleep-mode allows the processor to draw less than 15μA of current. The μP supports 8-bit operations (e.g., addition, subtraction) and is able to perform multiplication in only two clock cycles.

AT45DB041 The AT45DB041 component is used as data storage. It is also manufactured by Atmel [7] and adds 512KB serial FLASH memory to MICAz. The additional storage can be used to store measurement data, hence it is also dubbed as "logger flash".

[2]Some networks use special protocols to elect aggregators dynamically. We focus on networks which use statically pre-defined roles.

CC2420 The CC2420 module is a single-chip 2.4GHz IEEE 802.15.4 compliant RF transceiver [68], ready for use in ZigBee [6] applications. It is manufactured by Texas Instruments. With a dipol-antenna attached, an outdoor range from 75m to 100m is possible.

The capability to perform only 8-bit arithmetic in 4KB of RAM has to be taken into account when implementing and optimizing cryptograhic algorithms. Fast algorithms lead to fast computations and short wake-cycles during a mote's lifetime. The lifetime of a mote is obviously limited by its energy resources. MICAz are powered by standard AA batteries (1000 mAh). By sleeping 99% of its lifetime, a mote is supposed to "survive" for years "in the field", once deployed and active [54]. Efficient power management is imperative to achieve this lifetime – as are low currents when micro-processor and/or radio-hardware are sleeping.

Power management is – among other critical functions and services – provided by the operation system. Several operating systems exist for embedded applications, such as BTNUT [21], MANTIS [9], or TinyOS [29]. We chose to use TinyOS, because it is well supported and the de-facto standard among WSN researches. TinyOS is a real-time operating system designed specifically for the special needs of wireless sensor networks, a brief introduction follows.

2.3 TinyOS & NesC

TinyOS is an abbreviation for "**Tiny** microthreaded **O**perating **S**ystem" and orginates at the University of California, Berkeley [29]. TinyOS is an event based operating environment designed for use with networked sensors. More precisely, it is designed to support the concurrency intensive operations required by WSNs with minimal hardware requirements. TinyOS features a component-based architecture, enabling rapid and flexible implementations while minimizing code size.

The component-based design of TinyOS is implemented in NesC [22], which is an extension to the C programming language. It was explicitly developed to enhance the C programming language – which is still common in embedded developement – with the structuring concepts and execution model of TinyOS. The basic concepts behind NesC (and therefore TinyOS) are the following [1]:

1. "Separation of construction and composition: programs are built out of components, which are assembled ('wired') to form whole programs. Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt."

2. "Specification of component behaviour in terms of set of interfaces. Interfaces may be **provided** or **used** by components. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job."

3. "Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider ('commands') and a set to be implemented by the interface's user ('events'). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g., send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the `sendDone` event."

TinyOS is a set of NesC components that can be understood as a program-library for own applications, which have to be implemented as NesC components as well. The components of TinyOS offer various functions ranging from abstraction-layers to the underlying hardware over a scheduler to automatic routing mechanisms. TinyOS and custom components can be used by "wiring" them to own applications, thus allowing the NesC compiler to effectively determine the necessary dependencies. Most of NesC's components are optional to use, some are mandatory (such as the "MainC" component, which initializes all custom software).

The NesC compiler works as a pre-compiler. By identifying all dependencies the corresponding components are "compiled"[3] into one C-file. The resulting C-file is compiled using a heavily modified version of AVRGCC. The original AVRGCC compiler is a cross-compiler and part of the "GNU Compiler Collection" [55] The afore mentioned compilers only represent two of the numerous tools used during this thesis. In the next section we will introduce the remaining tools.

2.4 Tools used

A commandline simulator (TOSSIM [44]) for Linux (and related operating systems) is distributed with TinyOS, enabling the simulation of large-scale networks. TOSSIM captures the network interactions of TinyOS nodes and even simulates the hardware of a mote. The usage of TOSSIM is simple; with one additional commandline argument[4] an application can be compiled for simulation instead of mote hardware. However, TOSSIM has some limitations:

1. TOSSIM can simulate only **one** binary image for all network nodes. This constraint makes it difficult to seperate the implementation of different roles in a protocol.
2. When compiling an application for TOSSIM, custom ASM code for special mote hardware (e.g., ATMega128L) cannot be interpreted. This leads to problems when trying to simulate the behaviour of highly optimized cryptographic routines.

TOSSIM offers a Python [3] interface, so scripts have been developed in Python to setup and start the simulation of a pre-defined WSN topology.

For some of the cryptographic primitives implemented in this work (such as keypair generation), we used the "GNU Multi Precision" (GMP) library [2]. The GMP library offers highly optimized modular-arithmetic functions[5] for large integers.

The performance of our implementations has been evaluated with AVRStudio [14]. AVRStudio is a graphical software development environment, simulator and debugger designed to run on Windows. A wide range of Atmel micro-processors can be simulated with AVRStudio, such as the ATMega128L. The simulator offers a cycle-counter and a fine-grained timer, allowing the accurate measurement of an algorithm's performance. The drawback of AVRStudio is that it only supports simulating and debugging of C and/or ASM code. In order to measure the performance of our NesC algorithms, we had to port them to pure C code. Therefore, our performance measurements do not take any overhead caused by TinyOS into account. On the other side, measuring the performance of a C/ASM binary allows us to compare our implementations to other implementations which were developed in C/ASM. AVRStudio serves as a debugger for binarys that have been compiled with debug information in a certain format. The EXTCOFF format allows to debug mixed code, even structs can be viewed in the debugger. Despite its advantages in

[3] The NesC compiler translates the component/interface concept into pure C code.

[4] `make micaz sim` instead of `make micaz`

[5] The GMP offer various methods ranging from random number generators to prime-factoring functions. We mostly used modular arithmetic functions for multiplication, inversion, subtraction and addition

debugging and simulating EXTCOFF binaries, AVRStudio is not able to compile mixed code with debugging information embedded in the EXTCOFF format itself.

Therefore, the WinAVR toolchain [73] has been used to edit and compile the benchmarked C-code. The WinAVR toolchain includes the AVRGCC compiler in its latest version. The compiler is capable of compiling our back-ported C/ASM code in the EXTCOFF format, such that the binary image can be simulated with AVRStudio.

3 Introduction to cryptography

Cryptography is a subset of the science called cryptology, which deals with secret messages. Today, cryptography is considered to be a blend of mathematics and computer science. The counterpart of cryptography is the cryptanalysis, which deals with analysing secret messages and cryptographic schemes.

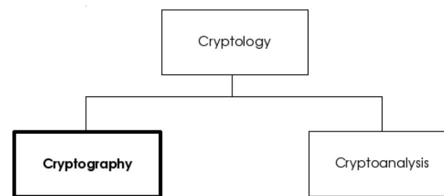

Figure 3.1: Cryptology and its subsets

However, this thesis will strongly focus on the cryptography part. Besides message secrecy (confidentiality), there are two other important security services that can be achieved by cryptography:

Confidentiality Confidentiality ensures that information is accessible only to those authorized to have access.

Authenticity By using authentication, the origin of a message can be reliable ascertained.

Integrity Methods to verify the integrity of a message ensure that a modification of data can be detected.

There are two kinds of cryptographic schemes: symmetric and asymmetric. Symmetric algorithms use the same key ("secret key") to encrypt and decrypt a message. Symmetric schemes are also called "**private** key schemes". An asymmetric cryptographic scheme uses two significantly different keys, the "public key" and a corresponding "private key". The two keys of asymmetric schemes are mathematically bound to each other. Asymmetric schemes are also called "**public** key schemes".

We will now introduce basic properties, well-known algorithms and applications of symmetric and asymmetric cryptography. Due to the fact that this thesis focuses on asymmetric algorithms, an introduction in discrete mathematics will be given, too.

3.1 Symmetric cryptography

The field of symmetric cryptography is divided into symmetric encryption methods and hash and MAC functions. While encryption schemes aim at providing confidentiality, the latter schemes map a message of arbitrary length to an output of fixed length. This output may be used to provide integrity. Both schemes are symmetric, because they use only one key for all involved entities.

To ensure confidentiality, the shared key must remain secret to the authorized parties using the scheme. This property makes symmetric schemes difficult to employ in cases where the need for confidentiality meets the need for cheap hardware in distributed networks (e.g., WSNs). The implied problem is called the "key management problem" and arises in different scenarios. To further illustrate the problem, we list some typical requirements where it is better to use asymmetric schemes:

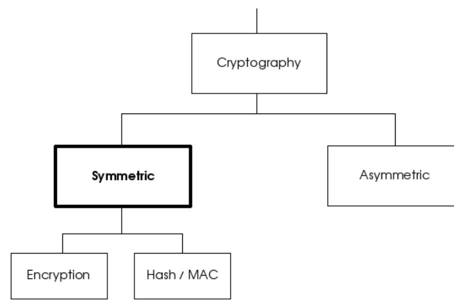

Figure 3.2: Symmetric cryptography and its subsets

1. To enable secured communication in WSNs, each network node has to store one (or more) key(s). This is a problem, because the hardware of motes (and hence the storage of the secret key) is not designed to protect against physical attacks aiming at extracting the key(s).

2. Entities sharing a key must be sure about the identities of each other. Otherwise they may reveal information to adversaries who are in posession of the secret key and disguise as valid participant of the network.

3. When one-to-one encryption is desirable, this would require $n-1$ distinct keys to be distributed across a network of n network nodes.

4. While the initial distribution of secret keys may be performed in a trusted environment, updating the key(s) of deployed motes must be done via a secure and authenticated channel.

Besides these properties that make symmetric schemes unsuitable for WSNs, schemes using a shared key have some advantages compared to asymmetric schemes.

1. Symmetric algorithms are usually a lot faster than asymmetric schemes.

2. The key-size needed to achieve a reasonable level of security is much smaller. A typical symmetric key is of 80 bit size, whereas asymmetric schemes require keys with 1024 bit and more.

3. Symmetric algorithms normally consume less RAM and ROM than asymmetric schemes.

3.1.1 Encryption

Figure **3.3** shows a typical scenario, where two parties use the same key to encrypt and successively decrypt a message. A symmetric encryption scheme consists of three algorithms:

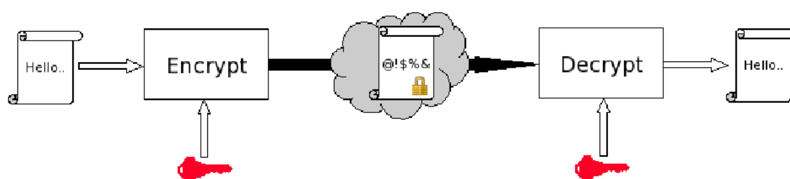

Figure 3.3: Schematic of a symmetric encryption scheme

key generation, encryption, and decryption. A secret key k is generated by selecting an appropriate "security parameter" κ and applying the generation algorithm to it. The plaintext of a secret message M is encrypted using the encryption algorithm. The resulting ciphertext C can only be decrypted by applying the decryption function with the right key.

Definition 1 (Symmetric Encryption Scheme [26, Def. 5.1.1]) *A* symmetric encryption scheme $\texttt{SES} := (\texttt{Gen}, \texttt{Enc}_k, \texttt{Dec}_k)$ *consists of the following algorithms:*

Gen *A probabilistic algorithm that on input of the security parameter κ outputs a symmetric key $k \in_R \{0,1\}^\kappa$, $k \leftarrow \texttt{Gen}(\kappa)$.*

Enc *A deterministic encryption algorithm that on input $k \in \{0,1\}^\kappa$ and a message $M \in \{0,1\}^*$ outputs a ciphertext $C \leftarrow \texttt{Enc}_k(M)$.*

Dec *A deterministic decryption algorithm that on input $k \in \{0,1\}^\kappa$ and a ciphertext $C \in \{0,1\}^*$ outputs $M \leftarrow \texttt{Dec}_k(C)$.*

Example: Alice wants to send a secret message to Bob over a public channel (e.g., the internet), without evil Eve being able to read the message. Alice generates a secret key and sends it to Bob via a secure[1] channel, for example by giving it to him directly. Now Bob and Alice are in posession of a shared secret that Eve does not know. Alice encrypts her message with an algorithm she has previously negotiated with Bob. Then, she sends the encrypted message via the public channel. Eve can eavesdrop the channel, but is unable to make any sense out of the data she sees, because she is lacking the key. Bob can decrypt the message with his key and reply in a similar fashion. This simple example partially demonstrates the key management problem we mentioned before: a secure channel must exist **before** secure communication can take place.

DES [52] and its successor AES [19] (also known as "Rijndael") are the most well-known symmetric encryption algorithms.

3.1.2 Hash and MAC functions

Hash algorithms are another class of symmetric algorithms, among them MD5 [56] and SHA1 [51]. A hash algorithm maps a message M of arbitrary length to a hash digest $\mu_{\texttt{hash}}$ of fixed length. The digest can be used as checksum or to "fingerprint" a block of data.

Definition 2 (Hash Function [25, Def. 2.2.1]) *A function $\texttt{H} : \{0,1\}^* \rightarrow \{0,1\}^r$, $r \in \mathbb{N}$ is called a collision resistant hash function if the following two conditions hold:*

Efficient computation *There exists an efficient, deterministic algorithm that on input of $M \in \{0,1\}^*$ outputs the hash value $\mu_{\texttt{hash}} \leftarrow \texttt{H}(M)$, $\mu_{\texttt{hash}} \in \{0,1\}^r$.*

Collision resistance *The probability that an adversary finds $M_1, M_2 \in \{0,1\}^*$ such that $M_1 \neq M_2$ and $\texttt{H}(M_1) = \texttt{H}(M_2)$ is negligible.*

Example: Due to their collision resistance, hash algorithms can be used to achieve integrity. A common application of this principle is found in package management programs of modern Linux systems. The management software downloads new applications together with a hash value. Then, the binary image of the newly downloaded software is hashed with the appropriate hash function. If the generated hash matches the transmitted hash digest, the software is installed, otherwise it is rejected as "defective".

The so-called "message authentication codes" (MACs) are quite similar to hash functions, with the difference that they take a second parameter as input. A secret key is generated similarly to the generation of a key for the encryption algorithm. Signing a message M with this key using the signature generation method outputs a MAC digest $\mu_{\texttt{MAC}}$ of fixed length. A receiver of the message and the digest, who also shares k with the sender, can use the verification function to test whether the tuple $(\mu_{\texttt{MAC}}, k, M)$ is valid (i.e., $\mu_{\texttt{MAC}}$ has been generated by k and M).

[1] By secure we denote a confidential, authentic channel that also provides integrity.

Definition 3 (Message Authentication Code [26, Const. 6.3.1]) *A* message authentication code $\mathtt{MAC} := (\mathtt{Gen}, \mathtt{Sign}_k, \mathtt{Verify}_k)$ *consists of the following algorithms:*

Gen *A probabilistic algorithm that on input of the security parameter κ outputs a symmetric key $k \in_R \{0,1\}^\kappa$, $k \leftarrow \mathtt{Gen}(\kappa)$.*

Sign *A deterministic algorithm that on input $k \in \{0,1\}^\kappa$ and a message $M \in \{0,1\}^*$ outputs a MAC value $\mu_{\mathtt{MAC}} \leftarrow \mathtt{Sign}(M)$.*

Verify *A deterministic algorithm that on input $k \in \{0,1\}^\kappa$, $M \in \{0,1\}^*$ and a candidate MAC value $\mu_{\mathtt{MAC}}$ outputs "accept" or "reject", indicating whether $\mu_{\mathtt{MAC}}$ is valid or not, $\mathtt{ind} \in \{\text{"accept"}, \text{"reject"}\}, \mathtt{ind} \leftarrow \mathtt{Verify}_k(\mu_{\mathtt{MAC}}, M)$.*

MAC functions can be constructed from hash functions and symmetric encryption schemes. They can be efficiently computed by sensor nodes. A MAC function constructed from a hash algorithm is called HMAC or "keyed hash function" [40]. The basic idea of HMAC functions is to hash a secret key k together with the message M,

$$\mu_{\mathtt{HMAC}} \leftarrow \mathtt{H}(M||k).$$

This way, only entities in posession of the secret key k can generate $\mu_{\mathtt{HMAC}}$. Due to this property, integrity **and** authentication can be achieved with (H)MAC functions.

3.2 Asymmetric cryptography

Asymmetric schemes are divided into two subsets (see Figure **3.4**), encryption schemes and signature schemes. Like symmetric encryption algorithms, asymmetric encryption schemes aim at protecting the confidentiality of a message. Signature schemes are comparable to MAC functions, they aim at binding the identity of an entity to a message. Besides these common aims, asymmetric schemes are completely different from their symmetric counterparts. Where symmetric schemes use one shared key for sender and receiver (e.g., Alice and Bob), asymmetric schemes use different keys for each role.

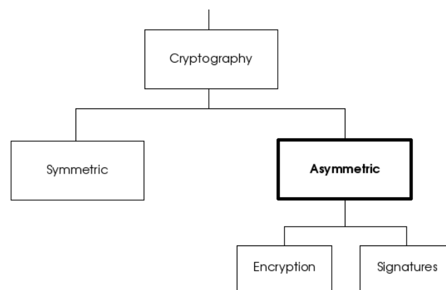

Figure 3.4: Asymmetric cryptography and its subsets

The concept of public-key cryptography was introduced in 1976 by Diffie and Hellman [20] in order to solve the key management problem we discussed earlier. Asymmetric systems use two keys, a private key $k_{\mathtt{priv}}$ that is kept secret and the corresponding public key $k_{\mathtt{pub}}$ that may be published in unencrypted form. The tuple $(k_{\mathtt{priv}}, k_{\mathtt{pub}})$ is called the keypair. Due to their mathematical connection, it is an important property of the keypair that the private key cannot be derived easily when the public key is known. Therefore, asymmetric systems are based on "hard" number theoretic problems and use a "trapdoor function".

Definition 4 (Trapdoor one-way function [25, Def. 2.2.1]) *A* trapdoor one-way function *is a function $f : \mathbb{X} \rightarrow \mathbb{Y}$ such that:*

1. *It is easy to compute $f(x)$ for all $x \in \mathbb{X}$.*

2. *Given a value y it is intractable to compute x such that $f(x) = y$ for all $y \in \mathbb{Y}$.*

3. *Given a value $y \in \mathbb{Y}$ and some additional trapdoor information it is easy to compute $x = f^{-1}(y)$.*

"Hard" means that solving the corresponding computational problem is believed to be infeasible. Several "hard" problems exist, but only few could be successfully used to build an asymmetric scheme, such as the "integer factorization problem". RSA [57], a widespread asymmetric algorithm, relies on the assumption that it is hard to factor a large integer n with $n = p \cdot q$, where p and q are large, unknown primes.

As stated before, asymmetric schemes are significantly slower than symmetric algorithms. For this reason, asymmetric schemes (e.g., RSA) have been considered to be unsuitable for low-power embedded devices (such as motes) for a long time. This has changed with the availability of high-speed implementations of alternative cryptography schemes.

We will now introduce the basics of asymmetric encryption and signature generation. After that, we will shortly list the most important definitions of discrete mathematics to introduce the basic concepts necessary to understand the asymmetric algorithms presented in this thesis.

3.2.1 Encryption

An asymmetric encryption scheme consists of three distinct algorithms. The key generation algorithm generates a keypair $(k_{\texttt{priv}}, k_{\texttt{pub}})$. The public key can be sent to anyone who is interested in sending encrypted messages to the creator. To produce the encrypted message C, the message M and key $k_{\texttt{pub}}$ are passed to the encryption function (see Figure **3.5**). The

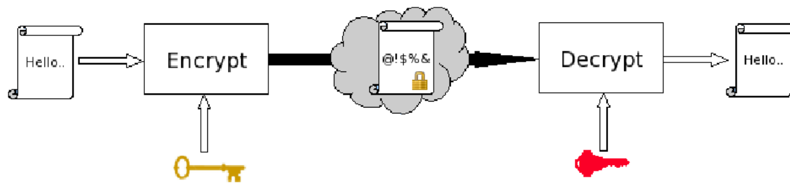

Figure 3.5: Schematic of an asymmetric encryption scheme

output has to be sent to the entity in posession of the corresponding private key. When the owner of a private key receives an encrypted message, he uses his secret key $k_{\texttt{priv}}$ to generate the plaintext message M with the decryption algorithm.

Definition 5 (Asymmetric Encryption Scheme [26, Def. 5.1.1]) *An* asymmetric encryption scheme $\texttt{AES} := (\texttt{Gen}, \texttt{Enc}_{k_{\texttt{pub}}}, \texttt{Dec}_{k_{\texttt{priv}}})$ *consists of the following algorithms:*

Gen *The* keypair $(k_{\texttt{pub}}, k_{\texttt{priv}})$ *corresponding to a set of security parameters* $\chi = \{\kappa_1, \kappa_2, ..\kappa_n\}$, $n \in \mathbb{N}$ *is generated by the probabilistic key generation algorithm,* $(k_{\texttt{pub}}, k_{\texttt{priv}}) \leftarrow \texttt{Gen}(\chi)$.

Enc *An encryption algorithm that on input of the public key $k_{\texttt{pub}}$ and a message $M \in \{0,1\}^*$ outputs a ciphertext $C \leftarrow \texttt{Enc}_{k_{\texttt{pub}}}(M)$.*

Dec *A decryption algorithm that on input $k_{\texttt{priv}}$ and a ciphertext $C \in \{0,1\}^*$ outputs $M \leftarrow \texttt{Dec}_{\texttt{k}_{\texttt{priv}}}(C)$.*

Example: For Alice, Bob, and Eve this means that the public key can be transmitted freely, eliminating the need for a preexisting, secure channel – thus solving the key management problem. In case Bob wants to send a message to Alice he just needs to request the public key via an authentic channel. Today, central servers in the internet provide a way to reliably publish and obtain public keys. After having received the public key, Bob encrypts his secret message with this key and forwards the result to Alice. Again, Eve is clueless about the content, although she is in posession of Alice's (and even Bob's) public key. Eve could use this key to send encrypted messages to Alice, too. Alice is the only person in posession of the matching secret key, so she can easily decrypt and read any message encrypted with her public key. To reply to Bob, Alice needs to obtain his public key and proceed as previously done by Bob.

3.2.2 Signatures

Diffie and Hellman also introduced the notion of digital signatures. A digital signature allows to uniquely bind a message to its sender. This connection can only be created by the sender (using the private key), but it can be verified by everybody with the public key. Again, the scheme consists of three distinct algorithms and a keypair, but compared

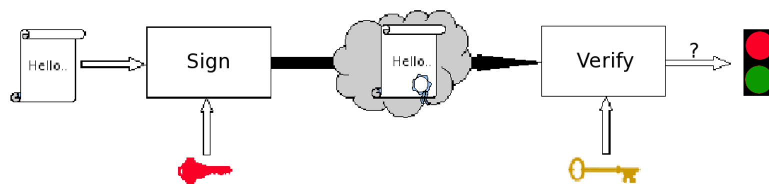

Figure 3.6: Schematic of digital signature scheme

to asymmetric encryption schemes the keys are used the other way round, i.e., the private key is used for "encryption". The key generation algorithms of encryption and signature schemes based on the same mathematical problem are often interchangeable – but this is not always the case. An entity that wants to send a signed message has to pass its private key $k_{\texttt{priv}}$ together with a message M to the signature generation function. The resulting signature S can be transmitted freely over the internet, together with the message M (see Figure **3.6**). Whoever wants to verify the signature S for a message M uses "public key infrastructures" (PKI) – such as servers in the internet – to find the corresponding public key and applies the signature verification algorithm to the tuple $(M, S, k_{\texttt{pub}})$.

Definition 6 (Digital Signature Scheme [26, Def. 6.1.1]) *A digital signature scheme* $\texttt{DSS} := (\texttt{Gen}, \texttt{Sign}_{k_{\texttt{priv}}}, \texttt{Verify}_{k_{\texttt{pub}}})$ *consists of the following algorithms:*

Gen *The keypair* $(k_{\texttt{pub}}, k_{\texttt{priv}})$ *corresponding to a set of security parameters* $\chi = \{\kappa_1, \kappa_2, ..\kappa_n\}$, $n \in \mathbb{N}$ *can be generated efficiently by the probabilistic key generation algorithm,* $(k_{\texttt{pub}}, k_{\texttt{priv}}) \leftarrow \texttt{Gen}(\chi)$.

Sign *An algorithm that on input of the private key* $k_{\texttt{priv}}$ *and a message* $M \in \{0,1\}^*$ *outputs a digital signature* $S \leftarrow \texttt{Sign}_{k_{\texttt{priv}}}(M)$.

Verify *An algorithm that on input* $k_{\texttt{pub}}$, $M \in \{0,1\}^*$, *and a candidate signature* S *outputs "accept" or "reject", indicating" whether* S *is valid or not,* $\texttt{ind} \in \{\text{"accept"}, \text{"reject"}\}$, $\texttt{ind} \leftarrow \texttt{Verify}_{k_{\texttt{pub}}}(S, M)$.

Example: Alice wants to sign a digital contract with Bob. She uses her unique private key to sign the digital document. Then, she submits the document and the generated signature to Bob. Now, Eve can read the message. Bob and Eve are able to verify the signature by using Alice's public key. Verification only works, when document and signature are transmitted unmodified and the signature has been generated with the key corresponding

to the public key Bob and Eve associate with Alice. In case Eve manipulates the contract (e.g., during transmission), this would be detected by Bob because the verification algorithm would fail (i.e., the algorithm outputs "*reject*"). Eve cannot forge a signature of Alice, because she does not have the private key matching the known public key – nor is she able to compute it.

3.2.3 Mathematical background

We will now give some important definitions successively leading to the final definition of the "prime field". The prime field is the most important mathematical construct for this thesis. All of the presented schemes are based on arithmetic operations in prime fields.

Definition 7 (Finite Group [64]) *A* group $(\mathbb{G}, \circ)$ *is a finite set* $\mathbb{G}$ *together with a binary operation* $\circ$ *(called the* group operation*), both satisfy the following properties:*

Closure *For all* $a, b \in \mathbb{G}$ *the result of the binary operation* $\circ$ *yields an element also in* $\mathbb{G}$*:* $c = a \circ b$ *with* $c \in \mathbb{G}$.

Associativity *The defined* group operation *is associative:* $a \circ (b \circ c) = (a \circ b) \circ c$, $\forall a, b, c \in \mathbb{G}$.

Identity *There is an identity element* 1 *such that:* $a \circ 1 = 1 \circ a = a$, $\forall a \in \mathbb{G}$.

Inverse *There must be an inverse element* a^{-1} *for all* $a \in \mathbb{G}$*, such that* $a \circ a^{-1} = 1$, $\forall a \in \mathbb{G}$.

Definition 8 (Subgroup [67]) *Let* $(\mathbb{G}, \circ)$ *be a finite group.* $\mathbb{S} \subseteq \mathbb{G}$ *is a* subgroup *of* $\mathbb{G}$ *when* $(\mathbb{S}, \circ)$ *is a group, too. Let* $a \in \mathbb{G}$*, repeatedly applying* $\circ$ *to* a *generates a subgroup of* $\mathbb{G}$, $<a> = \{a^k | k \in \mathbb{N}\} = \mathbb{T}$. *a is called the generator of* $\mathbb{T}$.

Definition 9 (Abelian Group [62]) *An* abelian group *is a group* $(\mathbb{G}, \circ)$ *with* $\circ$ *as the* group operation *for which the* commutative law *holds:* $a \circ b = b \circ a$, $\forall a, b \in \mathbb{G}$.

Definition 10 (Ring [66]) *A set* $\mathbb{R}$ *with two binary operations such as addition '+' and multiplication '·' is called a* ring*, when for* $(\mathbb{R}, \cdot, +)$ *the following properties are satisfied:*

1. $(\mathbb{R}, +)$ *is an abelian group with additive identity denoted by* 0.
2. *For* $(\mathbb{R}, \cdot)$*, the following axioms hold:*

   Closure $a \cdot b \in \mathbb{R}$*, for all* $a, b \in \mathbb{R}$.
   Associativity $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
   Identity $a \cdot 1 = 1 \cdot a = a \in \mathbb{R}$.

3. *The distributive law holds:* $(a + b) \cdot c = c \cdot a + c \cdot b$, $\forall a, b, c \in \mathbb{R}$.

Definition 11 (Finite field [63]) Finite fields *are abstractions of familiar number systems and their properties. A* finite field *consist of a finite set* $\mathbb{F}_n = \{0, 1, ..n - 1\}$ *together with two binary operations: e.g., addition '+' and multiplication '·'. These building blocks form a* finite field $(\mathbb{F}_n, \cdot, +)$*, when the following properties are satisfied:*

1. $(\mathbb{F}_n, +)$ *is an abelian group with additive identity denoted by* 0.
2. $(\mathbb{F}_n \setminus \{0\}, \cdot)$ *is an abelian group with muliplicative identity denoted by* 1.
3. *The distributive law holds:* $(a + b) \cdot c = c \cdot a + c \cdot b$, $\forall a, b, c \in \mathbb{F}_n$.

Definition 12 (Prime Field [65]) *Let* p *be a prime,* $\mathbb{F}_p = \{0, 1, ..., p - 1\}$ *be a finite set. Then* $(\mathbb{F}_p, \cdot, +)$ *with addition and multiplication performed modulo* p *denotes a* prime field.

4 XTR-DSA

In this chapter we will detail our implementation of the XTR-DSA [42] signature scheme. We start by giving an overview of the scheme and present related work. Then, we decompose the scheme in several layers and present our implementation of each layer. We close this chapter by analyzing the performance of each evaluated algorithm and the overall implementation of XTR-DSA.

4.1 Introduction

The XTR public key scheme has been proposed by Lenstra in 2000 [43]. XTR is an abbreviation for "Efficient and Compact Subgroup Trace Representation" (ECSTR). XTR uses "traces" to represent and calculate powers of elements of a subgroup of a prime field. According to the authors of XTR, this scheme is able to achieve security equivalent to RSA-1024 by using subgroup elements of 160-170 bit size. Small operands are imperative for fast arithmetic on constrained devices, so the XTR based signature scheme XTR-DSA is a promising candidate for fast asymmetric signature primitives.

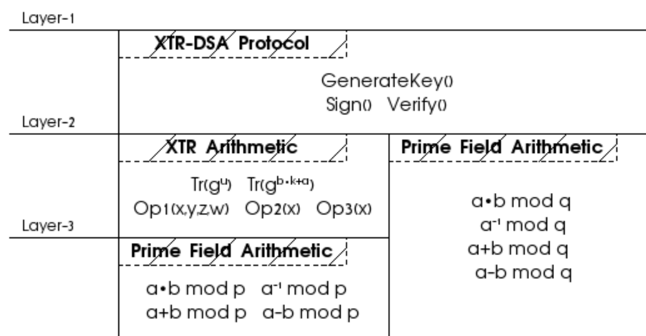

Figure 4.1: Three layers of XTR-DSA

XTR-DSA is a signature scheme with three layers (see Figure **4.1**). On the lowest layer there is the prime field arithmetic, which includes basic arithmetic operations such as modular addition, subtraction, multiplication, and inversion in $\mathbb{Z}_n$, with n being a large prime number. The XTR layer uses this arithmetic for three arithmetic operations we denote by `Op1`$(\cdot)$, `Op2`$(\cdot)$, and `Op3`$(\cdot)$. From these three operations we can build methods to compute traces of elements in $\mathbb{Z}_{p^6}$, which is required by all algorithms of the XTR-DSA layer. The XTR-DSA layer additionally involves prime field arithmetic, but with a different modulus q.

4.1.1 Related work

XTR is patented, therefore we were unable to find any publicly available, open-source software implementations of XTR-DSA. However, we found a paper by Blass *et al.* [10] where the authors compare the implementations of several asymmetric schemes for low-power devices – one of them is XTR-DSA. We also found papers discussing the hardware implementation of XTR on FPGAs [53, 50].

4.1.2 Mathematical background

This section will give a brief introduction to the underlying mathematical principles of XTR. The information and definitions presented are based on two papers by the XTR authors [42, 43]. For a more detailed analysis of the arithmetic and the corresponding proofs please refer to these papers.

XTR's groups

In 1997, Lenstra proposed a method to use cyclotomic polynomials to construct efficient public key cryptosystems [41]. For the construction of XTR, two primes q and $p \equiv 2 \bmod 3$ of 160-170 bit size are used to implement Lenstra's idea. XTR operates in a subgroup of order q (the "XTR-subgroup") of the order $\Phi_6(p)$ subgroup of $\mathbb{Z}^*_{p^6}$ (the "XTR-supergroup"), where $\Phi_6(x) = x^2 - x + 1$ is called the 6th degree cyclotomic polynomial.

Definition 13 (Cyclotomic polynomial [41]) *Let $\Phi_t(x) = \prod_{\xi}(x - \xi)$, be the tth degree cyclotomic polynomial, where ξ ranges over the primitive t^{th} roots of unity.*

For the security of XTR it is mandatory that q is chosen such that $q > 6$ and $q|\Phi_6(p)$, otherwise the "hard" problem on which XTR is based, may be broken (see below).

Definition 14 (XTR-supergroup and XTR-(sub)group [42, p. 2]) *Let $p \equiv 2 \bmod 3$ and q be primes such that $q > 6$ and $q|\Phi_6(p)$.*

XTR-supergroup *The XTR-supergroup is an order $\Phi_6(p) = p^2 - p + 1$ subgroup of $\mathbb{Z}^*_{p^6}$ generated by g.*

XTR-(sub)group *The XTR-subgroup is a subgroup of the XTR-supergroup of order q.*

Trace representation

By using the trace function over $\mathbb{Z}_{p^2}$, a compact and efficient representation of elements of the XTR-supergroup is found.

Definition 15 (Conjugates [42, p. 4]) *The* conjugates *of $h \in \mathbb{Z}_{p^6}$ over $\mathbb{Z}_{p^2}$ are h, h^{p^2}, and h^{p^4}.*

Definition 16 (Trace [43, p. 4]) *The sum of conjugates over $\mathbb{Z}_{p^2}$ of $h \in \mathbb{Z}_{p^6}$ is known as the* trace $\mathtt{Tr}(h)$*, where $\mathtt{Tr}(h) = h + h^{p^2} + h^{p^4} \in \mathbb{Z}_{p^2}$.*

Recall that g generates the XTR-supergroup. We follow the notation of XTR's authors and denote $\mathtt{Tr}(g^1) = \mathtt{Tr}(g)$ by $c_1 = c$, for any integer exponent k we write $\mathtt{Tr}(g)^k = \mathtt{Tr}(g^k)$ as c_k. XTR uses the property that there is a way to compute $\mathtt{Tr}(g^k)$ efficiently when $\mathtt{Tr}(g)$ is given, thereby replacing $g^n \in \mathbb{Z}_{p^6}$ by $\mathtt{Tr}(g^n) \in \mathbb{Z}_{p^2}$ and obtaining a reduction of a factor three in operand size. In Section **4.3.2** we give three ways of obtaining c_k from c. This is done by computing three consecutive powers of c in every step of the exponentiation.

Definition 17 ($S_n(c)$ [43, Def. 2.3.6]) *Let $S_n(c) = (c_{n-1}, c_n, c_{n+1}) = (\mathtt{Tr}(g^{n-1}), \mathtt{Tr}(g^n), \mathtt{Tr}(g^{n+1})) \in \mathbb{Z}^3_{p^2}$, where $g \in \mathbb{Z}^*_{p^6}$ and $c_{n-1}, c_n, c_{n+1} \in \mathbb{Z}_{p^2}$. Let $S_0 = (c^p, 3, c)$ and $S_1 = (3, c, c^2 - 2 \cdot c^p)$.*

Elements and arithmetic in $\mathbb{Z}_{p^2}$

XTR uses the trace over $\mathbb{Z}_{p^2}$ to represent elements of the supergroup. By using $\mathtt{Tr}(\cdot)$, we never have to explicitly represent any $h \in \mathbb{Z}^*_{p^6}$, we rather work with elements in $\mathbb{Z}_{p^2}$. An element $c \in \mathbb{Z}_{p^2}$ is represented as $c = x_1 \cdot \alpha + x_2 \cdot \alpha^2$ where α and $\alpha^p = \alpha^2$ are the roots of the polynomial $X^2 + X + 1$, which is irreducible over $\mathbb{Z}_p$. In this representation, an element $t \in \mathbb{Z}_p$ is represented as $-t \cdot \alpha - t \cdot \alpha^2$, so $\mathtt{Tr}(g^0) = c_0 \in \mathbb{Z}_{p^2}$ is represented as $-3 \cdot \alpha - 3 \cdot \alpha^2$.

We have observed that for a successful implementation of XTR only three distinct mathematical operations are sufficient. We denote them by $\mathtt{Op1}(x, y, z, w), \mathtt{Op2}(x)$, and $\mathtt{Op3}(x)$. These three operations have been derived based on the facts given by Stam [60, Facts 2.3] and Lenstra [42, Corollary 2.3.5]. Discussing the implications and proofs of these basic facts is not in scope of this thesis, the interested reader is refered to the original documents. The following arithmetic in $\mathbb{Z}_{p^2}$ and $\mathbb{Z}_p$ is performed by these basic operations:

Op1(x,y,z,w) This operation requires four elements $w, x, y, z \in \mathbb{Z}_{p^2}$. The resulting element $c \in \mathbb{Z}_{p^2}$ is computed by two multiplications in $\mathbb{Z}_{p^2}$,

$$c = \mathtt{Op1}(x, y, z, w) = x \cdot z - y \cdot z^p + w.$$

$\mathtt{Op1}(\cdot)$ is an XTR layer operation (see Figure **4.1**), the actual arithmetic is performed by simple prime field operations in $\mathbb{Z}_p$. Four modular multiplications in $\mathbb{Z}_p$ are necessary. $c = c_1 \cdot \alpha + c_2 \cdot \alpha^2$ is computed by,

$$c_1 = w_1 + (z_1 \cdot (y_1 - x_2 - y_2) + z_2 \cdot (x_2 - x_1 + y_2)) \bmod p, \tag{4.1}$$

$$c_2 = w_2 + (z_1 \cdot (x_1 - x_2 + y_1) + z_2 \cdot (y_2 - x_1 - y_1)) \bmod p. \tag{4.2}$$

Op2(x) $c = \mathtt{Op2}(x)$ is computed by one squaring in $\mathbb{Z}_{p^2}$,

$$c = \mathtt{Op2}(x) = x^2 - 2 \cdot x^p.$$

Note that computation of x^p does not require any arithmetic in $\mathbb{Z}_p$, because

$$x^p = x_1^p \cdot \alpha^p + x_2^p \cdot \alpha^{2 \cdot p} = x_1 \cdot \alpha^p + x_2 \cdot \alpha$$

holds. So x_1 and x_2 are just swapped and only two multiplications are performed:

$$c_1 = x_2 \cdot (x_2 - 2 \cdot x_1) - 2 \cdot x_2 \bmod p, \tag{4.3}$$

$$c_2 = x_1 \cdot (x_1 - 2 \cdot x_2) - 2 \cdot x_1 \bmod p. \tag{4.4}$$

Op3(x) This operation is based on $\mathtt{Op2}(\cdot)$ and can be computed as

$$c = \mathtt{Op3}(x) = (\mathtt{Op2}(x) - x^p) \cdot x + 3 = x^3 - 3 \cdot x^{p+1} + 3.$$

We compute c_1 and c_2 in two steps, first we set t_1 and t_2 such that

$$t_1 = x_2 \cdot (x_2 - 2x_1) - 3 \cdot x_2 \bmod p, \quad t_2 = x_1 \cdot (x_1 - 2x_2) - 3 \cdot x_1 \bmod p. \tag{4.5}$$

From (t_1, t_2) we can compute (c_1, c_2) directly. By applying the ideas of Karatsuba [5] we compute $(x_1 \cdot x_2), (t_1 \cdot t_2)$, and $(x_1 + x_2) \cdot (t_1 + t_2)$. Therefore, we only need a total of five multiplications in $\mathbb{Z}_p$ (recall that $3 \in \mathbb{Z}_{p^2}$ is represented as $-3 \cdot \alpha - 3 \cdot \alpha^2$),

$$c_1 = (x_2 \cdot t_2) - ((x_1 + x_2) \cdot (t_1 + t_2) - (x_1 \cdot x_2) - (t_1 \cdot t_2)) - 3 \bmod p, \tag{4.6}$$

$$c_2 = (x_1 \cdot t_1) - ((x_1 + x_2) \cdot (t_1 + t_2) - (x_1 \cdot x_2) - (t_1 \cdot t_2)) - 3 \bmod p. \tag{4.7}$$

Given $c = \mathtt{Tr}(g)$, we can use these three operations to compute $S_n(c) = (c_{n-1}, c_n, c_{n+1})$. By computing $S_n(c)$ we implicitly compute the nth power $\mathtt{Tr}(g^n)$ of $\mathtt{Tr}(g)$, where $n \in \mathbb{Z}_q$ and $g \in \mathbb{Z}^*_{p^6}$. The computation of powers based on c_1 is the core operation of XTR, therefore the performance of $\mathtt{Op1}(\cdot), \mathtt{Op2}(\cdot)$, and $\mathtt{Op3}(\cdot)$ is most critical for the overall performance of XTR and XTR-DSA.

"Hard" problem

The security of XTR is based on the *Discrete Logarithm Problem* (DLP). In case of XTR this means that given c and d in the XTR-subgroup, it is "hard" to find an integer x such that $c^x = d$. By choosing q such that $q > 6$ and $q|\Phi_6(p)$, q does not divide the group order of a subgroup of $\mathbb{Z}^*_{p^6}$. Therefore, no subgroup of order q can be embedded in any of $\mathbb{Z}^*_p$, $\mathbb{Z}^*_{p^2}$, and $\mathbb{Z}^*_{p^3}$. This implies that in order to solve the DLP problem in the XTR-(sub)group, it has to be solved in the XTR-supergroup, which is believed to be computationally intractable.

4.2 The XTR-DSA protocol

XTR-DSA is a digital signature scheme based on the elementary operations (Layer-2, Figure **4.1**) of XTR. The scheme has been proposed by Lenstra in 2000 [42], shortly after the introduction of XTR. In this section we will introduce the algorithms provided by XTR-DSA. Before presenting the algorithms, we need to define the parameters of XTR-DSA.

1. p is a prime of about 170 bit size.
2. q is a prime dividing $\Phi_6(p) = p^2 - p + 1$ and chosen to be of 160 bit size.

The interested reader is refered to the Appendix (Section **A.1.1**), where we list the primes p and q, which are used by our implementation.

Our implementation of XTR-DSA provides three algorithms as required by Definition **6**. Generating a keypair is considered to be performed offline, therefore we implemented a keygenerator in C utilizing the GMP library. The signature generation and verification primitives were implemented in NesC.

4.2.1 Keypair generation

Not only does the keypair generation algorithm output the keypair, but also two primes of the required form. This algorithm requires computation of $S_k(c)$ for a random $k \in \mathbb{Z}_q$ and $c \in \mathbb{Z}_{p^2}$, which is performed by $\texttt{expHelper}(\cdot), \texttt{Op1}(\cdot)$, and $\texttt{Op2}(\cdot)$. We will describe these algorithms in Section **4.3**. For the key generation process it is important to know that $\texttt{expHelper}(\cdot)$ computes $(S0, S1, S2) = S_{k-1}(c)$ in case k is even and $(S0, S1, S2) = S_k(c)$ in case k is odd. We can derive $S_k(c)$ from $S_{k-1}(c)$ in three additional steps ($\texttt{Step1}, \texttt{Step2}, \texttt{Step3}$).

Algorithm 2: XTR-DSA: Keypair and prime generation

Input: Q: Length of q, P: Length of p
Output: Primes p, q, keypair $(k_{\texttt{priv}}, k_{\texttt{pub}})$
Find integer r such that $q = r^2 - r + 1$ is a Q-bit prime;
Find integer s such that $p = r + s \cdot q$ is a P-bit prime and $p \equiv 2 \bmod 3$;
Choose $x1, x2$ to be random elements in $\mathbb{Z}_p$;
Set generator, $c = x1 \cdot \alpha + x2 \cdot \alpha^2 \leftarrow \texttt{Tr}(g)$;
Choose a random integer $k_{\texttt{priv}} \in_R [2, q-3]$;
$(S0, S1, S2) \leftarrow (3, c, \texttt{Op2}(c))$; $r \leftarrow \lceil \texttt{log2}(k_{\texttt{priv}}) \rceil$;
Compute $(S0, S1, S2) \leftarrow \texttt{expHelper}(S0, S1, S2, k_{\texttt{priv}}, r)$;
if $k_{\texttt{priv}}$ *is even* **then**
Step1 $\texttt{swap}(S1, S0)$;
Step2 $\texttt{swap}(S1, S2)$;
Step3 $S2 \leftarrow \texttt{Op1}(S1, S0, c, S2)$;
$k_{\texttt{pub}} \leftarrow (S0, S1, S2)$;
Return $(p, q, k_{\texttt{priv}}, k_{\texttt{pub}})$;

The keypair of the XTR-DSA scheme is defined by $(k_{\texttt{priv}}, k_{\texttt{pub}})$. XTR-DSA's double exponentiation algorithms require c and c_k to be known, because $S_k(c)$ (or $S_{k-1}(c)$) can be computed from them. However, there exist two variants: the first variant computes $S_k(c)$ before performing the actual exponentiation, the other algorithm computes $S_{k-1}(c)$ instead. To avoid this additional, "slightly more involved computation" [42] on the verifiers

Variant	Key	Value
1	$k_{\texttt{priv}}$	k
	$k_{\texttt{pub}}$	$S_k(c) = (\texttt{Tr}(g^{k-1}), \texttt{Tr}(g^k), \texttt{Tr}(g^{k+1})$
2	$k_{\texttt{priv}}$	k
	$k_{\texttt{pub}}$	$S_{k-1}(c) = (\texttt{Tr}(g^{k-2}), \texttt{Tr}(g^{k-1}), \texttt{Tr}(g^k)$

Table 4.1: The XTR-DSA keypair

side, we chose to include the surrounding powers of c_k (or c_{k-1} for the second variant) into the public key (see Table **4.1**).

4.2.2 Signature generation

An XTR-DSA signature can be generated using the following algorithm. Observe that we need to compute the power of a trace, i.e., we calculate $c_u = \texttt{Tr}(g^u)$ in the `Exp` step. We denote the "Secure Hashing Algorithm 1" [51] by $\texttt{SHA1}(\cdot)$, which is a hashing algorithm according to Definition **2** in Section **3.1.2**. By choosing q to be of 160 bit size, we rarely need to reduce the output of the hash function modulo q, because the output size is 20 bytes.

Algorithm 3: XTR-DSA: Signature generation [42, Alg. 5.41]

Input: The private key $k_{\texttt{priv}} = k$, message m
Aux. Input: Prime q, generator c
Output: XTR-DSA signature $S_{\texttt{XTR-DSA}} = (r, s)$
Select random integer $u \in_R [2, q-3]$;
Exp Exponentiate, $x1 \cdot \alpha + x2 \cdot \alpha^2 \leftarrow \texttt{singleExp}(c, u)$;
$r = (x1 + p \cdot x2) \bmod q$;
if $r = 0$ **then**
  Restart algorithm;
Hash the message, $h \leftarrow \texttt{SHA1}(m) \bmod q$;
$s \leftarrow u^{-1} \cdot (h + k \cdot r) \bmod q$;
Return $S_{\texttt{XTR-DSA}} = (r, s)$;

Please note that we introduce a so-called "auxiliary input" for the notation of algorithms. By auxiliary, we refer to certain parameters that must be available to the algorithm, although we do not explicitly pass them as arguments when refering to it. This helps to focus on the important, mandatory parameters which are listed as normal "input" to the algorithm and always passed as arguments by us.

4.2.3 Signature verification

The following algorithm performs verification of XTR-DSA signatures. Depending on the variant of the exponentiation algorithms, we need to use $S_k(c)$ or $S_{k-1}(c)$ as public key. Signature verification requires the computation of $\texttt{Tr}(g^{b \cdot k+a})$, where $k = k_{\texttt{priv}}$ is unknown. The algorithm to perform this exponentiation is known as double exponentiation algorithm. We will show the algorithm to perfom this step in Section **4.3.1**.

Algorithm 4: XTR-DSA: Signature verification [42, Alg. 5.42]

Input: Public key $k_{\texttt{pub}} = S_{k-1}(c)$ or $S_k(c)$, message m, signature $S_{\texttt{XTR-DSA}} = (r, s)$
Aux. Input: Primes p, q
Output: $\texttt{ind} \in \{\text{"accept"}, \text{"reject"}\}$
if $(r, s) \leq 0$ **or** $(r, s) \geq q$ **then**
  Return "reject";
$w \leftarrow s^{-1} \bmod q$;
$h \leftarrow \texttt{SHA1}(m) \bmod q$;
$u1 \leftarrow w \cdot h \bmod q$; $u2 \leftarrow w \cdot r \bmod q$;
DblExp Double Exponentiate, $z1 \cdot \alpha + z2 \cdot \alpha^2 \leftarrow \texttt{dblExp}(k_{\texttt{pub}}, u1, u2) = \texttt{Tr}(g^{u2 \cdot k + u1})$;
$v \leftarrow (z1 + p \cdot z2) \bmod q$;
if $v = r$ **then**
  Return "accept";
else
  Return "reject";

4.3 XTR arithmetic

Having discussed the primitives of the signature scheme, we can now introduce the algorithms to perform single and double exponentiation. The algorithms presented here are based on Stam's paper on performance improvements for XTR [60]. We start with two double exponentiation methods, because two of the three single exponentiation algorithms are based on the second double exponentiation algorithm.

Algorithm 5: `expHelper`$(\cdot)$: Exponentiation helper algorithm [42, Alg. 2.3.7]

Input: $S0, S1, S2 \in \mathbb{Z}_{p^2}$, exponent d, threshold r
Aux. Input: Prime p
Output: $S0, S1, S2$
if d *is odd* **then**
  $e \leftarrow (d-1)/2$;
else
  $e \leftarrow (d/2) - 1$;
Write e as bitstring, $e = (e_{r-1}, \cdots, e_0)_2$;
for i *from* $r-1$ *downto* 0 **do**
  if $e_i = 1$ **then**
    Pow $S0 \leftarrow \texttt{Op1}(S2, c, S1, S0^p)$;
    $S1 \leftarrow \texttt{Op2}(S1)$; $S2 \leftarrow \texttt{Op2}(S2)$;
    $\texttt{swap}(S0, S1)$;
  else
    Pow $S2 \leftarrow \texttt{Op1}(S0, c^p, S1, S2^p)$;
    $S1 \leftarrow \texttt{Op2}(S1)$; $S0 \leftarrow \texttt{Op2}(S0)$;
    $\texttt{swap}(S2, S1)$;
Return $(S0, S1, S2)$;

For all algorithms presented in this section, a method we denote by `expHelper`$(\cdot)$ is required. The algorithm takes three elements $S0, S1, S2 \in \mathbb{Z}_{p^2}$, an exponent e, and a value r, indicating the highest bit in e as input. e is scanned beginning from e_{r-1} downto e_0 and $(S0, S1, S2)$ are modified accordingly. The final values of $S0 - S3$ are returned. Recall that computing the pth power (see the `Pow` steps in Algorithm **5**) of an element $c = x1 \cdot \alpha + x2 \cdot \alpha^2 \in \mathbb{Z}_{p^2}$ can be done by simply swapping $x1$ and $x2$. By `swap`$(\cdot)$ we denote swapping two $\mathbb{Z}_{p^2}$ elements which allows efficient computation of each exponentiation step without additional variables. For optimal performance, swapping operations should be implemented by swapping adresses of variables instead of their contents.

4.3.1 Double exponentiation

We have evaluated two double exponentiation algorithms. By `dblExp1`$(\cdot)$ we denote Stam's *matrix-less XTR double exponentiation* [60, Alg. 2.5], by `dblExp2`$(\cdot)$ we denote the *improved double exponentiation algorithm* [60, Alg. 3.1]. Recall that a double exponentiation algorithm is required by the verification method in order to compute $\texttt{Tr}(g^{u_2 \cdot k + u_1})$ where k is unknown.

Matrix-less XTR double exponentiation

The following algorithm represents a major improvement over Lenstra's double exponentiation algorithm [42, Alg. 5.27], as it does not require matrices for the computation. Removing the matrices does not improve the overall performance of the exponentiation step, but makes the implementation significantly easier. For `dblExp1`$(\cdot)$ the public key has to be given such that $k_{\texttt{pub}} = S_k(c) = (c_{k-1}, c_k, c_{k+1})$.

Algorithm 6: `dblExp1`$(\cdot)$: Matrix-less double exponentiation [60, Alg. 2.5]

Input: Public key $k_{\texttt{pub}} = (c_{k-1}, c_k, c_{k+1})$, exponents a, b
Aux. Input: Generator c, prime q
Output: $\texttt{Tr}(g^{b \cdot k + a})$
$r \leftarrow \lceil \texttt{log2}(q) \rceil - 1$; $d \leftarrow b/2^{r+1} \bmod q$; $t = a/d \bmod q$;
Step1 Init $(S0, S1, S2) \leftarrow (c_{k-1}, c_k, c_{k+1})$;
$S0 \leftarrow \texttt{Op1}(S2, c, S1, S0)$; $S1 \leftarrow \texttt{Op2}(S1)$; $S2 \leftarrow \texttt{Op2}(S2)$; $\texttt{swap}(S0, S1)$;
Exponentiate, $(S0, S1, S2) \leftarrow \texttt{expHelper}(S0, S1, S2, t, r)$;
if t *is even* **then**
  $\texttt{swap}(S1, S2)$;
Step2 $r \leftarrow \lceil \texttt{log2}(d) \rceil$;
Exponentiate, $(S0, S1, S2) \leftarrow \texttt{expHelper}(3, S1, \texttt{Op2}(S1), d, r)$;
if t *is even* **then**
  Return $S2$;
else
  Return $S1$;

The main "work" in this algorithm is performed by `expHelper`$(\cdot)$, so it is obvious that the helper method has to be implemented in a very optimized manner. The desired result is obtained in two steps. First $c_{2^{r+1} \cdot k + t \bmod q}$ is computed from $S_k(c)$. In the second step, $c_{d \cdot (2^{r+1} \cdot k + t) \bmod q}$ is computed, where

$$c_{d \cdot (2^{r+1} \cdot k + t) \bmod q} = c_{\frac{b}{2^{r+1}} \cdot (2^{r+1} \cdot k + a \cdot \frac{2^{r+1}}{b}) \bmod q} \stackrel{!}{=} c_{b \cdot k + a \bmod q}.$$

Improved double exponentiation

Stam also introduces a further improved version of his double exponentiation algorithm. Although Stam describes the more generalized computation of $c_{b \cdot k + a \cdot l}$, we tailor this generalization for our requirements by assuming that $l = 1$ is fixed.

In the new algorithm, $c_{b \cdot k + a}$ is computed in one iteration (compared to the two calls of `expHelper`$(\cdot)$ in Algorithm **6**). $c_{b \cdot k + a}$ is computed based on $k_{\texttt{pub}} = S_{k-1}(c)$ by maintaining and continously decreasing two variables $d = b$ and $e = a$. d and e are updated according to a variant of the continued fraction method [49] which may be applied to solve second degree recurrences such as the *Lucas sequence*, which is a generalization of the *Fibonacci sequence*.

Definition 18 (Lucas sequence [49, p. 1]) *For two relative prime numbers* p, q *satisfying* $p^2 - 4 \cdot q \neq 0$, *the* Lucas sequence $L_n(p,q)$ *is defined as* $L_0(p,q) = 0, L_1(p,q) = 1$, *and the recurrent relation* $L_n(p,q) = p \cdot L_{n-1}(p,q) + q \cdot L_{n-2}(p,q)$ *for* $n > 1$.

Definition 19 (Fibonacci sequence [49, p. 1]) *The* Fibonacci sequence F_n *is given by the* Lucas sequence *with* $p, q = 1$. $F_n = L_n(1,1)$ *is defined as* $F_0 = 0, F_1 = 1$, *and* $F_n = F_{n-1} + F_{n-2}$ *for* $n > 1$.

The following algorithm operates on a tuple $(d, e, S0, S1, S2, S3)$, where $S0 - S3$ are elements in $\mathbb{Z}_{p^2}$. For better readability, we have broken the whole algorithm into several smaller pieces. The sub-algorithms used to update the afore mentioned tuple are denoted by `update1`$(\cdot)$-`update6`$(\cdot)$. Algorithm **7** combines the sub-algorithms to the *improved double exponentiation* method.

Algorithm 7: `dblExp2`$(\cdot)$: Improved double exponentiation [60, Alg. 3.1]

Input: Public key $k_{\texttt{pub}} = (c_{k-2}, c_{k-1}, c_k)$, exponents a, b
Aux. Input: Generator c, primes p, q
Output: $\texttt{Tr}(g^{b \cdot k + a})$
Init $(S0, S1, S2, S3) \leftarrow (c_k, c, c_{k-1}, c_{k-2})$; $d \leftarrow b$; $e \leftarrow a$;
while d *is even* **and** e *is even* **do**
  $d \leftarrow d/2$; $e \leftarrow e/2$; $f2 \leftarrow f2 + 1$;
while d *divisible by* 3 **and** e *is divisible by* 3 **do**
  $d \leftarrow d/3$; $e \leftarrow e/3$; $f3 \leftarrow f3 + 1$;
while $d \neq e$ **do**
  if $d > e$ **then**
    if $d \leq 4 \cdot e$ **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update1`$(d, e, S0, S1, S2, S3)$;
    else if d *is even* **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update2`$(d, e, S0, S1, S2, S3)$;
    else if e *is odd* **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update3`$(d, e, S0, S1, S2, S3)$;
    else
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update4`$(d, e, S0, S1, S2, S3)$;
  if $e > d$ **then**
    if $e \leq 4 \cdot d$ **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update5`$(d, e, S0, S1, S2, S3)$;
    else if e *is even* **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update4`$(d, e, S0, S1, S2, S3)$;
    else if d *is odd* **then**
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update6`$(d, e, S0, S1, S2, S3)$;
    else
      $(d, e, S0, S1, S2, S3) \leftarrow$ `update2`$(d, e, S0, S1, S2, S3)$;
$S1 \leftarrow$ `Op1`$(S0, S2, S1, S3)$; $r \leftarrow \lceil \texttt{log2}(d) \rceil$;
Exponentiate, $(S0, S1, S2) \leftarrow$ `expHelper`$(3, S1, \texttt{Op2}(S1), d, r)$;
if d *is even* **then**
  `swap`$(S1, S2)$;
for i *from* 1 *up to* $f2$ **do**
  $S1 \leftarrow$ `Op2`$(S1)$;
for i *from* 1 *up to* $f3$ **do**
  $S1 \leftarrow$ `Op3`$(S1)$;
Return $S1$;

Algorithm 8: `update1`$(\cdot)$: Update algorithm 1 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: Updated $(d, e), (S0, S1, S2, S3)$
`swap`(d, e); `swap`$(S3, S2)$; `swap`$(S2, S1)$; `swap`$(S1, S0)$;
$S0 \leftarrow$ `Op1`$(S1, S3, S2, S0)$; $S3 \leftarrow S3^p$; $e \leftarrow e - d$;
Return $(d, e), (S0, S1, S2, S3)$;

Algorithm 9: `update2`$(\cdot)$: Update algorithm 2 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: Updated $(d, e), (S0, S1, S2, S3)$
`swap`$(S3, S2)$; $S2 \leftarrow S2^p$; $S2 \leftarrow$ `Op1`$(S0, S1, S3, S2)$;
$d \leftarrow d/2$; $S3 \leftarrow$ `Op2`$(S3)$; $S0 \leftarrow$ `Op2`$(S0)$;
Return $(d, e), (S0, S1, S2, S3)$;

Algorithm 10: `update3`$(\cdot)$: Update algorithm 3 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: Updated $(d, e), (S0, S1, S2, S3)$
$d \leftarrow (d - e)/2$; `swap`$(S1, S3)$; $S1 \leftarrow$ `Op1`$(S0, S2, S3, S1)$;
$S0 \leftarrow$ `Op2`$(S0)$; $S3 \leftarrow$ `Op2`$(S3)$; $S3 \leftarrow S3^p$;
Return $(d, e), (S0, S1, S2, S3)$;

Algorithm 11: `update4`$(\cdot)$: Update algorithm 4 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: Updated $(d, e), (S0, S1, S2, S3)$
`swap`(d, e); `swap`$(S0, S1)$; `swap`$(S2, S3)$; $d \leftarrow d/2$;
$S0 \leftarrow$ `Op2`$(S0)$; $S3 \leftarrow S3^p$; $S2 \leftarrow S2^p$; $S3 \leftarrow$ `Op2`$(S3)$;
Return $(d, e), (S0, S1, S2, S3)$;

Algorithm 12: `update5`$(\cdot)$: Update algorithm 5 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Output: Updated $(d, e), (S0, S1, S2, S3)$
$e \leftarrow e - d$; `swap`$(S0, S3)$; `swap`$(S2, S3)$; $S0 \leftarrow$ `Op1`$(S2, S3, S1, S0)$;
Return $(d, e), (S0, S1, S2, S3)$;

Algorithm 13: `update6`$(\cdot)$: Update algorithm 6 for improved double exponentiation

Input: $d, e \in \mathbb{Z}_q$ and $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: Updated $(d, e), (S0, S1, S2, S3)$
`swap`(e, d); $d \leftarrow (d - e)/2$; `swap`$(S0, S1)$; `swap`$(S1, S3)$;
$S1 \leftarrow$ `Op1`$(S3, S2, S0, S1)$; $S3 \leftarrow$ `Op2`$(S3)$; $S2 \leftarrow S2^p$; $S3 \leftarrow S3^p$; $S0 \leftarrow$ `Op2`$(S0)$;
Return $(d, e), (S0, S1, S2, S3)$;

4.3.2 Single exponentiation

We have implemented three single exponentiation algorithms. The first algorithm is based on the `expHelper`$(\cdot)$ method, while the second and third method are based on the more complex *improved double exponentiation* method. The algorithms to compute `Tr`(g^e) are denoted by `singleExp1`$(\cdot)$, `singleExp2`$(\cdot)$, and `singleExp3`$(\cdot)$.

Basic single exponentiation

The main exponentiation steps in this algorithm are performed by `expHelper`$(\cdot)$. We only have to take care that for an even exponent e, the resulting tuple $(S0, S1, S2)$ represents $S_{e-1}(c)$. In case e is odd we have $S1 = c_e$, in the afore mentioned case $S2 = c_e$. The original algorithm [43] computes all three powers $S_e(c) = (c_{e-1}, c_e, c_{e+1})$ from $S_{e-1}(c)$. However, we do not need $S_e(c)$, so we only take care of finding $c_e = \mathtt{Tr}(g^e)$.

Algorithm 14: `singleExp1`$(\cdot)$: Single exponentiation algorithm [60, Alg. 2.4]

Input: Generator c, exponent e
Output: $\mathtt{Tr}(g^e)$
Init, $(S0, S1, S2) \leftarrow (3, c, \mathtt{Op2}(c))$; $r \leftarrow \lceil \mathtt{log2}(e) \rceil$;
$(S0, S1, S2) \leftarrow \mathtt{expHelper}(S0, S1, S2, e, r)$;
if *e is even* **then**
  Return $S2$;
else
  Return $S1$;

Improved single exponentiation

The *improved single exponentiation algorithm* makes use of the *improved double exponentiation algorithm*. It is obvious that by choosing $b \cdot k + a = e$, we can compute $\mathtt{Tr}(g^{b \cdot k + a}) = \mathtt{Tr}(g^e)$. The freedom of choice for a, b, and k is exploited in such a way that Algorithm **7** favors those `update`$(\cdot)$ methods that require only few operations, while quickly decreasing d and e. The desired behaviour is achieved by choosing $k = 1$, therefore we need only

$$S_{k-1}(c) \overset{\text{k=1}}{=} S_0(c) = (c^p, 3, c)$$

for the exponentiation. In the next step, a and b are chosen such that $a = \frac{3-\sqrt{5}}{2} \cdot e$ and $b = e - a$. Observe that by choosing a and b in this specific manner, we have a ratio of

$$\frac{b}{a} = \frac{e - \frac{3-\sqrt{5}}{2} \cdot e}{\frac{3-\sqrt{5}}{2} \cdot e} = \frac{1 - \frac{3-\sqrt{5}}{2}}{\frac{3-\sqrt{5}}{2}} = \frac{\sqrt{5}-1}{3-\sqrt{5}} \approx 1.618033989...$$

The resulting irrational number is known as the "golden ratio" Φ, which is closely related to the *Fibonacci sequence* (see Definition **19**), because $F_{n+1}/F_n \approx \Phi$ (for a large n). By choosing $\frac{b}{a} = \frac{b+a}{a} = \Phi$, sub-algorithm `update1`$(\cdot)$ is repeatedly applied. Setting $e = d - e$ and $d = e$ (with $d = b$ and $e = a$ initially) as done by `update1`$(\cdot)$ still conserves this ratio, because updating d and e this way equals a step back in the *Fibonacci sequence*. That means, given $\tilde{F}_n = d$ and $\tilde{F}_{n-1} = e$ we set

$$e = \tilde{F}_{n-2} = \tilde{F}_n - \tilde{F}_{n-1} = d - e$$

and $d = \tilde{F}_{n-1}$ first. In the next iteration of the for-loop the algorithm computes

$$e = \tilde{F}_{n-3} = \tilde{F}_{n-1} - \tilde{F}_{n-2} = d - e$$

and $d = \tilde{F}_{n-2}$ and so on. Ideally, this would go on until $d = 0$ and $e = 1$ (or vice versa) – however, due to unavoidable rounding errors, this behaviour is lost at some point.

Algorithm 15: `singleExp2`$(\cdot)$: Improved single exponentiation [60, Alg. 5.1]

Input: Generator c, exponent e
Aux. Input: $\delta = 2^{160}$, $\gamma = \lfloor \frac{3-\sqrt{5}}{2} \cdot \delta \rfloor$
Output: $\mathtt{Tr}(g^e)$
$a \leftarrow (\gamma \cdot e)/\delta$; $b = e - a$;
$S1 \leftarrow \mathtt{dblExp2}(c^p, 3, c, a, b)$;
Return $S1$;

In order to avoid floating-point arithmetic while deriving a and b, we have pre-computed the values $\delta = 2^{160}$ and $\gamma = \lfloor \frac{3-\sqrt{5}}{2} \cdot \delta \rfloor$. The choice of δ depends on the bit size of q (here, q is chosen to have 160 bits). Dividing $(\gamma \cdot e)$ by δ is free, since we can perform 20 8-bit right-shifts instead of dividing by 2^{160}. This way

$$a = \lfloor ((\lfloor \frac{3-\sqrt{5}}{2} \cdot \delta \rfloor) \cdot e)/\delta \rfloor \approx \frac{3-\sqrt{5}}{2} \cdot e$$

can be obtained by integer arithmetic. Note that the we pass the tuple

$$(c_{k-2}, c_{k-1}, c_k) \overset{\text{k=1}}{=} (c_{-1}, c_0, c) = (c^p, 3, c)$$

as the "public key" to $\texttt{dblExp2}(\cdot)$.

Single exponentiation with pre-computation

Stam also proposed a method that involves pre-computing $S_{t-1}(c)$ for $t = 2^{\lceil \texttt{log2}(q)/2 \rceil}$ once and storing it for all following single exponentiations. S_{t-1} can be generated with the following algorithm.

Algorithm 16: Precomputation algorithm for single exponentiation [60, Alg. 4.1]

Input: Generator c
Aux. Input: Prime q
Output: $S_{t-1}(c)$
$r \leftarrow \lceil \texttt{log2}(q)/2 \rceil; \quad t \leftarrow 2^r - 1;$
$(S0, S1, S2) \leftarrow \texttt{expHelper}(3, c, \texttt{Op2}(c), t, r);$
Return $(S0, S1, S2)$;

Given the pre-computed values, the actual exponentiation method is supposed to be significantly faster than the two previous algorithms. Recall that Algorithm **7** computes $\texttt{Tr}(g^{b \cdot k+a})$, but requires $S_{k-1}(c)$ as input. Instead of computing $\texttt{Tr}(g^{b \cdot k+a})$ based on $S_{k-1}(c)$, we can alternatively compute $\texttt{Tr}(g^{b \cdot t+a})$ based on $S_{t-1}(c)$ with $\texttt{dblExp2}(\cdot)$. With t fixed to $2^{\lceil \texttt{log2}(q)/2 \rceil}$, we exploit the freedom of choice for a and b and set $b \cdot t + a = e$. With e and t known, a and b can be derived by division with remainder, i.e., computing e/t where b is the quotient and a the remainder of the division.

Algorithm 17: $\texttt{singleExp3}(\cdot)$: Single exponentiation with $S_{t-1}(c)$ given [60, Alg. 4.2]

Input: Generator c, exponent e
Aux. Input: t, pre-computed values $S_{t-1}(c) = (c_{t-2}, c_{t-1}, c_t)$
Output: $\texttt{Tr}(g^e)$
Compute a, b using division with remainder such that $e = b \cdot t + a$;
if $a = 0$ **then**
  $S1 \leftarrow \texttt{singleExp2}(c_t, b)$
else if $b = 0$ **then**
  $S1 \leftarrow \texttt{singleExp2}(c, a)$
else
  $S1 \leftarrow \texttt{dblExp2}(c_{t-2}, c_{t-1}, c_t, a, b);$
Return $S1$;

The smaller a and b are, the faster the algorithm proceeds. Due to the choice of t, the operands a and b have a size of about $\frac{1}{2} \cdot \lceil \texttt{log2}(e) \rceil$ bits. We can expect that, in this case, $\texttt{dblExp2}(\cdot)$ is twice as fast compared to a "normal" double exponentiation.

4.3.3 Op1, Op2, and Op3

With respect to the overall performance, the operations $\texttt{Op1}(\cdot)$ - $\texttt{Op3}(\cdot)$ and the underlying field arithmetic are most critical. In order to make the following operations as fast as

possible, we have substituted modular multiplication by *Montgomery multiplication* with integrated reduction. Due to the integration of multiplication and reduction, we did not incorporate the idea of separating and delaying the reduction steps as proposed by Stam [60]. Consequently, the following algorithms are a direct implementation of Equations **4.1**–**4.7**, but with modular multiplication replaced by *Montgomery multiplication*.

Algorithm 18: `Op1`$(\cdot)$: Special operation 1 for XTR layer arithmetic

Input: $x, y, z, w \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: $r = x \cdot z - y \cdot z^p + w$
$t1 \leftarrow \texttt{modAdd}(x2, y2, p)$; $t2 \leftarrow \texttt{modSub}(y1, t1, p)$;
$r1 \leftarrow \texttt{montMul}(z1, t2, p)$; $t2 \leftarrow \texttt{modSub}(t1, x1, p)$;
$t2 \leftarrow \texttt{montMul}(t2, z2, p)$; $r1 \leftarrow \texttt{modAdd}(t2, r1, p)$;
$r1 \leftarrow \texttt{modAdd}(r1, w1, p)$; $t1 \leftarrow \texttt{modAdd}(x1, y1, p)$;
$t2 \leftarrow \texttt{modSub}(y2, t1, p)$; $r2 \leftarrow \texttt{montMul}(z2, t2, p)$;
$t2 \leftarrow \texttt{modSub}(t1, x2, p)$; $t2 \leftarrow \texttt{montMul}(t2, z1, p)$;
$r2 \leftarrow \texttt{modAdd}(t2, r2, p)$; $r2 \leftarrow \texttt{modAdd}(r2, w2, p)$;
Return $r = r1 \cdot \alpha + r2 \cdot \alpha^2$;

Algorithm 19: `Op2`$(\cdot)$: Special operation 2 for XTR layer arithmetic

Input: $x = x1 \cdot \alpha + x2 \cdot \alpha^2 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: $r = x^2 - 2 \cdot x^p$
$r1 \leftarrow \texttt{modSub}(x2, x1, p)$; $r1 \leftarrow \texttt{modSub}(r1, x1, p)$;
$r1 \leftarrow \texttt{montMul}(r1, x2, p)$; $r2 \leftarrow \texttt{modSub}(x1, x2, p)$;
$r2 \leftarrow \texttt{modSub}(r2, x2, p)$; $r2 \leftarrow \texttt{montMul}(r2, x1, p)$;
$r1 \leftarrow \texttt{modSub}(r1, x2, p)$; $r1 \leftarrow \texttt{modSub}(r1, x2, p)$;
$r2 \leftarrow \texttt{modSub}(r2, x1, p)$; $r1 \leftarrow \texttt{modSub}(r2, x1, p)$;
Return $r = r1 \cdot \alpha + r2 \cdot \alpha^2$;

Algorithm 20: `Op3`$(\cdot)$: Special operation 3 for XTR layer arithmetic

Input: $x = x1 \cdot \alpha + x2 \cdot \alpha^2 \in \mathbb{Z}_{p^2}$
Aux. Input: Prime p
Output: $r = x^3 - 3 \cdot x^{p+1} + 3$
$r = r1 \cdot \alpha + r2 \cdot \alpha^2 \leftarrow \texttt{Op2}(x)$;
$r1 \leftarrow \texttt{modSub}(r1, x2, p)$; $r2 \leftarrow \texttt{modSub}(r2, x1, p)$;
$s \leftarrow \texttt{modAdd}(x1, x2, p)$; $t \leftarrow \texttt{modAdd}(r1, r2, p)$;
$u \leftarrow \texttt{montMul}(s, t, p)$; $s \leftarrow \texttt{montMul}(x1, r1, p)$;
$t \leftarrow \texttt{montMul}(x2, r2, p)$; $u \leftarrow \texttt{modSub}(u, s, p)$;
$u \leftarrow \texttt{modSub}(u, t, p)$; $r1 \leftarrow \texttt{modSub}(t, u, p)$;
Sub1 $r2 \leftarrow \texttt{modSub}(s, u, p)$; $r1 \leftarrow \texttt{modSub}(r1, 3, p)$;
Sub2 $r2 \leftarrow \texttt{modSub}(r2, 3, p)$;
Return $r = r1 \cdot \alpha + r2 \cdot \alpha^2$;

In these algorithms, we denote modular subtraction by `modSub`$(\cdot)$, modular addition by `modAdd`$(\cdot)$, and *Montgomery multiplication* by `montMul`$(\cdot)$. The algorithms for the modular arithmetic[1] are given in the following section.

4.4 Prime field arithmetic

Implementing fast and efficient arithmetic in $\mathbb{Z}_q$ and $\mathbb{Z}_p$ is the key to a fast overall implementation. Although performance optimizations on higher levels (such as the XTR layer) also add to an improved performance, the major gains in computational speed are obtained

[1] Note that the last three algorithms only perform arithmetic in $\mathbb{Z}_p$

by fast prime field arithmetic. In the following sections we present the basic arithmetic routines that perform multi-precision addition, subtraction, multiplication, and inversion.

4.4.1 Modular addition and subtraction

Modular subtraction and addition are trivially to implement when we take care that the inputs to the addition and subtraction method are always reduced by the modulus. The multi-precision integers are added/subtracted word-wise with respect to carry propagation. In case the addition result is bigger than the modulus, a single subtraction by the modulus is sufficient to perform reduction – in case the subtraction result is negative one addition of the modulus is sufficient.

We will now give the basic routines that perform addition and subtraction **without** reduction. From these methods we can construct the modular arithmetic. Note that $c_i, b_i, a_i, borrow, carry, u$, and v are of 8-bit size in our implementation. (u, v) represents a word of 16-bit size. Please note further that a and b are not necessarily of the same size and that c will have as many words as the largest operand.

Algorithm 21: `add`$(\cdot)$: Multi-precision addition

Input: $a = (a_{n-1}, \cdots, a_0)_{2^8}$, $b = (b_{m-1}, \cdots, b_0)_{2^8}$
Aux. Input: Sizes n, m
Output: $c = a + b$, last carry $carry$

$carry \leftarrow 0$;
Step1 **for** i *from* 0 *to* `MIN`$(n, m) - 1$ **do**
  $(u, v) \leftarrow (carry + a_i + b_i)$;
  $c_i \leftarrow v$; $carry \leftarrow u$;
Step2 **if** $n > m$ **then**
  for i *from* i *to* $n - 1$ **do**
    $(u, v) \leftarrow (carry + a_i)$;
    $c_i \leftarrow v$; $carry \leftarrow u$;
else if $m > n$ **then**
  for i *from* i *to* $m - 1$ **do**
    $(u, v) \leftarrow (carry + b_i)$;
    $c_i \leftarrow v$; $carry \leftarrow u$;
Return $c, carry$;

A multi-precision subtraction of $a - b$ is performed by adding b_i's two-complement representation to a_i. To derive this representation, our algorithm inverts all of b_i's bits by applying the `XOR`$(\cdot)$ function, which performs a logical 'exclusive-or' and yields b_i' This is the first of two steps to get the two-complement representation, the second step requires adding '1' to the b_i'. By initially setting $borrow = 1$, both steps are executed in `TwoComp1`, until the sum (u, v) is lower than 2^8, i.e., $borrow \leftarrow u \stackrel{!}{=} 0$. In case $u = 0$ occurs, this means that the result of $a_i - b_i$ is negative (i.e., $a_i < b_i$). According to standard schoolbook subtraction, the algorithm "borrows" a part of the next digit a_{i+1} of a. This is achieved by setting $borrow = 0$: only the first step of the two-complement generation is executed when computing $a_{i+1} - b_{i+1}$, therefore the algorithm propagates a "negative carry".

Algorithm 22: `sub`(·): Multi-precision subtraction

Input: $a = (a_{n-1}, \cdots, a_0)_{2^8}$, $b = (b_{m-1}, \cdots, b_0)_{2^8}$
Aux. Input: Sizes n, m
Output: $c = a - b = (c_{o-1}, \cdots, c_0)_{2^8}$, last borrow *borrow*

```
          borrow ← 1;
Step1     for i from 0 to MIN(n, m) − 1 do
TwoComp1      (u, v) ← (borrow + a_i + XOR(b_i, 0xFF));
              c_i ← v;   borrow ← u;
Step2     if n > m then
              for i from i to n − 1 do
TwoComp2          (u, v) ← (borrow + a_i + 0xFF);
                  c_i ← v;   borrow ← u;
          else if m > n then
              for i from i to m − 1 do
TwoComp3          (u, v) ← (borrow + XOR(b_i, 0xFF));
                  c_i ← v;   borrow ← u;
          Return c, borrow;
```

Algorithm **21** and Algorithm **22** proceed in two steps: first, the corresponding words of a and b are added/subtracted, until either a or b has no remaining words (i.e., $n \neq m$). In the second step, the carry/borrow is propagated to the $|n-m|$ remaining words of either a or b. This behaviour is visible in the subtraction algorithm: step `TwoComp2` propagates the *borrow* value to the remaining words of a_i, by using the two-complement representation $b'_i =$ `0xFF` of $b_i = 0$. In `TwoComp3`, the algorithm proceeds as previously explained, but respects that $a_i = 0$.

Given `add`(·) and `sub`(·) we can construct their modular counterparts easily.

Algorithm 23: `modAdd`(·): Modular multi-precision addition

Input: Operands $a = (a_{n-1}, \cdots, a_0)_{2^8}$, $b = (b_{m-1}, \cdots, b_0)_{2^8}$, $q = (q_{o-1}, \cdots, q_0)_{2^8}$
Aux. Input: Sizes n, m, o
Output: $c = a + b \bmod q$

```
      (c, carry) ← add(a, b);
Cmp   if (carry = 1 and n = o and m = o) or c > q then
          (c, borrow) ← sub(c, q);
      Return c;
```

Algorithm 24: `modSub`(·): Modular multi-precision subtraction

Input: Operands $a = (a_{n-1}, \cdots, a_0)_{2^8}$, $b = (b_{m-1}, \cdots, b_0)_{2^8}$, $q = (q_{o-1}, \cdots, q_0)_{2^8}$
Output: $c = a - b \bmod q$

```
      (c, borrow) ← sub(a, b);
Cmp   if borrow = 0 then
          (c, carry) ← add(c, q);
      Return c;
```

Observing the `Cmp` steps executed by Algorithm **23** and Algorithm **24** reveals that it is more efficient to test for a negative result than to test whether $c > q$ (which involves `MIN`$(o,$ `MAX`$(n, m))$ single word comparisons in the worst case).

The presented routines can be applied to any combination of values and lengths of the input operands. However, to really enhance the performance of our implementation, we decided to additionally implement these routines in highly specific ASM code, for the sake of speeding up `Op1`(·)-`Op3`(·)'s arithmetic in $\mathbb{Z}_p$.

4.4.2 Multiplication

Modular multiplication is the most time consuming arithmetic operation performed by `Op1`(·)-`Op3`(·). Consquently, we chose Montgomery multiplication rather than standard modular multiplication. However, we also need modular multiplication for arithmetic on the protcol layer. We will now present and discuss the properties of the implemented (Montgomery) multiplication methods.

Modular multiplication

We chose to implement the *Comba multiplication* algorithm [13], wich is the column-wise adaption of the schoolbook multiplication method. The Comba method is assumed to be up to 30% faster than the naive approach, which is due to its improved carry handling [13].

Algorithm 25: `mul`(·): Multi-precision multiplication with Comba's method

Input: $a = (a_{n-1}, \cdots, a_0)_{2^8}$, $b = (b_{m-1}, \cdots, b_0)_{2^8}$
Opt. Input: Threshold $s =$ `MAXVAL`
Aux. Input: Sizes n, m
Output: $c = a \cdot b = (c_{n+m-1}, \cdots, c_0)_{2^8}$ **or** $(c_{s-1}, \cdots, c_0)_{2^8}$

```
for i from 0 to MIN(s, n+m) - 1 do
    bo ← MIN(m - 1, i);   ao ← i - bo;
    for j from 0 to MIN(n - ao, bo + 1) - 1 do
MAC     (t, u, v) ← (t, u, v) + a_{ao+j} · b_{bo-j};
    c_i ← v;   v ← u;   u ← t;   t ← 0;
Return c;
```

By (t, u, v) we denote a 24-bit value, where t, u, and v are 8-bit values. The variable t holds the carry of the "Multiply-and-Add-with-Carry" (see `MAC` step in Algorithm **25**) operation. One iteration of the outer loop computes one resulting word c_i. This way, we only have to take care of propagating the carry t from c_0 to c_{n+m-1}. Note that we pass an optional "treshold" value to the multiplication algorithm, which may be used to limit the result to s words. The threshold is used only in the reduction step. In case s is not given by the caller, it is set to the maximum value (`MAXVAL`), so that it does not interfere with the end condition of the outer for-loop.

We implemented the *Barrett reduction* algorithm, which is applied to the result of the *Comba multiplication*. Normally, modular reduction of $a = (a_{2\cdot(k-1)}, \cdots, a_0)_b$ by $m = (m_{k-1}, m_{k-2}, \cdots, m_0)_b$ (with $m_{k-1} \neq 0$) can be achieved by dividing and returning the remainder[2]. However, as division is rather slow (division is not supported by the ATMEGA128L), Barrett had the idea of replacing $\lfloor a/m \rfloor$ by the following,

$$\lfloor a/m \rfloor = \lfloor \frac{a}{b^{k-1}} \cdot \underbrace{\frac{b^{2\cdot k}}{m}}_{\mu} \cdot \frac{1}{b^{k+1}} \rfloor. \tag{4.8}$$

Although the right part of Equation **4.8** looks more complicated, its computation is significantly faster than normal divsion – when two conditions are fulfilled. First, b has to be chosen such that $b - 1$ is the largest value that can be represented by a single word. The ATMEGA128L is an 8-bit micro-processor, so we chose $b = 2^8$. By choosing b in this specific way, we can perform division by b^{k-1} and b^{k+1} by right-shifting the dividend. Second, the quantity $\mu = b^{2\cdot k}/m$ has to be already computed (the modulo does not change, therefore μ is fixed), so that Equation **4.8** can be calculated by two right-shifts and one multiplication with μ.

[2] Note that we represent a and m as bitstrings of width b

Algorithm 26: `mod`$(\cdot)$: Modular reduction with Barrett's algorithm [46, Alg. 14.42]

Input: $a = (a_{2\cdot(k-1)}, \cdots, a_0)_{2^8}$, $m = (m_{k-1}, \cdots, m_0)_{2^8}$
Aux. Input: Precomputed quantity $\mu = 2^{8\cdot(2\cdot k)}/m$, size k
Output: $r = a \bmod m = (r_{k-1}, r_{k-2}, \cdots, r_0)_{2^8}$
Step1 $q1 \leftarrow \lfloor a/2^{8\cdot(k-1)} \rfloor$; $q2 \leftarrow q1 \cdot \mu$; $q3 \leftarrow \lfloor q2/2^{8\cdot(k+1)} \rfloor$;
Step2 $r1 \leftarrow a \bmod 2^{8\cdot(k+1)}$; $r2 \leftarrow \texttt{mul}(q3, m, k+1)$; $r \leftarrow r1 - r2$;
if $r < 0$ **then**
  $r \leftarrow r + 2^{8\cdot(k+1)}$;
while $r \geq m$ **do**
  $r \leftarrow r - m$;
Return r;

In `Step1`, we can perform the division by two right-shifts by $k-1$ and $k+1$ words respectively. Modulo reduction in `Step2` is performed by removing all of a's words above a_k. We also see that by using $\texttt{mul}(\cdot, k+1)$, our implementation of Comba's algorithm generates only the first $k+1$ words of $r2$. This is equal to computing $r2$ with a standard multiplication followed by a modulo reduction such that

$$r2 = q3 \cdot m \bmod 2^{8\cdot(k+1)},$$

but significantly faster.

Based on the foregoing, we constructed the modular multiplication in the following algorithm.

Algorithm 27: `modMul`$(\cdot)$: Modular multiplication

Input: Operands $a = (a_{k-1}, \cdots, a_0)_{2^8}, b = (b_{k-1}, \cdots, b_0)_{2^8}$, $m = (m_{k-1}, \cdots, m_0)_{2^8}$
Output: $c \equiv a \cdot b \bmod m = (c_{k-1}, c_{k-2}, \cdots, c_0)_{2^8}$
$c \leftarrow \texttt{mul}(a, b)$;
$c \leftarrow \texttt{mod}(c, m)$;
Return c;

Montgomery multiplication

A single application of Peter Montgomery's multiplication algorithm [48] is not faster than our normal modular multiplication as in Algorithm **27**. This is due to the fact that *Mongomery multiplication* is performed on transformations of the respective operands instead of their given values. A complete *Montgomery multiplication* involves transforming operands, multiplying them, and transforming the result back. The overhead of (re-)transforming these integers makes *Montgomery multiplication* slower than modular multiplication. However, there are certain applications where the transformation operation is followed by a sequence of multiplications of the transformed integers (such as in modular exponentiation) – in this case, *Montgomery multiplication* is superior.

We first explain Montgomery's idea and subsequently we emphasize how it can be applied to our implementation. Let $a, b < m$ be the integers we want to multiply modulo m. m is an l-bit modulus, i.e., $2^{l-1} < m < 2^l$. Let $R = 2^l$, we call

$$\tilde{a} \equiv a \cdot R \bmod m, \quad \tilde{b} \equiv b \cdot R \bmod m$$

the Montgomery representations of a and b where multiplying with R is the transformation step. When $\texttt{GCD}(R, m) = 1$ is given, the set

$$\{a \cdot R \bmod m | 0 \leq a < m\}$$

contains all numbers between 0 and $m-1$. There is a one-to-one correspondence between the numbers in the interval $[0, m-1]$ and the numbers in the above set. Montgomery's

reduction step exploits this correspondence by introducing a much faster reduction. Given two integers $\tilde{a}$ and $\tilde{b}$ the *Montgomery multiplication* algorithm computes

$$\tilde{c} \equiv \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m \equiv a \cdot R \cdot b \cdot R \cdot R^{-1} \bmod m \equiv a \cdot b \cdot R \bmod m,$$

where $R^{-1} \cdot R \equiv 1 \bmod m$. The next algorithm gives the basic idea of the Montgomery algorithm. Prior to this, we have to introduce another integer m' such that $R \cdot R^{-1} - m' \cdot m \equiv 1 \bmod m$.

Algorithm 28: Montgomery multiplication (sketch)

Input: $\tilde{a}, \tilde{b}$, modulus m
Aux. Input: Precomputed quantites m', R
Output: $\tilde{c} \equiv \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m$
$\tilde{t} \leftarrow \tilde{a} \cdot \tilde{b}$;
$\tilde{c} \leftarrow (\tilde{t} + (\tilde{t} \cdot m' \bmod R) \cdot m)/R$;
if $\tilde{c} \geq m$ **then**
  $\tilde{c} \leftarrow \tilde{c} - m$;
Return $\tilde{c}$;

R was chosen as a power of two, so multiplication and division are operations that can be performed by left- and right-shifting respectively. Therefore, *Montgomery multiplication* is assumed to be faster than our modular multiplication algorithm `modMul`$(\cdot)$.

Our implementation of XTR's arithmetic uses `Op1`$(\cdot)$ − `Op3`$(\cdot)$ for exponentiation. These routines only operate on a maximum of four elements $S0, S1, S2, S3 \in \mathbb{Z}_{p^2}$. By transforming the initial values of $S0, S1, S2$, and $S3$ into Montgomery representation we can perform addition, subtraction, and *Montgomery multiplication* successively, because adding

$$\tilde{c} \equiv \tilde{a} + \tilde{b} \bmod m \equiv a \cdot R + b \cdot R \bmod m \equiv (a+b) \cdot R \bmod m \equiv c \cdot R \bmod m,$$

and subtracting

$$\tilde{c} \equiv \tilde{a} - \tilde{b} \bmod m \equiv a \cdot R - b \cdot R \bmod m \equiv (a-b) \cdot R \bmod m \equiv c \cdot R \bmod m,$$

does not destroy the Montgomery representation. After the main exponentiation step has finished, our algorithms only need to transform the result back to normal representation.

We have evaluated two *Montgomery multiplication* methods with integrated reduction: the *Coarsely Integrated Operand Scanning* (CIOS) and the *Finely Integrated Product Scanning* (FIPS) method. Both algorithms were implemented based on a paper by Koç *et al.* [39]. The authors compare several variants of *Montgomery multiplication* algorithms and finally recommend to use the CIOS method (at least on their general purpose processor).

Algorithm 29: `montMul1`$(\cdot)$: CIOS-Montgomery multiplication [39]

Input: $\tilde{a} = (\tilde{a}_{k-1}, \cdots, \tilde{a}_0)_{2^8}, \tilde{b} = (\tilde{b}_{k-1}, \cdots, \tilde{b}_0)_{2^8}$, modulus m
Aux. Input: Montgomery quantity m'_0, size k
Output: $\tilde{c} \equiv \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m = (\tilde{c}_{k-1}, \cdots, \tilde{c}_0)_{2^8}$
Init $\tilde{c} = (\tilde{c}_{k+1}, \tilde{c}_k, \cdots, \tilde{c}_0)_{2^8} \leftarrow 0$; $(u, v) \leftarrow 0$;
for *i from* 0 *to* $k-1$ **do**
  Loop1 — **for** *j from* 0 *to* $k-1$ **do**
    $(u, v) \leftarrow u + \tilde{a}_j \cdot \tilde{b}_i + \tilde{c}_j$;
    $\tilde{c}_j \leftarrow v$;
  $(u, v) \leftarrow \tilde{c}_k + u$; $\tilde{c}_k \leftarrow v$; $\tilde{c}_{k+1} \leftarrow u$;
  Mod — $n \leftarrow \tilde{c}_0 \cdot m'_0 \bmod 2^8$; $(u, v) \leftarrow \tilde{c}_0 + n \cdot m_0$;
  Loop2 — **for** *j from* 1 *to* $k-1$ **do**
    $(u, v) \leftarrow u + n \cdot m_j + \tilde{c}_j$;
    RShift — $\tilde{c}_{j-1} \leftarrow v$;
  $(u, v) \leftarrow \tilde{c}_k + u$; $\tilde{c}_{k-1} \leftarrow v$; $\tilde{c}_k = \tilde{c}_{k+1} + u$;
Return $\tilde{c} = (\tilde{c}_{k-1}, \tilde{c}_{k-2}, \cdots, \tilde{c}_0)_{2^8}$;

Algorithm 30: `montMul2`$(\cdot)$: FIPS-Montgomery multiplication [39]

Input: $\tilde{a} = (\tilde{a}_{k-1}, \cdots, \tilde{a}_0)_{2^8}, \tilde{b} = (\tilde{b}_{k-1}, \cdots, \tilde{b}_0)_{2^8}$, modulus m
Aux. Input: Montgomery quantity m'_0, size k
Output: $\tilde{c} \equiv \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m = (\tilde{c}_{k-1}, \cdots, \tilde{c}_0)_{2^8}$
Init $\tilde{c} = (\tilde{c}_k, \tilde{c}_{k-1}, \cdots, \tilde{c}_0)_{2^8} \leftarrow 0$; $(t, u, v) \leftarrow 0$;
for *i from* 0 *to* $k-1$ **do**
  for *j from* 0 *to* $i-1$ **do**
    $(t, u, v) \leftarrow (t, u, v) + \tilde{a}_j \cdot \tilde{b}_{i-j}$;
    $(t, u, v) \leftarrow (t, u, v) + \tilde{c}_j \cdot m_{i-j}$;
  Mod $(t, u, v) \leftarrow (t, u, v) + \tilde{a}_i \cdot \tilde{b}_0$; $\tilde{c}_i \leftarrow v \cdot m'_0 \bmod 2^8$;
  $(t, u, v) \leftarrow (t, u, v) + \tilde{c}_i \cdot m_0$;
  $v \leftarrow u$; $u \leftarrow t$; $t \leftarrow 0$;
for *i from* k *to* $2 \cdot k - 1$ **do**
  for *j from* $i - k + 1$ *to* $k-1$ **do**
    $(t, u, v) \leftarrow (t, u, v) + \tilde{a}_j \cdot \tilde{b}_{i-j}$;
    $(t, u, v) \leftarrow (t, u, v) + \tilde{c}_j \cdot m_{i-j}$;
  $\tilde{c}_{i-k} \leftarrow v$;
  $v \leftarrow u$; $u \leftarrow t$; $t \leftarrow 0$;
Return $\tilde{c} = (\tilde{c}_{k-1}, \tilde{c}_{k-2}, \cdots, \tilde{c}_0)_{2^8}$;

The listed algorithms have been adapted for our purposes. We have implemented them with respect to the micro-processor's operand size of 8 bit. Similiar to our implementation of Comba's algorithm, we denote a 24-bit tuple by (t, u, v). Note that the `Mod` steps in Algorithm **29** and **30** are not explicitly implemented, because all variables can only store 8 bits – so reduction is rather implicit. Both algorithms interleave a multiplication step with a reduction step, therefore c remains quite small (as opposed to normal multiplication). Although both methods have some common properties, there are also major differences. The CIOS method calculates $\tilde{c} = \tilde{a} \cdot \tilde{b}_i$ in `Loop1`. `Loop2` applies the reduction step to $\tilde{c}$, here $m_j \cdot n = m_j \cdot (\tilde{c}_0 \cdot m'_0 \bmod 2^8)$ are the coefficients of $(t \cdot m' \bmod R) \cdot m$. These coefficients are directly added to $\tilde{c}_j$. Division by R is applied by writing the result of the reduction step to $\tilde{c}_{j-1}$, thereby performing one leftshift in every iteration of `Loop2`. The FIPS method computes the columns of c – this approach is comparable to the *Comba multiplication* presented earlier. In addition, the FIPS method comprises of four loops, which suggests an added overhead compared to the CIOS method.

We also implemented both methods in ASM, optimized for use with a modulus p of 22 words size. The CIOS method was implemented according to Algorithm **29**. For the FIPS method we chose to generate ASM code by an additional computer program, which automatically unrolls all loops. Our program generates a rather long ASM listing with the advantage of no added overhead due to loop control.

4.4.3 Modular inversion

Modular inversion is required infrequently, the core XTR arithmetic does not need inversion at all. We implemented the *binary inversion algorithm* which is a variant of Euclid's algorithm to compute the greatest common divisor (GCD) of two numbers a and b. The algorithm is based on four observations:

1. If a and b are even, $\mathtt{GCD}(a, b) = 2 \cdot \mathtt{GCD}(a/2, b/2)$.
2. If a is odd and b is even, $\mathtt{GCD}(a, b) = \mathtt{GCD}(a, b/2)$.
3. If a is even and b is odd, $\mathtt{GCD}(a, b) = \mathtt{GCD}(a/2, b)$.
4. If both numbers are odd, $(a - b)$ is even, $\mathtt{GCD}(a, b) = \mathtt{GCD}(a, (a - b)/2)$.

Repeatedly applying these rules to an integer a and a modulus m is a fast way to derive the modular inverse $a^{-1} \bmod m$.

Note that Algorithm **31** uses another "trick": in case $x1$ or $x2$ is odd, we can make them even again by adding the odd modulus. We can do this because we effectively perform halving modulo m, thus we do not change $x1$'s or $x2$'s value modulo m by adding the modulus. The halving step performed afterwards ensures that $x1, x2 \leq m$.

Algorithm 31: `modInvert`$(\cdot)$: Binary inversion method [46, Alg. 14.62]

```
Input: Integer a < m, modulus m
Output: a^-1 mod m
u ← a;  v ← m;  x1 ← 1;  x2 ← 0;
while u ≠ 1 and v ≠ 1 do
    while u is even do
        u ← u/2;
        if x1 is odd then                      (Trick)
            x1 ← x1 + m;
        x1 ← x1/2;
    while v is even do
        v ← v/2;
        if x2 is odd then                      (Trick)
            x2 ← x2 + m;
        x2 ← x2/2;
    if u > v then
        u ← u − v;  x1 ← x1 − x2;
    else
        v ← v − u;  x2 ← x2 − x1;
if u = 1 then
    Return x1;
else
    Return x2;
```

4.5 Results

We will now detail the benchmarking results of most of the implemented algorithms. While the preceding part of this chapter was presented from XTR-DSA layer to prime field arithmetic, we will now give the results in reverse order. The reason for this is that all upper layers depend on the lower layers. Therefore we start with evaluating the algorithms on the lowest layer and then benchmark the following layers based on the winning candidates.

4.5.1 Prime field arithmetic

First, we present the timing results for modular addition and subtraction. We have implemented two versions of both operations, one in NesC and one in ASM. The results are given for addition and subtraction of two 22-word integers modulo p. Due to the varying runtime of both operations (depending on whether the result has to be modulo reduced or not) we average the timings over ten measurements. Table **4.2** reveals that subtraction is faster than addition – as expected in Section **4.4.1**. This is most likely due to the fact that determining the actual "reduction step" (i.e., adding/subtracting the modulus) in both algorithms is of different complexity. We also see that implementing the arithmetic in ASM yields a siginificant gain in performance, the ASM routines are more then ten times faster.

Montgomery multiplication is the third operation responsible for the performance of `Op1`$(\cdot)$ − `Op3`$(\cdot)$. We have implemented the CIOS and FIPS method, both in NesC and

Method	Language	$t_{\texttt{average}}$
Modular addition	NesC	0.536ms
	ASM	0.048ms
Modular subtraction	NesC	0.512ms
	ASM	0.046ms

Table 4.2: Performance of modular addition/subtraction algorithms

ASM. The NesC-methods are very flexible and can be used with any combination of input operands, the ASM-arithmetic is fixed to primes with 22 words. We see the usual

Method	Language	$t_{\texttt{mul}}$
Montgomery CIOS-Multiplication	NesC	12.615ms
	ASM	2.087ms
Montgomery FIPS-Multiplication	NesC	10.733ms
	ASM	1.301ms

Table 4.3: Performance of Montgomery multiplication algorithms

difference between the ASM and NesC implementations. Interestingly, in contrast to the recommendation by Koç *et al.* [39], the CIOS method is slower than the FIPS method. This does not only affect the ASM variant, where we unrolled all loops, but also the NesC code.

4.5.2 XTR arithmetic

The arithmetic in `Op1`$(\cdot) -$ `Op3`$(\cdot)$ is performed based on the ASM versions of the addition, subtraction, and the FIPS multiplication method. We did not benchmark these atomic

Method	$t_{\texttt{precomp}}$	$t_{\texttt{exp}}$
Single exponentiation	-	2.053s
Improved single exponentiation	-	1.621s
Single exponentiation w/ pre-computation	0.977s	0.882s

Table 4.4: Performance of single exponentiation methods

operations, but the exponentiation algorithms built on top of them. The results for the single exponentiation algorithms are shown in Table **4.4**. Note that we also included the pre-computation time required by `singleExp3`$(\cdot)$. Stam expected a speedup of 60% compared to `singleExp1`$(\cdot)$. In our implementation, the third exponentiation algorithm is about 57% faster although we did not implement the "faster field arithmetic" [60].

Table **4.5** lists the performance of the two double exponentiation algorithms. We did not implement the first method proposed by Lenstra, due to its complicated nature. However, Stam estimates that the performance of the matrix-less algorithm is comparable to Lenstras matrix based method. The improved method is twice as fast as the first method. Compared with the single-exponentiation method with pre-computation, the double exponentiation method is half as fast. We already expected that, because in case of single exponentiation, the operands a and b are only half as long.

Method	$t_{\texttt{dblexp}}$
Matrix-less XTR double exponentiation	3.979s
Improved double exponentiation	1.920s

Table 4.5: Performance of double exponentiation methods

4.5.3 Overall performance of XTR-DSA

We have benchmarked our implementation of XTR-DSA's signature generation and verification primitives. The results were obtained by selecting the fastest prime field arithmetic and the best single and double exponentiation methods. The scheme has not been optimized in any way on the protocol layer. A complete evaluation of our implementation of

Method	t
XTR-DSA: Signature generation	0.965s
XTR-DSA: Signature verification	2.009s

Table 4.6: Overall performance of our XTR-DSA implementation

XTR-DSA will be given in comparison to the other asymmetric schemes in Chapter **7**.

4.6 Overview of optimizations

We have evaluated and implemented several algorithms and optimizations, the main optimizations are summarized in the following listing:

ASM Implementing all of `Op1`$(\cdot)-$`Op3`$(\cdot)$'s arithmetic in ASM boosts the overall performance of our implementation. However, due to the very specific optimizations for primes p of 22 words size, we cannot adapt these routines easily for primes of other sizes.

Montgomery representation Performing addition and subtraction in *Montgomery representation* does not cause any overhead compared to adding "normal" integers. The main reason for performing `Op1`$(\cdot)-$`Op3`$(\cdot)$'s arithmetic in Montgomery representation is that we can replace our normal modular multiplication by *Montgomery multiplication*, which is supposed to be faster.

Single Exponentiation Choosing the single exponentiation algorithm with pre-computation significantly accelerates our implementation. Precomputation of $S_{t-1}(c)$ is performed only once, so the costs can be neglected.

Double Exponentiation The improved double exponentiation algorithm is faster than the original proposal by Lenstra [43]. Additionally, it is much easier to implement.

Precomputations Precomputing the Montgomery representation of the generator c, the $\mathbb{Z}_{p^2}$-tuples represented by $S_k(c), S_{k-1}(c)$ or $S_{t-1}(c)$ eliminates the computational costs of the transformation step. We only have to transform the result of our exponentiation algorithms back to normal representation. We also pre-compute the Montgomery representation of $3 = -3 \cdot \alpha - 3 \cdot \alpha^2$. In addition, we pre-compute the required quantities μ, m' for Barrett's and Montgomery's reduction methods.

XTR is not as well examined as other asymmetric schemes (e.g. elliptic curve cryptography), so we could not find many optimization proposals. However, Stam presented some major improvements over the original algorithms, so the first step has been done. We suggest that, while optimizing the XTR's exponentiation algorithms is not trivial, it might be worthwhile to replace more of the NesC code by ASM routines.

5 ECDSA

In this chapter we treat the ECDSA signature scheme. First we give an introduction in Section **5.1**, then we present and discuss the details of our implementation and the evaluated algorithms in Section **5.2** and Section **5.3**. Finally we analyze the performance of our implementation and detail the applied optimizations in Section **5.5** and Section **5.6** respectively.

5.1 Introduction

The "Elliptic Curve Digital Signature Algorithm" (ECDSA) scheme is the standard signature scheme based on elliptic curve cryptography (ECC). Elliptic curves have been studied by number theorists since the middle of the nineteenth century. In 1985 (after the invention of asymmetric cryptography in 1976) Neal Koblitz and Victor Miller independently discovered the possibility to use elliptic curves for the purpose of asymmetric cryptography [38, 47]. The major advantage of ECC compared to the well-known RSA scheme [57] is the decreased key size and operand length. It is a well established fact that a 160-bit elliptic curve provides security equivalent to a 1024-bit RSA key. This property makes ECC perform better than RSA when memory requirements and computational speed are critical constraints, such as it is the case in WSNs.

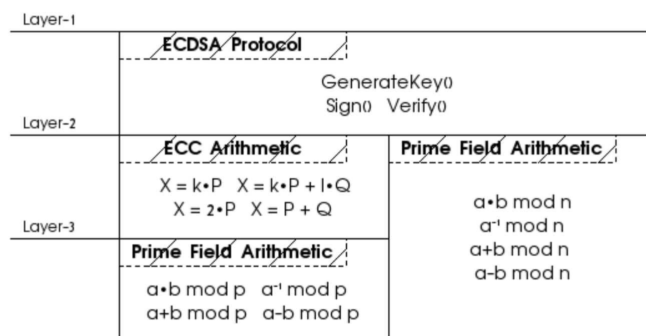

Figure 5.1: Three layers of ECDSA

Similarly to the XTR-DSA scheme, we can divide the ECDSA scheme into three layers. Arithmetic on the lowest layer is performed in a (finite) prime field modulo the prime p. This arithmetic is required by the second layer to perform mathematical operations on the elliptic curve. Based on the (scalar) point multiplication functions provided by the ECC layer, the ECDSA protocol is constructed. Figure **5.1** also reveals that the first layer additionally requires prime field arithmetic modulo n, where n is the prime order of the elliptic curve's basepoint.

After a short overview of the currently known implementations, we will introduce the fundamentals of elliptic curve cryptography in Section **5.1.2**.

5.1.1 Related work

The implementation presented in this chapter is based on the efforts of Uhsadel [69, 70]. Uhsadel implemented highly optimized ASM routines for the prime field arithmetic performed on Layer-3 (see Figure **5.1**) of the ECDSA scheme. His goal was to enable high-

speed ECC based schemes on the ATMEGA128L micro-processor. Uhsadel's code performes arithmetic in $\mathbb{Z}_p$, where the prime number p is defined as part of the recommended elliptic curve domain parameter set for 160-bit elliptic curves (SECP160R1) [61]. Modular multiplication in $\mathbb{Z}_p$ can be considered to be most critical for the overall performance, since 77% of a point multiplication is spent multiplying in $\mathbb{Z}_p$ on an 8-bit μP [27]. The "hybrid multiplication algorithm" implemented by Uhsadel requires 0.39ms to perform a single, non-modular 160-bit multiplication.

Uhsadels work was initially inspired by Gura *et al.* [27]. In 2004, Gura compared fast implementations of ECC and RSA. By implementing the Layer-3 and Layer-2 arithmetic in ASM, Gura achieved a multiplication of 160-bit field-elements in 0.42ms and a single point multiplication in 0.81s. The source-code of his implementation is not publicly available and is intellectual property of SUN Microsystems. Gura's implementation of ECC has later on been analyzed with respect to energy consumption [71] and a mote's lifetime [54].

A. Liu and P. Ning [45] have implemented not only ECC primitives, but the whole ECDSA scheme in NESC. Their software package ("TinyECC") is freely available – it supports several elliptic curves[1] and also the parameters defined by SECP160R1. In the latest version (0.3), TinyECC performs a signature generation in 1.925s, a verification is executed in 2.433s.

A more recent article by Wang and Li describes their implementation of ECC and RSA on MICAZ motes [72]. Wang *et al.* use the SECP160R1 parameter set for their ECC implementation. 160-bit integer multiplication is performed in 0.47ms. The ECDSA implementation based on their ECC library can generate signatures in 1.35s and verify signatures in 2.85s.

5.1.2 Mathematical background

This subsection introduces and illustrates the mathematical foundations and core arithmetic of elliptic curve cryptography.

Elliptic curves

An elliptic curve E over the prime field $\mathbb{Z}_p$ is defined by the *Weierstrass equation* [28, p. 76, Def. 3.1],

$$E(\mathbb{Z}_p) : y^2 + a_1 \cdot x \cdot y + a_3 \cdot y = x^3 + a_2 \cdot x^2 + a_4 \cdot x + a_6$$

where $a_1, a_2, a_3, a_4, a_5, a_6 \in \mathbb{Z}_p$. The simplified definition of elliptic curves is derived in several transformation steps under certain assumptions [28, pp. 78-79]. We will use the simplified equation of elliptic curves for the remainder of this chapter:

$$E(\mathbb{Z}_p) : y^2 = x^3 + a \cdot x + b, \mathtt{char}(\mathbb{Z}_p) \neq 2, 3. \tag{5.1}$$

The tuple (x, y) with $x, y \in \mathbb{Z}_p$ satisfying Equation **5.1** is called a "point in affine coordinates" on the elliptic curve $E(\mathbb{Z}_p)$. For a working ECDSA scheme, there are three distinct mathematical operations (Layer-2, Figure **5.1**) on points required. Point addition, point doubling, and multiplying a point with a scalar.

ECC arithmetic is applied to points represented as x- and y-coordinates, therefore we can draw them as a curve in a two-dimensional coordinate system. With the help of this coordinate system we may describe addition and doubling geometrically (see Figure **5.2**).

Point addition Let $P_1, P_2 \in E(\mathbb{Z}_p)$ be two distinct points. The sum $Q = P_1 + P_2$ where $Q \in E(\mathbb{Z}_p)$ is defined as follows: draw a line through P_1 and P_2. The reflection (on the x-axis) of the intersection of this line with the curve is the resulting point Q.

[1] i.e., SECP128R1, SECP128R2, SECP160K1, SECP160R2, SECP192K1 and SECP192R1

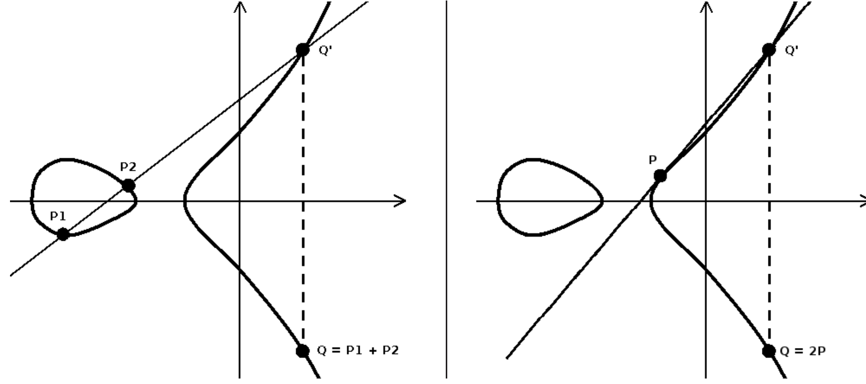

Figure 5.2: Geometric (**a**) point addition and (**b**) point doubling

Point doubling Let $P \in E(\mathbb{Z}_p)$ be a point on E. The double $Q = 2 \cdot P, Q \in E(\mathbb{Z}_p)$ is derived by drawing a tangent line to the curve in P. The intersection of this line with the curve Q' has to be reflected on the x-axis and yields Q.

Although geometric addition and doubling of points is more demonstrative, we need formulas to actually compute the resulting points in software. Points $P = (x, y)$ with $x, y \in \mathbb{Z}_p$ that satisfy Equation **5.1** form an abelian group (see Definition **9**), therefore arithmetic is defined by the group law.

Definition 20 (Group law [28, p. 80]) *The* group law *for points satisfying the elliptic curve equation* $E(\mathbb{Z}_p) : y^2 = x^3 + a \cdot x + b, \texttt{char}(\mathbb{Z}_p) \neq 2, 3$ *is defined as follows:*

Identity ∞ *is the identity of the group,* $P + \infty = \infty + P = P$, $\forall P \in E(\mathbb{Z}_p)$. ∞ *is called the "point at infinity".*

Negatives *If* $P = (x, y) \in E(\mathbb{Z}_p)$, *then* $(x, y) + (x, -y) = \infty$. *The point* $(x, -y)$ *is denoted by* $-P$.

Point addition *Let* $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E(\mathbb{Z}_p)$, *where* $P_1 \neq \pm P_2$. *Then* $Q = P_1 + P_2 = (x_3, y_3)$ *where*

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \bmod p, \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \bmod p. \quad (5.2)$$

Point doubling *Let* $P = (x_1, y_1) \in E(\mathbb{Z}_p)$, *where* $P \neq -P$. *Then* $Q = P + P = 2 \cdot P = (x_2, y_2)$ *is defined by*

$$x_2 = \left(\frac{3 \cdot x_1^2 + a}{2 \cdot y_1}\right)^2 - 2 \cdot x_1 \bmod p, \quad y_2 = \left(\frac{3 \cdot x_1^2 + a}{2 \cdot y_1}\right)(x_1 - x_2) - y_1 \bmod p. \quad (5.3)$$

From point addition and point doubling we can derive (scalar) point multiplication methods. We distinguish two types of point multiplication, single point multiplication and simultaneous point multiplication. Note that the following definitions are given to demonstrate the multiplication methods, our actual implementation relies on improved and therefore different approaches.

Single (scalar) point multiplication Let $P \in E(\mathbb{Z}_p)$ be a point on E, n is the prime order the curve's basepoint G and $k \in \mathbb{Z}_n$ is an integer. Computing $Q = k \cdot P$ where $Q \in E(\mathbb{Z}_p)$ may be performed by k point additions,

$$Q = \underbrace{P + P + \cdots + P}_{k}.$$

Simultaneous (scalar) point multiplication Let $P_1, P_2 \in E(\mathbb{Z}_p)$, n is the prime order of E and $k_1, k_2 \in \mathbb{Z}_n$ are two scalar integers. Simultaneous point multiplication $Q = k_1 \cdot P_1 + k_2 \cdot P_2$ where $Q \in E(\mathbb{Z}_p)$ may be performed by two single point multiplications, $X_1 = k_1 \cdot P_1, X_2 = k_2 \cdot P_2$ and one point addition such that $Q = X_1 + X_2$.

Point representation

Inversion in $\mathbb{Z}_p$ is far more expensive than modular multiplication on the ATMEGA128L micro-processor, because no division operation is supported. Observing Equation **5.2** and Equation **5.3** reveals that point addition and doubling require several division steps. In order to avoid these steps, the arithmetic can be performed in Jacobian representation instead. Recall that $P = (x, y) \in E(\mathbb{Z}_p)$ is a point in affine coordinates, adding a redundant coordinate z is a common method to avoid the frequent use of inversion routines. The tuple $(\tilde{x}, \tilde{y}, \tilde{z})$ denotes the Jacobian coordinates[2] of a point $\tilde{P}$. The Jacobian representation $\tilde{P} = (\tilde{x}, \tilde{y}, \tilde{z})$ of a point $P = (x, y)$ in affine coordinates is derived easily by letting

$$\tilde{x} = x, \quad \tilde{y} = y, \quad \tilde{z} = 1.$$

Transforming a point $\tilde{P}$ in Jacobian coordinates back to P in affine coordinates is computationally expensive, because it involves inversion,

$$x = \frac{\tilde{x}}{\tilde{z}^3}, \tag{5.4}$$

$$y = \frac{\tilde{y}}{\tilde{z}^2}. \tag{5.5}$$

Other representations (e.g., “Chudnovsky” coordinates [11]) transform $P = (x, y)$ into $\hat{P} = (\hat{x}, \hat{y}, \hat{z}, \hat{z}^2, \hat{z}^3)$ and add even more reduncancies. However, we did not consider them for our implementation, because they consume up to three times more memory than points in affine coordinates. This conflicts with our intention to use as less RAM as possible, wich is imperative on a constrained device such as the ATMEGA128L.

Coordinates	Operation	#Mul	#Sq	#Inv
Affine	$Q = P_1 + P_2$	2	1	1
Jacobian	$\tilde{Q} = \tilde{P}_1 + \tilde{P}_2$	12	4	0
Jacobian/Affine	$\tilde{Q} = \tilde{P}_1 + P_2$	8	3	0
Chudnovsky	$\hat{Q} = \hat{P}_1 + \hat{P}_2$	11	3	0
Affine	$Q = 2 \cdot P$	2	2	1
Jacobian	$\tilde{Q} = 2 \cdot \tilde{P}$	4	4	0
Chudnovsky	$\hat{Q} = 2 \cdot \hat{P}$	5	4	0

Table 5.1: Computational costs of point addition and doubling in different coordinate systems [28, Tab. 3.3]

We will now evaluate the number of squarings, multiplications and inversions in $\mathbb{Z}_p$ required by the addition and doubling algorithms for all three point representations. We do not consider multi-precision subtraction and addition, because the computational costs are negligible compared to the listed operations. We also list combinations of two representations, in these cases the ECC arithmetic is performed in “mixed coordinates”. Table **5.1** indicates that point addition should be performed in mixed coordinates. In this case, a

[2] This is not to be confused with the *Montgomery representation* used in the previous chapter.

point in Jacobian coordinates is added to a second point in affine coordinates. Note that the resulting point is in Jacobian coordinates, this is advantageous because we will perform point doubling in Jacobian coordinates.

"Hard" Problem

Elliptic curve cryptography relies on a "hard" problem, the so-called *Elliptic Curve Discrete Logarithm Problem* (ECDLP). It is based on the group law (see Definition **20**) which allows to construct point multiplication methods on top of point addition and point doubling. The ECDLP is stated as follows: given two points $P, Q \in E(\mathbb{Z}_p)$, find an integer k such that $Q = k \cdot P$. k is called the discrete logarithm of Q to the base P. Solving this problem in large groups with carefully chosen curve parameters is assumed to be "hard".

5.2 The ECDSA Protocol

The ECDSA protocol based on elliptic curve cryptography was proposed and accepted as ANSI standard in 1999 (ANSI X9.62). Furthermore it was accepted in 2000 as IEEE (IEEE-1363-2000) and FIPS (FIPS 186-2) standards. We will now introduce the necessary subset of the elliptic curve domain parameters defined by the SEC group [61].

1. p is a large prime and the order of the underlying prime field $\mathbb{Z}_p$ of the elliptic curve.
2. The parameters $a, b \in \mathbb{Z}_p$ define the elliptic curve (see Equation **5.1**).
3. The basepoint G is given in affine coordinates. G has prime order n and is the generator of the elliptic curve group.

As stated before, our implementation uses the parameters defined by the SECP160R1 parameter set [61], we list their respective values in the Appendix in Section **A.1.2**. Due to the good availability of ECDSA keypairs, only the signature generation and verification methods were actually implemented in NesC. However, we will describe the basic algorithms for all three primitives in the remainder of this Section.

5.2.1 Keypair generation

The keypair (Q, d) may be generated by using Algorithm **32**. d is selected randomly and used as the public key, Q is a point - derived by multiplying the scalar d with the basepoint G - and serves as the private part of the keypair. The algorithm benefits from all optimizations for the single point multiplication (denoted by `singlePointMul`$(\cdot)$), because this is its main operation.

Algorithm 32: ECDSA: Keypair generation [28, Alg. 4.28]

Input: Prime order n, basepoint G
Output: Public key Q, private key d
Randomly select $d \in_R [1, n-1]$;
Compute $Q \leftarrow$ `singlePointMul`(d, G);
Return (Q, d);

Note that it is quite simple to derive a working keypair, once the main ECC arithmetic (Layer-2, Figure **5.1**) is implemented.

5.2.2 Signature generation

An ECDSA signature $S_{\texttt{ECDSA}}$ for a given message m consists of two values (r, s) that can be obtained by following this algorithm. Note that we denote the SHA1 hash algorithm [51] by $\texttt{SHA1}(\cdot)$. Only the `PMul` step requires single point multiplication, the remaining arithmetic is computed in $\mathbb{Z}_n$.

Algorithm 33: ECDSA: Signature generation [28, Alg. 4.29]

Input: Private key d, message m
Aux. Input: Prime order n, basepoint G
Output: $S_{\texttt{ECDSA}} = (r, s)$
Randomly select $k \in_R [1, n-1]$;
PMul Compute $X = (x1, y1) \leftarrow \texttt{singlePointMul}(k, G)$;
$r \leftarrow x1 \bmod n$;
if $r = 0$ **then**
  Restart algorithm;
$e \leftarrow \texttt{SHA1}(m)$; $s \leftarrow k^{-1}(e + d \cdot r) \bmod n$;
if $s = 0$ **then**
  Restart algorithm;
Return (r, s);

Please note further that point multiplication uses G as multiplicand, G is a parameter of SECP160R1 and therefore fixed as long as the SECP160R1 curve is used, since our implementation supports only this parameter set.

5.2.3 Signature verification

Verification of a tuple $(m, S_{\texttt{ECDSA}})$ is performed by using the following algorithm. The verifier should also check whether the signer's public key is valid. We have omitted this step in our implementation because we assume that the public key is not (ex-)changed during a mote's lifetime.

Algorithm 34: ECDSA: Signature verification [28, Alg. 4.30]

Input: Public key Q, message m, signature $S_{\texttt{ECDSA}} = (r, s)$
Aux. Input: Prime order n, basepoint G
Output: `ind` $\in \{$"accept", "reject"$\}$
$e \leftarrow \texttt{SHA1}(m)$; $w \leftarrow s^{-1} \bmod n$;
$u1 \leftarrow e \cdot w \bmod n$; $u2 = r \cdot w \bmod n$;
PMul $X = (x1, y1) \leftarrow \texttt{dblPointMul}(u1, G, u2, Q)$;
if $X = \infty$ **then**
  Return "reject";
$v = x1 \bmod n$;
if $v = r$ **then**
  `ind` $\leftarrow$ "accept";
else
  `ind` $\leftarrow$ "reject"
Return `ind`;

This algorithm requires simultaneous point multiplication denoted by $\texttt{dblPointMul}(\cdot)$. So far, we have only introduced one method to compute $u1 \cdot G + u2 \cdot Q$ – by applying $\texttt{singlePointMul}(\cdot)$ to $(u1, G)$ and $(u2, Q)$ and adding the result. However, we will see that there are far better methods, which are computationally equivalent to the single application of $\texttt{singlePointMul}(\cdot)$. All other operations are calculated in $\mathbb{Z}_n$.

5.3 ECC arithmetic

In this section we will present the ECC layer arithmetic (Layer-2, Figure **5.1**). This is the layer where most of our optimizations and algorithm evaluations took place. We will begin with introducing the (scalar) point multiplication methods built on top of the point addition and doubling algorithms, the latter algorithms are presented thereafter.

5.3.1 Single scalar point multiplication

Single point multiplication is needed rather infrequently. Although we could have implemented the simultaneous point multiplication by using two single multiplications, we implemented a more sophisticated approach (see Section **5.3.2**). Due to this different approach, single point multiplication is only involved when signing a message or creating a keypair.

We have evaluated three algorithms: by `singlePointMul1`$(\cdot)$ we denote the *binary left-to-right method*, `singlePointMul2`$(\cdot)$ refers to the *sliding window method*. The third algorithm is called the *fixed-point method*. All algorithms compute $k \cdot P$ where $P \in E(\mathbb{Z}_p)$, $k \in \mathbb{Z}_n$.

Binary left-to-right multiplication

The *binary left-to-right* multiplication algorithm is straightforward and can be understood as the naive approach to scalar point multiplication. Let $t = \lceil \texttt{log2}(k) \rceil$, the algorithm iterates over all t bits of $k = (k_{t-1}, k_{t-2}, \cdots, k_0)_2$, beginning at the leftmost bit $k_{t-1} \neq 0$. The following algorithm is the ECC equivalent of the *square-and-multiply* method for exponentiation, squaring is substituted by point doubling (denoted by `pointDbl`$(\cdot)$) and point addition (denoted by `pointAdd`$(\cdot)$) is executed instead of multiplying.

Algorithm 35: `singlePointMul1`$(\cdot)$: Binary left-to-right method [28, Alg. 3.27]

Input: Scalar $k = (k_{t-1}, \cdots, k_1, k_0)_2$, point P
Aux. Input: Bit size t
Output: $k \cdot P$
$\tilde{T} = (\tilde{x}, \tilde{y}, \tilde{z}) \leftarrow \infty$;
for i *from* $t-1$ *downto* 0 **do**
  $\tilde{T} \leftarrow$ `pointDbl`$(\tilde{T})$;
  if $k_i = 1$ **then**
    $\tilde{T} \leftarrow$ `pointAdd`$(\tilde{T}, P)$;
ReTransf Return $X \leftarrow (\tilde{x}/\tilde{z}^3, \tilde{y}/\tilde{z}^2)$;

Note that $\tilde{T}$ is a point in Jacobian representation. In order to return the point X in affine coordinates, the `ReTransf` step is necessary. The affine representation is derived according to Equations **5.4** and **5.5**.

Sliding window multiplication

The *sliding window method* employs a "window" of w bits that is moving left-to-right over the bits of k, skipping consecutive sequences of zeros. Recall the preceding algorithm, where the scalar is scanned bitwise. The advantage of the new method is that the bits are scanned in "chunks" and are interpreted as small integers. By pre-computing multiple $\Gamma_u = u \cdot P$ for all possible integers u represented by a bitstring of length w (i.e., $u \in [0, 2^w - 1]$), each "chunk" can be processed by one point addition and w doublings.

However, the *sliding window method* presented here does not require Γ_u for all $u \in [0, 2^w - 1]$ – only multiples of P where u is odd are computed and stored. This effectively

halves the size of the pre-computation table and the time required to generate it. The missing Γ_u for even values of u must be respected when passing the window over k, therefore it has to be taken care that the last bit in the window is set. The following algorithm generates a table consisting of $\{P, 3 \cdot P, 5 \cdot P, \cdots, (2^w - 1) \cdot P\}$. Note that we index the pre-computed points in a continuous manner, which makes more sense with respect to the implementation, e.g., $\Gamma_1 = P, \Gamma_2 = 3 \cdot P, \cdots, \Gamma_{2^{w-1}} = (2^w - 1) \cdot P$.

Algorithm 36: Precomputation for the sliding window method

Input: Point $P = (x, y)$
Aux. Input: Bit size t, window size w
Output: $\{P, 3 \cdot P, 5 \cdot P, \cdots, (2^w - 1) \cdot P\}$
Copy $\tilde{T1} = (\tilde{x1}, \tilde{y1}, \tilde{z1}) \leftarrow (x, y, 1)$; $\tilde{T2} \leftarrow$ `pointDbl`$(\tilde{T1})$; $\Gamma_1 \leftarrow P$;
for i *from* 2 *to* $2^{w-1} - 1$ **do**
Transf | Copy $T1 \leftarrow \Gamma_{i-1}$; $\tilde{T1} \leftarrow (x1, y1, 1)$;
| $\tilde{T1} = (\tilde{x1}, \tilde{y1}, \tilde{z1}) \leftarrow$ `pointAdd`$(\tilde{T1}, \tilde{T2})$;
ReTransf | $\Gamma_i \leftarrow (\tilde{x1}/\tilde{z1}^3, \tilde{y1}/\tilde{z1}^2)$;
Return $\{\Gamma_1, \Gamma_2, \cdots, \Gamma_{2^{w-1}}\}$;

Our algorithm starts by stroring P in Γ_1 and then successively adds $\tilde{T2} = 2 \cdot P$ to Γ_{i-1}, thus computing $\Gamma_2 = \Gamma_1 + 2 \cdot P = 3 \cdot P$, $\Gamma_3 = \Gamma_2 + 2 \cdot P = 5 \cdot P$ and so forth. The `Transf` step is required to convert the previously stored Γ_{i-1} from affine representation back to Jacobian coordinates in $\tilde{T1}$. The `ReTransf` step retransforms the Jacobian result of the mixed coordinate point addition to an affine point. This is a mandatory step, as we will see now.

Algorithm 37: `singlePointMul2`$(\cdot)$: Sliding window method [28, Alg. 3.38]

Input: Scalar $k = (k_{t-1}, \cdots, k_0)_2$
Aux. Input: Bit size t, w, $\Gamma_{(u+1)/2} = u \cdot P \ \forall u \in \{1, 3, \cdots, 2^w - 1\}$
Output: $k \cdot P$
$\tilde{T} = (\tilde{x}, \tilde{y}, \tilde{z}) \leftarrow \infty$; $i \leftarrow t - 1$;
while $i \geq 0$ **do**
  if $k_i = 0$ **then**
    $l \leftarrow 1$; $u \leftarrow 0$;
  else
    $l \leftarrow 0$;
    if $i \leq w$ **then**
      $j \leftarrow 0$;
    else
      $j \leftarrow i - w$;
    for j *from* j *to* $i - 1$ **do**
      if $k_j = 1$ **then**
        if $l \neq 0$ **then**
          $l \leftarrow i - j$;
        if $l = 0$ **then**
          $u \leftarrow u + 2^{j+l-i}$;
RepDbl $\tilde{T} \leftarrow$ `repPointDbl`$(\tilde{T}, l)$;
  if $u \neq 0$ **then**
Adjust $u \leftarrow (u + 1)/2$;
AddTab $\tilde{T} \leftarrow$ `pointAdd`$(\tilde{T}, \Gamma_u)$;
  $i = i - l$;
Return $X \leftarrow (\tilde{x}/\tilde{z}^3, \tilde{y}/\tilde{z}^2)$;

Here we use the `repPointDbl`$(\cdot)$ algorithm to perform l consecutive doublings. This can be performed either by calling `pointDbl`$(\cdot)$ for l times, or by a sligthly better method that will be discussed later on. The `Adjust` step calculates the correct index for a given u which

is induced by our wish to continously index the points Γ_u (see Algorithm **36**). The `AddTab` step reveals that the second argument to `pointAdd`$(\cdot)$, which is the pre-computed value Γ_u, must be given in affine coordinates. This is also preferable, because storing the same amount of points in Jacobian representation would require 50% more RAM.

Considering the RAM limitations of the ATMEGA128L, it seems a valid assumption that w is chosen to be small (e.g., $w \in [2, 6]$). For a small w, pre-computing the whole table on-the-fly (i.e., before each multiplication with an unknown point P) can be done in little time. But there is a better, more efficient way: recalling Algorithm **33** it is obvious that only k is random in the `PMul` step. G is - as stated before - the basepoint of the curve and therefore defined by SECP160R1. Due to this, we can safely pre-compute all $\Gamma_{(u+1)/2} = u \cdot G$ only once (e.g., when the mote is powered on) and then use the table for all occuring single scalar point multiplications.

Fixed-point method

The *(single) fixed-point method* relies on a pre-computed table to speed up the multiplication, which makes it comparable to the *sliding window method*. However, the technique used to scan the scalar is a completely different one, which makes pre-computing a single point approximately as expensive as a normal scalar point multiplication. Therefore, tables should not be generated on-the-fly (as opposed to the *sliding window method*), which implies that the *fixed-point method* can only be applied reasonably when G is fixed and known.

The pre-computation step of the *fixed-point method* proceeds as follows. Let $t = \lceil \texttt{log2}(n) \rceil$, $d = \lceil t/w \rceil$, where n is the order of the basepoint. All scalars are elements in $\mathbb{Z}_n$, therefore we can represent each scalar in t bits. In order to accelerate the point multiplication, we pre-compute the points $\Gamma_\iota = i \cdot G$ where

$$i = (I^{w-1}, \cdots, I^0)_{2^d} = (\underbrace{0, \cdots, 0}_{d-1}, \iota_{w-1}, \underbrace{0, \cdots, 0}_{d-1}, \iota_{w-2}, \cdots \underbrace{0, \cdots, 0}_{d-1}, \iota_1, \underbrace{0, \cdots, 0}_{d-1}, \iota_0)_2, \quad (5.6)$$

for all possible $\iota = (\iota_{w-1}, \iota_{w-2}, \cdots, \iota_1, \iota_0)_2 \in [0, 2^w - 1]$. This representation of i reveals sequences I^j with d bits where only the last bit may be set. Observing the previously presented point multiplication algorithms, we see that each '0' leads to a point doubling, each '1' to a point addition. Therefore, pre-computation of all Γ_ι can be rougly compared to 2^w scalar multiplications with "sparse" scalars. As stated before, pre-computation is only assumed to be performed once, in the moment when the mote is powered on. Under this assumption, the computational overhead of performing 2^w initial point multiplications can be neglected.

The next algorithm can be used to generate all Γ_ι as required by the actual point multiplication algorithm. Note that we index the pre-computed points Γ_ι for all $\iota \in [0, 2^w - 1]$ by the small integers ι and not by the lengthy representation of i (as in Equation **5.6**).

Algorithm 38: Precomputation for the fixed-point method

Input: Basepoint G
Aux. Input: Window size w, bit size t, "chunk" size $d = \lceil t/w \rceil$
Output: $\{\Gamma_0, \Gamma_1, \cdots, \Gamma_{2^w-1}\}$

```
for ι from 0 to 2^w − 1 do
    T̃ = (x̃, ỹ, z̃) ← ∞;
    Write ι as bitstring: ι = (ι_{w−1}, ··· , ι_1, i_0)_2 of length w;
    for j from w − 1 to 0 do
        T̃ ← repPointDbl(T̃, d);
        if ι_j = 1 then
            T̃ ← pointAdd(T̃, G);
    Γ_ι ← (x̃/z̃^3, ỹ/z̃^2);
Return {Γ_0, Γ_1, ··· , Γ_{2^w−1}};
```

Each pre-computed point has to be converted from the Jacobian representation of $\tilde{T}$ to affine coordinates, this leads to 2^w inversions. The following algorithm computes $k \cdot G$, when $\Gamma_\iota = i \cdot G$ is given, with the *fixed-point method*.

Algorithm 39: `singlePointMul3`$(\cdot)$: The fixed-point method [28, Alg. 3.41]

Input: Scalar $k = (K^{w-1}, \cdots, K^1, K^0)_{2^d}$
Aux. Input: Bit size $t = \lceil \texttt{log2}(k) \rceil$, d, points $\Gamma_\iota = i \cdot G \ \forall \iota \in [0, 2^w - 1]$
Output: $k \cdot G$

```
     Write each K^j as bitstring, K^j = (K^j_{d−1}, ··· , K^j_1, K^j_0)_2;
Init T̃ = (x̃, ỹ, z̃) ← ∞;
     for i from d − 1 downto 0 do
Dbl      T̃ ← pointDbl(T̃);
Scan     κ ← (K^{w−1}_i, ··· , K^1_i, K^0_i)_2;
Add      T̃ ← pointAdd(T̃, Γ_κ);
     Return X ← (x̃/z̃^3, ỹ/z̃^2);
```

The *fixed-point method* executes only $d = \lceil t/w \rceil$ point additions and doublings in total, so the performance strongly depends on the choice of w. On the other hand, 2^w pre-computed points in affine representation have to be stored in memory. For a 160-bit curve (e.g., SECP160R1) this would result in a memory usage of $2^w \cdot 40$ bytes.

The following example demonstrates, how the *fixed-point method* constructs the final result. Suppose we wanted to multiply $k = 3212$ with G, where $t = \lceil \texttt{log2}(3212) \rceil = 12$. We assume a window size of $w = 4$, therefore we have $d = \lceil 12/4 \rceil = 3$. We assume further that the points $\Gamma_\iota = i \cdot G$ with

$$i = (0, 0, \iota_3, 0, 0, \iota_2, 0, 0, \iota_1, 0, 0, \iota_0)_2$$

for all $\iota = (\iota_3, \iota_2, \iota_1, \iota_0) \in [0, 15]$ have already been calculated. We start the actual point multiplication step by decomposing 3212 into its binary representation,

$$(\underset{11}{1}, \underset{10}{1}, \underset{9}{0}, \underset{8}{0}, \underset{7}{1}, \underset{6}{0}, \underset{5}{0}, \underset{4}{0}, \underset{3}{1}, \underset{2}{1}, \underset{1}{0}, \underset{0}{0})_2.$$

Due to $d = 3$, the algorithm will proceed in three iterations of the for-loop. Table **5.2** shows which bits are scanned, the resulting index κ and the intermediate value of $\tilde{T}$ for each iteration of Algorithm **39**. As hinted by referring to the *square-and-multiply* method, "normal" exponentiation is comparable to point multiplication which is obvious by looking at the intermediate values of $\tilde{T}$.

Step	Bits scanned	κ	$\tilde{T}$
Init	-	-	∞
Dbl	-	-	∞
Scan	11, 8, 5, 2	$(1,0,0,1)_2 = 9$	∞
Add	-	-	$(0,0,\mathbf{1},0,0,\mathbf{0},0,0,\mathbf{0},0,0,\mathbf{1})_2 \cdot G$
Dbl	-	-	$(0,1,0,0,0,0,0,0,0,0,1,0)_2 \cdot G$
Scan	10, 7, 4, 1	$(1,1,0,0)_2 = 12$	-
Add	-	-	$(0,1,\mathbf{1},0,0,\mathbf{1},0,0,\mathbf{0},0,1,\mathbf{0})_2 \cdot G$
Dbl	-	-	$(1,1,0,0,1,0,0,0,0,1,0,0)_2 \cdot G$
Scan	9, 6, 3, 0	$(0,0,1,0)_2 = 2$	-
Add	-	-	$(1,1,\mathbf{0},0,1,\mathbf{0},0,0,\mathbf{1},1,0,\mathbf{0})_2 \cdot G$

Table 5.2: Steps performed by the fixed-point method

5.3.2 Simultaneous scalar multiplication

Signature verification involves calculating $u1 \cdot G + u2 \cdot Q$, which could be done by applying one of the previously discussed algorithms twice. However, we have not further pursued this idea because there are better ways to perform simultaneous scalar multiplication. We start with presenting the so-called *Shamir's trick*, then we derive a variant that is more suitable for constrained devices such as the ATMega128L. After discussing *Shamir's trick* in combination with a truncated pre-computation table, we show how the *fixed-point method* can be applied to simultaneous point multiplication.

Shamir's trick

Let $t = \lceil \texttt{log2}(n) \rceil$, by choosing w we divide scalars in $d = \lceil t/w \rceil$ "chunks". *Shamir's trick* uses a window of w bits width, similarly to the *sliding window method*. Suppose we want to simultaneously multiply $k \cdot P_1 + l \cdot P_2$. In order to accelerate the multiplication we have

a	b	Precomputed sum
0	0	$\Gamma_{0,0} \leftarrow 0$
0	1	$\Gamma_{0,1} \leftarrow P_1$
$\vdots$	$\vdots$	$\vdots$
0	$2^w - 1$	$\Gamma_{0,2^w-1} \leftarrow (2^w - 1) \cdot P_1$
1	0	$\Gamma_{1,0} \leftarrow P_2$
1	1	$\Gamma_{1,1} \leftarrow P_2 + P_1$
$\vdots$	$\vdots$	$\vdots$
$2^w - 1$	$2^w - 1$	$\Gamma_{2^w-1,2^w-1} \leftarrow (2^w - 1) \cdot P_1 + (2^w - 1) \cdot P_2$

Table 5.3: Precomputation for Shamir's trick

to calculate the result of several simultaneous multiplications $\Gamma_{a,b} = a \cdot P_1 + b \cdot P_2$ for all small integers $a, b \in [0, 2^w - 1]$ in advance. Table **5.3** demonstrates the pre-computation process.

We have developed a pre-computation algorithm for *Shamir's trick*. By introducing additional parameters, we can re-use the following algorithm for the remaining simultaneous multiplication methods. The "treshold" parameter q defines the upper bound of the interval we choose a and b from. Setting $q = 2^w$ will pre-compute $\Gamma_{a,b} = a \cdot P_1 + b \cdot P_2$ for all $a, b \in [0, 2^w - 1]$. A second parameter c indicates the number of doublings that are performed in the `Dbl` step. For *Shamir's trick* without modification we set $c = 1$.

Algorithm 40: Precomputation for Shamir's trick (and all variants)

Input: Points $P1, P2$, threshold q, double-count c
Output: $\Gamma_{a,b}\ \forall a,b \in [0, q-1]$
$w \leftarrow \lceil \texttt{log2}(q) \rceil$;
for a *from* 0 *to* $q-1$ **do**
  for b *from* 0 *to* $q-1$ **do**
    $\tilde{T1} = (\tilde{x1}, \tilde{y1}, \tilde{z1}) \leftarrow \infty$; $\tilde{T2} = (\tilde{x2}, \tilde{y2}, \tilde{z2}) \leftarrow \infty$;
    Write a, b as bitstrings, $a = (a_{w-1}, \cdots, a_0)_2$, $b = (b_{w-1}, \cdots, b_0)_2$;
    for i *from* 0 *to* $w-1$ **do**
Dbl $\tilde{T1} \leftarrow \texttt{repPointDbl}(\tilde{T1}, c)$; $\tilde{T2} \leftarrow \texttt{repPointDbl}(\tilde{T2}, c)$;
      if $a_i = 1$ **then**
        $\tilde{T1} \leftarrow \texttt{pointAdd}(\tilde{T1}, P1)$;
      if $b_i = 1$ **then**
        $\tilde{T2} \leftarrow \texttt{pointAdd}(\tilde{T2}, P2)$;
Inv1 $\Gamma_{a,b} \leftarrow (\tilde{x1}/\tilde{z1}^3, \tilde{y1}/\tilde{z1}^2)$;
    $\tilde{T1} \leftarrow \texttt{pointAdd}(\tilde{T2}, \Gamma_{a,b})$;
Inv2 $\Gamma_{a,b} \leftarrow (\tilde{x1}/\tilde{z1}^3, \tilde{y1}/\tilde{z1}^2)$;
Return $\{0, P2, 2 \cdot P2, \cdots, (q-1) \cdot P2, P1 + P2, \cdots, (q-1) \cdot P1 + (q-1) \cdot P2\}$;

Note that pre-computation is performed using a simultaneous version of the standard *binary left-to-rigth algorithm* (see Algorithm **35**). For a 160-bit curve Algorithm **40** requires to store 2^{2w} points à 40 bytes in RAM. Please note further that 2^{2w+1} inversions are needed, since addition requires a point in affine coordinates (this is solved by the `Inv1` step) and outputs a point in Jacobian coordinates although we want to store the pre-computed points in affine coordinates (`Inv2` performs this transformation). We could pre-compute the table on-the-fly, but it is generally a better idea to use the fact that signature verification computes $u1 \cdot G + u2 \cdot Q$, where the points G (basepoint) and Q (public key) are fixed and known in advance. Therefore we can generate the table once for a mote's lifetime. For the rest of this section we will assume that all $\Gamma_{a,b} = a \cdot G + b \cdot Q$ with $a, b \in [0, 2^w - 1]$ have been pre-computed.

Point multiplication with *Shamir's trick* is quite similar to the *sliding window method*. The difference is that the window is passed over both scalars l and k simultaneously from left-to-right with a fixed step-size, while the *sliding window method* skips sequences of zeros. We will now briefly demonstrate how *Shamir's trick* scans the scalars and uses the resulting integers to find the corresponding pre-computed point. Let l and k be t-bit integers, then we can break their binary representation down in $d = \lceil t/w \rceil$ strings of w bits:

$$k = (\overbrace{k_{t-1}, k_{t-2}, \cdots, k_{t-w}}^{K_{d-1}}, \overbrace{k_{t-w-1}, k_{t-w-2}, \cdots, k_{t-2w}}^{K_{d-2}}, \cdots, \overbrace{k_{w-1}, k_{w-2}, \cdots, k_0}^{K_0})_2,$$

$$l = (\underbrace{l_{t-1}, l_{t-2}, \cdots, l_{t-w}}_{Ld-1}, \underbrace{l_{t-w-1}, l_{t-w-2}, \cdots, l_{t-2w}}_{K_{d-2}}, \cdots, \underbrace{l_{w-1}, l_{w-2}, \cdots, l_0}_{K_0})_2.$$

The d bitstrings of l and k are now successively selected and interpreted as integers K and L. By using these integers the table can be indexed to select the corresponding pre-computed sum $\Gamma_{K,L}$. The following algorithm proceeds in this manner and requires only d steps with one point addition and w doublings to compute $l \cdot G + k \cdot Q$.

Algorithm 41: `dblPointMul1`$(\cdot)$: Shamir's trick [28, Alg. 4.48]

Input: Scalars $k = (K_{d-1}, \cdots, K_0)_{2^w}$, $l = (L_{d-1}, \cdots, L_0)_{2^w}$
Aux. Input: w, $d = \lceil t/w \rceil$, points $\Gamma_{a,b} = a \cdot G + b \cdot Q \; \forall a, b \in [0, 2^w - 1]$
Output: $k \cdot G + l \cdot Q$
$\tilde{T} = (\tilde{x}, \tilde{y}, \tilde{z}) \leftarrow \infty$;
for i *from* $d-1$ *downto* 0 **do**
  $\tilde{T} \leftarrow$ `repPointDbl`$(\tilde{T}, w)$;
  $\tilde{T} \leftarrow$ `pointAdd`$(\tilde{T}, \Gamma_{K_i, L_i})$;
Return $X \leftarrow (\tilde{x}/\tilde{y}^3, \tilde{z}/\tilde{y}^2)$;

Shamir's trick with truncated pre-computation table

As we have seen in the preceding sections, pre-computation tables can grow large - especially *Shamir's trick* quadruples its RAM requirements for every increase of the window size w. Big window sizes are desirable because $d = \lceil t/w \rceil$ determines the number of additions in Algorithm **41**, but huge pre-computation tables should be avoided on constrained devices. We have developed a variant of *Shamir's trick* that allows, for a given window size w, to pre-compute only q^2 points where $2^{w-1} < q < 2^w$. By allowing q to be chosen in this manner, we have found a method to better scale the memory requirements of *Shamir's trick*.

As hinted before, we can re-use the pre-computation algorithm for *Shamir's trick* by setting the threshold argument q such that $2^{w-1} < q < 2^w$. By observing Algorithm **40** we see that the threshold value determines the for-loops' break conditions, i.e., the algorithm will only iterate a and b over $[0, q-1]$. By passing G and Q as input points the algorithm will output $\Gamma_{a,b} = a \cdot G + b \cdot Q$ for all $a, b \in [0, q-1]$. The "double-count" parameter c remains untouched, i.e., $c = 1$.

We had to modify Algorithm **41** in order to account for the smaller pre-computation table. Our modified algorithm starts by scanning the scalars k and l left-to-right, but the obtained integers K and L are handled differently. In case $K < q, L < q$ they can be used to obtain the corresponding point $\Gamma_{K,L} = K \cdot G + L \cdot Q$. When the algorithm encounters $K \geq q$ or $L \geq q$, K and L cannot be used to access the table, because in this case $\Gamma_{K,L}$ has not been computed. However, by simply right-shifting K and L by one bit we derive $K' = \lfloor K/2 \rfloor$ and $L' = \lfloor L/2 \rfloor$, thereby effectively reducing the window size that was used to obtain K and L. Due to $2^{w-1} < q$ we can now safely index our table with K' and L'.

The following algorithm assumes that the points $\Gamma_{a,b} = a \cdot G + b \cdot Q$ for $a, b \in [0, q-1]$ have already been pre-computed. Precomputation is not done on-the-fly but only once (similarly to the unmodified algorithm).

Algorithm 42: `dblPointMul2`$(\cdot)$: Shamir's trick with truncated pre-computation table

Input: Scalars $k = (k_{t-1}, \cdots, k_1, k_0)_2, l = (l_{t-1}, \cdots, l_1, l_0)_2$
Aux. Input: Window size w, t, $d = \lceil t/w \rceil$, threshold q, points $\Gamma_{a,b}\ \forall a, b \in [0, q-1]$
Output: $k \cdot G + l \cdot Q$

```
T̃ = (x̃, ỹ, z̃) ← ∞;   i = t;
while i > 0 do
    if i ≥ w then
        s = w;
    else
        s = i
    K ← (k_{i-1}, ··· , k_{i-s})_2;    L ← (l_{i-1}, ··· , l_{i-s})_2;
    if K ≥ q or L ≥ q then
        K ← ⌊K/2⌋;    L ← ⌊L/2⌋;
        s = s - 1;
    T̃ ← repPointDbl(T̃, s);
    T̃ ← pointAdd(T̃, Γ_{K,L});
    i = i - s;
Return X ← (x̃/ỹ^3, z̃/ỹ^2);
```

Simultaneous fixed-point method

In order to combine the idea of fixed-point computations (pre-computing products where the scalar has a special form) with simultaneous point multiplication (pre-computing sums $a \cdot G + b \cdot Q$ for some integers a and b) we have developed a third multiplication algorithm. In the following we will call this the *simultaneous fixed-point method*. Again, the algorithm requires a set of pre-computed points to speed up the calculation of $u1 \cdot G + u2 \cdot Q$.

We will now shortly illustrate the general idea of our algorithm. As done previously, we choose a window size w, set $t = \lceil \texttt{log2}(n) \rceil$ and $d = \lceil t/w \rceil$. In the pre-computation phase we calculate $\Gamma_{\alpha,\beta} = a \cdot G + b \cdot Q$ where

$$a = (A^{w-1}, \cdots, A^0)_{2^d} = (\overbrace{0, \cdots, 0}^{d-1}, \alpha_{w-1}, \overbrace{0, \cdots, 0}^{d-1}, \alpha_{w-2}, \cdots \overbrace{0, \cdots, 0}^{d-1}, \alpha_1, \overbrace{0, \cdots, 0}^{d-1}, \alpha_0)_2,$$

$$b = (B^{w-1}, \cdots, B^0)_{2^d} = (\underbrace{0, \cdots, 0}_{d-1}, \beta_{w-1}, \underbrace{0, \cdots, 0}_{d-1}, \beta_{w-2}, \cdots \underbrace{0, \cdots, 0}_{d-1}, \beta_1, \underbrace{0, \cdots, 0}_{d-1}, \beta_0)_2,$$

for all $\alpha = (\alpha_{w-1}, \cdots, \alpha_0)_2, \beta = (\beta_{w-1}, \cdots, \beta_0)_2 \in [0, 2^w - 1]$ by using Algorithm **40** and providing G and Q as input points. The threshold value q is set to $q = 2^w$. The so-called "double count" parameter c is set to $c = d$. In the `Dbl` step of our pre-computation method we see that $c = d$ enforces d point doublings thereby achieving d leftshifts, which result in sequences of d consecutive zeros. It is important to notice that generating all $\Gamma_{\alpha,\beta}$ (for $w > 1$) will take a long time, because we effectively compute 2^{2w+1} single point multiplications (although with "sparse" scalars) with the binary left-to-right method. We need 2^{2w+1} transformations to affine coordinates and 2^{2w} extra point additions to derive the sum from the single point multiplications.

Comparing the *simultaneous fixed-point method* with its counterpart for single multiplication (see Algorithm **39**) shows that both algorithms are quite similar. Ignoring the different pre-computations, both algorithms can be considered to be equally fast for the same window size w.

Algorithm 43: `dblPointMul3`$(\cdot)$: Simultaneous fixed-point method

Input: Scalars $k = (K^{w-1}, \cdots, K^0)_{2^d}$, $l = (L^{w-1}, \cdots, L^0)_{2^d}$
Aux. Input: w, t, $d = \lceil t/w \rceil$, points $\Gamma_{\alpha,\beta} = a \cdot G + b \cdot Q \; \forall \alpha, \beta \in [0, 2^w - 1]$
Output: $k \cdot G + l \cdot Q$
Write each K^j, L^j as bitstrings, $K_j = (K^j_{d-1}, \cdots, K^j_0)_2$, $L^j = (L^j_{d-1}, \cdots, L^j_0)_2$;
$\tilde{T} = (\tilde{x}, \tilde{y}, \tilde{z}) \leftarrow \infty$;
for i *from* $d-1$ *downto* 0 **do**
  $\tilde{T} \leftarrow$ `pointDbl`$(\tilde{T})$;
  $\alpha \leftarrow (K_i^{w-1}, \cdots, K_i^1, K_i^0)_2$; $\beta \leftarrow (L_i^{w-1}, \cdots, L_i^1, L_i^0)_2$;
  $\tilde{T} \leftarrow$ `pointAdd`$(\tilde{T}, \Gamma_{\alpha,\beta})$;
Return $X \leftarrow (\tilde{x}/\tilde{z}^3, \tilde{y}/\tilde{z}^2)$;

5.3.3 Point addition

Point addition is an atomic operation, i.e., we cannot further improve the steps executed here, because they are pre-defined by the curve's equation and the chosen coordinate system.

Algorithm 44: `pointAdd`$(\cdot)$: Addition of points $\tilde{P1}, P2$ [28, Alg. 3.22]

Input: Points $\tilde{P1} = (\tilde{x1}, \tilde{y1}, \tilde{z1})$, $P2 = (x2, y2)$
Aux. Input: Prime p
Output: $\tilde{P3} = (\tilde{x3}, \tilde{y3}, \tilde{z3}) = \tilde{P1} + P2$
if $P2 = \infty$ **then**
  Return $\tilde{P3} \leftarrow (\tilde{x1}, \tilde{y1}, \tilde{z1})$;
if $\tilde{P1} = \infty$ **then**
  Return $\tilde{P3} \leftarrow (x2, y2, 1)$;
$t1 \leftarrow$ `modMul`$(\tilde{z1}, \tilde{z1}, p)$; $t2 \leftarrow$ `modMul`$(t1, \tilde{z1}, p)$; $t1 \leftarrow$ `modMul`$(t1, x2, p)$;
$t2 \leftarrow$ `modMul`$(t2, y2, p)$; $t1 \leftarrow$ `modSub`$(t1, \tilde{x1}, p)$; $t2 \leftarrow$ `modSub`$(t2, \tilde{y}1, p)$;
if $t1 = 0$ **then**
  if $t2 = 0$ **then**
    $\tilde{P3} \leftarrow (x2, y2, 1)$; Return $\tilde{P3} \leftarrow$ `pointDbl`$(\tilde{P3})$;
else
  Return $\tilde{P3} \leftarrow \infty$;
$\tilde{z3} \leftarrow$ `modMul`$(z1, t1, p)$; $t3 \leftarrow$ `modMul`$(t1, t1, p)$; $t4 \leftarrow$ `modMul`$(t1, t3, p)$;
$t3 \leftarrow$ `modMul`$(t3, \tilde{x1}, p)$; $t1 \leftarrow$ `modAdd`$(t3, t3, p)$; $\tilde{x3} \leftarrow$ `modMul`$(t1, t1, p)$;
$\tilde{x3} \leftarrow$ `modSub`$(\tilde{x3}, t1, p)$; $\tilde{x3} \leftarrow$ `modSub`$(x3, t4, p)$; $t3 \leftarrow$ `modSub`$(t3, \tilde{x3}, p)$;
$t3 \leftarrow$ `modMul`$(t3, t2, p)$; $t4 \leftarrow$ `modMul`$(t4, \tilde{y}1, p)$; $\tilde{y3} \leftarrow$ `modSub`$(t3, t4, p)$;
Return $\tilde{P3} = (\tilde{x3}, \tilde{y3}, \tilde{z3})$;

Recall that we chose to compute point addition in mixed coordinates (see Section **5.1.2**). The input to Algorithm **44** consists of two points $\tilde{P1}, P2 \in E(\mathbb{Z}_p)$. $\tilde{P1}$ is a point in Jacobian representation, $P2$ is in affine coordinates. The output is derived according to an adaption of Equation **5.2** for mixed representation. Observe that we denote modular subtraction by `modSub`$(\cdot)$, addition by `modAdd`$(\cdot)$, halving by `modHalf`$(\cdot)$ and modular multiplication by `modMul`$(\cdot)$. Modular arithmetic is performed modulo p.

5.3.4 Point doubling

We have evaluated two distinct algorithms for point doubling. The first algorithm will compute $2 \cdot \tilde{P}$, where $\tilde{P}$ is a point in Jacobian representation. The second method will

compute $2^m \cdot \tilde{P}$ – that means m successive doublings. We start with listing the single point doubling algorithm.

Algorithm 45: `pointDbl`$(\cdot)$: Point doubling in Jacobian coordinates [28, Alg. 4.21]

Input: Point $\tilde{P1} = (\tilde{x1}, \tilde{y1}, \tilde{z1})$
Aux. Input: Prime p
Output: $\tilde{P2} = 2 \cdot \tilde{P1}$
if $P1 = \infty$ **then**
  Return $\tilde{P2} \leftarrow \infty$;
$t1 \leftarrow$ `modMul`$(\tilde{z1}, \tilde{z1}, p)$; $t2 \leftarrow$ `modSub`$(\tilde{x1}, t1, p)$; $t1 \leftarrow$ `modAdd`$(\tilde{x1}, t1, p)$;
$t2 \leftarrow$ `modMul`$(t2, t1, p)$; $t2 \leftarrow$ `modAdd`$(t2, t2, p)$; $t2 \leftarrow$ `modAdd`$(t2, t2, p)$;
$\tilde{y2} \leftarrow$ `modAdd`$(\tilde{y1}, \tilde{y1}, p)$; $\tilde{z2} \leftarrow$ `modMul`$(\tilde{y2}, \tilde{z1}, p)$; $\tilde{y2} \leftarrow$ `modMul`$(\tilde{y2}, \tilde{y2}, p)$;
$t3 \leftarrow$ `modMul`$(\tilde{y}2, \tilde{x}1, p)$; $\tilde{y2} \leftarrow$ `modMul`$(\tilde{y2}, \tilde{y2}, p)$; $\tilde{y2} \leftarrow$ `modHalf`$(\tilde{y2}, p)$;
$\tilde{x2} \leftarrow$ `modMul`$(\tilde{t2}\tilde{t2}, p)$; $t1 \leftarrow$ `modAdd`$(t3, t3, p)$; $\tilde{x2} \leftarrow$ `modSub`$(\tilde{x2}, t1, p)$;
$t1 \leftarrow$ `modSub`$(t3, \tilde{x2}, p)$; $t1 \leftarrow$ `modMul`$(t1, t2, p)$; $\tilde{y2} \leftarrow$ `modSub`$(t1, \tilde{y}2, p)$;
Return $\tilde{P2} \leftarrow (\tilde{x2}, \tilde{y2}, \tilde{z2})$;

Suppose we have to compute $2^m \cdot \tilde{P}$, which especially occurs in the pre-computation step. This can be achieved either by calling the algorithm presented above m-times or by using the next algorithm only once. The possibly faster method listed in the following was previously denoted by `repPointDbl`$(\cdot)$ and takes m as an extra argument.

Algorithm 46: `repPointDbl`$(\cdot)$: Repeated point doubling in Jacobian coordinates [28, Alg. 4.23]

Input: Point $\tilde{P1} = (\tilde{x1}, \tilde{y1}, \tilde{z1})$, integer $m > 0$
Aux. Input: Prime p
Output: $\tilde{P2} = 2^m \cdot P1$
if $\tilde{P} = \infty$ **then**
  Return $\tilde{P2} \leftarrow \infty$;
$\tilde{x2} \leftarrow \tilde{x1}$; $\tilde{z2} \leftarrow \tilde{z1}$; $r \leftarrow m$;
$\tilde{y2} \leftarrow$ `modAdd`$(\tilde{y1}, \tilde{y1}, p)$; $w \leftarrow$ `modMul`$(\tilde{z1}, \tilde{z1}, p)$; $w \leftarrow$ `modMul`(w, w, p);
while $r > 0$ **do**
  $a \leftarrow$ `modMul`$(\tilde{x2}, \tilde{x2}, p)$; $a \leftarrow$ `modSub`(w, a, p); $c \leftarrow$ `modAdd`(a, a, p);
  $a \leftarrow$ `modSub`(a, c, p); $y \leftarrow$ `modMul`$(\tilde{y2}, \tilde{y2}, p)$; $b \leftarrow$ `modMul`$(\tilde{x2}, y, p)$;
  $\tilde{x2} \leftarrow$ `modMul`(a, a, p); $c \leftarrow$ `modAdd`(b, b, p); $\tilde{x2} \leftarrow$ `modSub`$(\tilde{x2}, c, p)$;
  $\tilde{z2} \leftarrow$ `modMul`$(\tilde{z2}, \tilde{y2})$; $y \leftarrow$ `modMul`(y, y, p);
  $r \leftarrow r - 1$;
  if $r > 0$ **then**
    $w \leftarrow$ `modMul`(w, y, p);
  $b \leftarrow$ `modSub`$(b, \tilde{x2}, p)$; $a \leftarrow$ `modMul`(a, b, p); $a \leftarrow$ `modAdd`(a, a, p);
  $\tilde{y2} \leftarrow$ `modSub`(a, y, p);
$\tilde{y2} \leftarrow$ `modHalf`$(\tilde{y2}, p)$;
Return $\tilde{P2} \leftarrow (\tilde{x2}, \tilde{y2}, \tilde{z2})$;

The algorithm trades $m - 1$ modular additions, halvings and one multiplication for two squarings (compared to applying Algorithm **45** m-times). Due to the fact that we do not distinguish between squaring and multiplying (i.e., we use the same algorithm for both), we have to carefully evaluate for which choice of m the repeated doubling algorithm is faster than Algorithm **45**.

5.4 Prime field arithmetic

We have divided the prime field arithmetic in two subsets as seen in Figure **5.1**. Arithmetic in $\mathbb{Z}_p$ is required by the point doubling and point addition methods and therefore executed frequently during the calculation of $k \cdot G$ or $u1 \cdot G + u2 \cdot Q$. Arithmetic modulo n is performed only by the ECDSA protocol layer and therefore not as critical for the overall performance as arithmetic in $\mathbb{Z}_p$.

Due to these observations, we have implemented adapted versions of the same algorithms that were already applied to XTR-DSA (see Section **4.4.3**, Section **4.4.2**, and Section **4.4.1**). The latter algorithms are used for computations in $\mathbb{Z}_n$ only.

Addition, subtraction, halving, and multiplication in $\mathbb{Z}_p$ have been implemented as a part of Uhsadel's master thesis [69]. His ASM routines are highly optimized and based on the fixed and special form of the modulus p. Please refer to Uhsadel's thesis for more details.

5.5 Results

In this section we will detail the benchmark results of our implementation. The presentation is divided in two parts. We start with analyzing the results for the ECC arithmetic layer (point multiplication methods), then we select the best candidates to evaluate the overall behaviour of the signature generation and verification primitives. Benchmarking results of the prime field arithmetic can be found in Section **4.5.1** and Uhsadel's thesis [69].

5.5.1 ECC arithmetic

We did not benchmark point addition and point doubling algorithms, because we have implemented methods for mixed coordinates only. Instead, we start with comparing the single point multiplication methods. The parameter w (window size) listed in each table ranges from the necessary minimum to the maximum possible on the ATMEGA128L. Recall that a stored point for SECP160R1 needs at least 40 bytes of RAM memory and the ATMEGA128L offers 4KB . The maximum value of w is $w = 7$ for the *sliding window*

Method	**w**	**Points stored**	$t_{\texttt{precomp}}$	$t_{\texttt{mul}}$
Binary left-to-right	–	–	–	1.190s
Sliding window method	3	$2^2 = 4$ pts	0.196s	0.970s
	4	$2^3 = 8$ pts	0.391s	0.893s
	5	$2^4 = 16$ pts	0.777s	0.881s
	6	$2^5 = 32$ pts	1.552s	0.845s
	7	$2^6 = 64$ pts	3.099s	0.814s
Fixed point method	3	$2^3 = 8$ pts	2.695s	0.572s
	4	$2^4 = 16$ pts	6.847s	0.457s
	5	$2^5 = 32$ pts	15.517s	0.369s
	6	$2^6 = 64$ pts	34.569s	0.312s

Table 5.5: Performance of single point multiplication algorithms

method and $w = 6$ for the *fixed-point method*. With this choice of w, each pre-computation

table occupies $64 \cdot 40$ bytes = 2560 bytes in RAM. The next increase of w yields a table of 5.1KB size – which is far more RAM than offered by the μP.

Looking at Table **5.5**, it is no surprise that the *fixed-point method* performs best. Assuming that tables have to be pre-computed only once during a mote's lifetime, even a pre-computation time of 34s can be tolerated. Interestingly, the *sliding window method* with $w = 7$ performs worse than the *fixed-point method* whith $w = 3$, when comparing pre-computation times – while the *fixed-point method* requires only 1/8 stored points and performs 30% faster.

The simultaneous multiplication methods are based on the so-called *Shamir's trick* which was also benchmarked by us. The first modification of *Shamir's trick* represents a refinement of the original, allowing for a better scalability on low-power devices. Table **5.6** reflects this property, because the steps between *Shamir's trick* with $w = 2$ and $w = 3$ are not possible with the original algorithm. By setting q such that $q \in [2^2 + 1, 2^3 - 1]$ we truncate the pre-computation table and achieve a performance that increases in smaller steps for every increase of q. It is obvious that setting $q = 5$ and $w = 3$ has the best ratio of added performance to increased table size. Even better, the step from $(w = 2, q = 4)$ to $(w = 3, q = 5)$ yields already 50% of the performance increase that is achieved by setting $(w = 3, q = 8)$ – while requiring only 40% of the 64 points. Comparing the *simultaneous fixed-point method* with its single point multiplication counterpart reveals that for $w = 3$ both methods are approximately equally fast. The enormous pre-computation overhead induced by selecting $w = 3$ was already expected in the previous sections.

Method	**w**	**q**	**Points stored**	$t_{\texttt{precomp}}$	$t_{\texttt{mul}}$
Shamir's trick w/ truncated table	2	3	$3^2 = 9$ pts	0.398s	1.276s
Shamir's trick	2	4	$(2^2)^2 = 16$ pts	0.982s	1.140s
Shamir's trick w/ truncated table	3	5	$5^2 = 25$ pts	1.778s	1.083s
	3	6	$6^2 = 36$ pts	2.867s	1.044s
	3	7	$7^2 = 49$ pts	4.202s	1.037s
Shamir's trick	3	8	$(2^3)^2 = 64$ pts	5.857s	1.026s
Simultaneous fixed-point method	2	4	$(2^2)^2 = 16$ pts	6.687s	0.779s
	3	8	$(2^3)^2 = 64$ pts	43.718s	0.575s

Table 5.6: Performance of simultaneous point multiplication algorithms

5.5.2 Overall performance of ECDSA

To use both ECDSA primitives on a constrained device, a suitable parameter set for single and simultaneous multiplications has to be selected. We chose to benchmark three distinct parameter sets with different goals: (**1**) best signature generation performance, (**2**) best verification performance, and (**3**) "economic" performance. By "economic" we denote a parameter set that achieves a good overall performance with a reasonable pre-computation and storage overhead. Note that (**1**) and (**2**) are extreme setups and occupy almost all RAM to store their respective pre-computation tables.

Operation	$t_{\texttt{sign-max}}$	$t_{\texttt{verify-max}}$	$t_{\texttt{eco}}$
ECDSA: Signature generation	0.652s	0.918s	0.918s
ECDSA: Signature verification	1.746s	0.938s	1.486s

Table 5.7: Overall performance of our ECDSA implementation

To achieve the best signature generation performance, we decided to use the *fixed-point method* with $w_{\texttt{sngl}} = 6$. We have combined this setup with the modified *Shamir's trick* algorithm where we used $w_{\texttt{dbl}} = 2$ and $q = 3$, thereby truncating the table to only $3^2 = 9$ entries. The speed is quite notable, although it is bought at the cost of 35 seconds for pre-computation. Table **5.7** also shows that we can achieve a symmetrical behaviour for the signature generation and verification methods, when using *fixed-point methods* with $w_{\texttt{sngl}} = 3$ and $w_{\texttt{dbl}} = 3$. However, this performance implies that we store $8 + 64$ points in RAM and have a pre-computation phase of 47 seconds. The best choice for real-world applications would be the "eco" setup. Here, we choose the *single fixed-point method* with $w_{\texttt{sngl}} = 3$ and *Shamir's trick* with $w_{\texttt{dbl}} = 3$ and the threshold $q = 5$. We have to store only $8 + 25$ points computed in about 4.5 seconds.

Comparing the listed times with the execution time of the multiplication methods we see an overhead of roughly 40%. This already hints at a great optimization potential, because the 40% of extra time is induced only by the ECDSA protocol and hence the arithmetic in $\mathbb{Z}_n$. We will further discuss the implementation in Chapter **7**.

5.6 Overview of optimizations

We will now give an overview of optimizations for the ECDSA signature scheme. We start by listing the optimizations that were actually applied to our implementation.

ASM All prime field arithmetic in $\mathbb{Z}_p$ has been implemented in ASM and is optimized for the ATMEGA128L micro-processor.

Curvespecific optimizations The curve defined by the SECP160R1 parameter set uses a *Mersenne* prime p for the underlying field $\mathbb{Z}_p$. The special form of p allows the application of special optimizations. The fact that only one modulus is supported by the ASM routines is exploited by unrolling all loops and "hardcoding" the prime.

Point representation Using the mixed point representation for point addition and doubling performs best compared to other representations.

Fixed-point scalar multiplication The point multiplication $k \cdot G$ is performed using a *fixed-point method*. Instead of re-using the single multiplication for the verification primitive of ECDSA, we have developed and implemented a simultaneous variant of the *fixed-point method*. This method allows to compute $k \cdot G + l \cdot Q$ nearly as fast as $k \cdot G$ when using the single *fixed-point method*.

We focused on algorithms and improvements that were promising notable gains in speed, while implementing them in NESC was feasible for the given amount of time. Due to the fact that elliptic curve cryptography is relatively well studied, it has been subject to various attempts to speed up the computation. We will now list some other optimiziations that might be worthwhile to test.

Exponent re-coding Methods that encode the scalar k in non-adjacent forms (NAF) [37] and others exploit the property of ECC that point addition is as expensive as point subtraction. The goal of re-coding k is to achieve longer sequences of zeros, thus reducing the hamming-weight of the scalar. The lower the hamming-weight of k, the fewer point additions/subtractions are needed when computing $k \cdot G$.

More ASM When comparing the speed of our binary left-to-right method to Gura's implementation, we see that our NESC implementation is significantly slower. The reason for this is obvious, we have used ASM code only for the core field arithmetic, whereas Gura implemented the whole ECC arithmetic (point addition, doubling, and scalar multiplication) in ASM. Implementing our sophisticated multiplication algorithms in pure ASM is likely to dramatically increase the overall performance of ECDSA.

Offline pre-computation In order to pre-compute even more points while decreasing the RAM usage it might be worthwhile to perform pre-computation offline. Precomputation tables can be stored in the 512KB FLASH memory provided by the MICAz mote. By using bigger pre-computation tables, the speed of our implementation could be improved even more. Further, no pre-computation has to be performed by the mote itself.

Point compression Given only the x coordinate of a point (e.g., the public key) and the parameters of an elliptic curve, we can derive y from x by solving the curve equation (see Equation **5.1**). This property can be applied to achieve “point compression”, thereby reducing the affine representation of a point on a 160-bit curve from two 160-bit integers (x and y coordinate) to one 161-bit integer (x coordinate plus extra bit). The additional bit is required in order to distinguish between the two values y_1 and y_2 solving the quadratic curve equation. Although point compression does not increase the performance, it is a suitable way to decrease the memory and bandwidth required for storing and transmitting the key.

6 NTRUSign

This chapter presents our implementation of the NTRUSign signature scheme. It is organized as follows: we start with an introduction on NTRUSign and its mathematical foundations. Then, we elaborate on the implementation of the protocol's primitives with focus on the key generation process. We conclude this chapter with the evaluation of the core operation's performance and the overall execution times.

6.1 Introduction

NTRUSign is a relatively new public key scheme by Hoffstein *et al.*, published in its final version in 2003 [32]. NTRUSign is the successor of the "NTRU Signature Scheme" (NSS), which was published [34] and successfully attacked [23] in 2001. NTRUSign is based on NTRU [33], whose main arithmetic operation is the convolution of two polynomials $f(X), g(X) \in \mathbb{Z}[X]/X^N - 1$, where $h(X) = f(X) * g(X)$ with coefficients

$$h_k = \sum_{i+j \equiv k \bmod N} f_i \cdot g_j, \quad h_k, f_i, g_j \in \mathbb{Z},$$

is called the *convolution product*. This operation can be executed efficiently in software, which makes NTRU based schemes (NSS, NTRUSign) promising candidates for signature generation and verification primitives on constrained devices such as WSN motes. In analogy to the identification of seperate layers for ECDSA and XTR-DSA, we can divide the signature generation and verification primitives in: (**1**) the NTRUSign protocol layer and (**2**) the NTRU arithmetic layer where convolution is the only, atomic operation. Generating an NTRUSign keypair is a fairly complex process and involves more algorithms which we will not classify in terms of layers.

More on the mathematical background of NTRU will be explained in the next section, while the remainder of this chapter focuses on the algorithms required to implement the NTRUSign protocol layer.

6.1.1 Related work

NTRUSign is patented, therefore no public implementation is available. We found one paper by Blass, Junker, and Zitterbart [10] where they compared (H)ECC, XTR, NTRU, and RSA. The authors have implemented these asymmetric schemes for the ATMega128L micro-processor and measured the overall performance and the performance of the most time consuming operations.

6.1.2 Mathematical background

We will now briefly introduce the mathematical foundations of the NTRUSign lattice and the implications of polynomial convolution.

Lattices

An integer lattice is a discrete subgroup of $\mathbb{Z}^m$. Each element v of this subgoup is called *lattice point* and is often given as a vector. Lattices are most commonly represented by giving their bases. Given a complete basis, the lattice can be constructed by all linear combinations of each linearly independent basis vector. Note that the same lattice can be constructed using different bases, which is a property that allows the construction of NTRUSign.

Definition 21 (Lattice [36, p. 1]) *Let $b_1, b_2, .., b_n \in \mathbb{Z}^m$ be n linearly independent basis-vectors with m elements in $\mathbb{Z}$. Then L is called the* lattice*, where*

$$L(b_1, b_2, \cdots, b_n) = \sum_{i=1}^{n} b_i \cdot \mathbb{Z} = \left\{ \sum_{i=1}^{n} t_i \cdot b_i | t_1, t_2, \cdots, t_n \in \mathbb{Z} \right\}.$$

The vectors $(b_1, b_2, \cdots, b_n)$ are called the lattice basis. The dimension of L equals n, $\mathtt{DIM}(L) = n$.

Polynomials and vectors in NTRUSign

NTRU schemes employ polynomials chosen from the set $\mathbb{R} = \mathbb{Z}[X]/X^N - 1$, where N is a small integer. A polynomial $f(X) \in \mathbb{R}$ with $N - 1$ coefficients is written as

$$f(X) = f_0 \cdot X^0 + f_1 \cdot X^1 + f_2 \cdot X^2 + \cdots + f_{N-1} \cdot X^{N-1} = \sum_{i=0}^{N-1} f_i \cdot X^i, \quad f_i \in \mathbb{Z}.$$

The polynomial's coefficients can also be arranged as a row-vector $[f_0, f_1, \cdots, f_{N-1}]^T$. Note that we will write f instead of $f(X)$ as a shorthand for polynomials in the remainder of this chapter. Two operations are defined for $\mathbb{R}$: multiplication (known als convolution) denoted by '$*$' and addition denoted by '$+$'. $(\mathbb{R}, *, +)$ is a ring (see Definition **10**) of polynomials. Multiplication works similary to ordinary polynomial multiplication, but respects the relation

$$X^{N+k} \equiv X^k \bmod X^N - 1$$

for any $0 \leq k < N$. According to this relation, the convolution of two polynomials $f, g \in \mathbb{R}$ is defined as $h = f * g$, where h's coefficients are given by

$$h_k \equiv \sum_{i+j \equiv k \bmod N} f_i \cdot g_j \quad (0 \leq k < N). \tag{6.1}$$

Addition is performed coefficient-wise and reduction modulo $X^N - 1$ is not applied,

$$h_k \equiv \sum_{k=0}^{k<\mathtt{MAX}(i,j)} f_k + g_k.$$

There exists a matrix M_a associated with any element $a \in \mathbb{R}$, known as the circulant matrix. The rows of M_a are the rotations of a written as row-vector. Given the matrix-representations M_a and M_b of $a, b \in \mathbb{R}$, the polynomial convolution of $a * b$, can also be expressed as the product of their respective matrices $M_a \cdot M_b$.

Definition 22 (Circulant matrix [31, p. 4]) *Let $a \in \mathbb{R}$, M_a is the associated $N \times N$-matrix indexed in the range $[0, N-1]$,*

$$M_a = \begin{pmatrix} a_0 & a_1 & \cdots & a_{N-1} \\ a_{N-1} & a_0 & \cdots & a_{N-2} \\ \vdots & \vdots & & \vdots \\ a_1 & a_2 & \cdots & a_0 \end{pmatrix}.$$

The matrix M_a is known as the circulant matrix *of a. The rows are circular rotations of a's row-vectors.*

The NTRUSign lattice

All NTRU based schemes require the generation of two polynomials $f, g \in \mathbb{R}$. For NTRUSign, these polynomials are chosen to be binary (i.e., all coefficients are in $\{0, +1\}$) or trinary (coefficients in $\{-1, 0, +1\}$). By choosing the coefficients from these specific sets it is assured that the generated polynomials are "small", i.e., their centered norm denoted by $|| \cdot ||$ is "small".

Definition 23 (Centered norm [36, p. 3, Equ. 1]) *Let $f = f_0 \cdot X^0 + \cdots + f_{N-1} \cdot X^{N-1} \in \mathbb{R}$ be a polynomial of degree $N-1$. Let μ_f be the average of the coefficients of f, where $\mu_f = \frac{1}{N} \sum_{i=0}^{N-1} f_i$. The* centered norm *of the polynomial f is defined as $||f||$, where*

$$||f||^2 = \sum_{i=0}^{N-1} (f_i - \mu_f)^2.$$

A modulus q is chosen as a power of two, $\mathbb{R}_q$ is called the truncated polynomial ring, i.e., the coefficients are in $\mathbb{Z}_q$. When choosing f and g it has to be taken care that both are invertible in $\mathbb{R}_q$, i.e., there exist f^{-1} and g^{-1} such that $f^{-1} * f \equiv g * g^{-1} \equiv 1 \bmod q$. In order to generate the full NTRUSign lattice, two *base completion vectors* F and G must be found such that $f * G - g * F = q$. The details of generating a valid keypair will be discussed in Section **6.2**. Assume that we have found (f, g, F, G), we can set the public polynomial h such that

$$h \equiv f^{-1} * g \bmod q \equiv g^{-1} * f \bmod q \tag{6.2}$$

(when using a "standard" basis) and (by construction)

$$f * h = g + k_1 \cdot q, \quad g * h = f + k_2 \cdot q, \tag{6.3}$$

for some constants $k_1, k_2 \in \mathbb{R}$.

The NTRUSign-lattice L_h is generated by all linear combinations of the rows of the following $2N \times 2N$-matrix

$$\begin{pmatrix} M_1 & M_h \\ M_0 & M_q \end{pmatrix},$$

known as the public basis. Here, M_1, M_0, M_h, and M_q represent the $N \times N$ circulant matrices of the N-sized row-vectors $[1, 0, \cdots, 0]^T, [0, 0, \cdots, 0]^T, [h_0, h_1, \cdots, h_{N-1}]^T$, and $[q, 0, \cdots, 0]^T$ respectively. Additionally, L_h can also be generated by a different, private basis

$$\begin{pmatrix} M_f & M_g \\ M_F & M_G \end{pmatrix},$$

where each sub-matrix M_i represents the circulant matrix of f, g, F, and G respectively. Using Equation **6.2** and Equation **6.3** the relation between the private and the public basis can be expressed as

$$\begin{pmatrix} M_f & -M_{k_1} \\ M_F & -M_{k_2} \end{pmatrix} \cdot \begin{pmatrix} M_1 & M_h \\ M_0 & M_q \end{pmatrix} = \begin{pmatrix} M_f & M_g \\ M_F & M_G \end{pmatrix}.$$

The difference between both bases is the fact that the private basis is "small" (i.e., the polynomials have reasonably small coefficients), while half of the public basis (i.e., M_q and M_h) is not "small". Therefore, only the private basis can be used to generate "small" polynomials in L_h.

The NTRUSign scheme can also be used with a "transposed" lattice, i.e., the lattice generated by the transposed matrix of the private base,

$$\begin{pmatrix} M_f & M_g \\ M_F & M_G \end{pmatrix}^T = \begin{pmatrix} M_f & M_F \\ M_g & M_G \end{pmatrix}.$$

The corresponding public polynomial h is set such that

$$h \equiv f^{-1} * F \bmod q \equiv f^{-1} * G \bmod q.$$

"Hard" problem

The "hard" problem NTRUSign is based on is known as the *Closest Vector Problem* (CVP). Given an arbitrary vector (polynomial) $f \in \mathbb{Z}^m$, it is assumed to be "hard" to find a *lattice point* g "close" to f, i.e., $||f - g||$ is "small". This applies to NTRUSign as follows: given the private, small polynomials (f, g) and the *completion pair* (F, G) (and thereby a basis of "short" vectors) the signer finds a *lattice point* s close to a "message point[1]", where s is published as the signature.

6.2 The NTRUSign protocol

In this section we will present the algorithms required to generate NTRUSign keys, signatures, and to verify the generated signatures. However, first we need to introduce the parameters of NTRUSign. Depending on the choice of binary or trinary polynomials $f, g \in \mathbb{R}$, the parameter-sets differ. The basic parameter-set $(N, q, \texttt{NormBound}^2, B, t)$ is made up of:

1. N is the security parameter and defines the degree of all polynomials in $\mathbb{R} = \mathbb{Z}[X]/X^N - 1$.

2. q is used as the modulus, it defines the underlying field $\mathbb{Z}_q$ of the truncated polynomial ring $\mathbb{R}_q$. For efficiency reasons, q is always chosen to be a power of two.

3. $\texttt{NormBound}^2$ is an integer indicating the maximum "distance" between two vectors, i.e., when $f, g \in \mathbb{R}$ satisfy $||f - g|| < \texttt{NormBound}$ they are said to be "close". This parameter is important during the verification process, it will be discussed in detail later on.

4. $B \geq 0$ defines how many additional private bases are generated by the keypair generation algorithm. For $B = 0$, only one keypair is generated. For all $B \geq 1$, B additional, independent bases are generated. These extra bases are known as *perturbation bases* and are used to "perturb" a "message point". Perturbations are employed to harden the NTRUSign scheme against so-called "transcript attacks" [24].

[1] i.e., a polynomial representative of a digital message generated using a hash function

5. $t \in \{$"transpose","standard"$\}$ indicates whether the standard

$$(f_i, f'_i = F_i, h_i \equiv f_i^{-1} * g_i \bmod q)$$

or the transposed

$$(f_i, f'_i = g_i, h_i \equiv f_i^{-1} * F_i \bmod q)$$

lattice bases are used for all $i \in [0, B]$.

In cases where binary polynomials $f, g \in \mathbb{R}$ are desired, the foregoing parameter-set is completed by (d_f, d_g), defining the number of '+1' for the polynomials f and g. When trinary polynomials are used (this lowers the requirements for N) then the parameters are completed by d, where $d+1$ equals the number of '+1' and d the number of '-1' for both polynomials. Our implementation uses $N = 127$ and $q = 256$ combined with transposed bases. The interested reader is refered to the Appendix (Section **A.1.3**) for the complete listing of our choice of NTRUSIGN parameters.

According to Definition **6**, the NTRUSIGN scheme provides of three different primitives. The signature generation and verification routines have been implemented as one NESC component. A program to generate NTRUSIGN keypairs was implemented in C with excessive use of the GMP multi-precision library.

6.2.1 Keypair generation

The generation of a valid NTRUSIGN keypair requires a source of (pseudo-)random bits for the construction of the binary/trinary polynomials f and g. The following algorithm[2] will output the public polynomial h and $B+1$ polynomials (f_k, f'_k, h_k), which we will use to perturb a "message point" and generate the corresponding signature. Additionally, the *basis completion pair* is generated. Note that the basis completion cannot be computed in cases where f and g are not relative prime.

Algorithm 47: NTRUSIGN: Keypair generation

Input: d_f, d_g for binary polynomials, d for trinary polynomials, desired bases B
Aux. Input: Prime q
Output: Public key h, private key consisting of bases $\{f_{*,k}, f'_{*,k}, h_{*,k}\}$ $\forall k \in [0, B]$

```
for k from B downto 0 do
    Generate random trinary/binary polynomials f_{*,k}, g_{*,k};
    Complete lattice basis, (F_{*,k}, G_{*,k}) ← completeBasis(f_{*,k}, g_{*,k});
    Invert polynomial, fq ← invertPoly(f_{*,k}, q);
    if t = "standard" then
        f'_{*,k} ← F_{*,k};
        Set polynomial h_{*,k} ← fq * g_{*,k} mod q;
    else
        f'_{*,k} ← g_{*,k};
        Set polynomial h_{*,k} ← fq * F_{*,k} mod q;
Return h ← h_{*,0}, {f_{*,0}, f'_{*,0}, h_{*,0}, ··· , f_{*,B}, f'_{*,B}, h_{*,B}};
```

No testvectors were available during the implementation, so we had to embed testing routines to check whether certain conditions such as

$$f * G - g * F \stackrel{?}{=} q$$

hold. However, we will not discuss these test any further, because they are neither very sophisticated nor of any relevance for the overall key generation.

[2]Note that we face a little problem with the notation: we cannot use f_k to index a set of polynomials, because by f_k we refer to the kth coefficient of f. To solve this, we will index a set of polynomials by $f_{*,k}$, where $f_{i,k}$ is the ith coefficient of the kth polynomial.

Instead, we will now elaborate on the most important algorithms to create valid NTRUSIGN keypairs. As hinted in the introduction, we have to strictly separate the key generation and signature generation/verification primitives, because they only share the convolution operation. Generating the NTRUSIGN lattice basis is not trivial (such as computing $k \cdot P$ for ECDSA keys when given a point multiplication algorithm), therefore we will explicitly cover the following two tasks: (**1**) computing the *basis completion pair* (F, G) for a given pair of polynomials (f, g) such that $f * G - g * F = q$ requires computing the resultants of f and g. In the algorithm above, we denote this step by `completeBasis`$(\cdot)$. (**2**) Inverting f in $\mathbb{R}_q$ involves inverting the polynomial modulo 2 and then using *Newton iteration* to compute $f^{-1} \bmod 2^e$, where $e = \mathtt{log2}(q)$. Please note that the key generation process is performed on a standard PC.

Completing the basis

Completing a basis with f and g chosen randomly as binary/trinary polynomials involves computing of the resultants of f and g. The resultant R_f of f with $X^N - 1$ is the product of f evaluated at all complex roots of $X^N - 1$.

Definition 24 (Resultants [12, Chap. 3]**)** *Let $f \in \mathbb{R}$ be a polynomial with N coefficients. Let $\Phi = \sum_{i=0}^{N-1} X^i$ be a modulus in $\mathbb{R}$. Let $R_f \in \mathbb{Z}$ denote the resultant of f with $X^N - 1$ where*

$$R_f \equiv \mathtt{resultant}(f, X^N - 1) \bmod \Phi \equiv \prod_{i=1}^{N-1} f(x^i) \bmod \Phi. \tag{6.4}$$

Deriving R_f and R_g is achieved by computing the resultants of f and g with $X^N - 1$ in $\mathbb{R}_{p_i}$ for a set of small prime numbers p_i. The resultants modulo p_i are combined to a single resultant in $\mathbb{Z}$ using a second algorithm. If the product $\prod_{\forall i} p_i$ is greater than the maximum possible resultant, the result will be obtained exactly. By defining ρ_f and ρ_g such that

$$\rho_f \equiv \prod_{i=2}^{N-1} f(x^i) \bmod \Phi, \quad \rho_g \equiv \prod_{i=2}^{N-1} g(x^i) \bmod \Phi, \tag{6.5}$$

and combining Equation **6.4** and Equation **6.5** and for some $k_f, k_g \in \mathbb{Z}[X]$ we can write

$$\rho_f \cdot f + k_f \cdot (X^N - 1) = R_f \bmod \Phi, \quad \rho_g \cdot g + k_g \cdot (X^N - 1) = R_g \bmod \Phi.$$

Assuming that R_f and R_g are relatively prime (i.e., $\mathtt{GCD}(R_f, R_g) = 1$) we can now use the *Extended Euclidean Algorithm* (EEA) to find $\alpha, \beta \in \mathbb{Z}$ such that

$$\alpha \cdot R_f + \beta \cdot R_g = 1. \tag{6.6}$$

By setting

$$F = -q \cdot \beta \cdot \rho_g, \quad G = q \cdot \alpha \cdot \rho_f, \tag{6.7}$$

finally a solution to $f * G - g * F = q$ is found. For more details and the corresponding proofs the interested reader is refered to [32].

Algorithm **48** computes the resultant $R'_f \bmod p$ and a polynomial $\rho_f \in \mathbb{R}_p$ for a given polynomial $f \in \mathbb{R}$, where p is a "small" prime. By `deg`$(\cdot)$ we denote a function that returns the degree of a polynomial.

Algorithm 48: `compResMod`$(\cdot)$: Resultant computation mod p [4, Alg. 2.2.7.1]

Input: Polynomial $f \in \mathbb{R}$, small prime p
Aux. Input: Integer N
Output: Polynomial $rhoF \in \mathbb{R}_p$, resultant $r \in \mathbb{Z}_p$
Init polynomials $a \leftarrow X^N - 1$; $b \leftarrow f$;
Init polynomials $v1 \leftarrow 0$; $v2 \leftarrow 1$;
$da \leftarrow \texttt{deg}(a)$; $db \leftarrow \texttt{deg}(b)$;
$ta \leftarrow da$; $c \leftarrow 0$; $r \leftarrow 1$;
while $b > 0$ **do**
  $c \leftarrow b_{db}^{-1} \cdot a_{da} \bmod p$;
  Set polynomial $a \leftarrow a - c * b * X^{da-db}$ in $\mathbb{Z}_p[X]$;
  Set polynomial $v1 \leftarrow v1 - v2 * c * X^{da-db}$ in $\mathbb{Z}_p[X]$;
  if $\texttt{deg}(a) < db$ **then**
    $r \leftarrow r \cdot b_{db}^{ta-\texttt{deg}(a)} \bmod p$;
    if ta *is odd* **and** db *is odd* **then**
      $r \leftarrow -r \bmod p$;
    $\texttt{swap}(a, b)$; $\texttt{swap}(v1, v2)$; $ta \leftarrow db$;
  $da \leftarrow \texttt{deg}(a)$; $db \leftarrow \texttt{deg}(b)$;
$r \leftarrow r \cdot b_0^a \bmod p$; $c \leftarrow B_0^{-1} \bmod p$;
Set polynomial $rhoF \leftarrow v2 * c * r \bmod p$;
Return $(rhoF, r)$;

Algorithm 49: `compRes`$(\cdot)$: Resultant computation over the integers [4, Alg. 2.2.7.2]

Input: Polynomial $f \in \mathbb{R}$
Aux. Input: Integer N
Output: Polynomial $rhoF \in \mathbb{R}$, resultant $r \in \mathbb{Z}$
Generate a list of P primes $\{p_0, p_1, \cdots, p_{P-1}\}$;
$pProd \leftarrow 1$; $r \leftarrow 1$; $rhoF \leftarrow 1$;
for j *from* 0 *to* $P-1$ **do**
  $t \leftarrow p_j \cdot pProd$;
  Resultants Compute resultants $(rhoF, resF) \leftarrow \texttt{compResMod}(f, p_j)$;
  GCD Find α, β such that $\alpha \cdot p_j + \beta \cdot pProd = 1$;
  $r \leftarrow r \cdot \alpha \cdot p_j + resF \cdot \beta \cdot pProd \bmod t$;
  Set polynomial $rhoF \leftarrow rhoF * \alpha * p_j + rhoF * \beta * pProd \bmod t$;
  $pProd \leftarrow t$;
Return $(rhoF, r)$;

The method listed above uses the resultant computation modulo p (see `Resultants` step) to finally compute the resultant R_f over the integers and $\rho_f \in \mathbb{R}$. The usage of Algorithm **48** is denoted by `compResMod`$(\cdot)$. However, another supplementary algorithm is required to perform the `GCD` step. We use EEA to compute α and β – the GMP program library offers a function called `mpz_gcdext`$(\cdot)$ for this purpose.

As stated before, we can complete the basis by projecting $f, g \in \mathbb{R}$ down to $\mathbb{Z}$ via resultants. The next algorithm calls the previously introduced methods to generate the resultants of f and g and then applies Equation **6.6** and Equation **6.7** (see `Eq1`, `Eq2`) to derive the basis completion vectors F and G. The part of the algorithm following the `Shrink` mark determines and subtracts multiples of f and g from F and G, in order to make the *basis completion pair* as small as possible.

Algorithm 50: `completeBasis`$(\cdot)$: Completing the basis [4, Alg. 3.5.1.1]

Input: Random polynomials $f, g \in \mathbb{R}$
Aux. Input: Integer N
Output: Completion polynomials $F, G \in \mathbb{R}$
$(rhoF, resF) \leftarrow$ `compRes`(f);
$(rhoG, resG) \leftarrow$ `compRes`(g);
Eq1 Find α, β, gcd such that $\alpha \cdot resF + \beta \cdot resG = gcd$;
Eq2 Set polynomials, $F \leftarrow -rhoG * \beta * q$; $G \leftarrow rhoF * \alpha * q$;
Shrink Init "reverse" polynomials, $fRev_0 = f_0$; $gRev_0 = g_0$;
for i *from* 1 *to* $N-1$ **do**
  $fRev_i = f_{N-i}$; $gRev_i = g_{N-i}$;
Set polynomial $t \leftarrow f * fRev + g * gRev$;
$(rhoT, resT) \leftarrow$ `compRes`(t);
Set polynomial $c \leftarrow rhoT * (fRev * F + gRev * G)$;
for i *from* 0 *to* $N-1$ **do**
  $c_i \leftarrow$ `floor`$(c_i/resT + .5)$;
$F \leftarrow F - c * f$; $G \leftarrow G - c * g$;
Return (F, G);

Inversion

Generating a valid NTRUSIGN basis also requires inversion of f in $\mathbb{R}_q$. Using an adaption of EEA for polynomials seems to be the best choice – the only problem is: q is no prime itself, but a power of a prime (remember that q was chosen as power of two for efficiency reasons). In order to solve this problem we use the EEA to compute $f' = f^{-1} \bmod 2$. Then, we pass $(f, f', 2, \texttt{log2}(q))$ to a an algorithm that uses *Newton iteration* [58] to derive $f^{-1} \bmod q$ from f'.

Algorithm 51: `extGcdPoly`$(\cdot)$: Extended Euclidean algorithm for polynomials

Input: Small polynomial $f \in \mathbb{R}$, prime p
Aux. Input: Integer N
Output: $f^{-1} \in \mathbb{R}_p$
Init polynomials $b \leftarrow 0$; $c \leftarrow 1$; $g \leftarrow f$; $d \leftarrow X^N - 1$;
while `deg`$(d) > 0$ **do**
  if `deg`$(d) >$ `deg`(g) **then**
    `swap`(d, g); `swap`(b, c);
  $j \leftarrow \texttt{deg}(d) - \texttt{deg}(g)$; $\alpha \leftarrow -d_{\texttt{deg}(f)} \cdot g^{-1}_{\texttt{deg}(g)} \bmod p$;
  Set polynomial $d \leftarrow d + \alpha * X^j * g$;
  Set polynomial $b \leftarrow b + \alpha * X^j * g$;
Return $b \leftarrow b * d_0^{-1} \bmod p$;

Algorithm 52: `iterateInverse`$(\cdot)$: Inversion modulo the power of a prime [58, p. 2]

Input: Polynomial $f \in \mathbb{R}$, prime p, inverse f' such that $f' * f \equiv 1 \bmod p$, integer e
Output: Polynomial $b \in \mathbb{R}_{p^e}$ such that $b * f \equiv 1 \bmod p^e$
$q \leftarrow p$;
Init polynomial $b \leftarrow f'$;
while $q < p^e$ **do**
  $q \leftarrow q^2$;
  Set polynomial $t \leftarrow (2 - (f * b)) \bmod q$;
  Set polynomial $b \leftarrow b * t \bmod q$;
Return $b \leftarrow b \bmod p^e$;

Given the previous two methods, we can construct the required `invertPoly`$(\cdot)$ method

easily.

Algorithm 53: `invertPoly`$(\cdot)$: Inversion of a polynomial

Input: Polynomial $f \in \mathbb{R}$, power of two $q = 2^e$
Output: Polynomial $f^{-1} \in \mathbb{R}_{p^e}$
$e \leftarrow \texttt{log2}(q)$;
$h \leftarrow \texttt{extGcdPoly}(f, 2)$;
$h \leftarrow \texttt{iterateInverse}(f, 2, h, e)$;
Return h;

6.2.2 Signature generation

Signing requires a hash function according to Definition **2**. The hash function is used to seed a "Pseudo Random Number Generator" (PRNG) which in turn generates a (pseudo-) random polynomial $i \in \mathbb{R}_q$. The polynomial i is what we previously called the "message point", i.e., a polynomial "fingerprint" of a digital message. The polynomial is derived by the next algorithm. We have implemented the `PRNG`$(\cdot)$ function as advised [4, Alg. 3.7.3.1] in the "Efficient Embedded Security Standards" (EESS) document .

Algorithm 54: `genMsgPoly`$(\cdot)$: Generation of a "message point" [4, Alg. 3.6.1.1]

Input: Digitial message m
Aux. Input: Integers N, q
Output: Message representative $i \in \mathbb{R}_q$
$c \leftarrow \texttt{log2}(q)$; $b \leftarrow \texttt{ceil}(c/8)$;
Generate N pseudo-random bytes, $H = (H_{N-1}, \cdots, H_0)_{2^8} \leftarrow \texttt{PRNG}(\texttt{SHA1}(m), N)$;
for t *from* 0 *to* $N-1$ **do**
  $i_t \leftarrow \texttt{AND}(H_t, 2^{8-(8 \cdot b - c)} - 1)$;
Return i;

By `AND`$(\cdot)$ we refer to a function that performs the bitwise logical 'and' operation on bytes. We pass $2^{8-(8 \cdot b-c)} - 1$ as bit mask for the operation, thereby setting all $(8 \cdot b - c)$ high-order bits of H_t to zero. The newly generated representative polynomial i is required by the signature generation and verification algorithms.

Recall that the polynomials $\{f_{*,k}, f'_{*,k}\}$ are chosen to be small. For $B = 1$ we have to store $N \cdot 4$ "small" coefficients in RAM. In order to save RAM, we have encoded the "small" polynomials f and f' from a naive representation with one byte per coefficient to a more compressed form with only $\lceil (N/2) \rceil$ bytes. This way, we can use one byte to

$(-1,-1)$	$(-1,0)$	$(-1,1)$	$(0,-1)$	$(0,0)$	$(0,1)$	$(1,-1)$	$(1,0)$	$(1,1)$
0xFF	0xF0	0xF1	0x0F	0x00	0x01	0x1F	0x10	0x11

Table 6.1: Mapping two trinary coefficents to one byte

represent two coefficients. We use trinary coefficients, therefore we can simply map two consecutive coefficients to one byte as displayed in Table **6.1**. By encoding the polynomials, we achieve a compression of 50%. However, note that we use a method `decodePoly`$(\cdot)$ which "decompresses" the encoded polynomials to N words in our signature generation algorithm.

Algorithm 55: NTRUSign: Signature generation [4, Alg. 3.5.2.1]

```
Input: Message m, private key {f_{*,k}, f'_{*,k}, h_{*,k}} ∀k ∈ [0, B]
Aux. Input: Base count B, integers N, q
Output: Digital signature s
i ← genMsgPoly(m);   k ← B;
while (1) do
  [Decompress]  p1 ← decodePoly(f_{*,k});
                p2 ← multiplyPoly(p1, i);
                for j from 0 to N − 1 do
                    p2_j ← floor(p2_j/q + 0.5);
                    i_j ← −i_j;
  [Decompress]  p1 ← decodePoly(f'_{*,k});
  [DblConv]     psC ← multiplyPoly(p1, p2);   p2 ← multiplyPoly(p1, i);
                for j from 0 to N − 1 do
                    p2_j ← floor(p2_j/q + 0.5);
                    i_j ← −i_j;
  [Decompress]  p1 ← decodePoly(f_{*,k});
                p3 ← multiplyPoly(p2, p1);
                for j from 0 to N − 1 do
                    psC_j ← psC_j + p3_j;
                    if k = B then
                        s_j ← psC_j;
                    else
                        s_j ← s_j + psC_j;
                if j = 0 then
                    Break;
                for j from 0 to N − 1 do
                    p1_j ← h_{j,k} − h_{j−1,k} mod q;
                i ← multiplyPoly(psC, p1) mod q;
Return s;
```

This algorithm involves computing $5 \cdot B - 1$ total convolutions. We have observed that it might be possible to combine the `DblConv` step into one single algorithm, i.e., perhaps we can construct some kind of "double convolution algorithm". More on this observation will be discussed in Section **6.3.2**. We also see how the B distinct private bases are applied. In fact, the message is effectively signed B times.

6.2.3 Signature verification

The verification algorithm calculates the distance between the polynomials $t = h_{*,0} * s$ and i, where $h_{*,0}$ is the public key, s the signature and i the message representative. When the distance is small enough (this is defined by the `NormBound`^{2} parameter) we can be sure that the signature is a *lattice point* generated with the private lattice basis $\{f_{*,0}, f'_{0,*}, h_{*,0}\}$ corresponding to $h_{*,0}$.

Algorithm 56: NTRUSIGN: Signature verification [4, Alg. 3.5.3.1]

Input: Message m, signature polynomial s
Aux. Input: Public key $h_{*,0}$, integer N
Output: `ind` $\in \{$"accept", "reject"$\}$
$i \leftarrow$ `genMsgPoly`(m);
Compute polynomial $t \leftarrow$ `multiplyPoly`$(h_{*,0}, s) \bmod q$;
Compute polynomial $e2 \leftarrow i - t \bmod q$ (set all $e2_i$ in range $[0, q-1]$);
$e2Sum \leftarrow 0$; $sSum \leftarrow 0$; $sqSum \leftarrow 0$;
for i *from* 0 *to* $N-1$ **do**
  $e2Sum \leftarrow e2Sum + e2_i$;
  $sSum \leftarrow sSum + s_i$;
  $sqSum \leftarrow sqSum + e2_i^2 + s_i^2$;
$sqNorm \leftarrow (N \cdot sqSum - sSum^2 - e2sum^2)/N$;
if $sqNorm \leq$ `NormBound`^{2} **then**
  `ind` $\leftarrow$ "accept";
else
  `ind` $\leftarrow$ "reject";
Return `ind`;

In contrast to the original algorithm, we avoid calculating the square root by simply comparing $sqNorm$ ("squared norm") to `NormBound`^{2} – instead of computing $\sqrt{sqNorm}$ and comparing it to `NormBound`.

When implementing the preceding algorithm it has to be taken care that $sqSum$ may exceed the maximum operand size of the target hardware. Our implementation uses special variants of the previously discussed prime field arithmetic functions to compute the centered norm.

6.3 NTRU arithmetic

Here we will present the algorithms for performing NTRU's main arithmetic operation, the convolution. We have evaluated two approaches: (single) convolution of $g = p1 * p2$ and double convolution with three operands, i.e., $g1 = p1 * p2, g2 = p1 * p3$.

6.3.1 Single convolution

The convolution of polynomials is the core operation of any NTRU scheme, therefore the speed of NTRUSIGN mainly depends on the speed of this operation. We have evaluated two algorithms to perform the single convolution operation. The first algorithm represents the very naive approach, we dubbed it the *simple convolution algorithm*. To enhance the convolution, we have implemented and improved a recursive multiplication algorithm proposed by Silverman in a technical report [59]. We call this the *modified Karatsuba method*. We will now briefly introduce the candidate algorithms for performing a single convolution.

Simple convolution algorithm

The *simple convolution algorithm* is easy to implement and has a very small footprint in RAM and ROM. The drawback of this algorithm is its complexity of $O(N^2)$ for polynomials with N coefficients each. The algorithm is an implementation of Equation **6.1**.

Algorithm 57: `multiplyPoly1`$(\cdot)$: The simple convolution method

Input: Polynomials $f, g \in \mathbb{R}$
Aux. Input: Integer N
Output: Polynomial $h = f * g$
Initiate polynomial $h \leftarrow 0$;
for i *from* 0 *to* $N-1$ **do**
  for j *from* 0 *to* $N-1$ **do**
    $k \leftarrow (i+j) \bmod N$;
    MAC $h_k \leftarrow h_k + f_i \cdot g_j$;
Return h;

Note that this algorithm operates in $\mathbb{R}$, hence, the coefficients are not modulo reduced. In order to apply reduction modulo q – and thereby performing convolution in $\mathbb{R}_q$ – we can add a reduction step after each inner iteration.

In [59] Silverman stated a second proposal: it might be worthwhile to exploit the the fact that some polynomials only have coefficients in $\{-1, 0, 1\}$. Instead of using multiplication to determine the coefficient of h, he suggested to use a `switch` construct of the following form (C syntax):

```
switch (f.coeff[i]) {
  case 0 : break;
  case +1: h[k] += g.coeff[j]; break;
  case -1: h[k] -= g.coeff[j]; break;
  default: h[k] += g.coeff[j] * f.coeff[i];
}
```

By using this construct we trade the multiplication and addition executed in the `MAC` step in Algorithm **57** for one single addition/subtraction and a certain overhead implied by the use of the `switch` statement. However, we will now briefly discuss this idea by dissecting the low-level (i.e., ASM) behaviour of `switch` statments.

A `switch`(v) statement typically uses v to index a table of jump-addresses. We assume 16-bit adresses, then the code generated by using the high-level `switch` command would calculate

$$ptr \leftarrow \texttt{BASE_ADDRESS} + 2 \cdot v.$$

Here, the constant dubbed as BASE_ADDRESS equals the 16-bit start address of the jump-table. After the calculation step, the pointer *ptr* points to a certain entry in the table, depending on the value of v. By jumping to the code-block *ptr* points to, the `switch`(v) statement executes the corresponding code for each case v. For example $v = 3$ results in $ptr = \texttt{baseAdress} + 6$, the CPU would jump to the address indicated by the fourth entry in the table. We see that using `switch` commands does not avoid multiplication, therefore we chose to omit the implementation of Silverman's second proposal.

Modified Karatsuba algorithm

The *high speed multplication algorithm* presented by Silverman [59] is based on Karatsuba's idea [5] and reduces the number of single-precision multiplications for two N-word integers from N^2 to $\frac{3}{4} \cdot N^2$ when used non-recursively. By applying Karatsuba's idea recursively, the complexity can be reduced even more, while increasing the RAM usage.

Assume we want to simply multiply two arbitrary polynomials $f, g \in \mathbb{Z}[X]$, where $h = f \cdot g, h \in \mathbb{Z}[X]$. Applying Karatsuba's idea to the multiplication of polynomials requires splitting f and g in two parts of approximately the same length $N/2$. Let $n_1 = \lfloor N/2 \rfloor, n_2 = N - n_1$, decompose the multiplicands f and g such that

$$f = f_1 + f_2 \cdot X^{n_1}, \quad g = g_1 + g_2 \cdot X^{n_1}.$$

By writing f and g in this specific manner, the result h can be constructed as

$$h = f \cdot g = f_1 \cdot f_2 + (g_1 \cdot f_2 + g_2 \cdot f_1) \cdot X^{n_1} + f_2 \cdot g_2 \cdot X^{2n_1}. \tag{6.8}$$

At this moment, we have to compute four inner products with operands of about $N/2$ coefficients each. Although the multiplicands are only half as long, the number of coefficient multiplications remains $4 \cdot (N/2)^2 = N^2$. The "trick" to reduce this complexity is to rewrite the coefficient of X^{n_1} such that

$$(g_1 \cdot f_2 + g_2 \cdot f_1) = (f_1 + f_2) \cdot (g_1 + g_2) - f_1 \cdot g_2 - f_2 \cdot g_1.$$

Although it looks as if it got even worse, we have to keep in mind that $f_1 \cdot g_2$ and $f_2 \cdot g_1$ are already known due to the sub-products in Equation **6.8**. By following this observation we can construct h from three parts

$$h_1 = f_1 \cdot g_1, \quad h_2 = f_1 \cdot g_2, \quad h_3 = (f_1 + f_2) \cdot (g_1 + g_2), \tag{6.9}$$

such that

$$h = h_1 + (h_3 - h_1 - h_2) \cdot X^{n_1} + h_2 \cdot X^{2 \cdot n_1} = f \cdot g. \tag{6.10}$$

In summary (considering Equation **6.9** and Equation **6.10**) we can say that by applying Karatsuba's idea to the convolution of polynomials, we trade one multiplication of polynomials with $N/2$ coefficients for six additions/subtractions. This leads to only $\frac{3}{4}N^2$ coefficient multiplications. Applying Karatsuba's algorithm recursively to calculate h_1, h_2, and h_3 further reduces the number of multiplications. However, we have to keep in mind that 8-bit multiplications only requires two cycles on the ATMEGA128L, so the Karatsuba approach may not be superior in all cases (due to added overhead of recursion, splitting, etc.).

Algorithm **58** equals the original version of the *high speed convolution method* proposed by Silverman. Karatsuba's original algorithm was extended with an additional reduction step so that it respects the relation $X^N = 1$. `highSpeedConv`$(\cdot)$ calls itself recursively, until the length n of the polynomials f and g reaches a threshold defined by `CutOff`. When n is small enough, the product is computed directly using the `multiplyPoly1`$(\cdot)$ method mentioned previously.

Algorithm 58: `highSpeedConv`$(\cdot)$: Silverman's convolution algorithm [59]

Input: Two polynomials $f, g \in \mathbb{R}$, integer n
Aux. Input: Integer N, threshold `CutOff`
Output: Polynomial $h = f * g$
if $n <$ `CutOff` **then**
  Compute product directly, $h \leftarrow$ `multiplyPoly1`(f, g);
else
  $n1 \leftarrow$ `floor`$(n/2)$; $n2 \leftarrow n - n1$;
  Split polynomials, write $f = f1 + f2 \cdot X^{n1}$; $g = g1 + g2 \cdot X^{n1}$;
  Set polynomial $F \leftarrow f1 + f2$;
  Set polynomial $G \leftarrow g1 + g2$;
Recursion1 Recursively compute $h1 \leftarrow$ `highSpeedConv`$(f1, g1, n1)$;
Recursion2 Recursively compute $h2 \leftarrow$ `highSpeedConv`$(f2, g2, n2)$;
Recursion3 Recursively compute $h3 \leftarrow$ `highSpeedConv`$(F, G, n2)$;
  Set polynomial $h \leftarrow h1 + (h3 - h1 - h2) \cdot X^{n1} + h2 \cdot X^{2 \cdot n1}$;
Reduce **if** $2 \cdot n - 1 > N$ **then**
  for k *from* N *to* $2 \cdot n - 1$ **do**
    $h_{k-N} \leftarrow h_{k-N} + h_k$;
Return h;

One problem of this algorithm is its memory usage. Recall that we only have 4KB RAM available, in order to achieve the best performance possible, the algorithm must be able to call itself recursively for a certain number of times. We have identified two main problems which we will briefly discuss now.

1. As the last if-statement suggests, it is temporarily possible for the polynomial h to have more than N coefficients. In case this happens, h is directly reduced (by applying $X^N = 1$) to have less than or equal to N non-zero coefficients. However, this temporary property of having more than N coefficients leads to the need for more RAM: the function calling `highSpeedConv`$(\cdot)$ has to reserve RAM for $2 \cdot N$ coefficients of h, although h finally has only N coefficients.

2. The original algorithm requires three intermediate polynomials $h1, h2$, and $h3$. The polynomials are computed by the recursive calls to `highSpeedConv`$(\cdot)$ and have about n coefficients each.

We have made some observations and experimented with implementing a more optimized version of the algorithm. Two optimizations were applied in order to improve the algorithm's overall memory usage:

1. The relation $X^N = 1$ is only applied in the last iteration of `highSpeedConv`$(\cdot)$ before the function returns to its caller. In this case N equals n. We have modified the proposed algorithm such that the calling function only needs to reserve RAM for N coefficients of h. The idea of our modification is to split the step where h is "assembled" from $h1, h2$, and $h3$ into two cases. In case $2 \cdot n - 1 \leq N$, we assemble h according to Karatsuba's formula (Equation **6.10**). In case $2 \cdot n - 1 > N$, we assemble h in a similar fashion, but apply $X^N = 1$ by adding the corresponding coefficients of $h1, h2$, and $h3$ directly instead of adding them according to Equation **6.10** and applying the reduction step afterwards.

2. By observing which coefficients of $h1, h2$, and $h3$ are added/subtracted and carefully sorting the order of these atomic operations we were able to substitute the three polynomials (and hence three memory-buffers) by one polynomial (buffer) r. We will illustrate the direct correspondence of the polynomial's coefficients below.

Due to the fact that the final algorithm is rather complicated and in order to improve readability, we chose to split the notation of `multiplyPoly2`$(\cdot)$ in three parts. The first part is presented in Algorithm **59**. This is essentially the "body" of the algorithm, it calls two helper-methods. `assemblePoly1`$(\cdot)$ denotes a method that assembles h from the sub-products of $f1, g1, f2, g2, F$, and G without reduction. `assemblePoly2`$(\cdot)$ assembles h from these sub-products as well and additionally takes care of applying $X^N = 1$. In our actual implementation, we have embedded these steps in the `multiplyPoly2`$(\cdot)$ algorithm itself, without calling additional methods.

Algorithm 59: `multiplyPoly2`$(\cdot)$: Modified Karatsuba algorithm

```
Input: Two polynomials f, g ∈ R, integer n
Aux. Input: Integer N, threshold CutOff
Output: Polynomial h = f * g
if n < CutOff then
    Compute product directly, h ← multiplyPoly1(f, g);
else
    n1 ← floor(n/2);   n2 ← n − n1;
    Split polynomials, write f = f1 + f2 · X^n1;   g = g1 + g2 · X^n1;
    Set polynomial F ← f1 + f2;
    Set polynomial G ← g1 + g2;
    if 2 · n − 1 ≤ N then
        Set polynomial h ← assemblePoly1(f1, g1, f2, g2, F, G, n1, n2);
    else
        Set polynomial h ← assemblePoly2(f1, g1, f2, g2, F, G, n1, n2);
Return h;
```

Recalling Algorithm **58** we will now explicitly give a method for computing the coefficients of $h1, h2$, and $h3$ without actually writing them to three distinct memory buffers. Finally we will assemble the coefficients such that

$$h \leftarrow h1 + (h3 - h1 - h2) \cdot X^{n1} + h2 \cdot X^{2 \cdot n1}.$$

The intermediate polynomials $h1, h2$, and $h3$ are not used anymore, we rather store the three recursively computed convolution products in h or r.

Algorithm 60: `assemblePoly1`$(\cdot)$: A method to assemble h

Input: Polynomials $f1, g1, f2, g2, F, G \in \mathbb{R}$, integers $n1, n2$
Output: Polynomial h
Step1 Partly set polynomial $[h_0, h_1, \cdots, h_{n1\cdot 2-2}]^T \leftarrow$ `multiplyPoly2`$(f1, g1, n1)$;
$h_{n1\cdot 2-1} = 0; \quad j \leftarrow 0;$
Step2 Partly set polynomial $[h_{n1\cdot 2}, h_{n1\cdot 2+1}, \cdots, h_{n1\cdot 2+n2\cdot 2-2}]^T \leftarrow$ `multiplyPoly2`$(f2, g2, n2)$;
Set polynomial $r \leftarrow$ `multiplyPoly2`$(B, C, n2)$;
for *i from* $n1$ *to* $n1 \cdot 2 - 1$ **do**
  $r_j \leftarrow h_i + (r_j - h_j - h_{n1\cdot 2+j}); \quad j \leftarrow j + 1;$
$i \leftarrow n1 \cdot 2; \quad j \leftarrow j + 1;$
for *j from j to* $n1 \cdot 2 - 2$ **do**
  $h_i \leftarrow h_i + (r_j - h_j - h_{n1\cdot 2+j}); \quad h_j \leftarrow r_{i-n1\cdot 2}; \quad i \leftarrow i + 1;$
$j \leftarrow j + 1; \quad i \leftarrow i + 1; \quad h_j \leftarrow r_{i-n1\cdot 2};$
if $n1 \neq n2$ **then**
  $h_i \leftarrow h_i + (r_j - h_{n1\cdot 2+j});$
Return h;

By partly writing the coefficients of the polynomial h (note that we use the representation as row-vector) in two steps, we set $h = (f1 * g1) + (f2 * g2) \cdot X^{2\cdot n1}$. Overwriting the coefficients of r in the first loop and writing them back to h in the second loop enabled us to replace $h1, h2$, and $h3$ by only one buffer/polynomial.

The following algorithm works in a quite similar fashion. The difference is that the coefficients of the intermediate polynomials $h1, h2$, and $h3$ are added/subtracted in a different way. The previous algorithm was trivially to derive because the mapping was explicitly given by Equation **6.10**. The next algorithm required decomposing and re-ordering of the recombination and reduction steps. In Table **6.2** we illustrate the exact mapping for Equation **6.10** in combination with a reduction step modulo $X^N - 1$. The latter behaviour was implemented in Algorithm **61**.

Algorithm 61: `assemblePoly2`$(\cdot)$: A method to assemble and reduce h

Input: Polynomials $f1, g1, f2, g2, F, G \in \mathbb{R}$, integers $n1, n2$
Aux. Input: Integer N
Output: Polynomial $h \bmod X^N - 1$
Partly set polynomial $[h_0, h_1, \cdots, h_{n1\cdot 2-2}]^T \leftarrow$ `multiplyPoly2`$(f1, g1, n1)$;
Set polynomial $r \leftarrow$ `multiplyPoly2`$(f2, g2, n2)$;
$t \leftarrow h_{n1}; \quad j \leftarrow n1 + 1;$
for *i from 0 to* $n1 - 3$ **do**
  $h_{n1+i} \leftarrow h_{n1+i} - h_i - r_i + r_j; \quad h_i \leftarrow h_i - h_j - r_j + r_{i+1};$
  $j \leftarrow j + 1;$
$i \leftarrow i + 1; \quad h_{N-3} \leftarrow r_{N-2} + h_{N-3} - h_i - r_i; \quad h_i \leftarrow h_i - r_{N-2} + r_{i+1};$
$i \leftarrow i + 1; \quad h_{N-2} \leftarrow r_{N-1} - h_i - r_i; \quad h_i \leftarrow h_i + r_{i+1};$
$i \leftarrow i + 1; \quad h_{N-1} \leftarrow r_0 - t - r_i; \quad i \leftarrow n1;$
Set polynomial $r \leftarrow$ `multiplyPoly2`$(B, C, n2)$;
for *j from* 0 *to* $n1 - 2$ **do**
  $h_i \leftarrow h_i + r_j; \quad h_j \leftarrow r_{i+1};$
  $i \leftarrow i + 1;$
$h_{N-2} \leftarrow h_{N-2} + r_{j+1}; \quad h_{N-1} \leftarrow h_{N-1} + r_{j+2};$
Return h;

Although `assemblePoly1`($\cdot$) and `assemblePoly2`($\cdot$) are more complicated than Silverman's approach, the overall runtime of both Karatsuba based convolution algorithms should be quite comparable.

+/-	h_0	h_1	$\cdots$	h_{n1-2}	h_{n1-1}	h_{n1}	h_{n1+1}	$\cdots$	h_{N-2}	h_{N-1}
+	$h1_0$	$h1_1$	$\cdots$	$h1_{n1-2}$	$h1_{n1-1}$	$h1_{n1}$	$h1_{n1+1}$	$\cdots$		
+	$h2_1$	$h2_2$	$\cdots$	$h2_{n1-1}$	$h2_{n1}$	$h2_{n1+1}$	$h2_{n1+2}$	$\cdots$	$h2_{N-1}$	$h2_0$
+	$h3_{n1+1}$	$h3_{n1+2}$	$\cdots$	$h3_{N-2}$		$h3_0$	$h3_1$	$\cdots$	$h3_{n1-1}$	$h3_{n1}$
-	$h1_{n1+1}$	$h1_{n1+2}$	$\cdots$			$h1_0$	$h1_1$	$\cdots$	$h1_{n1-1}$	$h1_{n1}$
-	$h2_{n1+1}$	$h2_{n1+2}$	$\cdots$	$h2_{N-2}$		$h2_0$	$h2_1$	$\cdots$	$h2_{n1-1}$	$h2_{n1}$

Table 6.2: Overview of sums that form h's coefficients in the last iteration of Algorithm **59**

Our modification drastically reduced the RAM-usage of Silverman's algorithm. We will now briefly estimate the achieved savings by focusing on the first, non-recursive iteration of the algorithm. The original version requires RAM for a double sized buffer (i.e., $N \cdot 2$ coefficients) for h, allocated by the caller of the convolution function. $h1, h2$, and $h3$ have approximately N coefficients, the polynomials F and G are stored with $N/2$ coefficients each. Assuming 16-bit coefficients, this totals to a RAM-usage of

$$2 \cdot (2 \cdot N + 3 \cdot N + N) = 12 \cdot N \text{ bytes}$$

for the first, non-recursive iteration. Every recursive iteration allocates memory for its own polynomials $h1, h2, h3, F$, and G, although with halved sizes. For $N = 127$ we can estimate the memory usage of Silverman's algorithm to be 1524 bytes. In contrast, the modified algorithm requires N bytes for h and r, which was introduced to replace $h1, h2$, and $h3$. The polynomials F and G have not been changed. Assuming 16-bit coefficients, this sums up to a RAM-usage of

$$2 \cdot (N + N + N) = 6 \cdot N \text{ bytes.}$$

We have reduced the RAM-usage by 50% while preserving the original speed. The drawback of our modification is that it is rather complicated and therefore difficult to implement.

6.3.2 Simultanous convolution

As observed in the `DblConv` step of Algorithm **55**, we can potentially combine the computation of $psC = p1 * p2$ and $p2 = i * p1$. We hope to further increase the speed of these two convolution steps, by calculating the resulting polynomials psC and $p2$ in parallel. We have developed and implemented two approaches: by *simple simultaneous convolution algortihm* we denote the simultaneous equivalent of the *simple convolution algorithm*. We call the second method in this section the *simultaneous (modified) Karatsuba algorithm*.

Simple simultaneous convolution algorithm

As stated before, we combine the computation of two polynomials $h1$ and $h2$ in one inner loop. Observe that we have only three input polynomials $f1, f2$, and g. We call this algorithm the `dblMultiplyPoly1`($\cdot$) method. The advantage of this algorithm is its simplicity. Further, by combining the computation of $h1$'s and $h2$'s coefficients in one inner loop, we hope to outperform the sequential execution of `multiplyPoly1`($\cdot$).

Algorithm 62: `dblMultiplyPoly1`($\cdot$): Simple simultaneous convolution

Input: Polynomials $f1, f2, g \in \mathbb{R}$
Aux. Input: Integer N
Output: Polynomials $h1 = f1 * g$, $h2 = f2 * g$
Initiate polynomial $h \leftarrow 0$;
for i *from* 0 *to* $N-1$ **do**
  for j *from* 0 *to* $N-1$ **do**
    $k \leftarrow (i+j) \bmod N$;
    $h1_k \leftarrow h1_k + f1_i \cdot g_j$;
    $h2_k \leftarrow h2_k + f2_i \cdot g_j$;
Return $(h1, h2)$;

Simultaneous Karatsuba algorithm

The *simultaneous Karatsuba algorithm* relies on the same principle as its counterpart for single convolution. The only difference here is that we to split three input polynomials denoted by $f1, f2$, and g. It is important to note that recursion is very RAM intensive, because every recursive call of `dblMultiplyPoly2`($\cdot$) creates a new function stack. The memory is only freed when the creating function returns. The size of each newly allocated memory block is approximately half as big as the size of the calling iteration's function stack.

We have to carefully evaluate whether this algorithm can be extecuted on the ATMega128L with RAM constrained to 4KB. Assuming that each polynomial has 16-bit coefficients, we can roughly estimate the RAM-usage of Algorithm **63** to be

$$2 \cdot (2 \cdot N + 2 \cdot N + 3/2N) = 11 \cdot N \text{ bytes}$$

in the first, non-recursive iteration. The previous equation summarizes the memory allocated for $h1, h2, r1$, and $r2$ (N coefficients each) with the $N/2$ coefficients of $F1, F2$, and G. For $N = 127$ we can assume a memory usage of 1397 bytes, which is nearly 35% of all RAM available. Recall that this is the memory occupied by the first iteration, i.e., when using one recurion this accumulates to roughly 2.1KB. Considering the memory constraints on the ATMega128L, it seems unlikely that high-recursion levels (meaning a low `CutOff` value) can be reached.

Algorithm 63: `dblMultiplyPoly2`($\cdot$): Simultaneous Karatsuba algorithm

Input: Three polynomials $f1, f2, g \in \mathbb{R}$, integer n
Aux. Input: Threshold `CutOff`, integer N
Output: Polynomials $h1 = f1 * g$, $h2 = f2 * g$
if $n <$ `CutOff` **then**
  Compute product directly, $(h1, h2) \leftarrow$ `dblMultiplyPoly1`$(f1, f2, g)$;
else
  $n1 \leftarrow n/2$; $n2 \leftarrow n - n1$;
  Split polynomials, write $f1 = f11 + f12 \cdot X^{n1}$; $f2 = f21 + f22 \cdot X^{n1}$;
  Split polynomial, write $g = g1 + g2 \cdot X^{n1}$;
  Set polynomials $F1 \leftarrow f11 + f12$; $F2 \leftarrow f21 + f22$;
  Set polynomial $G \leftarrow g1 + g2$;
  if $2 \cdot n - 1 \leq N$ **then**
    $(h1, h2) \leftarrow$ `assemblePolys1`$(f11, f12, f21, f22, g1, g2, F1, F2, G, n1, n2)$;
  else
    $(h1, h2) \leftarrow$ `assemblePolys2`$(f11, f12, f21, f22, g1, g2, F1, F2, G, n1, n2)$;
Return h;

Algorithm **63** is given in a shortened notation. Please find the double convolution methods to assemble the polynomials $h1$ and $h2$ in the Appendix (Section **A.2**). We have decided

to omit their demonstration here, because they work similarly to their single convolution counterparts, but for three instead of two polynomials.

It is entirely possible to extend Karatsuba's idea to the convolution of four, five, or even more polynomials in parallel. However, each new input polynomial requires additional memory (for intermediate polynomials), which is obviously very limited on the ATMega128L. We assume that three input polynomials are the maximum on this hardware, while allowing a reasonable level of recursion.

6.4 Results

In this section we will present and briefly analyze the measured execution times for each convolution algorithm. Finally, we list the overall performance of our NTRUSign implementation

6.4.1 NTRU arithmetic

We have implemented two single convolution methods. The *simple convolution algorithm* represents the naive approach and is easy to implement while occupying only a small amount of extra RAM memory. In contrast to the first approach, the second algorithm is very memory intensive. The values for `CutOff` were chosen such that we achieve one

Method	`CutOff`	$t_{\texttt{single-conv}}$
Simple convolution	-	0.563s
Modified Karatsuba method	64	0.078s
	32	0.064s
	16	0.061s
	8	0.071s

Table 6.3: Performance of single convolution algorithms

level of recursion for `CutOff` = 64, two levels of recursions for `CutOff` = 32, and four levels of recursion for `CutOff` = 8. In Table **6.3** we see that three recursion levels result in the fastest single convolution, being nine times faster than the naive algorithm. Adding another level of recursion slows the algorithm down, which is most likely due to of the overhead for building a function stack and calling the function. Although the RAM usage is only temporary (as opposed to the static tables required by ECDSA), it has to be kept in mind that high recursion levels may lead to the stack overwriting the heap, i.e., when other TinyOS applications statically allocate much RAM as well.

Simultaneous convolution was intended to replace the sequential application of single convolution algorithms. We will now briefly compare, how the simultaneous versions perform compared to applying the non-simultaneous algorithm twice. Observing the results for the *simple convolution* algorithms in Table **6.4** and Table **6.3** we can see that the simultaneous version is 40% faster. We also observe that for `CutOff` = 64 the *simultaneous*

Method	`CutOff`	$t_{\texttt{dbl-conv}}$
Simple simultaneous convolution	-	0.672s
Simultaneous Karatsuba method	64	0.131s
	32	0.107s

Table 6.4: Performance of simultaneous convolution algorithms

Karatsuba algorithm is about 17% faster. For `CutOff` = 32, the algorithm is even 13%

faster than the *single Karatsuba algorithm* with `CutOff` = 16. Although we were able to dramatically decrease the RAM usage of our Karatsuba variants by 50% (thus enabling the simultaneous algorithm on the ATMega128L), the algorithm still demands such a large amount of RAM that only two levels of recursion can be achieved. Setting `CutOff` = 16 yields error messages from AVRStudio[3].

6.4.2 Overall performance of NTRUSign

For the final benchmark of the overall performance we chose the *single Karatsuba algorithm* with `CutOff` = 16 and its simultaneous equivalent with `CutOff` = 32. This is the maximum on the ATMega128L with respect to the limited amount of memory. Interestingly, the

Operation	t
NTRUSign: Signature generation	0.619s
NTRUSign: Signature verification	0.078s

Table 6.5: Overall performance of our NTRUSign implementation

signature verification process is a lot faster than generating a valid signature. The reason for this is that the algorithm effectively signs a message twice (because we use one *perturbation basis*), while verifying requires only one single convolution.

6.5 Overview of optimizations

Here we will shortly list the applied tweaks. Note that the proper choice of parameters for NTRUSign was also part of the optimization process.

Trinary polynomials By using trinary polynomials and taking a certain probability for verification failures into account, we were able to choose a lower value for N. This helps speeding up the convolution operation, which has a complexity of $O(N^2)$.

Encoding We have encoded the trinary polynomials such that the overall RAM footprint of the private key could be reduced by about 40%. This step was mandatory in order to free RAM for our memory intensive convolution methods.

Karatsuba variants Employing two memory optimized variants of Silverman's convolution algorithm further decreases the amount of single-precision multiplications.

Transposed basis By using $t =$ "transpose" we use trinary polynomials $\{f_{*,k}, g_{*,k}\}$ for all $k \in [0, B]$ during the signature generation, thus potentially increasing the speed of coefficient multiplications.

We have observed that the resulting signatures are rather "small" (i.e., the centered norm is small). This can possibly be exploited by encoding the signature in a way similar to the "compression" of trinary polynomials. Encoding the signature could reduce the size by approximately 25%. However, we have not further pursued this idea.

[3] AVRStudio reports "excessive stack overlow[s]" and aborts the simulation.

7 Evaluation

In this chapter we analyze and compare the implemented NesC components with respect to (**1**) RAM and ROM footprint, (**2**) signature sizes and key sizes, and (**3**) performance. We assign "points" for each discipline, i.e., the winning component gets three points, the second best two points, and the worst only one. We summarize the results of this competition in the following chapter.

Note that we use the "economic" parameter setup of ECDSA. This parameter set achieves a medium performance while demanding reasonable ressources. Please note further that we do not benchmark our implementation of the key generation primitives.

7.1 RAM and ROM footprint

In the following table we list the RAM and ROM footprint of our NesC components. The presented figures are estimated by the NesC compiler when compiling our code for the MICAz hardware. Clearly, the NTRUSign component is the winner in this discipline. The reason is obvious, the arithmetic required by the NTRUSign signature generation and verification primitives is restricted to a single operation, the convolution. However, as one can see the ECDSA component demands less RAM than the XTR-DSA component, while occupying twice as much ROM memory. Compared to XTR-DSA, our ECDSA compo-

Scheme	RAM	ROM	Points
XTR-DSA	1625 bytes	24308 bytes	• •
ECDSA	1035 bytes	43290 bytes	• •
NTRUSign	571 bytes	14886 bytes	• • •

Table 7.1: Comparing the RAM and ROM requirements of our implementations

nent has one advantage: better scalability, i.e., we could use different parameters for the point multiplication algorithms, thereby easily increasing the RAM usage to 3.5KB – but we could also use algorithms without any pre-computation tables. The XTR-DSA component does not have this advantage, the only "scalability" we have is choosing whether we want to use the single exponentiation algorithm with pre-computation or not. The performance impact of this decision is quite immense, while we can scale ECDSA's performance in small steps.

The RAM utilization illustrated in Table **7.1** has one considerable drawback: the NesC compiler apparently estimates only the statically allocated RAM, i.e., memory reserved on the heap. It does not account for memory that is allocated dynamically. This does not reflect NTRUSign's enormous (but temporary) demand for stack memory, which is due to the recursive nature of its convolution algorithms. ECDSA and XTR-DSA do not require much extra memory, so the RAM estimation is quite fitting.

7.2 Keypair and signature size

Table **7.2** reveals that ECDSA is the overall winner of this discipline. The signature sizes of XTR-DSA and ECDSA are identical because a modulus of the same size is used to calculate the signature values. The size of their private key is also quite comparable, but

ECDSA is the overall winner. In order to accelerate XTR-DSA, we chose to include the surrounding powers of c_k or c_{k-1} in the public key, thereby drastically increasing the keysize. NTRUSign requires the biggest keypair, this is due to the use of *perturbation bases*, i.e., we have to store two distinct private keys. When naively storing each polynomial in N bytes, the size of the keypair would be even bigger. To free RAM memory and enable

Scheme	$\|S\|$	$\|k_{\texttt{priv}}\|$	$\|k_{\texttt{pub}}\|$	Points
XTR-DSA	40 bytes	20 bytes	176 bytes	• •
ECDSA	40 bytes	21 bytes	40 bytes	• • •
NTRUSign	127 bytes	383 bytes	127 bytes	•

Table 7.2: Comparing the sizes of the generated signatures and keypairs

the simultaneous Karatsuba algorithm, we have encoded the four trinary polynomials in strings of 64 bytes each. This step reduced the size of the private key by approximately 40%.

Recall that we aim at providing digital signature primitives for WSNs, which implies that signatures should be as small as possible. Assuming that we frequently transmit signed messages, this seems to make NTRUSign an unsuitable choice for WSN hardware. However, we found a recent publication on energy consumption of cryptographic primitves in comparison to reception and transmission of signatures [54]. The authors have measured an energy consumption of 26.46mWs induced by executing a computation step in 0.61 seconds [54, Tab. 6] on the MICAz mote. They also measured a power consumption of 172.03μWs for sending an RSA-1024 signature with the CC2420 chip [54, Tab. 11]. We can directly map these figures to our implementation, because NTRUSign generates signatures of $8 \cdot 127 \approx 1024$ bits in 0.619s. Between transmitting a signature and creating it are about two orders of magnitude. The impact of sending NTRUSign's longer signature becomes only significant when directly compared to its fast signature verification.

7.3 Performance

NTRUSign is superior to the other signature schemes when comparing signature generation and verification time. Recall that these results were achieved with a pure NesC implementation without any pre-computation. XTR-DSA and ECDSA are heavily as-

Scheme	$t_{\texttt{precomp}}$	$t_{\texttt{sign}}$	$t_{\texttt{verify}}$	Points
XTR-DSA	1.2s	0.965s	2.009s	•
ECDSA	4.5s	0.918s	1.486s	• •
NTRUSign	-	0.619s	0.078s	• • •

Table 7.3: Comparing the performance of our implementations

sembly enhanced and nearly equally fast in generating a signature, but ECDSA wins the verification benchmark. However, ECDSA's behaviour requires a longer pre-computation phase.

We have found two comparable implementations of ECDSA, both use the SECP160R1 parameters. Liu and Ning have developed the so-called "TinyECC" software package and use the *sliding window method* for single point multiplication. Additionally, they have adapted *Shamir's trick* for simultaneous multiplication. On their website, they provide performance characteristics for several mote types. Table **7.4** illustrates the results for MICAz motes. Wang and Li presented their implementation of ECDSA for MICAz hardware in 2006. They use the *sliding window method*, but give no detailed information on the required pre-computation time. As one can see, our ECDSA implementation is the

Scheme	**Authors**	$t_{\texttt{precomp}}$	$t_{\texttt{sign}}$	$t_{\texttt{verify}}$
ECDSA	Liu and Ning [45]	3.548s	1.925s	2.433s
ECDSA	Wang and Li [72]	N/A	1.35s	2.85s
RSA	Gura *et al.* [27]	N/A	10.99s	0.43s

Table 7.4: Performance of other ECDSA and RSA implementations

fastest implementation. Even better, XTR-DSA is also faster although being the slowest candidate in our internal benchmark. Recall that we have listed the "economic" setup of ECDSA, i.e., we could perform signature generation in 0.625s or signature verification in 0.938s.

We also list Gura's implementation of RSA, but comparing the times required to sign a message obviously reveals that RSA should not be considered for WSNs. The quite impressive verification time is achieved by using a special, small modulus – however, NTRUSign still has the fastest signature verification.

To the best of our knowledge, there are no comparable implementations of XTR-DSA or NTRUSign.

8 Conclusion

In this thesis, we have implemented and optimized three asymmetric signature schemes. We have improved several algorithms and achieved highly optimized code. We did this in order to evaluate, whether there are other – possibly better suited – signature schemes for WSN, besides ECDSA. During the implementation process, we have learned that it requires substantial effort to achieve a resonable performance. We also learned that the performance difference between a naive implementation and a more sophisticated approach can be of one to two orders of magnitude[1].

The amount of time required for implementing NTRUSign's key generation primitive is comparable to the time consumed by the implementation of a completely optimized ECDSA scheme. The reason for this is that ECDSA is well documented and highly optimized ASM code was already available. On the other hand, implementing NTRUSign's remaining methods was quite easy, although we faced a major problem: no testvectors were available for NTRUSign and XTR-DSA. This made debugging XTR-DSA a complex task and is one reason why implementing XTR-DSA took most of the time. The second reason is that we implemented nearly all of the proposed enhancements. We also chose to improve the prime field arithmetic with special ASM routines.

Finally, we have summarized the respective performance benchmarks in one chapter, where we perform a competetive analysis of the implementation. The evaluation yields no clear winner, NTRUSign ($3 + 1 + 3 = 7$ points) and ECDSA ($2 + 3 + 2 = 7$ points) are both tied for the first place. However, we see that NTRUSign is the winner in two categories, while ECDSA wins only one. This makes NTRUSign the first choice, when (static) RAM and ROM requirements and overall performance are critical. It has to be kept in mind that ECDSA is more "scalable", i.e., we can improve the performance by choosing different parameters, while increasing the static memory usage and pre-computation time. But even with extreme parameters, we were unable to achieve NTRUSign's performance. NTRUSign, on the other hand, is also "scalable" in some way, we can decrease the dynamic memory usage by allowing fewer recursions – this does not even impact the overall performance very much.

[1] Our very first, proof-of-concept implementation of XTR-DSA required 115s for generating a signature!

9 Bibliography

[1] NesC website. http://nescc.sourceforge.net/.

[2] The GNU MP bignum library. http://gmplib.org.

[3] The Python programming Language. http://www.python.org.

[4] Efficient Embedded Security Standard (EESS) #1: Draft 2.0, 2003. http://www.ceesstandards.org.

[5] A. A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Soviet Physics-Doklad*, 7:595–596, 1963.

[6] ZigBee Alliance. ZigBee Specification. Technical Report 1.0, 2005.

[7] Atmel Corporation. *4-Megabit 2.7-volt Only Serial DataFlash*.

[8] Atmel Corporation. *8-bit Microcontroller with 128K Bytes In-System Programmable Flash*. http://www.nodna.com/fileadmin/download/MEGAROBOTICS/ATMega128.pdf.

[9] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mobile Networks and Applications*, 10:563–579(17), January 2005.

[10] E.-O. Blass, H. Junker, and M. Zitterbart. Effiziente Implementierung von Public-Key-Algorithmen für Sensornetze, 2005. http://doc.tm.uka.de/2005/blass-sensornetz-key-algo-2005.pdf.

[11] G. Chudnovsky and D. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primilarity factoring tests. *Advances in Applied Mathematics*, 7:385–434, 1987.

[12] H. Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. 1993.

[13] P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal 29*, 4:526–536, 1990.

[14] Atmel Corporation. AVR Studio, 4.12.490, Service Pack 3. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725.

[15] Crossbow Technology Inc. *Imote2 – High performance wireless sensor network node*.

[16] Crossbow Technology Inc. *MICA2 – WIRELESS SENSOR PLATFORM*, 2007. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

[17] Crossbow Technology Inc. *MICAz – WIRELESS MEASUREMENT SYSTEM*, 2007. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[18] Crossbow Technology Inc. *TelosB – TELOSB NODE PLATFORM*, 2007. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[19] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998.

[20] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976.

[21] ETH Zurich. *BTnut System Overview*, 2007. http://www.btnode.ethz.ch/static_docs/doxygen/btnut/.

[22] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[23] C. Gentry, J. Jonsson, J. Stern, and M. Szydlo. Cryptanalysis of the NTRU Signature Scheme (NSS) from Eurocrypt '01. In *Advances in Cryptology – Asiacrypt 2001, Proceedings*, volume 2248, pages 123–131. Springer, 2001.

[24] C. Gentry and M. Szydlo. Cryptanalysis of the Revised NTRU Signature Scheme. In *Advances in Cryptology – EUROCRYPT '02*, volume 2332, pages 299–320. Springer, 2002.

[25] O. Goldreich. *Foundations of Cryptography*, volume 1. 1998. Available at http://www.wisdom.weizmann.ac.il/~oded/frag.html.

[26] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004. Available at http://www.wisdom.weizmann.ac.il/~oded/frag.html.

[27] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Elliptic curve cryptography and RSA on 8-bit CPUs. In *Proceedings of Workshop on Crpytographic Hardware and Embedded Systems (CHES 2004), 6th International Workshop*, pages 119 – 132, 2004.

[28] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[29] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*, 2000.

[30] J. L. Hill and D. Culler. MICA: a Wireless Platform for Deeply Embedded Networks. *Micro, IEEE*, 22(6):12–24, 2002.

[31] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte. NTRUSign: Digital signatures using the NTRU lattice, 2001. Preliminary draft presented at *Asia Crypt*.

[32] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte. NTRUSign: Digital signatures using the NTRU lattice. In *Proceedings of the RSA conference*, 2003.

[33] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. *Lecture Notes in Computer Science*, 1423:267–??, 1998.

[34] J. Hoffstein, J. Pipher, and J. H. Silverman. NSS: An NTRU Lattice-Based Signature Scheme. In *Advances in Cryptology - EUROCRYPT 2001, Proceedings*, volume 2045, pages 211–228, 2001.

[35] J. Hoffstein and N. Howgrave-Graham and J. Pipher and J. H. Silverman and W. Whyte. Performance Improvements and a Baseline Parameter Generation Algorithm for NTRUSign, 2005.

[36] J. Pipher. Lectures on the NTRU encryption algorithm and digital signature scheme, 2002. Available at `http://www.math.brown.edu/~jpipher/grenoble.pdf`.

[37] M. Joye and C. Tymen. Compact Encoding of Non-adjacent Forms with Applications to Elliptic Curve Cryptography. *Lecture Notes in Computer Science*, 1992:353–??, 2001.

[38] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[39] C. K. Kocç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[40] H. Krawczyk, M. Bellare, , and R. Canetti. HMAC: Keyed-hashing for message authentication. *RFC 2104*, 1997.

[41] A. K. Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields. In *Proceedings of Australian Conference on Information Security and Privacy*, pages 127–138. Springer Verlag, 1997.

[42] A. K. Lenstra and E. R. Verheul. An overview of the XTR public key system. In *The Proceedings of the Public Key Cryptography and Computational Number Theory Conference*, 2000.

[43] A. K. Lenstra and E. R. Verheul. The XTR Public Key System. *Lecture Notes in Computer Science*, 1880:1+, 2000.

[44] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[45] A. Liu and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks, 2006. `http://discovery.csc.ncsu.edu/software/TinyECC/`.

[46] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.

[47] V. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology Proceedings of CRYPTO'85*, pages 417–426. Springer Verlag, 1986.

[48] P. L. Montgomery. Multiplication without trial division. *Math. Computation*, 44(170):519–521, 1985.

[49] P. L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1992. Available at `ftp://ftp.cwi.nl/pub/pmontgom/Lucas.ps.gz`.

[50] M. Neve, E. Peeters, G. M. de Dormale, and J.-J. Quisquater. Faster and smaller hardware implementation of XTR. In *Proceedings of SPIE, Symposium on Optics & photonics, Advanced Signal Processing Algorithms, Architectures, and Implementations*, 2006.

[51] NIST. Secure Hash Standard. *FIPS 180-1*, 1995.

[52] NIST. Data Encryption Standard. *FIPS 46-3*, 1999.

[53] E. Peeters, M. Neve, and M. Ciet. XTR Implementation on Reconfigurable Hardware. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

[54] K. Piotrowsky, P. Langendoerfer, and S. Peter. How public key cryptography influences wireless sensor node lifetime. In *SASN 06: Proceedings of the fourth ACM workshop on Security of Ad Hoc and Sensor Networks*, pages 169–176. ACM Press, 2006.

[55] GNU Project. GNU Compiler Collection. http://gcc.gnu.org.

[56] R. L. Rivest. The MD5 message digest algorithm. *RFC 1321*, 1992.

[57] R. L. Rivest, A. Shamir, and L. M. Adleman. *A method for obtaining digital signatures and public-key cryptosystems*, pages 120 – 126. 1978.

[58] J. H. Silverman. Almost Inverses and Fast NTRU Key Generation. Technical Report 14, NTRU, 1999.

[59] J. H. Silverman. High Speed Multiplication of (Truncated) Polynomials. Technical Report 10, NTRU, 1999.

[60] M. Stam and A. K. Lenstra. Speeding up XTR. In *Advances in Cryptology – Asiacrypt 2001*, pages 125–143. Springer Verlag, 2001.

[61] Standards for Efficient Cryptography Group. SEC 2 - Recommended Elliptic Curve Domain Parameters. Ver. 1.0, 2000. http://www.secg.org.

[62] T. Rowland and E. Weisstein. *Abelian Group*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/AbelianGroup.html, accessed 25.9.2007.

[63] T. Rowland and E. Weisstein. *Finite Field*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/FiniteField.html, accessed 25.9.2007.

[64] T. Rowland and E. Weisstein. *Group*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/Group.html, accessed 25.9.2007.

[65] T. Rowland and E. Weisstein. *Prime Field*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/PrimeField.html, accessed 25.9.2007.

[66] T. Rowland and E. Weisstein. *Ring*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/Ring.html, accessed 25.9.2007.

[67] T. Rowland and E. Weisstein. *Subgroup*. From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/Subgroup.html, accessed 25.9.2007.

[68] Texas Instruments. *2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*.

[69] L. Uhsadel. Comparison of low-power public key cryptography on MICAz 8-bit micro controllers. Master's thesis, Ruhr-University Bochum, 2007.

[70] Leif Uhsadel, Axel Poschmann, and Christof Paar. Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes. In *Proceedings of the 4th European Workshop on Security and Privacy in Ad hoc and Sensor Networks — ESAS 2007*, July 2007.

[71] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz. Energy analysis of public-key cryptography for wireless sensor net-works. In *Third IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2005.

[72] H. Wang and Q. Li. Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper). In *International Conference of Information and Communication Security (ICICS)*, pages 519–528, Raleigh, NC, December 2006.

[73] WinAVR. Suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. http://winavr.sourceforge.net.

[74] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

A Appendix

A.1 Parameters

In this section we will shortly list the parameters that were used to implement the three public key systems. The parameters were chosen so that the security level provided by each signature scheme is comparable. We aimed at implementing a security level equivalent to the security of an 80-bit symmetric encryption system. Parameters of a certain size (>16 bit) are given in hexadecimal, most-significant bit first (MSB) notation.

A.1.1 XTR-DSA parameters

We chose to use a prime q of 160 bit size and p to be of 170 bit size. According to the authors of XTR, this provides a security level at least equivalent to RSA with a 1024-bit modulus and equally comparable to ECC curves with 170 bit operands [43, 42].

Parameter	Value
q	93c5e4fb 2f3c8757 11c761bb a144a62s 35ab3a19
p	0275 c45eda7a 4ef4c810 c4b64520 0ba9fb81 14c924b3

Table A.1: Parameters of our XTR-DSA implementation

A.1.2 ECDSA parameters

As stated previously, the implementation only supports the SECP160R1 curve. This curve is supposed to provide security equivalent to the security of symmetric systems with an 80 bit key [61]. The following table lists all parameters that are associated with this specific curve as defined by the SEC group.

Parameter	Value
q	ffffffff ffffffff ffffffff ffffffff 7fffffff
a	ffffffff ffffffff ffffffff ffffffff 7ffffffc
b	1c97befc 54bd7a8b 65acf89f 81d4d4ad c565fa45
G.x	4a96b568 8ef57328 46646989 68c38bb9 13cbfc82
G.y	23a62855 3168947d 59dcc912 04235137 7ac5fb32
n	1 00000000 00000000 0001f4c8 f927aed3 ca752257

Table A.2: Parameters of our ECDSA implementation

A.1.3 NTRUSign parameters

Several variants of NTRUSIGN parameters are available. One of the more recent papers on a parameter generation algorithm served as reference for chosing a parameter-set with a security equivalent to the other schemes [35]. It is crucial to chose a parameter-set with a low value for N, because the computational complexity of the convolution is $O(N^2)$. This parameter-set is given for trinary polynomials f and g. The authors mention that this parameter-set has a higher probability for verification-failures on valid signatures – practical experiments found that this propability is negligible.

Parameter	Value
N	127
q	256
`NormBound`^{2}	$122^2 = 14884$
B	1
d	31
t	"transpose"

Table A.3: Parameters of our NTRUSign implementation

A.2 Double convolution with the modified Karatsuba algorithm

We have chosen to relocate the lengthy algorithms `assemblePolys1`$(\cdot)$ and `assemblePolys2`$(\cdot)$ from the NTRUSign chapter to the appendix. However, we will not further discuss the algorithms here. Please find the algorithms in the following.

Algorithm 64: `assemblePolys1`$(\cdot)$: A method to assemble polyomials $h1, h2$

Input: Polynomials $f11, f12, f21, f22, g1, g2, F1, F2, G,$ sizes $n1, n2$
Output: Polynomials $h1, h2$
Partly set $([h1_0, h1_1, \cdots, h1_{n1\cdot 2-2}]^T, [h2_0, h2_1, \cdots, h2_{n1\cdot 2-2}]^T)$
... $\leftarrow$ `dblMultiplyPoly2`$(f11, f21, g1, n1)$;
$h1_{n1\cdot 2-1} = 0$; $h2_{n1\cdot 2-1} = 0$;
Partly set $([h1_{n1\cdot 2}, h1_{n1\cdot 2+1}, \cdots, h1_{n1\cdot 2+n2\cdot 2-2}]^T, [h2_{n1\cdot 2}, h2_{n1\cdot 2+1}, \cdots, h2_{n1\cdot 2+n2\cdot 2-2}]^T)$
... $\leftarrow$ `dblMultiplyPoly2`$(f12, f22, g2, n1)$;
Set polynomial $(r1, r2) \leftarrow$ `dblMultiplyPoly2`$(F1, F2, G, n2)$;
for i *from* $n1$ *to* $n1 \cdot 2 - 1$ **do**
  $r1_j \leftarrow h1_i + (r1_j - h1_j - h1_{n1\cdot 2+j})$; $r2_j \leftarrow h2_i + (r2_j - h2_j - h2_{n1\cdot 2+j})$; $j \leftarrow j + 1$;
$i \leftarrow n1 \cdot 2$;
for j *from* $j + 1$ *to* $n1 \cdot 2 - 2$ **do**
  $h1_i \leftarrow h1_i + (r1_j - h1_j - h1_{n1\cdot 2+j})$; $h1_j \leftarrow r1_{i-n1\cdot 2}$;
  $h2_i \leftarrow h2_i + (r2_j - h2_j - h2_{n1\cdot 2+j})$; $h2_j \leftarrow r2_{i-n1\cdot 2}$; $i \leftarrow i + 1$;
$j \leftarrow j + 1$; $i \leftarrow i + 1$;
$h1_j \leftarrow r1_{i-n1\cdot 2}$; $h2_j \leftarrow r2_{i-n1\cdot 2}$;
if $n1 \neq n2$ **then**
  $h1_i \leftarrow h1_i + (r1_j - h1_{n1\cdot 2+j})$;
  $h2_i \leftarrow h2_i + (r2_j - h2_{n1\cdot 2+j})$;
Return $h1, h2$;

Algorithm 65: `assemblePolys2`$(\cdot)$: A method to assemble and reduce $h1, h2$

Input: Polynimials $f11, f12, f21, f22, g1, g2, F1, F2, G,$ sizes $n1, n2$
Aux. Input: Integer N
Output: Polynomials $h1, h2$
Partly set $([h1_0, h1_1, \cdots, h1_{n1\cdot 2-2}]^T, [h1_0, h1_1, \cdots, h1_{n1\cdot 2-2}]^T)$
... $\leftarrow$ `dblMultiplyPoly2`$(f11, f21, g1, n1)$;
Set polynomials $(r1, r2) \leftarrow$ `dblMultiplyPoly2`$(f12, f22, g2, n2)$;
$t1 \leftarrow h1_{n1}; \quad t2 \leftarrow h2_{n1}; \quad j \leftarrow n1+1;$
for *i from 0 to* $n1-2$ **do**
  $h1_{n1+i} \leftarrow h1_{n1+i} - h1_i - r1_i + r1_j; \quad h2_{n1+i} \leftarrow h2_{n1+i} - h2_i - r2_i + r2_j;$
  $h1_i \leftarrow h1_i - h1_j - r1_j + r1_{i+1}; \quad h2_i \leftarrow h2_i - h2_j - r2_j + r2_{i+1};$
  $j \leftarrow j+1;$
$i \leftarrow i+1; \quad h1_{N-3} \leftarrow r1_{N-2} + h1_{N-3} - h1_i - r1_i;$
$h2_{N-3} \leftarrow r2_{N-2} + h2_{N-3} - h2_i - r2_i; \quad h1_i \leftarrow h1_i - r1_{N-2} + r1_{i+1};$
$h2_i \leftarrow h2_i - r2_{N-2} + r2_{i+1};$
$i \leftarrow i+1; \quad h1_{N-2} \leftarrow r1_{N-1} - h1_i - r1_i; \quad h2_{N-2} \leftarrow r2_{N-1} - h2_i - r2_i;$
$h1_i \leftarrow h1_i + r1_{i+1}; \quad h2_i \leftarrow h2_i + r2_{i+1};$
$i \leftarrow i+1; \quad h1_{N-1} \leftarrow r1_0 - t1 - r1_i; \quad h2_{N-1} \leftarrow r2_0 - t2 - r2_i; \quad i \leftarrow n1;$
Set polynomials $(r1, r2) \leftarrow$ `dblMultiplyPoly2`$(F1, F2, G, n2)$;
for *j from* 0 *to* $n1-2$ **do**
  $h1_i \leftarrow h1_i + r1_j; \quad h1_j \leftarrow r1_{i+1};$
  $h2_i \leftarrow h2_i + r2_j; \quad h2_j \leftarrow r2_{i+1};$
  $i \leftarrow i+1;$
$h1_{N-2} \leftarrow h1_{N-2} + r1_{j+1}; \quad h1_{N-2} \leftarrow h1_{N-2} + r1_{j+1};$
$h2_{N-1} \leftarrow h2_{N-1} + r2_{j+2}; \quad h2_{N-1} \leftarrow h2_{N-1} + r2_{j+2};$
Return $h1, h2$;