Breaking ECC2K-112: 0x276c233740d817000b80478fde46

Paul Wolfger
paul.wolfger@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Dipl.-Ing. Dr.techn. Erich Wenger
Assessor: Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

May, 2014

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Paul Wolfger

Acknowledgements

At first, I would particularly like to thank my supervisor Erich Wenger, who has mentored me since my Bachelor Thesis. Special thanks also to Kastl Wolfgang and Jürgen Fuss from the FH Oberösterreich, who provided us with 16 FPGA boards. Without these FPGA boards, we would most certainly not have been able to really establish a new record. Further I owe a very important debt to my fellow students Philipp Dunst, Oliver Söll, and Mario Werner, who supported me in all matters. I am also deeply grateful to my family, my siblings and parents especially, who supported me not only financially. Last but not least, I am particularly grateful for my girlfriend Anna's assistance during this very busy time. She and her two dogs sometimes were my only reason to see daylight, especially in the stressful final phase.

Abstract

Historically cryptography was primarily used by military and intelligence organizations. Nowadays, however, as security related topics are increasingly familiar front page news, the public's demand for security grows substantially. In particular, the rapid evolution of mobile and wireless technologies and their omnipresent usage raises a serious issue. How is it possible to provide security objectives as confidentiality, integrity, and authenticity despite the significantly less computational power and bandwidth available on these devices. Especially cryptographic objectives provided by common asymmetric cryptographic primitives fail to meet these requirements. In constrained environments, elliptic curve cryptography is most suitable, as for its underlying hard problem, the elliptic curve discrete logarithm problem, no sub-exponential time algorithm has been found. Thus it provides a higher cryptographic security per bit, which leads to smaller key sizes, ciphertexts, and signatures compared to RSA and El-Gamal systems. This is one reason, why since its introduction in 1985, by Neal Koblitz and Victor Miller, elliptic curve cryptography has gained broad acceptance in the industry and the academic community. Nowadays, it is considered as a new standard for public key cryptography. Estimations concerning the size of the parameters for elliptic curve cryptography to provide proper security to date vary. The prevalence of elliptic curve cryptosystems increases the demand of practically verifying if the security estimations hold. In order to further encourage investigations in this matter, we attack a 113-bit binary ECDLP on a Koblitz curve using a parallelized version of Pollard's rho algorithm implemented on FPGAs. The arithmetic of the underlying 113-bit binary field has been improved extensively. Basis transformations of binary field elements are applied to exploit the benefits of polynomial and normal basis representations. We report having solved ECC2K-112, which is a new binary ECDLP record, within 46 days using only 19 FGPA boards. The design is based on a fully autonomous, cyclic, self-sufficient Pollard's rho iteration function performing up to 291 million iterations per second. The piplined design maximizes the hardware utilization by keeping all 223 pipline stages permanently active. Together with a small generic serial interface the design consumes 57% of all available slices on a Virtex-6 XC6VLX240T FPGA. The results gathered provide influential estimations concerning the security of various key lengths for elliptic curve cryptography.

Keywords: Elliptic Curve Cryptography, Elliptic Curve Discrete Logarithm Problem, Field-Programmable Gate Array

Kurzfassung

Historisch gesehen wurde Kryptographie hauptsächlich von Militär- und Geheimdienstorganisationen verwendet. Heutzutage jedoch werden sicherheitsrelevante Themen immer öfter medial aufbereitet, wodurch das öffentliche Interesse für Sicherheit beträchtlich wächst. Insbesondere die rasante Entwicklung von mobilen Geräten und von drahtlosen Technologien und deren omnipräsente Verwendung, wirft ein ernstzunehmendes Problem auf. Wie ist es möglich die Sicherheitsziele Vertraulichkeit, Integrität und Authentizität trotz deutlich weniger Rechenleistung und Bandbreite auf diesen Geräten zu Verfügung zu stellen. Vor allem die kryptographischen Ziele, die mit Hilfe von asymmetrische Kryptographie erreicht werden, erfüllen diese Anforderungen nicht. Elliptische-Kurven-Kryptographie ist vor allem in solchen eingeschränkten Umgebungen sehr gut geeignet, da es für das zugrunde liegende Problem des diskreten Logarithmus in elliptischen Kurven keine subexponentiellen Algorithmen gibt. Deswegen bietet Elliptische-Kurven-Kryptographie eine höhere Sicherheit per Bit, was im Vergleich zu RSA und El-Gamal Systemen zu kleineren Schlüsselgrößen, Chiffretexten und Signaturen führt. Das ist ein Grund, warum seit der Einführung im Jahre 1985 von Neal Kobitz und Victor Miller, die Elliptische-Kurven-Kryptographie auf eine breite Akzeptanz in der Industrie und in der akademischen Gemeinschaft stößt. Heute wird Elliptische-Kurven-Kryptographie als ein neuer Standard für öffentliche Schlüsselsysteme betrachtet. Schätzungen über die Größe der Parameter für Elliptische-Kurven-Kryptographie, um zum jetzigen Zeitpunkt zweckmäßige Sicherheit zu gewährleisten, variieren. Die weite Verbreitung von Elliptische-Kurven-Kryptographie erhöht die Nachfrage praktisch zu überprüfen, ob die Sicherheitsabschätzungen stimmen. Um weitere Untersuchungen in dieser Sache zu fördern, versuchen wir den diskreten Logarithmus einer 113-bit Koblitz Binärkörperkurve mit einer parallelisierten Variante von Pollards rho Algorithmus, realisiert auf Feld programmierbaren Gatter-Anordnungen, zu lösen. Die Arithmetik des zugrunde liegenden 113-bit Binärkörpers wurde umfassend verbessert. Basistransformationen von Binärkörperelementen werden verwendet, um die Vorteile von Polynomial- und Normalbasisdarstellung auszunutzen. Wir verkünden ECC2K-112 mit 19 FPGA Karten in 46 Tagen gelöst zu haben, was ein neuer Rekord für das Lösen von diskreten Logarithmen in elliptischen Kurven auf Binärkörpern ist. Das Design basiert auf einer völlig autonomen, zyklischen, selbstversorgenden Pollard-rho-Iterationsfunktion, mit bis zu 291 Millionen Iterationen pro Sekunde. Das Design maximiert die Auslastung der Hardware indem alle 223 Piplinestufen ständig aktiv sind. Zusammen mit einer kleinen generischen seriellen Schnittstelle, braucht das Design 57% der Logikressourcen einer Virtex-6 XC6VLX240T FPGA. Die Ergebnisse liefern wichtige Abschätzungen im Bezug auf die Sicherheit von verschiedenen Schlüssellängen für Elliptische-Kurven-Kryptographie.

Stichwörter: Elliptische-Kurven-Kryptography, Problem des diskreten Logarithmus in elliptischen Kurven, Feld programmierbare Gatter-Anordnung

Contents

1 Introduction **1**

2 Cryptography **4**
2.1 What is Cryptography? 6
2.1.1 Goals and Terminology 7
2.2 Symmetric vs. Asymmetric Cryptography 7

3 Mathematical Background **10**
3.1 Groups 10
3.2 Rings 11
3.3 Fields 12
3.3.1 Prime fields 12
3.3.2 Binary Fields 12
3.3.3 Binary Extension Fields 13
3.3.4 Polynomial Basis 13
3.3.5 Normal Basis 14
3.3.6 Basis Transformation 15
3.4 Multiplicative Inverse 16
3.5 Modular Arithemic 17

4 Elliptic Curve Cryptography **19**
4.1 Arithmetic on Elliptic Curves 20
4.1.1 Group Law 22
4.1.2 Group Order and Group Structure 23
4.1.3 Scalar Multiplication 24
4.1.4 Koblitz Curves 24
4.1.5 Negation Map 25
4.2 The Elliptic Curve Discrete Logarithm Problem 25
4.2.1 Baby-Step Giant-Step 26
4.2.2 Pohlig-Hellman 26
4.2.3 Isomorphism Attacks 26
4.2.4 Pollard's Rho Attack 27

5 Solving the ECDLP **31**
5.1 Previous Work 31
5.2 Goals & Strategies 33
5.3 Selecting the Iteration Function 34
5.4 Overall Setup 36

5.5 FPGA Architecture . . . 36
5.5.1 Top Level Architecture . . . 37
5.5.2 Point Addition Module . . . 40
5.5.3 Point Automorphism Module . . . 41
5.5.4 $\mathbb{F}_{2^m}$ Inversion Module . . . 45
5.5.5 $\mathbb{F}_{2^m}$ Multiplication Module . . . 46
5.5.6 $\mathbb{F}_{2^m}$ Squaring (Reduction) Module . . . 48
5.5.7 $\mathbb{F}_p$ Addition Module . . . 48
5.5.8 $\mathbb{F}_p$ Multiplication Module . . . 49
5.6 Results . . . 49
5.6.1 ECC-Breaker . . . 49
5.6.2 ECC-Breaker on different FPGAs . . . 52
5.6.3 Expected Runtime for larger curves . . . 53
5.6.4 Solution to the posed ECDLP . . . 54

6 Conclusions **56**

A Parameter **58**

B Abbreviations **60**

Bibliography **61**

Chapter 1

Introduction

Not least because of the current NSA spying scandal, the public's interest in cryptography constantly grows. Thus, nowadays, cryptography gets even more ubiquitous and indispensable than ever. Especially a tremendous number of mobile devices and even tiny hardware devices as Radio Frequency Identification (RFID) tags or smart cards demand security. However, these devices are very limited in computational power and storage capacity. Elliptic Curve Cryptography (ECC) is an asymmetric cipher that meets these requirements. Since its introduction in 1985, by Neal Koblitz [43] and Victor Miller [55], the interest in ECC has constantly grown. ECC is a strong competitor to the well established RSA [68] and El-Gamal systems [24]. It is an asymmetric cipher, which provides and offers useful functionalities as digital signatures or key exchange mechanisms. Opposed to RSA and El-Gamal systems, ECC requires significantly shorter key sizes, ciphertexts, and signatures to offer the same level of security. In general, to date ECC offers the most security per bit of all current asymmetric cryptography schemes. This is because, in contrast to RSA and El-Gamal systems, no sub-exponential time algorithms for ECC's underlying hard problem, the Elliptic Curve Discrete Logarithm Problem (ECDLP), exist.

In daily life the arising question is, how big should the parameters of an elliptic curve cryptosystem be in order to avoid practical attacks. Parameters too large waste computational power, time, and space, parameters too small pose a security thread. There exist several estimations concerning the size of the parameters for ECC. In order to offer security to date the estimations vary. Lenstra and Verheul [45] specify a minimal key size of 154 bits, the European Network of Excellence in Cryptology II (ECRYPT II) [7] 160 bits, and the National Institute of Standards and Technology (NIST) [10] 224 bits. To encourage further investigations in this matter Certicom Corp [15] published a list of ECDLP challenges to be solved in order to practically evaluate the estimations. To date, the 109-bit binary and prime field challenges [16, 17] are the biggest challenges that have been solved. Apart from the Certicom challenges, Harley *et al.* [34] solved an ECDLP on a 109-bit Koblitz curve, and Bos *et al.* [13] even solved an ECDLP on a 112-bit prime field curve. There even exist efforts to solve the 131-bit challenge [8], however to date no results have been published.

There are many approaches trying to solve the ECDLP, some of them are software based solutions running on clusters of Graphics Processing Units (GPUs) [8] or Central Processing Units (CPUs) [8], some are Field Programmable Gate Array (FPGA) based designs [8, 13, 25, 32, 39, 49, 53]. However, since the 112-bit challenge was solved in 2012 [13]

no further results have been reported. These previous attacks were all carried out using Pollard's rho algorithm, which is the fastest known algorithm for solving ECDLPs. Depending on the elliptic curve's group order n, the solution is expected to be obtained after $\sqrt{\pi n/2}$ iterations of Pollard's rho iteration function. For Koblitz curves, a special subset of binary elliptic curves, the expected number of iterations can be reduced by a factor of m, m being the size of the underlying binary field. Koblitz curves are of great practical importance because for these curves a faster algorithm for elliptic curve point multiplications exist, which utilizes the Frobenius automorphism to avoid computational expensive point doublings. In order to further asses the practical security estimations of elliptic curve cryptosystems, we attack a 113-bit ECDLP on an Koblitz curve using a parallelized version of Pollard's rho algorithm. We are expected to obtain the solution after approximately $8.5 \cdot 10^{15}$ iterations. A state of the art CPU (Core-i7 i7-3537U [36]) needs 0.78 ms for a point addition, the most expensive part of a Pollard's rho iteration function when written in C. That means, the solution to a 113-bit binary ECDLP would be found after about 210,000 years. All the more we want to stress, that to obtain the solution to a 113-bit binary ECDLP , it of utmost importance to design hardware and/or software of high performance. Keep in mind, that some years ago, in the year 2000, a 113-bit binary curve, namely *sect113r1* [66], was standardized by Certicom.

Our Contribution. In this thesis, we present a high throughput FPGA design, *ECC-Breaker*, used for solving the ECDLP on binary curves. The attack on the ECDLP is conducted using a parallelized version of Pollard's rho algorithm. The solution to a pseudo-randomly generated 113-bit ECDLP was found within 46 days with 18 instances of ECC-Breaker implemented on Xilinx' ML605 [89] development boards. They come with a Virtex-6 XC6VLX240T FPGA. On this FPGA, ECC-Breaker achieves a maximal synthesizeable frequency of 291 MHz and uses 57% of its slices. ECC-Breaker is a cyclic, fully independent, completely piplined, and autonomous implementation of a Pollard's rho iteration function. Its 223 pipline stages are fed by a serial interface. Although carefully optimized for the 113-bit binary Koblitz curve, ECC-Breaker can also be used for breaking larger curves with some minor changes. The results gathered during this work are relevant for security estimations of ECC, as it shows how much effort in terms of time and money has to be invested to break an elliptic curve cryptosystem. Having broken a 113-bit binary elliptic curve, we estimate that even larger elliptic curves such as the 131 bit binary challenge are computationally feasible. Our results should be considered when selecting parameter size for elliptic curves, especially when limited computational power could mislead to questionable decisions. Apart from the results regarding the ECDLP, the ECC-Breaker provides important insights for high speed ECC hardware. The design is capable of performing up to 291 million elliptic curve operations per second.

Outline. This thesis is structured as follows. Chapter 2 gives an introduction to cryptography. It summarizes its history and its main principles. It emphasizes the differences between asymmetric and symmetric cryptography. We revisit some mathematical foundations needed for understanding ECC in Chapter 3, especially the underlying algebraic structures, ECC relies on. In Chapter 4, we describe how ECC works. We distinguish between prime field and binary field curves, we highlight Koblitz curves, and show known generic attacks applicable for solving ECDLPs. We focus on Pollard's rho attack, which is known to be the most powerful attack. The contribution of our work, the **ECC-Breaker** hardware design is described in Chapter 5. We show what previously has been achieved

regarding solving ECDLPs, present the strategies and goals we realized with our hardware design, and describe ECC-Breaker's architecture in detail. Moreover, we picture how we conducted our attempt to establish a new binary ECDLP record. It is concluded with detailed results of ECC-Breaker and its impacts regarding further ECDLP challenges. We summarize the thesis and risk taking a look into the future in Chapter 6.

Chapter 2

Cryptography

Cryptography comes from the greek words *kryptos* and *graphein*, which together mean *secret writing* (cf. [47]). The wish to *encrypt* something is almost as old as human history. Technical studies about cryptography are one of the oldest we have records of (cf. [20]). The history of cryptography starts in ancient Egypt in the 3rd millennium BC (cf. [21]). Back then it was forbidden to say and write the names of various Gods in public. In addition, the priests believed that it was their task to keep the God's secret lore. This was achieved by the use of hieroglyphs, different from those known to the public (cf. [82]). Such ciphers are called substitution ciphers.

Also the ancient Greeks are said to have known of ciphers. The Spartans used the *scytale* transposition cipher to hide information [20]. Transposition is a permutation of the plaintext. Although it was already mentioned in the 7th century BC by Archilochus, Plutarch [62] (50 - 120 AD) was the first to describe how the scytale is operated (cf. [42]). A scytale is a stick of wood, around which a paper is wound. To encrypt a message one starts to write, but instead of writing one letter next to each other, one has to write one letter beneath each other. Plutarch was convinced only someone with the same scytale or a wooden stick alike in length and thickness could discover the continuity of the message.

The *Caesar cipher* named after Julius Caesar, who used it in private correspondence, is a widely known substitution cipher. This means each letter is replaced with another. Caesar substituted, according to Suetonius [80], D for A and so on. Thus he performed a shift of three.

Another famous example in the rich history of Cryptography is the *ADFGX* [40] and *ADFGVX cipher* (an improvement to the ADFGX cipher) used by the Germans on the Western Front during World War I. It is based on substitution and transposition and was invented by Fritz Nebel. The first step is the substitution step, which is performed with the aid of a *Polybius square*. The Polybius square was originally used for the Greek alphabet. Each letter is represented by its coordinates within a square grid. The Germans replaced the coordinates with the letters A, D, F, G, and X. As such grids can only contain a square number of letters, one has to round down the number of letters to the next lowest square number by combining letters. The German, for instance, combined I and J. Additionally they started to fill the square with a code word. The remain of the square is filled with the remaining letters. Table 2.1 contains an example. Using this table "polybius" is transformed into "AGDAGAAFDXFGXAGX".

Table 2.1: Polybius square used by the ADFGX cipher with the codeword “cryptography”.

	A	D	F	G	X
A	C	R	Y	P	T
D	O	G	A	H	B
F	D	E	F	I	K
G	L	M	N	Q	S
X	U	V	W	X	Z

A second code word is used for the transposition step. This second code word forms a matrix with the substituted text. The code word written on the top of a matrix. Each letter of this code word is assigned to a number(e.g., in alphabetical order) and then the substituted text is writtten in this matrix line by line. The final text is produced line by line by rearranging each letter according to the numbers assigned to the code word. During the German “Frühjahresoffensive” the French Georges Painvin succeeded in breaking ADFGX, which enabled the French to locate the German army. This advantage is believed to be the reason why the Germans never occupied Paris and marks the watershed of this war [85]. The stated examples are counted to first of three epochs in cryptography. It is characterized by the use of pens, papers, or simple mechanical devices. Although there are still examples of unsolved ciphers (Beale-Chiffre [74]) the epoch of pen and paper cryptography came to an end because of the mentioned devastating experience in the first World War.

The next epoch of cryptography is characterized by the use of special machines built for encrypting and decrypting. Although there are many other examples, most famous is probably the *ENIGMA*. It was invented by the German electrical engineer Arthur Scherbius [3] in 1918. The earliest models in 1920 were commercial models. In the mid-1920s the machine stroke the German military’s eye. The ENIGMA reminds of an ordinary type writer. Its heart are three wheels. Each wheel contains on both sides 26 electrical contacts for the 26 letters of the German alphabet. These contacts are paired irregularly. After each keystroke the wheels are turned by a rotor. The key consists of the order of the wheels, the initial position of the rotors, the position of the alphabet relative to the rotor wiring, and how the electrical contacts (plugs) are connected. The number of possible configurations has been calculated to be around 10^{114} (cf. [54]). That is why the ENIGMA cipher is considered to be unbreakable. However the way ENIGMA was actually built raised some inherent problems. The machine’s greatest weakness though, was the way it was used. Without the operating shortcomings ENIGMA would, almost certainly, not have been broken [18].

During the Cold War, cryptography became more and more a secret science, as governments classified all information concerning cryptography confidential. Only with the advent of computers in 1970s, the public’s need for cryptography raised and so it became a public area of research. This marks the beginning of the third epoch of cryptography. To set a standard for en- and decrypting IBM and the National Security Agency (NSA) introduced the Data Encryption Standard [27] (DES) in 1976. Although in 2001 a new standard, the Advanced Encryption Standard [65] (AES), was published, the DES is still used, especially a modified improved version, the Triple-DES [27]. Another significant progress in cryptography happened in 1976. Until then it was always necessary to

exchange the key or the code word over a secure channel. Whitfield Diffie and Martin Hellman proposed an algorithm that allowed two parties to securely agree on a key over an insecure channel. Even if someone eavesdrop all the communication between the two parties, it is not be possible to reconstruct the key.

2.1 What is Cryptography?

World War I and the advent of computers significantly changed the meaning of cryptography. Until then, cryptography was effectively synonymous with encryption. Encryption is the transformation of information in a readable state to apparent nonsense. The re transformation is called decryption. The secret was the technique to decode the message. Everybody who knows the technique, including unwanted persons, can decrypt the message. This kind of strategy is called *security by obscurity.*

Modern cryptography, or cryptography as we know it, became rapidly more complex. Nowadays, it is an intersection between the various fields of mathematics, computer science, and electrical engineering. Probably the most important person in this context is Claude Shannon, often referred to as the "father of mathematical cryptography". It was his paper "Communication Theory of Secrecy System" [73], which constituted a new age of cryptography. Instead of keeping the techniques to decrypt a message secret, he introduced the concept of cryptography, which bases its security on mathematical problems. Shannon coined the phrases of theoretical secrecy (nowadays unconditional security) and practical secrecy (computational security). If an encryption systems is said to be unconditionally secure, it means that even if an adversary has unconditional resources (e.g., time, computational power...), the cryptosystem will not be broken. On the other hand, a computational secure system can be broken. However, it is assumed that an adversary does not have the required resources to do so. An example, the only example there is, for an unconditionally secure cipher is the one-time pad (OTP). In this encryption technique the plaintext is paired with a random and secret key, which is at least as long as the plaintext. If the key is truly random and never reused the ciphertext is impossible to break. However, the OTP is rather impractical. How to exchange the key? Where to store the key? And additionally, we need a new key every time we encrypt something!

In modern cryptography, the security of a cryptosystem (the hardness of breaking it) is specified in the amount of operations that have to be executed in order to break the cryptosystem. The amount of operations usually is given as a power of two or specified in bits. But what does, for instance, a 64-bit security level mean in practice? It means that in order to break such a cryptosystem, 2^{64} steps are required. There is a small story to get an idea of this kind of numbers narrated by Rudolf Taschner [78]: Once upon a time in India there was a wealthy and happy maharaja. But one day his wife died and nobody could cheer him up, until an old man came and taught the maharaja chess. This game fascinated him so much, he forgot about his wife. In order to thank the old man, the maharaja granted the old man a wish. He wished that the maharaja would start with one rice grain on the first field on the chessboard, on the second field there should be twice as much as on the first, on the third twice as much as on the second and so on. At first the maharaja was incensed by this apparent littleness. However by no later than the 20th field, the maharaja acknowledged to himself that he had underestimated the wish, because his servants already counted over one million rice grains. Altogether it would have been

18,446,744,073,709,551,615 rice grains, $2^{64} - 1$, about the same size as 40,000 pyramids of Cheops. And even with a single field more on the chessboard, the amount would almost double. This story should just show figuratively, that compared to the 64-bit security level the 128-bit security level is not only twice as secure, but 18,446,744,073,709,551,616 times more secure.

2.1.1 Goals and Terminology

By encryption we mean the transformation of a *plaintext* into a *ciphertext*. To perform the inverse transformation, the decryption, one needs to know the secret *key*. Cryptography's purpose is the protection of information. According to This is achieved by the following four main goals (cf. [52]):

- **Confidentiality** means preventing the disclosure of information to adversaries, or in general unauthorized individuals or systems. This is the most obvious purpose of a cryptosystem and is achieved by *encrypting* the information.
- **Integrity** proves the accuracy and consistency of information or data over its entire life cycle. Integrity prevents unauthorized third persons to modify the data in an unauthorized or undetected manner. Typically *hash functions* offer this service.
- **Authenticity** ensures the genuineness of data and transactions. It also validates the identity of involved parties. This property is often incorporated by *digital signatures*.
- **Non-repudiation** implies that no party of a transaction can deny this transaction. It ensures that one party has received the transaction and the other party has sent the transaction.

There are two types of cryptography used to accomplish these objectives.

- **Symmetric Cryptography** is also referred to as *symmetric-key*, *single-key*, or *secret-key* cryptography. Symmetric cryptography is characterized by the usage of a single cryptographic key. The keys for encryption and decryption are identical or can be derived from one another. The secret key must be shared between all parties involved.
- **Asymmetric Cryptography** is also called *asymmetric-key* or *public-key* cryptography. Asymmetric cryptography is based on two keys, a *public key* and a *private key*. The two keys are somehow mathematically linked. The public key may be distributed, however the private key must be kept secretly. In general, the public key is used to encrypt plaintext, the private key to decrypt the resulting ciphertext.

2.2 Symmetric vs. Asymmetric Cryptography

Symmetric cryptography mainly provides one single cryptographic feature, namely confidentiality. Not until the 1970s with the advent of asymmetric cryptography all theoretical purposes of cryptography can be realized. Until the discovery of digital signatures and hash functions, it was believed that confidentiality and authenticity were intrinsically linked (cf. [52]). However, it is essential to distinguish these two properties. Confidentiality gives no evidence about the identity of the the involved parties or the origin of the

encrypted data. This is not only useful, but essential in many different settings. For example, how to prove who signed a contract, if both parties are able to sign with the same key?

Moreover asymmetric cryptography addresses several inherent issues of symmetric cryptography:

- **Key Distribution Problem**. In many settings, symmetric is used to enable confidential communication over an insecure channel. The arising question is how to distribute the key. The insecure channel obviously cannot be used, otherwise an adversary gets hold of the key. The secret key can only be transmitted over an secure channel. Though, in general such secure channels hardly exist. In asymmetric cryptography the public key used to encrypt data can be transmitted over insecure channels. Only with the private key mathematically paired to the public key one can decrypt the ciphertext. A further issue of symmetric cryptography is that every unique pair of parties must have a unique key. However, this scales quadratic with the number of people. In a system with n people, there are $n(n-1)/2$ keys involved. Public-key cryptography inherently avoids this issue.

- **Trust Problem**. In symmetric key cryptography all involved parties have the same possibilities, as they all posses the same key. However, sometimes it is important to verify the originator of the message. In asymmetric cryptography this can be achieved, as there are different keys for encryption and decryption. If the private key is used to encrypt a message, the ciphertext is linked to the owner of the private key.

Asymmetric cryptography relies on a *trapdoor function*, which is easy to compute in one direction, yet believed or proven to be difficult in the opposite direction. Ideally would be a *one-way trapdoor function*, a function which cannot be inverted. However it is an unsolved problem in theoretical computer science if such functions even exist. Today, the three most prominent trapdoor functions which are of practical use, are based on the following problems:

- **Integer-Factorization Problem**. Integer factorization or prime factorization is the decomposition of an integer into its prime factors.

- **Discrete Logarithm Problem**. The Discrete Logarithm Problem (DLP) is the problem of computing the discrete logarithm in finite fields. Given a prime p , a generator α, an element $\beta \in \mathbb{Z}_p^*$ find $x \in \mathbb{Z}_p^*$ such that $\alpha^x \equiv \beta \bmod p$.

- **Elliptic Curve Discrete Logarithm Problem**. The Elliptic Curve Discrete Logarithm Problem (ECDLP) is a generalization of the DLP. More on this problem can be found in Section 4.2

In comparison to symmetric-key cryptography, public-key cryptography requires very long keys. To support this claim, have a look at Table 2.2.

Table 2.2: Key sizes for different encryption algorithms for various security levels.

Algorithm Family		**Security level** in bits *required key/group size* in bits			
		80	**128**	**192**	**256**
Symmetric		*80*	*128*	*192*	*256*
Integer factorization		*1,024*	*3072*	*7,680*	*15,360*
DLP	key	*160*	*256*	*192*	*512*
	group	*1024*	*3,072*	*7,680*	*15,360*
ECDLP		*160*	*256*	*384*	*512*

Chapter 3

Mathematical Background

In this section, we want to revisit some mathematical basics, as fundamentals of abstract algebra, in particular groups, rings, and fields, which are essential for understanding ECC. This is also important to establish a common language and formalism. We refer to Hankerson *et al.* [33] and Cohen *et al.* [19] for references, further details, and some neglected proofs.

3.1 Groups

Definition 3.1. Given a set S, an operation or *composition law* $\circ$ of S into itself, $\circ$ is a mapping from the Cartesian product $S \circ S$ to S.

Definition 3.2. A group is a set of elements G together with an composition law $\circ$ such that

- The composition law is *associative*, that is for all $a, b, c \in G$ we have $a \circ (b \circ c) = (a \circ b) \circ c$.
- $\circ$ has a *unit element* $e \in G$ such that for all $a \in G$ we have $a \circ e = e \circ a = a$.
- For every $a \in G$ there exists an *inverse element* a^{-1} such that $a \circ a^{-1} = e$.
- The group G is *commutative* (or *abelian*) if $a \circ b = b \circ a$ for all $a, b \in G$.

The unit element of a group G is necessarily always unique, as well as the inverse of an element. If G is commutative, the inverse of a is usually denoted by $-a$.

Definition 3.3. The *order of an element* $ord(a)$ of a group $(G, \circ)$ is the smallest positive integer k such that

$$a^k = a \circ a \circ a ... \circ a = e. \tag{3.1}$$

Definition 3.4. A group is cyclic if there is $\alpha \in G$ such that $ord(\alpha) = |G|$. α is then called a *generator* of G. The group G is called finite if the group's order is finite. Then it holds for every $a \in G$ that

- $a^{|G|} = e$.
- $ord(a)$ divides $|G|$.

Definition 3.5. A *subgroup* is a subset of a cyclic group G that is a group itself. Every element $a \in G$ with $ord(a) = s$ is the primitive element of a cyclic subgroup H with s elements. The subgroup H itself is associative and commutative, has the same neutral element e as G, and for every element $a \in H$ there is an $a^{-1} \in H$.

If a subgroup is of prime order (or *cardinality*) q it should be noted that there can be multiple generators. For subgroups with a prime group cardinality q it holds that all elements other than e have order q. In general for a group G each element $a \in G$ generates some subgroup H. Another important property follows *Lagrange's theorem.*

Theorem 3.1. (*Lagrange's theorem*) If H is a subgroup of G than $|H|$ divides $|G|$. For every integer k that divides the order n of a finite group G with the generator α there exists exactly one subgroup H of G of order k. The generator of this subgroup H is $\alpha^{n/k}$. The elements of H are exactly the elements $a \in G$ that satisfy $a^k = 1$ in G.

3.2 Rings

Definition 3.6. A set together with two composition laws $\times$ and $+$ is a *ring* R if

- $(R, +)$ is a commutative group.
- $\times$ is associative and has a unit element 1. The unit element of $(R, +)$ is different, it is 0.
- $\times$ is *distributive* over $+$, that means for all $a, b, c \in R$, $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$.

A ring R is commutative if its composition law $\times$ is commutative.

Definition 3.7. Given two ring $(R, +, \times)$ and $(R', \oplus, \otimes)$ a *homomorphism* ψ is an application from R to R' such that for all $a, b \in R$

- $\psi(a + b) = \psi(a) \oplus \psi(b)$.
- $\psi(a \times b) = \psi(a) \otimes \psi(b)$.
- $\psi(1) = 1$.

Definition 3.8. Given a ring R, let there be a natural homomorphism ψ from R to $\mathbb{Z}$:

$$\psi(n) = \begin{cases} 1 + 1 + ... + 1 & n \text{ times if } n \geq 0 \\ -(1 + 1 + ... + 1) & -n \text{ times if } n < 0. \end{cases} \tag{3.2}$$

If R is finite, $\psi(n)$ in Equation 3.2 must be zero for some n. The kernel of ψ, denoted $\ker(\psi(n))$, is specified to contain this/these element(s). If the kernel is the group of multiples of n ($n\mathbb{Z}$), we call n the characteristic of R, denoted by $\text{char}(R)$.

Definition 3.9. An element $a \in R$, R being a Ring, is called *invertible* if for some b it holds that $ab = ba = 1$. The *inverse* of a is denoted by a^{-1}. The set of invertible elements is a multiplicative group, denoted by $R*$.

3.3 Fields

Definition 3.10. A *field* $\mathbb{F}$ is a commutative ring and all its nonzero elements are invertible.

Let us summarize the definition of a field:

Definition 3.11. A set of elements is called a field $\mathbb{F}$ if

- The elements of $\mathbb{F}$ form an additive group with the composition law + and the neutral element 0.
- The elements of $\mathbb{F} \setminus \{0\}$ form an multiplicative group with the composition law $\times$ (or $\cdot$) and the neutral element 1.
- The distributive law holds, see Definition 3.6.

A field $\mathbb{F}$ is called *finite* if the set $\mathbb{F}$ is finite. The two operations are addition + and multiplication $\times$. *Subtraction* of elements is done by adding the *negative*. The negative of an element a is denoted by $-a$ and is the unique element in the field $\mathbb{F}$ such that $a+(-a) = 0$. Similarly, *division* is defined as multiplication with the inverse element. The inverse of an element b is denoted b^{-1} and is a unique element in $\mathbb{F}$ such that $b \cdot b^{-1} = 1$. The *order* of an field $\mathbb{F}$ is its number of elements.

Theorem 3.2. A field $\mathbb{F}$ of order q exists if and only if q is a prime power, $q = p^m$ for some positive integer m. If $m = 1$ then $\mathbb{F}$ is called a *prime field*, for $m \geq 2$ $\mathbb{F}$ is called *extension field*. p is called the *characteristic* of $\mathbb{F}$.

Fields of order q are denoted by $\mathbb{F}_q$. For every q there exists only one unique field $\mathbb{F}_q$, only the labeling of the elements may differ. That means that all fields $\mathbb{F}_q$ are *isomorphic* to each other.

3.3.1 Prime fields

A *prime field* or *Galois* field with a prime number of elements $\mathbb{F}_p$ is defined as the integer ring $\mathbb{Z}_p$. $\mathbb{Z}_p$ is the set of integers $\{0, 1, ..., p-1\}$. The multiplicative and additive invertibility of all nonzero elements is a property all fields share, as stated in Section 3.3. p is called the *modulus* of the prime field $\mathbb{F}_p$. The unique integer remainder r is defined as follows:

$$r = a \bmod p = a - \lfloor a/p \rfloor. \tag{3.3}$$

The operation of determining r is called *reduction modulo p*.

3.3.2 Binary Fields

Definition 3.12. A *binary field* $\mathbb{F}_2$ or GF(2) is also called *Galois Field*. It contains two elements, 0 and 1, the additive and the multiplicative, respectively, unity.

A binary field may be considered as a prime field of order two with $p = 2$. The arithmetic in binary fields can be implemented using Boolean algebra. Addition is a logical *XOR*, multiplication a logical *AND*. This property makes binary fields of particular interest.

3.3.3 Binary Extension Fields

Definition 3.13. Given two fields L and K, we call L an *extension field* of K if there exists a field homomorphism (Definition 3.7 is also applicable to fields) from K into L. We say L/K is a field extension of L over K.

Definition 3.14. Let L be a field extension over K. L/K can be viewed as K-dimensional vector space. The *degree* of an extension is the dimension of L/K.

A *Binary extension field* is an extension of the binary field. We denote these fields $\mathbb{F}_{2^m}$, m being the degree of extension. It contains $q = 2^m$ elements. Every element consists of a sequence of m coefficients $a_i \in \mathbb{F}_2$. There exist different ways to represent these elements. Relevant for our purposes are the *polynomial* (or canonical or standard) basis and the *normal* basis.

3.3.4 Polynomial Basis

In the polynomial basis, the elements are represented as polynomials in $\mathbb{F}_{2^m}$ of degree less than m. The polynomial basis is $\{1, \alpha, \alpha^2, ..., \alpha^{m-1}\}$, where α is a root of a primitive *irreducible polynomial* of degree m over $\mathbb{F}_{2^m}$. Every element can uniquely be expressed as a polynomial in α over $\mathbb{F}_{2^m}$ of degree less than m. Every element $A(\alpha) \in \mathbb{F}_{2^m}$ is thus represented as:

$$A(\alpha) = a_{m-1}\alpha^{m-1} + a_{m-2}\alpha^{m-2} + ... + a_1\alpha + a_0. \tag{3.4}$$

The binary vector $a = (a_{m-1}, a_{m-2}, ..., a_1, a_0)$ is associated with the field elements. The irreducible polynomial is the equivalent to the prime p in prime fields. Irreducible means that the polynomial cannot be factored into a product of polynomials with degree less than m.

Addition is performed as a polynomial addition with coefficient arithmetic performed modulo 2. Let $A(\alpha), B(\alpha) \in \mathbb{F}_{2^m}$.

$$C(\alpha) = A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i, \qquad c_i \equiv a_i + b_i \bmod 2. \tag{3.5}$$

Example 3.1. Let $A(\alpha) = \alpha^4 + \alpha^3 + \alpha + 1, B(\alpha) = \alpha^4 + \alpha^2 + 1 \in \mathbb{F}_{2^5}$. We want to compute $C(\alpha) = A(\alpha) + B(\alpha)$.

$$\begin{array}{lccccccccc}
A(\alpha) = & \alpha^4 & + & \alpha^3 & & & + & \alpha & + & 1 \\
B(\alpha) = & \alpha^4 & & & + & \alpha^2 & & & + & 1 \\
\hline
C(\alpha) = & 2\alpha^4 & + & \alpha^3 & + & \alpha^2 & + & \alpha^1 & + & 2 \\
C(\alpha) = & & & \alpha^3 & + & \alpha^2 & + & \alpha^1 & & \\
\hline\hline
\end{array}$$

Given $A(\alpha), B(\alpha) \in \mathbb{F}_{2^m}$ and an irreducible polynomial $f(\alpha)$, polynomial multiplication is defined as

$$C(\alpha) \equiv A(\alpha) \cdot B(\alpha) \bmod f(\alpha). \tag{3.6}$$

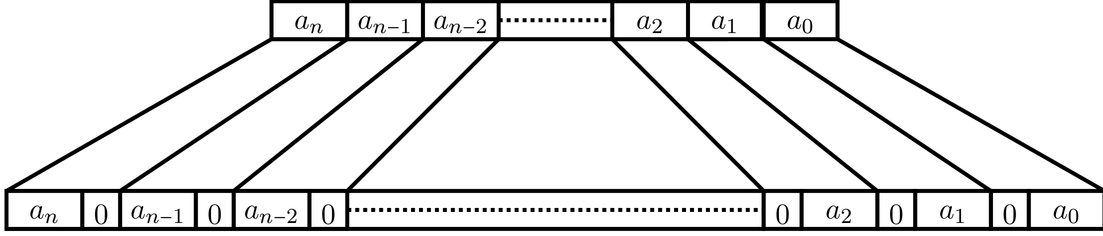

Figure 3.1: The squaring operation in binary fields when using polynomial representation.

Example 3.2. Let $A(\alpha) = \alpha^3 + \alpha^2 + 1, B(\alpha) = \alpha^3 + 1 \in \mathbb{F}_{2^5}$ with $f(\alpha) = \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + 1$. We want to calculate $C(\alpha) = A(\alpha) \cdot B(\alpha) \bmod f(\alpha)$. We begin by computing

$$D(\alpha) = A(\alpha) \cdot B(\alpha) = \alpha^6 + \alpha^5 + \alpha^3 + \alpha^2. \tag{3.7}$$

Modular reduction *modulo* $f(\alpha)$ can be done by individually reducing the partial terms of $D(\alpha)$. Terms with an exponent smaller than 5 are already reduced. Thus, we only have to perform the reduction for α^5 and α^6. We start with α^5:

$$\begin{aligned} f(\alpha) &= \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + 1 \\ \alpha^5 &= f(\alpha) + \alpha^4 + \alpha^3 + \alpha + 1 \\ \alpha^5 &\equiv \alpha^4 + \alpha^3 + \alpha + 1 \qquad \bmod f(\alpha). \end{aligned} \tag{3.8}$$

If we multiply α^5 by α we get α^6. The term α^5 in the resulting product has just been reduced, so it is replaced with the result of Equation 3.8.

$$\begin{aligned} \alpha^6 &= (\alpha^5) + (\alpha^3 + \alpha^2 + \alpha) \\ \alpha^6 &\equiv (\alpha^4 + \alpha^3 + \alpha + 1) + (\alpha^3 + \alpha^2 + \alpha) \quad \bmod f(\alpha) \\ \alpha^6 &\equiv \alpha^4 + \alpha^3 + 1 \qquad \bmod f(\alpha). \end{aligned} \tag{3.9}$$

The final result is obtained with the results of Equation 3.8 and Equation 3.9.

$$\begin{aligned} C(\alpha) &\equiv D(\alpha) \quad \bmod f(\alpha) \\ C(\alpha) &\equiv \alpha \quad \bmod f(\alpha). \end{aligned} \tag{3.10}$$

The squaring operation can be achieved by multiplying the value with itself. Due to the fact that the coefficient arithmetic is performed modulo 2 in binary extension fields, all odd powers vanish in the result and only even powers remain. Thus, the result always follows a simple pattern: Insert zeros between each coefficient of the original value, as shown in Figure 3.1. The reduction is performed afterwards.

3.3.5 Normal Basis

This section gives a short introduction to normal bases. A more detailed description would be beyond scope of this work. We refer to Gao and Lenstra [30], Lidl [48], and Mullin *et al.* [58] for details.

A *normal basis* is of form $\left\{\beta, \beta^2, ..., \beta^{2^{m-1}}\right\}$ and is said to be *generated* by the normal element $\beta \in \mathbb{F}_{2^m}$. So $A(\beta) \in \mathbb{F}_{2^m}$ is represented in the normal basis as

$$A(\beta) = a_{m-1}\beta^{2^{m-1}} + a_{m-2}\beta^{2^{m-2}} + ... + a_1\beta^2 + a_0\beta.$$

An important property of a normal basis $\mathbb{F}_{2^m}$ is

$$\beta^{2^m} = \beta. \tag{3.11}$$

A Multiplication in normal basis is, compared to polynomial basis, more complex. Multiplying two elements $A(\beta), B(\beta) \in \mathbb{F}_{2^m}$ results in a third element $C(\beta) \in \mathbb{F}_{2^m}$ and is defined via m *multiplication matrices* $\lambda^{(k)} \in \mathbb{F}_2$ as

$$c_k = \sum_{i}^{m-1} \sum_{j}^{m-1} \lambda_{ij}^{(k)} a_i b_j. \tag{3.12}$$

The space and time costs of this multiplication are related to the number of nonzero coefficients in its multiplication matrices. Mulin *et al.* [58] showed, that this number is at least $2m-1$ for $\mathbb{F}_{2^m}$. These bases are called *optimal normal bases* (ONB). There exist two types of ONBs, type 1 and type 2. Type 1 exists for composite integers m, type 2 for prime ones. They are related to Gauss periods and where specified by Gao and Lenstra [30]. For security reasons only prime extension degrees (so ONB type 2) are relevant for ECC.

A Squaring in normal basis representation is because of Equation 3.11 a simple *rotate left* operation:

$$\begin{array}{lllllllll} A(\alpha) & = a_{m-1}\alpha^{2^{m-1}} & + & a_{m-2}\alpha^{2^{m-2}} & +...+ & a_1\alpha^2 & + & a_0\alpha & \\ A^2(\alpha) & = a_{m-2}\alpha^{2^{m-1}} & + & a_{m-3}\alpha^{2^{m-2}} & +...+ & a_0\alpha^2 & + & a_{m-1}\alpha & . \end{array} \tag{3.13}$$

3.3.6 Basis Transformation

The efficiency of finite field arithmetic depends on how the elements are represented. The most resource consuming field operation is multiplication. The best space complexity results for type 2 ONB multipliers were achieved by Sunar *et al.* [77]. They need m^2 and $3m(m-1)/2$ AND respectively, XOR gates for an m-bit multiplier. This is the lower bound for normal basis multiplication. In comparison, the well known Karatsuba algorithm for polynomial basis multiplication has a space complexity of $\mathcal{O}(n^{log_2 3})$. However, squaring can be achieved in normal basis by means of a simple cyclic shift. In polynomial basis a modular reduction is required after each squaring.

Thus, if we perform a basis transformation, we can benefit from both representations. Recall that finite fields with the same number of elements are isomorphic to one another. Hence, there always exists a unique mapping between elements in different bases. A field element in polynomial basis $a(\alpha) = a_0\alpha^0 + a_1\alpha^1 + ... + a_{m_1}\alpha^{m-1}$ can be converted to an element in normal basis $a'(\beta) = a'_0\beta^{2^0} + a_1\beta^{2^1} + a_{m-1}\beta^{2^{m-1}}$ by a conversion matrix C. In order to perform the inverse transformation this matrix C has to be invertible. The existence of such an inverse C^{-1} depends on the irreducible polynomial. The matrix C is constructed in the following way: calculate $\beta, \beta^2, \beta^4, ..., \beta^{2^{m-1}} \bmod f(\alpha)$ in polynomial basis and fill each row with one result, see Equation 3.14.

$$\mathbf{C} = \begin{bmatrix} \beta & \bmod f(\alpha) \\ \beta^2 & \bmod f(\alpha) \\ \dots & \dots \\ \beta^{2^{m-1}} & \bmod f(\alpha) \end{bmatrix} = \begin{bmatrix} c_{11} \cdot \alpha^{m-1} & c_{12} \cdot \alpha^{m-2} & \dots & c_{1(m-1)} \cdot 1 \\ c_{21} \cdot \alpha^{m-1} & c_{22} \cdot \alpha^{m-2} & \dots & c_{2(m-1)} \cdot 1 \\ \vdots & \vdots & \ddots & \vdots \\ c_{(m-1)1} \cdot \alpha^{m-1} & c_{(m-1)2} \cdot \alpha^{m-2} & \dots & c_{(m-1)(m-1)} \cdot 1 \end{bmatrix} \tag{3.14}$$

3.4 Multiplicative Inverse

Recall that the multiplicative inverse of a nonzero element a in a finite field $\mathbb{F}_p$ is denoted a^{-1} and satisfies $aa^{-1} \equiv 1 \bmod p$. An efficient way to perform an inversion in finite fields is to apply the *extended Euclidian algorithm* (EEA), which is based on the *Euclidian algorithm*. The Euclidian algorithm solves the problem of computing the *greatest common divisor* (gcd) by reducing the problem of finding the gcd of two numbers recursively to finding the gcd of two smaller numbers, see Algorithm 3.1.

Algorithm 3.1 The Euclidian algorithm to find the gcd of two given numbers.

Input: a, b
Output: $\gcd(a, b)$

1: **function** EUCLID(a, b)
2:     **if** $b = 0$ **then**
3:         **return** a
4:     **else**
5:         **return** EUCLID($a, a \bmod b$)
6:     **end if**
7: **end function**

8: **return** EUCLID(a, b)

The EEA computes a linear combination of the form

$$\gcd(a, b) = d = sa + tb. \tag{3.15}$$

The idea behind this algorithm is to execute the standard algorithm and in every iteration the remainder is expressed as a linear combination of the inputs. In order to compute an inverse $a^{-1} \bmod p$, apply the EEA, see Algorithm 3.2, with the inputs a, p. If the resulting gcd d is equal to one, the integer a is invertible. In case of prime fields the gcd for all elements, expect zero, is 1. a^{-1} can be derived from Equation 3.15:

$$d = 1 = sa + tp \equiv sa \bmod p. \tag{3.16}$$

According to Definition 3.9 we have the inverse given as

$$1 = sa = a^{-1}a \bmod p. \tag{3.17}$$

Besides the EEA, there exists another way to perform inversion in finite fields. It is based on Fermat's little theorem, which states that given a prime q and an integer a

$$a^q \equiv a \bmod q. \tag{3.18}$$

If a and q are coprime to each other it follows that

$$a^{q-1} \equiv 1 \bmod q. \tag{3.19}$$

To calculate the inversion we divide Equation 3.19 by a and get

$$a^{q-2} \equiv a^{-1} \bmod q. \tag{3.20}$$

Algorithm 3.2 The extended Euclidian algorithm to find the multiplicative inverse.

Input: a, b

Output: The gcd of a and b in the form $\gcd(a, b) = d = sa + tb$

```
1: function EXTENDED EUCLID(a, b)
2:     if b = 0 then
3:         return (a, 1, 0)
4:     end if
5:     (d, u, v) ← EXTENDED EUCLID(b, a mod b)
6:     (s, t) ← (v, u − ⌊a/b⌋v)
7:     return (d, s, t)
8: end function

9: return EXTENDED EUCLID(a, b)
```

Expanding this technique to binary fields where $q = 2^m$ the inversion is defined as

$$a^{2^m - 2} \equiv a^{-1} \quad \text{over } \mathbb{F}_{2^m}. \tag{3.21}$$

However, the computational effort to perform an inversion according to Equation 3.21 is rather big. To calculate the $2^m - 2$th power several multiplication and squaring operation have to be performed. Especially the multiplications are complex. Itoh and Tsuji [38] introduced an algorithm to minimize the number of multiplications. It is based on the following three observations. Let $a \in \mathbb{F}_{2^m}$ then

$$a^{2^m - 2} = (a^{2^{m-1} - 1})^2. \tag{3.22}$$

Let i be even then

$$a^{2^i - 1} = (a^{2^{i/2} - 1})^{2^{i/2}} \cdot a^{2^{i/2} - 1} \tag{3.23}$$

and

$$a^{2^{i+1} - 1} = (a^{2^i - 1})^2 \cdot a. \tag{3.24}$$

Itoh and Tusji's algorithm only needs $\lceil \text{ld}(m) \rceil$ multiplication and m squaring operations. An example can be found in Section 5.5.4.

3.5 Modular Arithemic

Modular arithmetic is essential, in particular to perform computations in prime fields. A modular addition can simply be achieved by adding the two summands and subtracting the modulus if the sum is greater than the modulus. However, modular multiplication is much more complex. The hard part is not the multiplication, but the modular reduction. One can simply perform the modular reduction as defined in Equation 3.3. However, this implies a division, and division are in general harder than multiplications. This is why there exist special reduction algorithms, as the *Montgomery reduction algorithm* [56].

Montgomery Multiplication

The *Montgomery multiplication* is a combination of a multiprecision multiplication algorithm with the Montgomery reduction algorithm. It trades the computational expensive

trial divisions required by the reduction step for additional multiplications. Essentially this is achieved by changing the reduction modulo p to an reduction modulo R, with $R > p$ and $\gcd(R, p) = 1$. In general, R is chosen to be a power of two, to simplify the reduction modulo R. A further parameter, namely p', is computed using the extended euclidian algorithm: $RR^{-1} + pp' = 1$. To perform a multiplication, the factors have to be transformed into the *Montgomery domain*, which is done by multiplying them with R, as showed in Equation 3.25

$$\tilde{a} = aR \bmod p \tag{3.25}$$

Equation 3.26 shows how to transform integers from the Montgomery domain to the normal domain.

$$a = \tilde{a}R^{-1} \bmod p \tag{3.26}$$

The actual Montgomery multiplication is shown in Equation 3.27.

$$\begin{aligned} d &= \tilde{a}\tilde{b} = abR^2 \\ u &= ((dp' \bmod R)p + d)/R \\ \tilde{c} &= \begin{cases} u & \text{if } u < n \\ u - n & \text{else} \end{cases} \end{aligned} \tag{3.27}$$

The product dp' is reduced modulo R, which means that only the $\mathrm{ld}(R)$ least significant bits have to be calculated. The division by R in Equation 3.27 means, that only the $\mathrm{ld}(R)$ most significant bits of $(dp' \bmod R)p + d)$ are required to be computed. In order to avoid the expensive transformation to the Montgomery domain and back to the normal domain, it is reasonable to stay in the Montgomery domain if possible.

Chapter 4

Elliptic Curve Cryptography

The history of elliptic curves has its roots around the 2nd or 3rd century in ancient Alexandria. In *Diophantus of Alexandria's Arithmetica* [35] elliptic curves have their mathematical debut. Diophantus posed the following question: Given a number a, find x and y such that

$$y(a-y) = x^3 - x. \tag{4.1}$$

Diophantus solved the problem for $a = 6$ by transforming Equation 4.1 to have only a cubic and quadratic term by substituting $x = ky - 1$ and choosing $k = 3$. He obtained $x = 17/9$ and $y = 26/27$. A modern interpretation of Diophantus' solution is done in the following way: Construct the tangent line at the point $(0, -1)$ and get the point $(17/9, 26/27)$ where the tangent re-intersects the curve.
Diophantus himself had no idea about analytic geometry or about elliptic curves. But the mathematical problem he posed marks the beginning of *elliptic curve cryptography.* (cf. [14]).

Leonardo of Pisa, better known as *Fibonacci* [26], encountered a problem from arabic manuscripts of roughly the 8th century, namely to find rational numbers x such that both $x^2 + 5$ and $x^2 - 5$ are rational squares. Fibonacci found $x = 41/6$. He defined a positive integer n to be a congruent number if $u^2 - n$, u^2, and $u^2 + n$ are all nonzero squares for some rational number u. Also the product of these three numbers is a nonzero rational square v^2. What Fibonacci probably did not know is that - in modern terminology - he generated points on an elliptic curve: (u^2, v) is a point on the elliptic curve $E_n : y^2 = x^3 - n^2 x$ (cf. [67]).

After this first two appearances of elliptic curves and diophantine equations (polynomial equations with two or more unknowns, where only integer solutions are studied) disappeared until Fermat and Euler concerned themselves with them. Fermat's famous last theorem [22] originated in Fermat's studies about Diophantus' work.

In the 1670s, Newton [59] was the first to explain the mysteries behind Dophantus' problem. He pointed out, that Diophantus was actually intersecting a cubic curve with a line. He followed, that in general such an intersection consists of three points, unless the line is a tangent, then two points are the same.

Gauss' work [31] gave number theory a new direction, thus elliptic cures again were lost sight of. In the beginning of the 20th century elliptic curves reappeared. Mordell [57] and

Weil [83] proved Poincarés assumption, that the group of rational points on an elliptic curve is generated by a finite number of points. In the 1980s Lenstra [46] showed that elliptic curves could be used for factoring integers. This fact inspired Neal Koblitz [43] and Victor Miller [55] to independently propose a cryptographic system based on elliptic curves in 1985. In the late 1990's elliptic curves started to grow commercial and industrial acceptance.

This chapter contains an overview on elliptic curves. Relevant properties and definitions in general are given in Section 4.1. Beginning with Section 4.1.1 we mainly focus on elliptic curves over binary fields.

4.1 Arithmetic on Elliptic Curves

Elliptic curves are used in many cryptographic applications. One reason can be found in their fast group law and because no sub exponential attacks are known. The group law is based on arithmetic involving the points on an elliptic curve. Arithmetic on elliptic curves is defined in terms of arithmetic operations on an underlying field, e.g., a prime field or a binary extension field.

In this section we summarize the main properties. For references, further details, and some neglected proofs we refer to Hankerson *et al.* [33] and Cohen *et al.* [19] .

Definition 4.1. An *elliptic curve* E over a field K is denoted by E/K or $E(K)$ and is defined by the *Weierstrass equation*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a6 \tag{4.2}$$

with $a_1, ..., a_6 \in K$.

Definition 4.2. E as defined in Equation 4.2 is said to be *nonsingular* or *smooth* if for each *point* (x_1, y_1), the partial derivatives $2y_1 + a_1x_1 + a_3$ and $3x_1^2 + 2a_2x_1 + a_4 - a_1y_1$ do not become zero simultaneously. This means that for every point only one distinct tangent line exists. This property can also be expressed using the *discriminant* Δ of an curve E

$$\Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \tag{4.3}$$

with

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

A pair of elements (x, y), $x, y \in K$, is called a point, if it satisfies the equation of the curve.

The Weierstrass Equation 4.2 can be simplified considerably by an transformation called *admissible change of variables.* The admissible change of variables transforms an elliptic curve to another isomorphic (a bijective homomorphic) curve. For $u, r, s, t \in K$, $u \neq 0$ the admissible change of variables is defined is defined as

$$(x, y) \to (u^2x + r, u^3y + u^2sx + t). \tag{4.4}$$

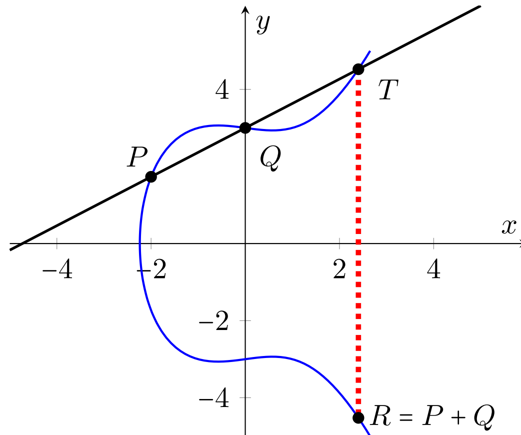

Figure 4.1: Point addition on an elliptic curve.

The admissible change of variables applied to curves over a field K with $\text{char}(K) = 2$ transforms Equation 4.2 to

$$y^2 + xy = x^3 + ax^2 + b. \tag{4.5}$$

For the sake of completeness, for fields K with a characteristic not equal to two or three (e.g., prime fields), the transformation results into the following equation

$$y^2 = x^3 + ax + b. \tag{4.6}$$

Let E be an elliptic curve defined over a field K. The set of points $E(K)$ forms an abelian group with ∞ serving as its identity. The group operation is called *addition of points* and is defined by the *chord-and-tangent rule*. Let $P = (x_1, y_1), Q = (x_2, y_2)$ be two points on an elliptic curve E over K. That means both pairs $(x_1, y_1), (x_2, y_2) \in K$ satisfy the equation E. The group operation addition applied to P and Q results in a third point $R = (x_3, y_3)$ with $R = P + Q$ and is constructed as follows: Draw a line from P to Q. The line intersects the curve at a point. Reflecting this point about the x-axis results in R. This *point addition rule* is depicted in Figure 4.1. Note that the intermediate point T is the negative of $R : -R$.

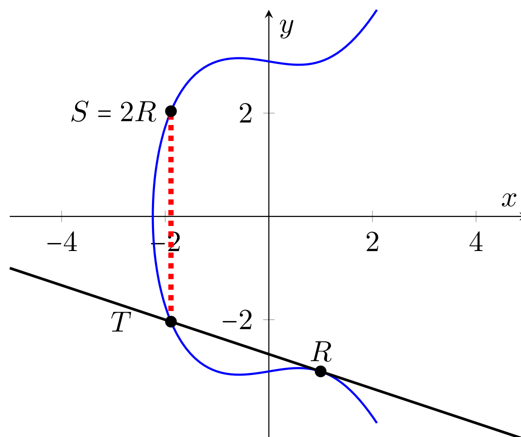

Figure 4.2: Point doubling on an elliptic curve.

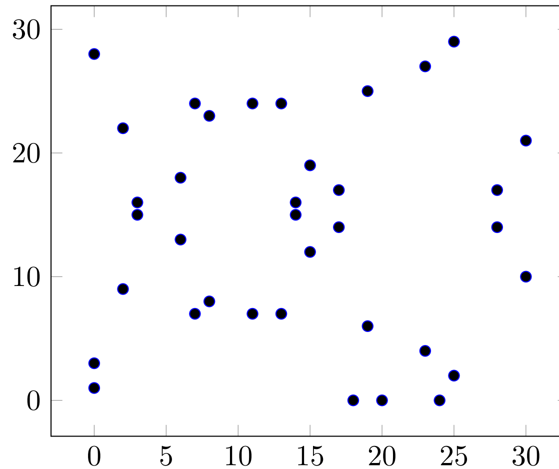

Figure 4.3: The elliptic curve $E : y^2 = x^3 - x + 9$ over $\mathbb{F}_{31}$.

Point doubling is a special case of a point addition. It is the addition of a point $R(x_1, x_2)$ with itself, resulting in a point $S = (x_3, y_3)$ with $S = R + R = 2 \cdot R$. The double of a point is constructed as follows: Draw the tangent line at P. The intersection (there is only one intersection) is depicted as T in Figure 4.2. Reflecting this point about the x-axis gives the point $S = 2R = R + R$. The intersection T is the negative of $S : -S$.

Figure 4.2 and Figure 4.1 show elliptic curves over $\mathbb{R}$. An elliptic curve over a finite field is not reminiscent of an curve anymore: Figure 4.3 shows the elliptic curve $E : y^2 = x^3 + x + 9$ (the same as in Figure 4.2 and Figure 4.1) over a prime field, namely $\mathbb{F}_{31}$. All the arithmetic is therefore done over $\mathbb{F}_{31}$. Elliptic curves over binary fields $\mathbb{F}_{2^m}$ cannot be plotted properly, one would need an m-dimensional plot.

4.1.1 Group Law

To form a group, several rules need to be considered, as stated in Section 3.1. In the following the group law for an elliptic curve E over a binary extension field $\mathbb{F}_{2^m}$ is summarized:

1. Identity: The point at infinity ∞. $P + \infty = P$ for all $P \in E(\mathbb{F}_{2^m})$.

2. Negatives: The negative of point $P = (x, y) \in E(\mathbb{F}_{2^m})$ is denoted $-P = (x, x + y)$. $-P$ is indeed a point on the elliptic curve. Note that $-P + P = \infty$.

3. Addition: Adding two points $P = (x_1, y_1), Q = (x_2, y_2) \in E(\mathbb{F}_{2^m})$, whereas $P \neq Q$ results in a third point $R = (x_3, y_3) \in E(\mathbb{F}_{2^m})$, where

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \end{aligned} \tag{4.7}$$

   with $\lambda = \frac{y_1 + y_2}{x_1 + x_2}$.

4. Doubling: Adding a point $P = (x, y) \in E(\mathbb{F}_{2^m})$ to itself, where $P \neq -P$ is called point doubling and results in a point $R = (x_3, y_3) \in E(\mathbb{F}_{2^m})$, where

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a \\ y_3 &= x_1^2 + \lambda x_3 + x_3 \end{aligned} \tag{4.8}$$

with $\lambda = x_1 \frac{y_1}{x_1}$.

4.1.2 Group Order and Group Structure

The *group order* is the number of points of an elliptic curve $E(\mathbb{F}_q)$. It is called the order of E over $\mathbb{F}_q$ and is denoted by $\#E(\mathbb{F}_q)$.

Theorem 4.1. Let E be an elliptic curve defined over $\mathbb{F}_q$. Then *Hasse's theorem* proves that

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq p + 1 + 2\sqrt{q}. \tag{4.9}$$

The group order depends on the elliptic curve's parameters a and b. For every prime field, there exists at least one elliptic curve having a prime order. However, curves over binary extension fields never have a prime group order. Their group order can be factored into several terms. These terms are referred to as *cofactor* h and *suborder* n. The suborder n is defined to be prime, so the remaining factors can be found as the cofactor. The cofactor is at least two and always a multiple of two.

Example 4.1. Let $E(\mathbb{F}_{31}) : y^2 = x^3 - x + 9$ as shown in Figure 4.3. According to Hasse's Theorem 4.1, $21 \leq \#E(\mathbb{F}_q) \leq 44$. Start with $P = (0,1)$ and compute the points $2P = P + P$, $3P = 2P + P$ until $[i]P = P$ for some $i > 0$.

$$\begin{array}{rcccll} P & = & & = & (0,3) & \\ 2P & = & P+P & = & (25,29) & \\ 3P & = & P+2P & = & (11,24) & \\ \dots & & & & & \\ 36P & = & \infty & & & \\ 37P & = & P+36P & = & (0,3) & = P. \end{array} \tag{4.10}$$

Equation 4.10 shows, that the point $P(0,3)$ has order 37. Actually these are all the points on this curve E. However not every point's order is equal to the group order. For instance if we start with $P(0,18)$ we get

$$\begin{array}{rcccll} P & = & & = & (0,18) & \\ 2P & = & \infty & & & \\ 3P & = & P+2P & = & (0,18) & = P. \end{array} \tag{4.11}$$

Example 4.2. We also want to give an example using a binary extension field. Let $E(\mathbb{F}_{2^3}) : y^2 + xy = x^3 + x^2 + x$. $\mathbb{F}_{2^3}$ is defined by the irreducible (or characteristic) polynomial $\alpha^3 + \alpha + 1$. According to Hasse's Theorem 4.1, $4 \leq \#E(\mathbb{F}_q) \leq 15$. We start by $P(\alpha^2 + \alpha + 1, \alpha^2 + \alpha + 1)$ and get

$$\begin{array}{rclrcrl} P & = & (& \alpha^2+\alpha+1 & , & \alpha^2+\alpha+1) & \\ 2P & = & (& \alpha^2+1 & , & \alpha^2+1) & \\ 3P & = & (& \alpha+1 & , & 0) & \\ 3P & = & (& \alpha+1 & , & \alpha+1) & \\ 4P & = & (& \alpha^2+1 & , & 0) & \\ 5P & = & (& \alpha^2+\alpha+1 & , & 0) & \\ 6P & = & \infty & & & & \\ 7P & = & (& \alpha^2+\alpha+1 & , & \alpha^2+\alpha+1) & = P. \end{array} \tag{4.12}$$

Actually the order of $E(\mathbb{F}_{2^3})$ is 14. But this particular point's order is only 7.

4.1.3 Scalar Multiplication

Let $E(\mathbb{F}_q)$ be an elliptic curve defined over $\mathbb{F}_q$. Given an integer $n \in \mathbb{N} \setminus \{0\}$ a *scalar multiplication* or *point multiplication* denotes the n-fold addition of a point P to itself. We denote the scalar by $[n]$. The scalar multiplication is then given as

$$[n]P = \underbrace{P + P + \ldots + P}_{n\,\text{times}} = Q. \tag{4.13}$$

By analogy with the *square-and-multiply* algorithm to compute an integer exponentiation more quickly, the *double-and-add* algorithm can be used to calculate the elliptic point scalar multiplication. Algorithm 4.1 shows how the point multiplication with an integer n can be done in at most $\text{ld}(n)$ point additions and point doublings instead of n additions.

Algorithm 4.1 Double-and-add algorithm for computing elliptic curve point multiplication.

Input: P, n with $n = \sum_{i=0}^{\text{ld}(n)} n_i 2^i$
Output: $R = [n]P$
1: $R \leftarrow 0$
2: **for** i from 0 to $\text{ld}(n) - 1$ **do**
3:     **if** $n_i = 1$ **then**
4:         $R \leftarrow R + P$
5:     **end if**
6:     $P \leftarrow P + P$
7: **end for**

4.1.4 Koblitz Curves

Koblitz curves or sometimes referred to as *Anomalous Binary Curves* (ABCs) are a special subset of elliptic curves defined over a binary field. This special subset of curves is applicable to an automorphism, called *Frobenius automorphism.* An automorphism is an isomorphism from an mathematical object or structure to itself. Koblitz curves have only a limited subset of possible parameters. They are defined as follows

$$E_{ABC} : y^2 + xy = x^3 + a_2 x + 1, \qquad \text{with } a_2 = 0 \text{ or } 1. \tag{4.14}$$

For these curves a characteristic polynomial of the Frobenius automorphism exists,

$$\chi(\lambda) = \lambda^2 - \mu\lambda + 2 \equiv 0 \bmod n \tag{4.15}$$

with $\mu = (-1)^{1-a_2}$.

The Frobenius automorphism itself is defined as

$$\Phi(P(x,y)) = P(x^2, y^2) = [\lambda]P(x,y) \tag{4.16}$$

with lambda from Equation 4.15. The automorphism is defined to map the point at infinity $P = \infty$ to itself.

We call the Frobenius automorphism an automorphism of size m as it can be applied up to $m-1$ times resulting in m different points (the original point P included):

$$\begin{array}{rcrcl} \Phi(P) & = & \lambda P & \neq & P \\ \Phi^2(P) & = & [\lambda^2]P & \neq & P \\ \Phi^3(P) & = & [\lambda^3]P & \neq & P \\ & \dots & & & \\ \Phi^{m-1}(P) & = & [\lambda^{m-1}]P & \neq & P \\ \Phi^m(P) & = & [\lambda^m]P & = & P. \end{array} \tag{4.17}$$

It follows from Equation 4.15 that point doublings can be replaced by point additions involving the Frobenius automorphism:

$$2P = \mu\Phi(P) - \Phi^2(P). \tag{4.18}$$

This property results in an efficient scalar multiplication method involving computations with the Frobenius automorphism and makes Koblitz curves of special interest for implementations. The Frobenius automorphism exists for every Koblitz curve. However, Koblitz curves have only the single free parameter a_2, see Equation 4.1.4. Thus for each prime extension degree m there exist only two different curves. We can compute their group order and check of how many subgroups the group order is composed of. For security reasons stated in Section 4.2, only curves with a small cofactor h being 2 or 4 can be used for security sensitive applications. Therefore we cannot generate a Koblitz curve applicable for cryptographic uses for each extension degee m.

4.1.5 Negation Map

The *negation map* is another automorphism. However, the size of the automorphism is only two. It holds for some λ_{neg} that

$$\Psi(P) = -P = \lambda_{\text{neg}}P. \tag{4.19}$$

4.2 The Elliptic Curve Discrete Logarithm Problem

The *Elliptic Curve Discrete Logarithm Problem* (ECDLP) is the foundation of elliptic curve cryptography. The security of all cryptographic protocols based on elliptic curves rely on the intractability of the ECDLP.

Definition 4.3. Let E be an elliptic curve defined over $\mathbb{F}_q$, P a point on E with order n, and Q a multiple of P, denoted $Q \in \langle P \rangle$. The ECDLP is the problem of finding the integer $k \in [0, n-1]$ such that $Q = [k]P$. The integer k is the *discrete logarithm* of Q to the base P, denoted $k = \log_P Q$.

Given the two points P and Q, an attacker has to calculate the integer k. The naive, straight forward approach, the *exhaustive search*, would be to calculate every scalar multiplication $(1P, 2P, 3P...)$ until the result equals Q. The average runtime of this attack is $\mathcal{O}(n)$. So to prevent this kind of attack one has to chose n big enough to make it

computationally infeasible.

However, there exist algorithms to solve the ECDLP faster. The best known algorithms have a running time of $\mathcal{O}(\sqrt{n})$. There exists no proof that no faster algorithm can be found to solve the ECDLP; there is no theoretical proof of the intractability of the ECDLP. In the following subsection we outline some algorithms to solve the ECDLP.

4.2.1 Baby-Step Giant-Step

Shank's *Baby-Step Giant-Step* [72] method is one approach to solve the ECDLP. It is based on a decomposition of k into

$$[k]P = [i\eta + j]P = Q. \tag{4.20}$$

A good choice for $\eta = \lceil\sqrt{n}\rceil$. As $k \in [0, n-1]$ it follows that $i\eta + j < n$, which means that $i, j \in [0, \eta - 1]$. The algorithm tries to find a pair that satisfies

$$[j]P = Q - [i\eta]P. \tag{4.21}$$

The left side of Equation 4.21 correspond to the *baby steps*, the right side to the *giant steps*. At first all possible baby steps have to be calculated and stored sorted into a table. Then all giant steps are calculated until a match is encountered. From this match the discrete Logarithm k can be reconstructed. From the bounds for i and j, it is possible to derive an expected running time of $\mathcal{O}(\sqrt{n})$. However, the algorithm is not applicable to curves with a large group order n because of its storage requirements of $\mathcal{O}(\sqrt{n})$.

4.2.2 Pohlig-Hellman

The *Pohlig-Hellman algorithm* [63] is based on the decomposition of the group order into smaller subgroups. Factorize

$$n = p_1^{e_1} p_2^{e_2} \dots p_z^{e_z} \tag{4.22}$$

with p_i being prime factors and e_i its powers.

The Pohlig-Hellman algorithm solves the system of congruences

$$\begin{aligned} k &\equiv k_1 \bmod p_1^{e_1} \\ k &\equiv k_2 \bmod p_2^{e_2} \\ k &\equiv \dots \\ k &\equiv k_z \bmod p_z^{e_z}. \end{aligned} \tag{4.23}$$

Finding the ECDLP is now as hard as finding the ECDLP in the largest subgroup of prime order and can therefore reduce the hardness of solving the ECDLP if the group order is a composite integer. An obvious countermeasure is to chose the group order to be of prime order or to be divisible by a larger prime.

4.2.3 Isomorphism Attacks

The mathematics behind *Weil and Tate paring* attack and a *Weil descent* attack is quite sophisticated. Basically, these attacks try to reduce the problem of solving an ECDLP to the problem of solving a DLP, for which faster algorithms are known. This can be done

if there exists an isomorphism between the group of elliptic points $\langle P \rangle$ and a group G. Let $P \in \mathbb{F}_q$ have prime order n and G be another group with order n. Thus $\langle P \rangle$ and G have order n and are both cyclic, which means they are isomorphic. So one tries to find an isomorphism

$$\Psi : \langle P \rangle \to G, \tag{4.24}$$

to reduce the ECDLP in $\langle P \rangle$ to an DLP in G. The difficulty lies in finding such an isomorphism.

An attack, the *Araki-Satoh-Semaev-Smart* [70, 71, 75] attack, on so called *prime-field-anomalous curves* exploits the isomorphism between $E(\mathbb{F}_p)$ and the additive cyclic group $\mathbb{F}^+$, which exists if $\#E(\mathbb{F}_p) = p$. In $\mathbb{F}^+$ the DLP can be found using the EEA described in Section 3.3.1.

The *Weil and Tate pairing attacks* [28, 51] are based on the following observation. Let $P \in \mathbb{F}_q$ have prime order n and $\gcd(n, q) = 1$. k is the smallest integer such that

$$q^k \equiv 1 \bmod n. \tag{4.25}$$

There exists a multiplicative group $\mathbb{F}^*_{q^k}$ with a subgroup G of order n. Solving a DLP in these subgroup leads to the solution of the ECDLP in $\mathbb{F}_q$. In order to avoid these attacks one needs to make sure that the base point's order n does not fulfill Equation 4.25 for small values of k. For group orders $n \geq 2^{160}$ it suffices to check for $k \leq 20$ (cf. [33]).

The *Weil descent* attack is properly the mathematically most complex attack. It is applicable to elliptic curves over binary field $\mathbb{F}_{2^m}$. If m is composite, the ECDLP in $E(\mathbb{F}_{2^m})$ can be reduced to a DLP of a curve with a larger genus defined over a subfield $E(\mathbb{F}_{2^l})$ with l being an integer divisor of m.

4.2.4 Pollard's Rho Attack

J. M. Pollard [64] introduced in 1975 a novel factorization method. His algorithm allowed to find a prime factor p in $\sqrt{p}$ steps as opposed to p steps with trial division. This factorization method is based on Floyd's cycle-finding algorithm and the fact, that random collisions are far more likely than a specific collision. Applied to the ECDLP, the algorithm finds two pairs of integers (c_1, d_1) and (c_2, d_2) such that:

$$c_1 P + d_1 Q = c_2 P + d_2 Q. \tag{4.26}$$

So we can compute the discrete logarithm k to the base P of Q as follows:

$$(c_1 - c_2)P = (d_2 - d_1)Q = (d_2 - d_1)kP. \tag{4.27}$$

$$k = (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n. \tag{4.28}$$

According to the birthday paradox we encounter such a collision after approximately $\sqrt{\frac{\pi n}{2}}$ steps. The naive approach is to store the calculated points and the pairs of integers in a table until a point is obtained for the second time. This however requires storage for averagely $\sqrt{\frac{\pi n}{2}}$ points and integers. The beauty of Pollard's rho attack is the negligible storage requirements and roughly the same execution time as the naive approach. The

idea behind this attack is to define an iterating function $f : \langle P \rangle \rightarrow \langle P \rangle$ given $X \in \langle P \rangle$ and $c_1, d_1 \in [0, n-1]$ with $X = c_1P + d_1Q$. This iteration function should easily compute $f(X) = \bar{X}$ and $c_2, d_2 \in [0, n-1]$ with $\bar{X} = c_2P + d_2Q$. f should also have the characteristics of a random function. It is common to partition $\langle P \rangle$ in to L branches $S_1, S_2, ..., S_L$ by assigning a point X to a branch according to its x-coordinate's least significant bits representing an integer j. So this partition function is $H(X) = j$ if $X \in S_j$. Let $a_j, b_j \in [0, n-1]$ for $0 \leq j < L$. Finally we can define our iterating function f as follows:

$$f(X) = X + a_jP + b_jQ \quad \text{where } j = H(X). \tag{4.29}$$

If $X = cP + dQ$, it follows that $f(X) = \bar{X} = \bar{c}P + \bar{d}Q$ where $\bar{c} = c + a_j$ and $\bar{d} = d + b_j$. Any starting point X_0 determines a sequence of points where $X_{i+1} = f(X_i)$. Eventually the sequence will collide and cycle forever, since the set $\langle P \rangle$ is finite. The collision can be found using Floyd's cycle-finding algorithm: Compute pairs (X_i, X_{2i}) until $X_i = X_{2i}$. We only have to store the current and the next pair.

Pollard's Rho attack as described above can easily be parallelized as shown by van Oorschot and Wiener [81]. The idea is to let every instance calculate points according to Equation 4.29. The parallelized Pollard's rho algorithm can be structured into three parts. An initialization part, a server part, and a client part. During the initialization part, M starting points for each client, and L points and integers of the iterating function f for all M clients are calculated. Each client selects one starting point and calculates subsequent points according to Equation 4.29. Note that during the initialization, the sum of a_jP+b_jQ were already calculated, so the client does not have to perform a point multiplication. If a client encounters a distinguished point, it sends the point and the integer pair to a server. A point is called distinguished, if for instance the leading t bits of the point's x-coordinate are zero or its Hamming weights is smaller than t. The server stores the received points in a data structure. Once the server receives a point twice and the integers c and d differ, the server calculates the scalar k. The pseudo code for the initialization, client, and server part is found in Algorithm 4.2, 4.3, and 4.4, respectively. Note that the possibility of failure in Algorithm 4.4 is negligible.

Algorithm 4.2 Initialization part of the parallelized Pollard's rho algorithm for the ECDLP.

Input: $P \in \mathbb{F}_{2^m}$ of prime order n, $Q \in \langle P \rangle$.
Output: Starting points $X_s = c_s + d_s$ and triples $R_j = aP_j + Qb_j$.
1: Select the number L of branches.
2: Specify the number M of clients.
3: Select a partition function H.
4: Select distinguishing property for points in $\langle P \rangle$, e.g. number t of leading zeros.
5: **for all** s from 0 to $M-1$ **do**
6: Select random $c_s, d_s \in [0, n-1]$.
7: $X_s \leftarrow c_sP + d_sQ$
8: **end for**
9: **for all** j from 0 to $L-1$ **do**
10: Select random $a_j, b_j \in [0, n-1]$.
11: $R_j \leftarrow a_jP + b_jQ$
12: **end for**

Algorithm 4.3 Client part of the parallelized Pollard's rho algorithm for the ECDLP.

Input: $R_j, X \in \mathbb{F}_{2^m}$ of prime order n; $a_j, b_j \in [0, n-1]$.
Output: Distinguished triple $X = cP + dQ$.

1: **loop** until finished
2: $j \leftarrow H(X)$
3: $X \leftarrow X + R_j$
4: $c \leftarrow c + a_j \bmod n$
5: $d \leftarrow d + b_j \bmod n$
6: **if** X is distinguished **then**
7: send X, c, d to the server.
8: **end if**
9: **end loop**

Algorithm 4.4 Server part of the parallelized Pollard's rho algorithm for the ECDLP.

Input: Distinguished point $X \in \mathbb{F}_{2^m}$ of prime order n and corresponding integers $c, d \in [0, n-1]$.
Output: The discrete logarithm $k = \log_P Q$.

1: **loop** until finished
2: Receive distinguished triple (X, c_1, d_1).
3: **if** X already in data structure **then**
4: Load triple with point (X, c_2, d_2) from data structure.
5: **if** $d_1 = d_2$ **then**
6: **return** Discard triple
7: **end if**
8: $k = (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n$
9: **return** k
10: **else**
11: store triple (X, c_1, d_1).
12: **end if**
13: **end loop**

Pollard's Rho Algorithm using Automorphisms

The iteration function described in Section 4.2.4 is called Teske's *r-adding walk* [79]. However there exist several iteration functions, which are not only defined on the points in $\langle P\rangle$ but on the *equivalence classes* defined by automorphisms. Given an automorphism $\Upsilon : \langle P\rangle \to \langle P\rangle$ of order w we have equivalence classes denoted by $[R]$. Such an equivalence class is defined as

$$[R] = R, \Upsilon(R), \Upsilon^2(R), ..., \Upsilon^{w-1}(R). \tag{4.30}$$

Let $\kappa \in [0, n-1]$ be an integer such that

$$\Upsilon(P) = \kappa P. \tag{4.31}$$

So if we have $X_i = [c_i]P + [d_i]Q$ we can compute

$$\Upsilon^l(X_i) = \tilde{X}_i = [\tilde{c_i}]P + [\tilde{d_i}]Q \tag{4.32}$$

with $\tilde{c_i} = c_i\kappa^l \bmod n$ and $\tilde{d_i} = d_i\kappa^l \bmod n$.

If the w equivalence classes are of approximately equal size the search space can be reduced from n to n/w. This holds for the Frobenius automorphism, an automorphism of size m and the negation map, an automorphism of size two. So a properly selected iteration function can speed up the attack by a factor of $\sqrt{2m}$ compared to the standard algorithm. In Section 5.3 we compare several iterations functions exploiting the Frobenius automorphism and the negation map.

Chapter 5

Solving the ECDLP

In order to compare different cryptosystems we use the concept of a security level specified in bits. However, this kind of comparison only allows a rough estimate. It only specifies the amount of iterations needed to break such a cryptosystem, but does not consider the expenses of a single iteration. The arising question is, how big should the parameters of an elliptic curve cryptosystem should be in order to avoid practical attacks. Parameters too large waste computational power, time, and space, parameters too small pose a security threat. A trade off between security and performance cannot be avoided. There exist several estimations concerning the size of the parameters for ECC. In order to offer security to date the estimations vary. Lenstra and Verheul [44] specify a minimal key size of 154 bits, the European Network of Excellence in Cryptology II (ECRYPT II) [23] 160 bits, and the National Institute of Standards and Technology (NIST) [9] 224 bits. To estimate the real effort of breaking a cryptosystem based on elliptic curves, Certicom [15] published a list of challenges. This list contains several different parameter settings for different curves.

In this chapter we outline previous attacks, summarize our design decisions, present our hardware design, the **ECC-Breaker**, in detail, and conclude with the results.

5.1 Previous Work

There exist three different types of challenges, which have to be distinguished: Curves over prime fields (ECCp), curves over binary fields (ECC2), and curves over binary field, where a Frobenius automorphism can be used (ECC2K). A challenge is denoted "ECCX-n" with n being the group order. For instance, a 113-bit Koblitz curve is an elliptic curve defined over a binary field defined by an irreducible polynomial with 113-bits. However, the group order of this curve is 112 bits (meaning $2^{111} \leq n < 2^{112}$) (see Section 4.1.2), so it is denoted ECC2K-112. Several challenges are official challenges published by Certicom, namely ECC2-109, ECC2K-108, ECC2-131, ECC2K-130, ECC2K-162, ECC2-193, and ECCp-109.

To attack a large curve, Pollard's rho algorithm is the algorithm of choice. It can be parallelized easily and efficiently and all instances can computee independently from each other. The computational effort of this attack is measured in repeated executions of its iteration function. Recall from Section 4.2.4 that on average the solution is found after $\sqrt{\pi n/2}$ iterations. A negation map speeds up the attack by a factor of $\sqrt{2}$, a Frobenius

automorphism by a factor of $\sqrt{m}$. These mechanisms are described in Section 5.3. All the following attacks we mention use Pollard's rho algorithm.

Until now, the hardest solved Certicom challenges are the 109-bit prime challenge (ECCp-109) and the 109-bit challenge for binary fields (ECC2-109). ECCp-109 was solved by Monico *et al.* [16] using about 10,000 PCs for 549 days in 2002. The ECC2-109 challenge was also also solved by Monico *et al.* [17] using about 2,600 PCs for around 510 days in 2004. Apart from the Certicom challenges, Harley *et al.* [34] solved an ECDLP on a 109-bit Koblitz curve (ECC2K-108). The mentioned attacks were running on general purpose CPU's, sometimes with public participation. ECCp-112 was solved by Bos *et al.* [13] in 2012. Bos *et al.* used PlayStation 3's [76] Cell CPUs. Each PlayStation 3 performs $42 \cdot 10^6$ Iterations Per Second (IPS). Other than the mentioned, there exist several approaches to solve ECDLPs with dedicated hardware. Most notably is the work of Fan *et al.* [25]. The proposed architecture targets Certicom's ECC2K-130 challenge for Koblitz curves and performs $111 \cdot 10^6$ IPS using Spartan-3 FPGAs [90]. An attempt to break ECC2K-130 was also started by Bailey *et al.* [8]. Besides implementations for Nvidia's GTX 295 graphics cards [60], Intel's Core 2 Extreme CPUs [37], they developed also a Spartan-3 architecture with $33.67 \cdot 10^6$ IPS throughput. Dormale *et al.* [53] target ECC2-112, ECC2-131, and ECC2-163 using Spartan-3 FPGAs performing up to $20 \cdot 10^6$ IPS. An architecture for ECCp-131 is contributed by Gueneysu *et al* [32], whose Spartan-3 architecture performs about $173 \cdot 10^3$ IPS. An Virtex-5 [91] architecture for ECC-112 by Judge *et al.* [39] executes $2.87 \cdot 10^6$ IPS, Mane *et al.*'s [49] architecture, which uses the same platform and attacks the same challenge, performs $660 \cdot 10^3$ IPS.

It is fairly difficult to compare different architectures targeting different curves. Nonetheless, Table 5.1 should give an overview about all mentioned architectures. Many of those designs are just proofs of concept and were never really used to solve an ECDLP. Some designs rely on additional hardware (e.g., PCs), which perform some of the arithmetic. The single design that really solved an ECDLP is the design by Bos *et al.* [13]. Possible ways to speed up Pollard's rho attack is the use of the Frobenius automorphism (denoted F in the table) and the negation map (N). The number of iterations takes implemented speedups into account. The throughput is given in IPS.

Table 5.1: Overview of previous attacks.

Ref.	**Curve**	**Speed ups**	**Iterations**	**Hardware**	**Throughput**	**Runtime**
[25]	ECC2K-130	F	$3 \cdot 10^{18}$	Spartan 3	$111 \cdot 10^6$	579 y
[8]	ECC2K-130	F & N	$2 \cdot 10^{18}$	C2E Q6850	$22 \cdot 10^6$	2853 y
[8]	ECC2K-130	F & N	$2 \cdot 10^{18}$	GTX 295	$25 \cdot 10^6$	2550 y
[8]	ECC2K-130	F & N	$2 \cdot 10^{18}$	PS3	$28 \cdot 10^6$	2315 y
[53]	ECC2-112	-	$90 \cdot 10^{15}$	Spartan 3	$20 \cdot 10^6$	143 y
[53]	ECC2-131	-	$46 \cdot 10^{18}$	Spartan 3	$2 \cdot 10^6$	$733 \cdot 10^3$ y
[53]	ECC2-163	-	$3 \cdot 10^{24}$	Spartan 3	$9 \cdot 10^6$	$11 \cdot 10^9$ y
[32]	ECCp-131	-	$49 \cdot 10^{24}$	Spartan 3	$173 \cdot 10^3$	$7 \cdot 10^6$ y
[49]	ECCp-112	-	$90 \cdot 10^{15}$	Virtex 5	$660 \cdot 10^3$	$4 \cdot 10^3$ y
[39]	ECCp-112	-	$90 \cdot 10^{15}$	Virtex 5	$3 \cdot 10^6$	998 y
[13]	ECCp-112	-	$90 \cdot 10^{15}$	PS3	$42 \cdot 10^6$	65 y

5.2 Goals & Strategies

Our goal is to establish a new binary ECDLP record. The largest solved binary ECDLP instance was ECC2-109, an ECDLP on a 109-bit binary curve. There is even an attempt to break ECC2K-130 [8], a 131-bit Koblitz curve. However, although the attack started in 2009, to date no results have been published.

To establish a new record we must beat ECC2-109. For security reasons stated in Section 4.2 the extension degree m should be of prime order. The primes following 109 are 113, 127, and 131. For the 127- and 131-bit challenges one would need to execute $12 \cdot 10^{18}$ and $46 \cdot 10^{18}$ iterations. The 113-bit binary curve is expected to be broken in $90 \cdot 10^{15}$ iterations, thus it is about 128 times easier than the 127-bit challenge. Exploiting the Frobenius automorphism can reduce the number of iterations by a factor of $\sqrt{m}$. Though, as stated in Section 4.1.4, we cannot generate secure Koblitz curves for each prime extension degree m, and unfortunately, for $m = 127$ this is the case. However, as it happens for m being 113 there exists a secure Koblitz curve. If we exploit the Frobenius automorphism, we can solve a 113-bit ECDLP in approximately $8 \cdot 10^{15}$ iterations. Although ECC2K-112 is in terms of computational effort easier than ECC2-127 or ECC2K-130, by breaking ECC2K-112 we provide important estimations regarding the computational complexity of ECC2-127, ECC2K-130, and ECC2-131.

Having chosen to target ECC2K-112, one of the next decisions we had to make, was whether we go for a general-purpose hardware approach (e.g. x86-CPUs) and use a huge amount of clients to solve the ECDLP, as Harley *et al.* [34] or Monico *et al.* [17] did, or if we go for a special-purpose hardware approach. In terms of throughput a special purpose hardware approach is vastly superior to a general-purpose hardware approach, however the amount of work involved in developing a special-purpose hardware is fairly big.

Nevertheless, we decided to develop a novel hardware architecture for a *Field Programmable Gate Array* (FPGA). FPGAs are programmable logic. An FPGA consists of so-called *slices*. A slice itself contains *Look Up Tables* (LUTs) to realize logic and *register* (regs) to store values. Additionally, FPGAs commonly offer *block RAM*, used for storing larger amounts of data, and *DSP*-slices, which can be used to efficiently perform integer arithmetic.

In comparison to many previous designs (see Section 5.1), we decided to use a state-of-the-art FPGA instead of a low-cost FPGA, as many previous works did by choosing the Spartan-3. The ML605 development board [89] with a Virtex-6 XC6VLX240T [89] (henceforth referred to as Virtex-6 only) seems to be the optimal choice. It is commonly available and it provides a fairly large amount of slices and additional DSP-slices. Our task is to perfectly parallelize the iteration function, to completely pipline it, such that in each clock cycle one result is calculated. We want to maximize the performance of a Pollard's rho iteration function on this particular board.

Such an iteration function basically consists of an elliptic curve point addition and some prime field arithmetic to keep track of the scalars of the linear combination $X = [c]P + [d]Q$. A point addition consists of two field multiplications, a field inversion and several field additions and field squarings. We refer to Section 4.1.1 for details. The field additions

are neglectable, the field squarings are also of low complexity, the field inversion itself is based on field squarings and field multiplications. The most complex part of an elliptic curve point addition is the field multiplication. Therefore, focusing development effort on evaluating several field multipliers in terms of size pays off. The results of this evaluation can be found in Section 5.5.5. The next crucial decision is which iteration function to use. The success of an attack relies on a properly selected iteration function. The selection process is described in Section 5.3.

Moreover, it is important to determine how the actual attack should be conducted. Despite the computational power of a Virtex-6 FPGA, an attempt to solve a 113-bit ECDLP requires several of those FPGAs calculating several days. It is crucial that these FPGAs are independent from each other to avoid a single point of failure. The results of the FPGAs have to be stored in a central database. We decided to go for a client-sever model: a sever is used to store the points and clients are devoted to calculate the points. These clients communicate with the FPGAs over a serial interface. They initialize the FPGAs, collect the calculated points and send them to the server. The detailed setup can be found in Section 5.4.

5.3 Selecting the Iteration Function

Prior to the actual hardware design, a proper iteration function has to be selected. This is a decision of great significancy, as the iterations function's performance is a upper bound for the overall performance. Recall from Section 4.2.4, that an iteration function $f : \langle P \rangle \rightarrow \langle P \rangle$ takes a linear combination of P and Q, namely $X = [c]P + [d]Q$ with $c, d \in [0, n-1]$ and computes a point $X' = [c']P + [d']Q$ with $c', d' \in [0, n-1]$. f should be easily computable and should have the characteristics of a random function. A proper iteration function determines the performance and success of Pollard's rho algorithm.

Teske's *r-adding walk* [79] is a close-to-optimum choice for the iteration function. It is based on a partition of the elliptic curve group into r distinct subsets $\{S_1, S_2, ..., S_r\}$ of roughly equal size. Depending on which subset S_j a point X is assigned to, the iteration functions computes $X' = X + [c_j]P + [d_j]Q$ with $c_j, d_j \in [0, n-1]$. For short we write $R_j = [c_j]P + [d_j]Q$. The pseudo code for this iteration function can be found in Section 4.2.4.

A speedup of $\sqrt{2}$ can be achieved by using the negation map, an automorphism of size two. After each iteration either X_i or $-X_i$ is selected depending on which point has a smaller y-coordinate when represented as an integer. The disadvantage of using the negation map is the possibility of trapping the iteration function in a loop. Cycles within this loop are called fruitless cycles. Suppose that X_i, X_{i+1} and X_{i+2} belong to the same set S_j and we select both times the negative point. Then $X_{i+1} = -(X_i + R_j)$ and $X_{i+2} = -(X_{i+1} + R_j) = X_i$. In order to avoid the occurrence of such loops, we have to apply additional techniques, as for instance described by Wiener and Zuccherato [84].

The Frobenius automorphism for Koblitz curves has not only size two, but size m, which allows the attack to be further sped up by a factor of $\sqrt{m}$. Wiener and Zuccherato [84] propose to calculate all points $\Phi^l(X_i + R_j)\ \forall\ l \in [0, m-1]$, and continue with the point which has the smallest x-coordinate when represented as an integer. Another iteration

function is introduced by Gallant *et al.* [29]. It is based on a labeling function $\mathcal{L}$, which maps the equivalence classes defined by the Frobenius automorphism to a set of representatives. The iteration function uses this map and is defined as $X_{i+1} = X_i + \Phi^l(X_i)$, where $l = \text{hash}_m(\mathcal{L}(X_i))$.

Table 5.2 shows an overview about the mentioned iteration functions. Expected and measured iterations are given for ECC2K-40. Measured iterations are given as average value, calculated from 100 iterations of Pollard's rho algorithm. For our experiments l in Gallant *et al*'s method is the Hamming weight calculated in normal basis. For our hardware design, we use Wiener and Zuccherato's [84] iteration functions, as our experiments render it superior.

Note, that the negation map was not considered, because we do not use it in our design for the following reason. Given L branches the probability to encounter a fruitless cycles is given by $1/(2mL)$. Let L be 1024. It follows that we encounter a fruitless cycle with a probability greater than 99% after about 1,000,000 iterations. This is far too likely to neglect. The mentioned technique to avoid fruitless cycles described by Wiener and Zuccherato [84] checks which branches the input point and the resulting point belongs to. If they belong to the same branch j, the result is discarded and the input point is considered to belong to the branch $j+1$ in the next iteration. This is repeated until the branches of the input point and the resulting point differ. This technique reduces the occurrence of fruitless cycles significantly, however does not avoid them completely. This can only be achieved with a loop detection, which requires considerably more control flow logic. Especially in a piplined hardware design, this is very impractical to implement.

To keep track of the integer scalars in Teske's r-adding walk we refer to Algorithm 4.3. When using Wiener and Zuccherato's iteration function the integer update is more complex. In addition to Teske's iteration function all points in the equivalence class of the point resulting from the addition are calculated. From these points, the point which has the smallest x-coordinate when represented as an integer is selected. This has the same effect as calculating all m points $\{X, [\lambda]X, [\lambda^2]X, ..., [\lambda^{m-1}]X\}$ and selecting the smallest. The integer scalar has additionally to be multiplied by λ^l, when selecting $[\lambda^l]X$. Consequently the update functions are given as

$$\begin{aligned} c_{i+1} &= (c_i + a_j) \cdot \lambda^l \bmod n \\ d_{i+1} &= (d_i + b_j) \cdot \lambda^l \bmod n \quad . \end{aligned} \tag{5.1}$$

Instead of performing the prime field multiplication from Equation 5.1, we can use an alternate representation of the scalars c_i and d_i to avoid this multiplication as shown by

Table 5.2: Number of iterations for different iteration functions.

Reference	**Iteration function f**	**Expected iterations**	**Measured iterations**
Teske [79]	$X_{i+1} = X_i + R_j$	$929 \cdot 10^3$	$906 \cdot 10^3$
Wiener and Zuccherato [84]	$X_{i+1} = \min\limits_{0 \le l < m} \left\{\Phi^l(X_i + R_j\right\}$	$145 \cdot 10^3$	$149 \cdot 10^3$
Gallant *et al.* [29]	$X_{i+1} = X_i + \Phi^l(X_i)$	$145 \cdot 10^3$	$205 \cdot 10^3$

Wiener and Zuccherato [84]. We rewrite

$$c_i = \lambda^{v_i} w_i, \tag{5.2}$$

and keep track of v_i and w_i. The update functions for w_i and v_i are given as

$$\begin{aligned} v_{i+1} &= v_i + j \bmod m \\ w_{i+1} &= w_i + \lambda^{-v_i} a_j \bmod n \quad . \end{aligned} \tag{5.3}$$

The use of a precalculated table of all values $\lambda^{-v_i} a_j$ (these are m values per branch, summing up to $2mL$ values for all branches a_j and b_j) reduces the computational effort to two additions, one modulo n (as in Equation 5.1) and one modulo m. The same principle can be applied to the second integer scalar d_i.

5.4 Overall Setup

A simple model to independently distribute the workload over several clients is needed. It should be possible to add new FPGAs to an existing client without affecting the running ones, or to even add a whole new client. We decided to transmit the triples collected by the clients via an *SFTP* channel to our sever. The server receives these triples and adds them to a MySQL [61] database. A schematic overview about this setup is depicted in Figure 5.1.

Depending on the number of leading zeros t of the point's x-coordinate when represented as an integer or its Hamming weight being less than t, we encounter a distinguished point after approximately 2^t iterations. The choice of t is a crucial part. A distinguishing property too small leads to huge storage requirements, a property too high leads a higher time overhead to detect a collision. We chose t to be 30, resulting in about $8 \cdot 10^6$ distinguished triples.

The clients communicate with the FPGAs via the serial interface (RS232). The client and server software is written in *JAVA*. Figure 5.2 shows a schematic class diagram of the software. The software consists of several threads running in parallel. On the client side there is the *Sender* and the *RS232Communicator*. These two threads communicate through a FIFO. The *RS232Communicator* communicates with the FPGA. It opens a serial connection and initializes the FPGA. Once the FPGA is started the thread periodically checks for available triples. These triples are verified for correctness by the JAVA software and then added to the common FIFO. The *Sender* thread idles until a certain amount of triples has been collected. Then these triples are serialized and sent via SFTP to a server. The server itself runs a *Receiver* thread. The thread periodically checks for received triples and then adds these triples to its database.

5.5 FPGA Architecture

In this section, we focus on the hardware architecture of a single FGPA. The main goal was to maximize the throughput per area. The design principle is a completely utilized hardware. This can be achieved with a single, unrolled, and fully piplined iteration function. The FPGA's registers can be used to realize pipline stages. A fully piplined hardware design produces one result each clock cycle. It follows that in each pipline stage one triple

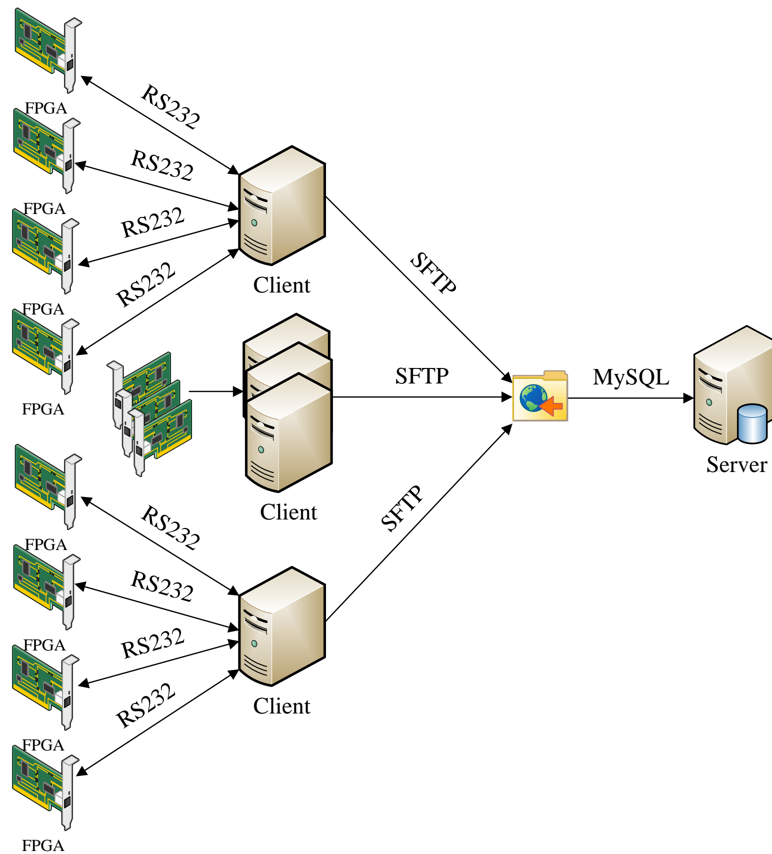

Figure 5.1: Overview

is processed. Thus, the number of pipline stages equals the number of triples processed per instance. There do not exist any idle pipline stages, which means the hardware is really 100% utilized, except for the interface. This approach is preferable to many smaller hardware instances packed into one FPGA, mainly because a high utilization can not be achieved otherwise.

It should be noted that the hardware is carefully optimized for the 113-bit Koblitz curve. However, the underlying architecture and many modules are generic and can be used for attacking other curves as well.

5.5.1 Top Level Architecture

There are two modules, an interface and the `ECC-Breaker` module. The interface module consists of an `RS232 Interface` and an `Interface` for the ECC-Breaker module. The `RS232 interface` converts the data transmitted over the serial port to a 32-bit address and 32-bit data or vice versa. The interface module for the ECC-Breaker is used to fill the pipline stages of the ECC-Breaker module and to read out the distinguished triples from it. It can store a triple used as input (`TripleIn`) and a triple read from the `Distinguished Triple Storage` (`TripleOut`). This is schematically depicted in Figure 5.3 and Figure 5.4,

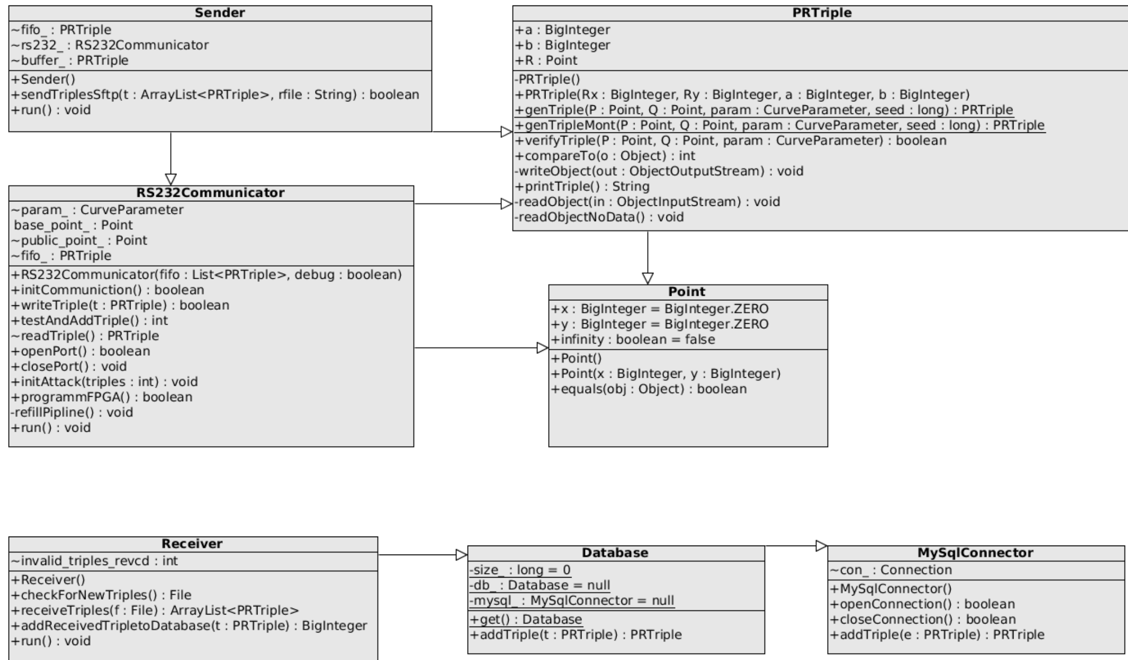

Figure 5.2: Class diagram of JAVA software.

respectively.

We built two different `ECC-Breaker` modules exploiting the Frobenius automorphism. Both realize Wiener and Zuccherato's iteration function described in Section 5.3. However, one calculates the scalars as stated in Equation 5.1, and the other one uses Equation 5.3 and its required table. To distinguish them we denote them `ECC-Breaker V1` and `ECC-Breaker V2`. The reason we built V1 was that our FPGA, the ML605, comes with DSP-slices. These DSP-slices contain several 18×18-bit integer multipliers, which can be clocked with up to about 600 MHz. Using these small multipliers we can built a big one, which consumes hardly any LUTs. In addition, we can demonstrate the efficiency of a prime field multiplication using this device. The reason to incorporate Wiener and Zuccherato's trick was that many FPGA's do not offer enough DSP-slices but still enough other resources to fit the remaining design.

Both versions realize a completely autonomous, circular, self-sufficient iteration function. The pipline is fed using the `TripleIn` register of the interface until each pipline stage is active. Distinguished triples are automatically added to the `Distinguished Triple Storage`, a FIFO containing up to 128 triples. When all pipline stages are initialized the interface can read the distinguished triples from this storage.

ECC-Breaker V1

Figure 5.3 shows `ECC-Breaker V1`. The multiplexer is used to either add a new triple or to continue with the previous result. By default the previous triple is used to feed the pipline.

The `Branching Table` contains 1024 branches, which easily fit into the block RAM of the

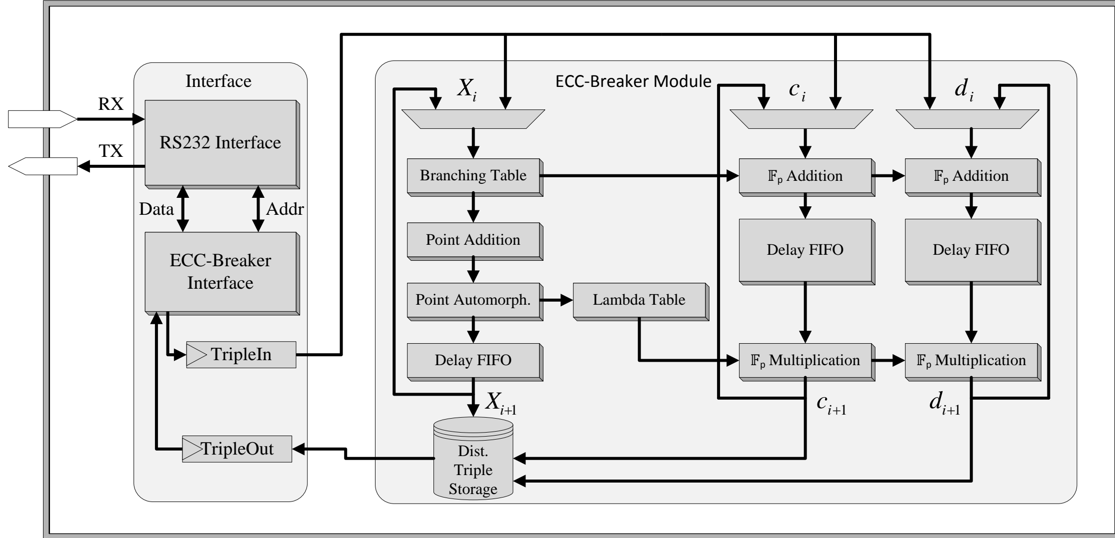

Figure 5.3: Overview of `ECC-Breaker V1`.

ML605. The branch number equals the 10 least significant bits of the point's x-coordinate when represented as an integer. The `Point Addition` module is fed with the point X_i and the output of the `Branching Table`. The result $S_i = X_i + R_j$ is then forwarded to the `Point Automorphism` module, which calculates the smallest $\Phi(S^l)$, with $l \in [0, m-1]$. The integer l is used as input for the `Lambda Table`.

The integers c_i and d_i are added to the integers a_j and b_j, respectively, which are outputs from the `Branching Table`. The $\mathbb{F}_p$ `Multiplication` module then multiplies the sum with λ^l, which comes as output from the `Lambda Table`. The `Lambda Table` containing all the powers of λ is also stored in the block RAM.

The `Delay FIFO`s are needed to synchronize the piplines for the point X_i and the integers c_i and d_i. We use the Montgomery domain to represent prime field elements. Hence, the $\mathbb{F}_p$ `Multiplication` module performs a Montgomery multiplication. The PC connected to the FPGA via a serial port transforms the integers from the Montgomery domain back to the normal domain.

We want to remark, that curves without Frobenius automorphism can also be attacked using `ECC-Breaker V1` by discarding the `Point Automorphism` and the $\mathbb{F}_p$ `Multipication` module. We denote this configuration `ECC-Breaker Plain`.

ECC-Breaker V2

`ECC-Breaker V2` is depcited in Figure 5.4. It is built from the same modules as V1. However, this version needs to keep track of four integers vc_i, vd_i, wc_i, wd_i. The `LambdaMul Table` contains the table of all precalculated $\lambda^l a_j$ and $\lambda^l b_j$ for all $l \in [0, m-1]$ and all branches a_j and b_j. The block RAM offers enough space to use 64 different branches. Still sufficient according to Teske [79], who showed that 20 branches are enough so that the iteration function acts as a pseudo random function.

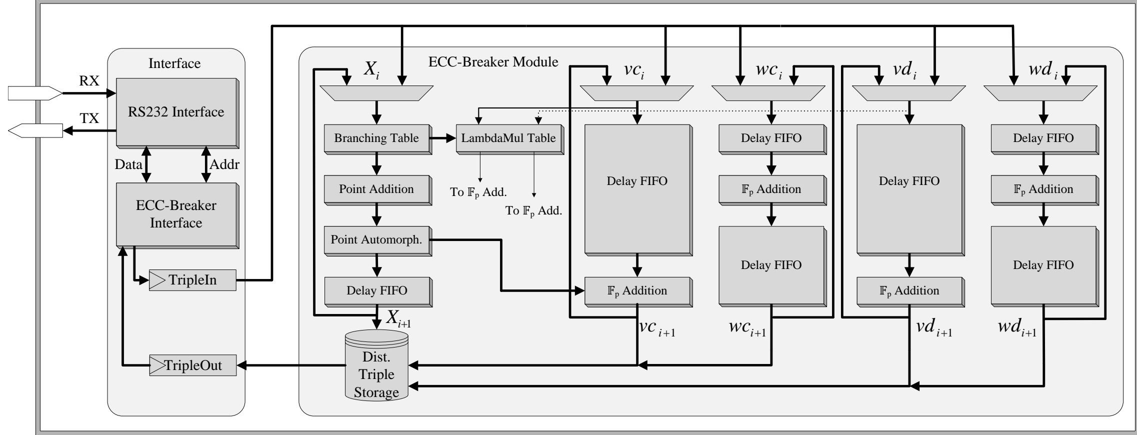

Figure 5.4: Overview of `ECC-Breaker V2`.

The `Distinguished Triple Storage` contains the distinguished point X and the integers vc and wc as well as vd and wd. The integers $c = \lambda^{vc}wc$ and $d = \lambda^{vd}wd$ satisfying $X = [c]P+[d]Q$ are not calculated by this hardware design. The client, to which the FPGA is connected to, performs the calculation. As this only needs to be done for distinguished points, the computational effort is negligible even for low cost CPUs.

5.5.2 Point Addition Module

The heart of `ECC-Breaker` is the `Point Addition` module. It performs the main task of our iteration function. The module takes 5 inputs, the summands' coordinates $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, as well as the elliptic curve constant a. The module is generic and can be used for all Koblitz curves. As the elliptic curve constant b is not a part of this point addition formula, the module is not applicable for generic elliptic curves based on binary fields. This module performs the point addition and produces the sum's coordinates $R = (x_3, y_3)$. It does not handle special cases of points being equivalent or inverse to each other, or the point at infinity. However, the likelihood of these occurrences is negligible.

The module is depicted schematically in Figure 5.5. It implements Equation 4.7 stated in Section 4.1.1. The symbols represent $\mathbb{F}_{2^m}$ addition, multiplication, inversion, and squaring, respectively. The figure does not contain FIFO modules which are required as buffers to keep all pipline stages active. The module takes 184 cycles to compute the result. The $\mathbb{F}_{2^m}$ inverter consumes most of these cycles and is in terms of area the module's largest building block, as it takes 83% of the overall slices needed by this module. The $\mathbb{F}_{2^m}$ inverter itself consists of eight $\mathbb{F}_{2^m}$ multipliers and 112 $\mathbb{F}_{2^m}$ squarers. Within the inverter, the eight $\mathbb{F}_{2^m}$ multipliers need the largest share of slices, namely 69%. The $\mathbb{F}_{2^m}$ adders and the dedicated $\mathbb{F}_{2^m}$ squarer hardly contribute to the size of the module and delay of the point addition.

It is not possible to get rid of the costly inversion (8 field multipliers and 112 field squarers) during the point addition even though there exist point addition formulas using projective coordinates to avoid the costly inversion. However Pollard's rho attack is only applicable

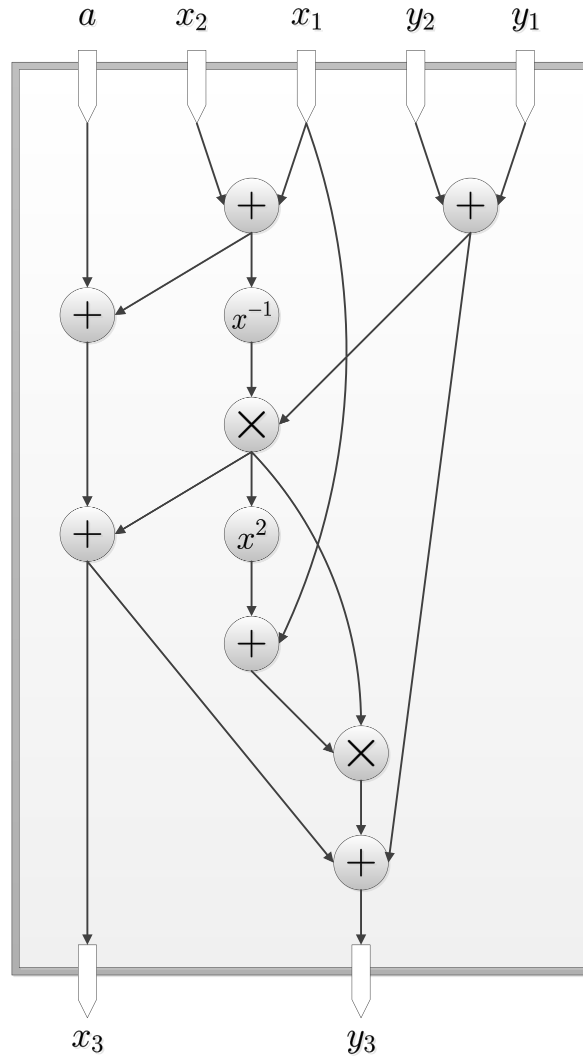

Figure 5.5: Simplified architecture of the point addition module.

when using affine coordinates (cf. [53]).

5.5.3 Point Automorphism Module

The point automorphism module contributes for a $\sqrt{m}$-fold speedup compared to an attack which does not exploit the Frobenius automorphism. The speedup is achieved by uniquely mapping all of the m points of each equivalence class to a certain point in this equivalence class. It is common to use the point with the smallest x-coordinate when represented as an integer. However, every unique mapping is valid. We denote the single point of an equivalence class that all other points are mapped to as *automorphed point*.

A module, which does m squarings and m comparisons as efficiently as possible is required to implement this automorphism. We built two of those modules: one does all the calculations in polynomial basis, the other one performs a basis transformation to normal basis and calculates the automorphism in normal basis. To distinguish them we call them

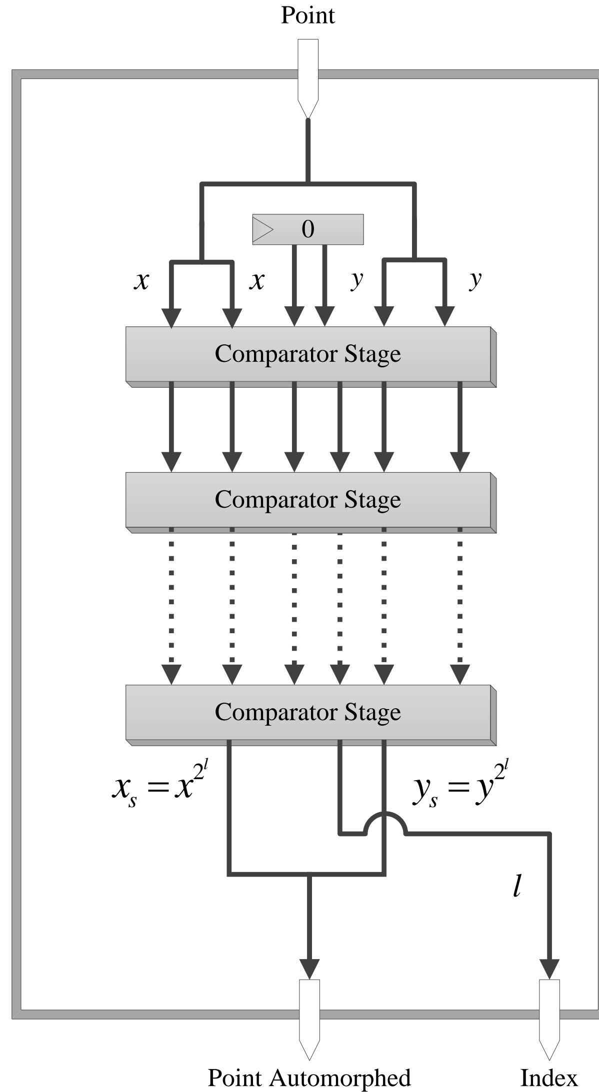

Figure 5.6: Overview of `Point Automorphism PB` module.

`Point Automorphism PB` and `Point Automorphism NB`, respectively.

For optimization, both modules only compare the $t = 70$ most significant bits. The probability that this limitation results in a wrong result of the comparison is negligible. If we perform $mi = m\sqrt{\frac{\pi n}{2m}}$ comparisons using this limitation, the probability for even once selecting the greater value is only $1 - (1 - 2^{-t})^{mi} = 0.00081$.

Point Automorphism PB

Figure 5.6 shows an overview of the `Point Automrphism PB` module. It consists of m `Comparator Stage`s, one of which is depicted in Figure 5.7. The first stage is fed twice with the input point's x and y coordinate The x- and y-coordinate of one of the two input coordinates is squared, resulting in x^2 and y^2. The squares are immediately forwarded to the output. Depending whether x or x^2 is smaller, x or x^2 is forwarded as x_s. The

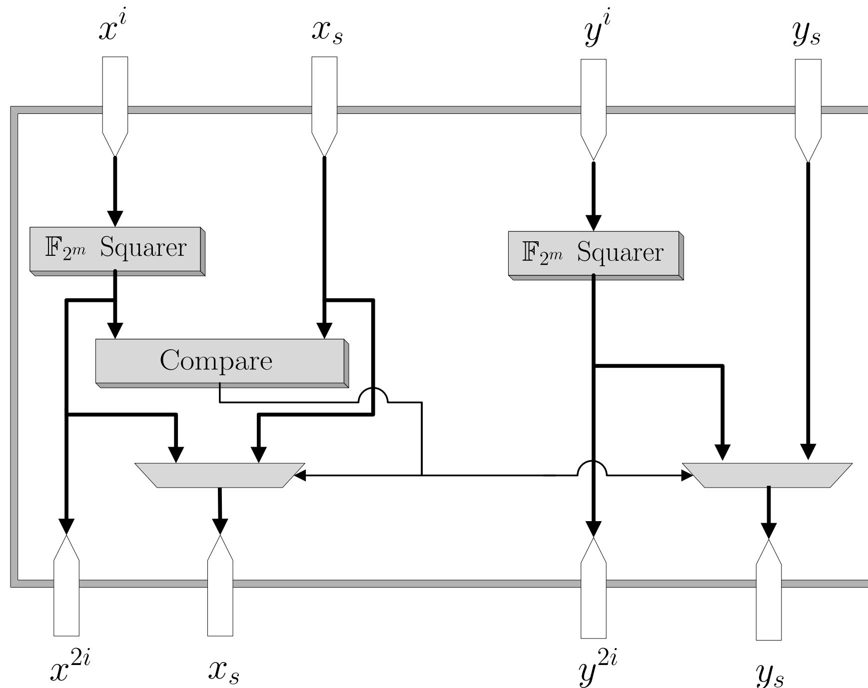

Figure 5.7: Schematic view of an `Comparator Stage`.

result of this comparison also determines whether y or y^2 becomes y_s. This is repeated m times with the previous results as inputs until in the last iterations the coordinates of the smallest point (henceforth referred to as *automorphed point*) is found. This module also computes an index l which reveals how often the initial values x and y have to be squared to get the automorphed point $S = (x^{2^l}, y^{2^l})$. This index is needed for the integer update function, more precisely for the $\mathbb{F}_p$ multiplication.

Point Automorphism NB

The second automorphism module, namely the `Point Automorphism NB` module, relies on a basis transformation. An overview of this module is shown in Figure 5.8. It starts by transforming the point's coordinates into the normal basis. This is done by multiplying each coordinate with a transformation matrix C, with all elements $c_{i,j} \in \mathbb{F}_2$. The matrix is generated as shown in Section 3.3.6. The matrix multiplication is realized as a network of XOR operations. The number of XOR operations is given by the number of nonzero elements in this matrix. The m squares of x in normal basis can be computed in a single cycle, as each square is only a cyclic shift by one of the previous results. The `Comparator Tree` computes the smallest of all these squares and the index l, denoting the number of squaring operations needed to get the smallest value. The `Barrel Rotate` module is an efficient module to perform variable cyclic shifts needed to compute y^{2^l}. It is implemented as a sequence of multiplexers. A barrel rotator performing cyclic shifts of up to m bit needs $\mathrm{ld}(m)$ layers of multiplexers per bit. The first sequence of these multiplexers realizes a cyclic shift by one, the second a cyclic shift by two, the third a cyclic shift by four, and so on. The results of the `Point Automorphism NB` module are generated by retransforming x^{2^l} and y^{2^l} to polynomial basis.

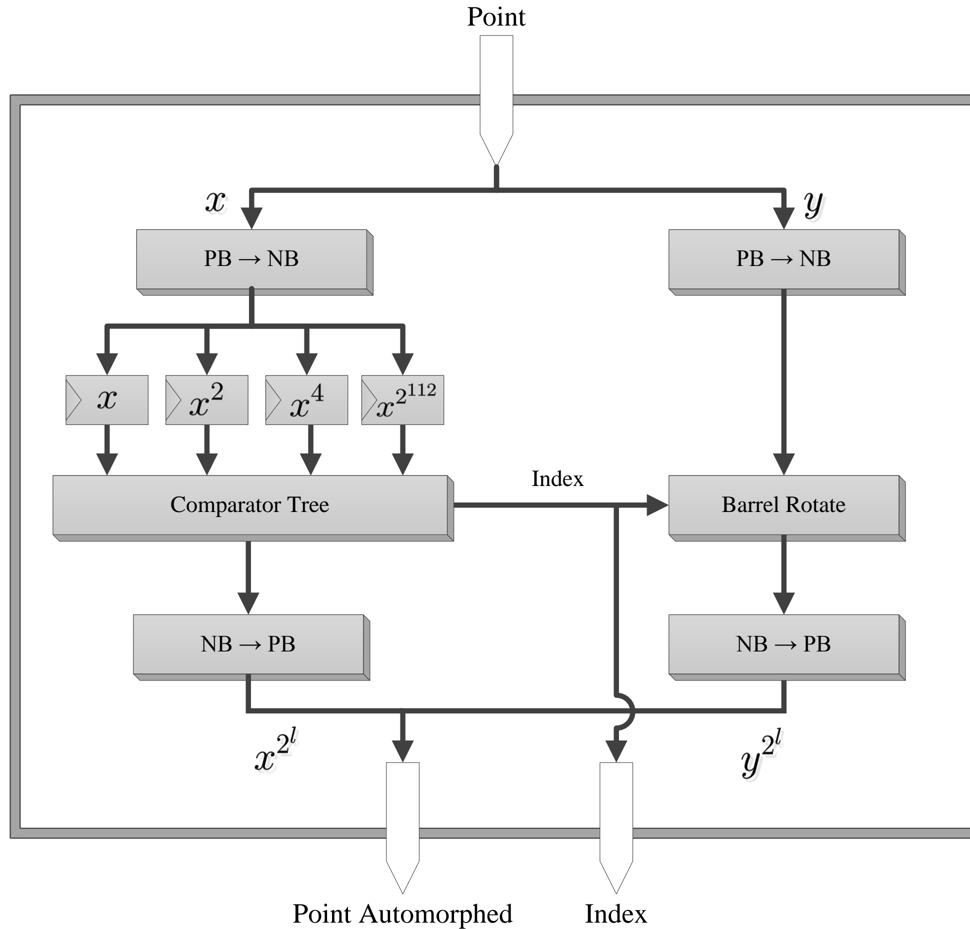

Figure 5.8: Overview of the automorphism module using basis transformations.

To get a picture of how the `Comparator Tree` is implemented, Figure 5.9 shows such a tree for four inputs. This tree consists of three building blocks, each of these blocks has four inputs: two squared x-coordinates and two integers denoting the squares' powers. A comparator compares the squares and forwards the smaller square and the power belonging to it.

The `Barrel Rotate` module replaces the m $\mathbb{F}_{2^m}$ `Squarer` needed to compute y^{2^l} in polynomial basis. Therefore we do not need any $\mathbb{F}_{2^m}$ `Squarer`. A further advantage is the significantly reduced number of pipline stages using the `Comparator Tree` compared to the pipline of m `Comparator Stages`.

Not unexpectedly, the `Point Automorphism NB` module is considerably more efficient in terms of size. Compared to the `Point Automorphism PB` module it needs 3.4 times less registers and 2.5 times less LUTs. The biggest part of the `Point Automorphism NB` module is the `Comparator Tree`, which makes up for about 75 % of the module's slices. Compared to that, the basis transformation is fairly cheap.

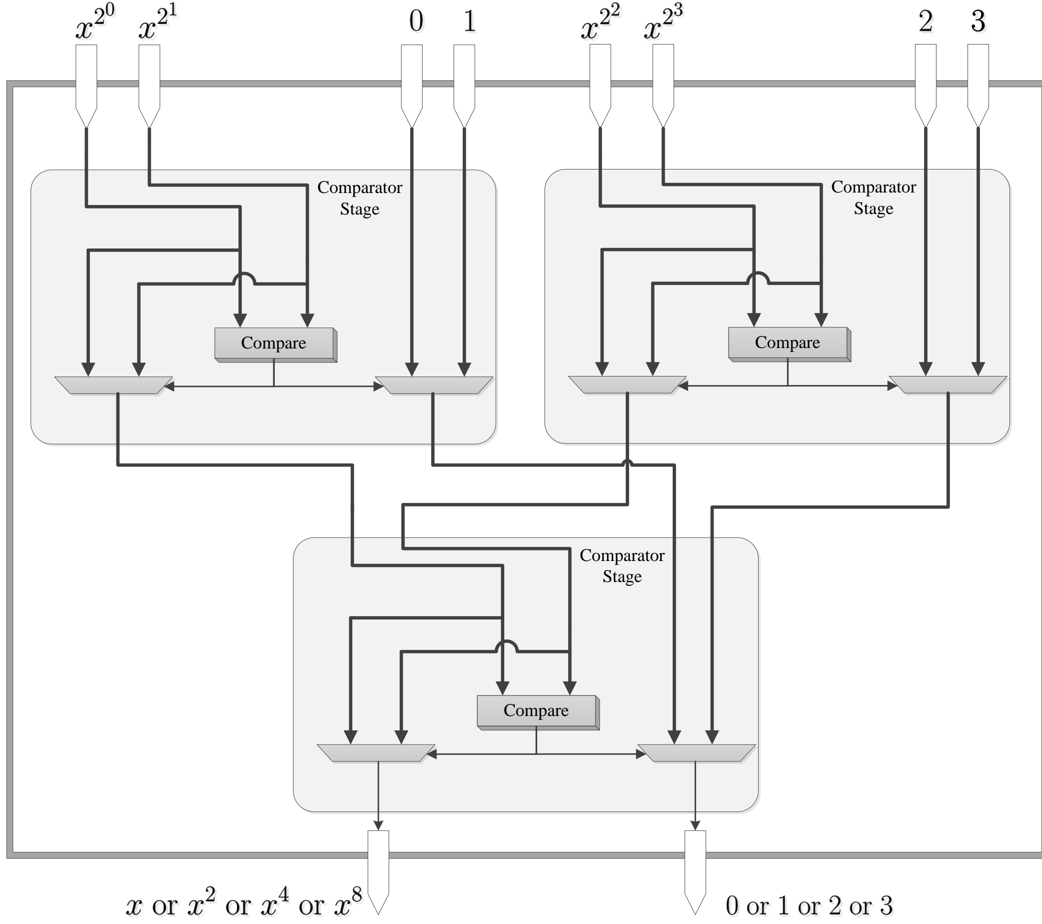

Figure 5.9: Comparator tree for 4 inputs.

5.5.4 $\mathbb{F}_{2^m}$ Inversion Module

The data path of inversion modules based on EEA (see Algorithm 3.2) is not deterministic, thus it is hard to implement in a pipelined hardware. Therefore, the inversion is computed using Fermat's little theorem, described in Section 3.4. Itoh and Tsuji's [38] exponentiation trick, see also Section 3.4, reduces the number of needed multiplications to eight the number of squarings to 112, for $m = 113$.

$$\begin{array}{rcll}
a^{2^{113}-2} & = & (a^{2^{112}-1})^2 & \\
a^{2^{112}-1} & = & (a^{2^{56}-1})^{2^{56}} & \cdot \; a^{2^{56}-1} \\
a^{2^{56}-1} & = & (a^{2^{28}-1})^{2^{28}} & \cdot \; a^{2^{28}-1} \\
a^{2^{28}-1} & = & (a^{2^{14}-1})^{2^{14}} & \cdot \; a^{2^{14}-1} \\
a^{2^{14}-1} & = & (a^{2^{7}-1})^{2^{7}} & \cdot \; a^{2^{7}-1} \\
a^{2^{7}-1} & = & (a^{2^{6}-1})^{2} & \cdot \; a \\
a^{2^{6}-1} & = & (a^{2^{3}-1})^{2^{3}} & \cdot \; a^{2^{3}-1} \\
a^{2^{3}-1} & = & (a^{2^{2}-1})^{2} & \cdot \; a \\
a^{2^{2}-1} & = & a^2 & \cdot \; a
\end{array} \tag{5.4}$$

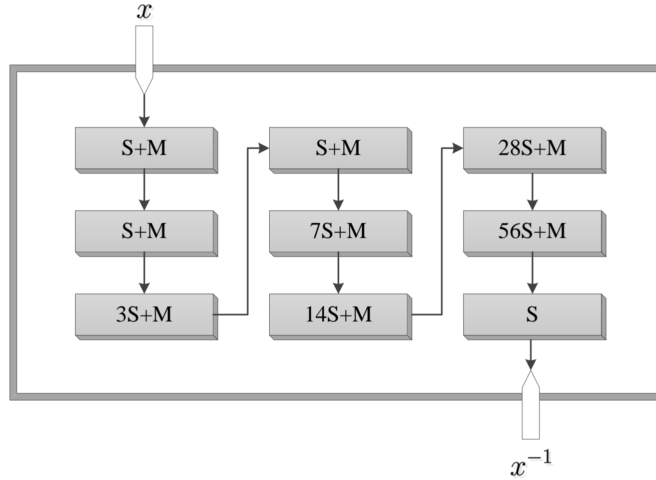

Figure 5.10: Schematic view of the inversion module.

Equation 5.4 shows how this exponentiation trick can be used to calculate the inverse in $\mathbb{F}_{2^m}$ with $m = 113$. In Figure 5.10 the module implementing Equation 5.4 to compute the inverse is depicted.

5.5.5 $\mathbb{F}_{2^m}$ Multiplication Module

The area footprint of our ECC-Breaker design is majorly influenced by the $\mathbb{F}_{2^m}$ multipliers. To reduce the impact, we evaluated several polynomial basis $\mathbb{F}_{2^m}$ multipliers on our target FPGA in terms of size. The results (post-synthesis) of our evaluation can be found in Table 5.3. LUTs are given as absolute values and in percent relative to the available numbers of LUTs on a Virtex-6 FPGA.

A 113-bit digit parallel multiplier is the most straight-forward design. Each bit of the second factor is multiplied (logical ADD) with all the bits of the first multiplicand, shifted according to its position, and then added (logical XOR) to the result.

A Mastrovito multiplier [50] represents the multiplication as a matrix-vector product. The reduction step is performed implicitly, as the matrix is generated depending on the irreducible polynomial. Unfortunately, this approach does not seem to be applicable for our purposes, as this multiplier achieved the worst results.

Table 5.3: Comparison of different polynomial basis $\mathbb{F}_{2^m}$ multipliers including reduction on a Virtex-6 FPGA.

Multiplier	**LUTs**	
Bit Parallel	5,505	3.7%
Mastrovito [50]	8,774	5.8%
BBE [11]	4,409	2.9%
Binary Karatsuba [69]	3,757	2.5%

Bernstein's Batch Binary Edwards (BBE) multiplier [11] needs the smallest amount of bit operations for a 113-bit multiplication. It is based on Toom and Karatsuba recursions. The code containing all bit operations is available at [12].

The multiplier we finally used in our hardware is a slightly modified version of the binary Karatsuba algorithm described by Rodrıguez-Henrıquez and Koç [69]. It is based on the Karatsuba multiplier, which originally was found by Karatsuba and Ofman [41]. Their algorithm allows to reduce the product of two large 2^k-bit polynomials to the sum of three smaller $2^{k/2}$-bit multiplications. This step can be applied recursively and reduce the number of single-digit multiplications to at most $3k^{\log_2 3}$ on overall. In practice it is usually more efficient to truncate the recursion at some point and compute the remaining multiplications using an other technique. However, the Karatsuba algorithm wastes several arithmetic operations when multiplying polynomials with arbitrary length $m = 2^k + d$, because then it is necessary to consider the polynomials as 2^{k+1}-bit polynomials. Rodrıguez-Henrıquez and Koç [69] introduced an algorithm, which prevents these unnecessary arithmetic operations. They pointed out that depending on m, it is more efficient to use a classical bit parallel multiplication for some partial results.

Algorithm 5.1 shows how we applied Rodrıguez-Henrıquez and Koç's [69] idea to perform an $m = 113$-bit multiplication. The following reduction step is not depicted. See Section 5.5.6 for this step. KS64 and KS32 denote 64-bit and 32-bit, respectively, binary Karatsuba multipliers. The Karatsuba multipliers are truncated at 16 bits and the remaining bits are calculated using a classical bit parallel multiplication.

Algorithm 5.1 Calculate $c = a \times b$, with a, b being m-bit binary polynomials.

Input: a, b

Output: $c = a \times b$

1: $m_{ab1} \leftarrow \text{KS64}(a[112..64] \oplus a[63..0],\ b[112..64] \oplus b[63..0])$
2: $c_{l1} \leftarrow \text{KS64}(a[63..0],\ b[63..0])$
3: $c_{l2} \leftarrow \text{KS32}(a[95..64],\ b[95..64])$
4: $c_{l3} \leftarrow a[111..96] \times b[111..96]$
5: $m_{ab2} \leftarrow \text{KS32}(a[95..64] \oplus a[111..96],\ b[95..64] \oplus b[111..96])$

6: $m_{a3} \leftarrow b[112] \times a[111..96]$
7: $m_{b3} \leftarrow a[112] \times b[111..96]$
8: $m_3 \leftarrow m_{a3} \oplus m_{b3}$
9: $c_3[32] \leftarrow a[112] \times b[112]$
10: $c_3[30..0] \leftarrow c_{l3}$
11: $c_3[31..16] \leftarrow c_3[31..16] \oplus m_3$

12: $m_2 \leftarrow m_{ab2} \oplus c_{l2} \oplus c_3$
13: $c_2[62..0] \leftarrow c_{l2}$
14: $c_2[97..64] \leftarrow c_3$
15: $c_2[94..32] \leftarrow c_2[94..32] \oplus m_2$

16: $m_1 \leftarrow m_{ab1} \oplus c_{l1} \oplus c_2$
17: $c[126..0] \leftarrow c_{l1}$
18: $c[225..128] \leftarrow c_2$
19: $c[190..64] \leftarrow c[190..64] \oplus m_1$

5.5.6 $\mathbb{F}_{2^m}$ Squaring (Reduction) Module

Squaring in normal basis is free. However, as we use the polynomial basis in the point addition module there is the need to compute several squarings in polynomial basis. The squaring module mainly consists of the reduction modulo the irreducible polynomial. Fortunately, it is not necessary to implement a generic reduction algorithm, as the irreducible polynomial as a fixed constant. The costs of the reduction depend on the number of the reduction polynomial's nonzero coefficients. Let u be the number of nonzero coefficients, then the reduction step consists of $2(u-1)$ m-bit additions.

Figure 5.11 shows schematically how to reduce a $2m = 226$-bit value using the reduction polynomial $f(\alpha) = \alpha^{113} + \alpha^5 + \alpha^3 + \alpha^2 + 1$. Note that the offsets in Figure 5.11 correspond to the nonzero elements of the irreducible polynomial $f(\alpha)$.

5.5.7 $\mathbb{F}_p$ Addition Module

The $\mathbb{F}_p$ `Addition` module is needed for updating the scalars. The addition is performed modulo the group order n. The module exploits the fact that both inputs are smaller than n. Thus their sum is smaller than $2n$. Either the sum is already smaller than n (therefore reduced modulo n) or it is sufficient to subtract the modulus n once in order to reduce the sum. Figure 5.12 schematically shows the module. It consists of two $\lceil \text{ld}(n) \rceil$-bit integer adders. These integer adders are realized as a chain of smaller adders to reduce the carry

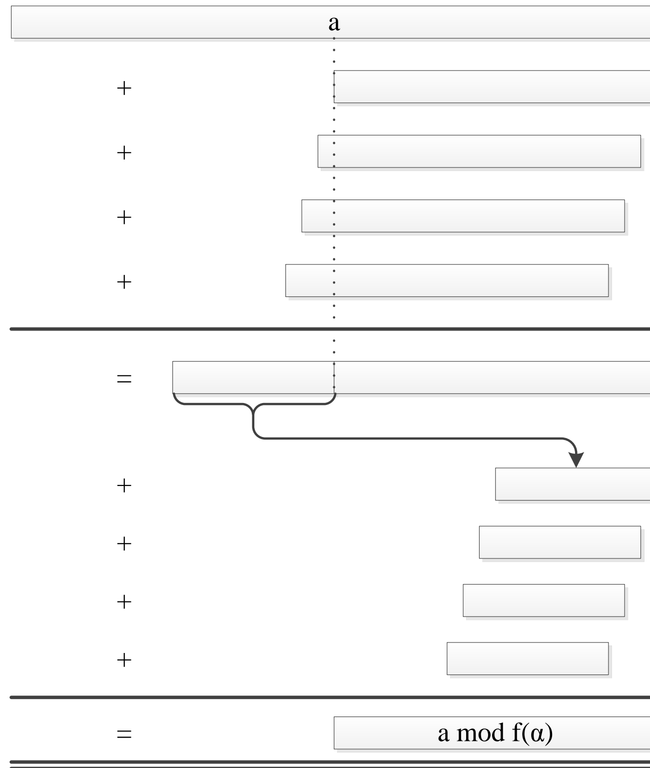

Figure 5.11: Schematic view of the reduction step.

propagation's length.

5.5.8 $\mathbb{F}_p$ Multiplication Module

`ECC-Breaker V1` also contains two $\mathbb{F}_p$ `Multiplication` modules. The modules conduct a Montgomery multiplication as described in Section 3.5. Each module consists of two full-sized and a half-sized $\lceil \mathrm{ld}(n) \rceil$-bit integer multipliers, which results are summed up. The partial products are calculated using the dedicated FPGA DSP-slices, which come with several 18×18-bit MAC units. The module carefully aligns the partial products to fully exploit the DSP-slices' potential.

5.6 Results

This section discusses the implementations. At first numbers are provided for our hardware design, the ECC-Breaker, followed with a generalization of ECC-Breaker to different FPGAs and targeting other curves, and concluded with the results of our conducted attempt so solve a 113-bit binary ECDLP. The results are generated using Xilinx ISE 14.4 [87].

5.6.1 ECC-Breaker

ECC-Breaker is the outcome of an collaborative development which started in March 2013. In a first approach a, compared to the ECC-Breaker, relatively small hardware design (a

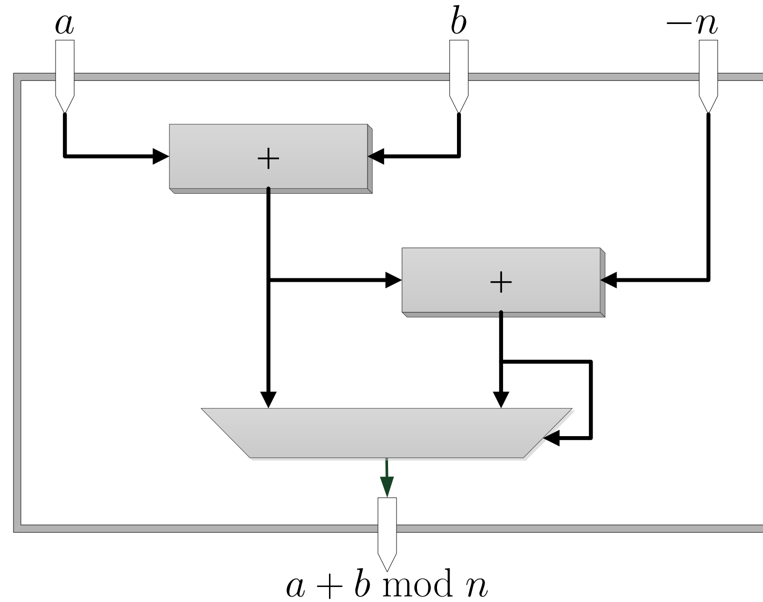

Figure 5.12: Prime field addition module.

core) was developed. Each core is capable of autonomously executing an iteration function and generating distinguished triples. About 30 instances of this core fit on a Virtex-6 FPGA. ECC2-83 was successfully solved within four days on average using a single Virtex-6 FPGA with 30 instances. Thereby, a major problem using this approach emerged, a very low hardware utilization. Altogether these 30 instances clocked at 60 MHz have only a throughput of about $10 \cdot 10^6$ IPS. This initial design helped us to identify the crucial parts and bottlenecks of a Pollard's rho implementation, and pointed out significant optimization potential.

The development of ECC-Breaker itself took about one person year. Starting from autumn 2013, we iteratively and continuously optimized the ECC-Breaker in terms of speed, area, and power optimization, never loosing sight of our main goal, namely maximizing the throughput per area. To optimize the performance on the target platform, the Virtex-6 FPGA, DSP-slices and block RAM are used whenever possible. Table 5.4 shows the post place-and-route device utilization of all ECC-Breaker versions on a Virtex-6. As the

Table 5.4: Post place-and-route device utilization of `ECC-Breaker V1`, `ECC-Breaker V2`, and `ECC-Breaker Plain` on a Virtex 6 XC6VLX240T FPGA.

XC6VLX240T		`ECC-Breaker V1`		`ECC-Breaker V2`		`ECC-Breaker Plain`	
Property	**avail**	**used**	**%**	**used**	**%**	**used**	**%**
Regs	301,404	80,902	26	68,873	22	53,291	17
LUTs	150,720	82,344	54	70,628	46	52,463	34
Slices	37,680	22,734	60	21,647	57	16,696	44
DSP-Slices	768	290	37	0	0	0	0
Block RAM	416	49	12	143	34	43	10

`Point Automorphism NB` module proved superior to the `Point Automorphism PB` module all data is generated using this module.

The versions with `Point Automorphism` module are only 30% respectively 36% large than `ECC-Breaker Plain`, but are about 11 times more efficient. The results further render `ECC-Breaker V2` advantageous to `ECC-Breaker V1`, only at the expense of block RAM. `ECC-Breaker V1` uses 1024 branches that consume 12% of the available block RAMs. `ECC-Breaker V2` has only 64 branches, which use 34% of the available block RAM resources. The absolute difference of 20% between `ECC-Breaker V1` and `ECC-Breaker Plain` originates from the `Lambda Table`. Further, the table shows that `ECC-Breaker V2` consumes slightly fewer slices than `ECC-Breaker V1`. This is because, although the $\mathbb{F}_p$ multipliers are carefully designed to utilize the DSP-Slices, they still need some LUTs.

We also practically evaluated some designs on a Spartan-6 LX150T development kit [6] from Avnet. Table 5.5 shows the results for `ECC-Breaker V2` and `ECC-Breaker Plain` on this FPGA. `ECC-Breaker V1` does not properly fit on this FPGA, and therefore no informative results can be given. The design gets mapped successfully, however the router is stretched to its limits, as the design only achieves unacceptable timings. This is due to the $\mathbb{F}_p$ multipliers, which exceed the FPGA's resources of DSP-slices and therefore are mapped into LUTs. `ECC-Breaker V2` needs only 24% more slices than `ECC-Breaker Plain` by providing an $\sqrt{m}$-fold speedup. The block RAM resources though, are almost completely utilized by 64 branches, mostly by the `LambdaMul Table`.

In Table 5.6 the post place-and-route results are summarized. Prices for development boards are taken from www.avnet.com [1] and do not contain taxes. The throughput is given per FPGA. `ECC-Breaker Plain` is capable of performing $200 \cdot 10^6$ IPS per instance, summing up to $400 \cdot 10^6$ IPS per Virtex-6 FPGA. All designs have successfully been validated on real hardware. The table shows that `ECC-Breaker V2` is about 21% faster than `ECC-Breaker V1`. This is due to the reduced routing effort, caused by the removal of the $\mathbb{F}_p$ multipliers. V2 trades a 113-bit prime field multiplication with a 7-bit prime field addition. The additional control flow logic does not have significant impact. On the Spartan-6 both designs achieve the same throughput, which further emphasizes the usage of the point automorphism module. Considering IPS per $ the Spartan-6 is even slightly superior to the Virtex 6 for the `ECC-Breaker V2` design. However, when using the `ECC-Breaker Plain` the Virtex 6 is about 50% more cost-effective than the Spartan 6. This is mainly due to the device utilization. The higher the device utilization, the more

Table 5.5: Post place-and-route device utilization of `ECC-Breaker V2` on a Spartan-6 XC6SLX150T FPGA.

XC6SLX150T		`ECC-Breaker V2`		`ECC-Breaker Plain`	
Property	**avail**	**used**	**%**	**used**	**%**
Regs	184,304	69,152	37	53,520	29
LUTs	91,152	67,315	73	48,885	53
Slices	23,038	20,563	89	16,664	72
DSP-Slices	180	0	0	0	0
Block RAM	268	246	92	56	21

cost-effective is the design. But keep in mind, financial expenses hardly ever scale linearly with throughput.

In order to estimate the costs of the modules in detail, Table 5.7 shows the device utilization summary. The table shows post-map results. They are not exact but should suffice to give an overview. Place-and-route values cannot be used, because the router performs optimization across all hierarchies, therefore the data is not available anymore after routing. The results are given for `ECC-Breaker V1` on a Virtex-6 FPGA only, because the proportionalities stay about the same for all FPGAs and ECC-Breaker versions. In addition it shows the impact of the $\mathbb{F}_p$ `Multiplication` modules to the overall system. Slices are given as percentage of the overall system. Registers, LUTs, DSP-slices, and block RAMs are absolute values. Registers and LUTs which are missing when adding up submodules are used for the control path logic in the parent modules. The table further reveals that the $\mathbb{F}_{2^m}$ inversion is with 64% of all slices the most slice consuming building block. Within the $\mathbb{F}_{2^m}$ inversion module the $\mathbb{F}_{2^m}$ multipliers have the major impact on size. These multiplier together with the two $\mathbb{F}_{2^m}$ multipliers used during point addition need about 55% of all slices. This emphasizes the importance of carefully choosing the $\mathbb{F}_{2^m}$ multiplier. The size of the `Point Automorphism` module is mainly influenced by the `Comparator Tree`. Compared to that the basis transformation is almost negligible. 32 of 59 block RAMs are used for `Delay FIFOs`, only 7 for the `Distinigushed Triple Storage`, and 13 for the `Branching Table`.

In summary, `ECC-Breaker V2` is the most efficient design in terms of IPS per slice and IPS per $.

5.6.2 ECC-Breaker on different FPGAs

In addition to the results presented previously, the hardware design `ECC-Breaker V2` was also evaluated on several different currently available state-of-the-art FPGAs. Although we optimized our design for a Virtex-6, the resulting VHDL-code is portable. In order to make a comparison as fair as possible the estimated maximal frequency post-synthesis is compared, because post place-and-route values are majorly influenced by configuration settings of the mapper and router and depend on the FPGA series. Slices are given as post place-route values. The optimizations done during routing can have a big impact on the slice utilization.

Xilinx latest FPGAs are divided into three series, one targeting low cost applications

Table 5.6: Post place-and-route Results of several different ECC-Breaker versions.

Design	**FPGA Series**	**Part Number**	**Price** [$]	**Slices** [%]	**Instances**	**Throughput** [10^6 **IPS**]
`V1`	Virtex 6	XC6VLX240T	2,495	60	1	165
`V2`	Virtex 6	XC6VLX240T	2,495	57	1	200
`V2`	Spartan 6	XC6SLX150T	995	89	1	100
`Plain`	Virtex 6	XC6VLX240T	2,495	79	2	400
`Plain`	Spartan 6	XC6SLX150T	995	72	1	100

Table 5.7: Device utilization summary for `ECC-Breaker V1` on a Virtex-6.

Entity	Instances	Slices	Registers	LUTs	BRAM	DSP
`Interface`	1	100%	80,899	82,344	52	290
`ECC-Breaker`	1	100%	80,730	82,127	52	290
`Branches`	1	0%	0	0	13	0
`Point Addition`	1	77%	57,195	80,611	20	0
$\mathbb{F}_{2^m}$ `Multiplication`	2	11%	7,926	8343	0	0
$\mathbb{F}_{2^m}$ `Inverse`	1	64%	47,095	46,023	0	0
$\mathbb{F}_{2^m}$ `Multiplication`	8	44%	31,704	32,736	0	0
$\mathbb{F}_{2^m}$ `Squarer`	112	20%	15,391	13,287	0	0
`Delay FIFOs`	5	2%	702	243	20	0
`Lambda Table`	1	0%	0	0	4	0
`Point Automorphism`	1	17%	15,194	21,876	0	0
`Comparator Tree`	1	14%	13,241	16,846	0	0
`Barrel Rotate`	1	1%	768	795	0	0
`PB->NB`	1	1%	226	1,990	0	0
`NB->PB`	1	1%	226	2,002	0	0
`Delay FIFOs`	1	1	702	243	0	0
$\mathbb{F}_p$ `Multiplication`	2	2%	4,066	2,078	0	290
$\mathbb{F}_p$ `Addition`	2	1%	1,406	1,219	0	0
`Delay FIFOs`	4	2%	1,363	162	8	0
`Distinguished Tripe Storage`	1	1%	471	104	7	0

(Artix-7 series [86]), one high performance designs (Virtex-7 series [92]), and one tries to offer the best cost effectiveness (Kintex-7 series [88]). The results of `ECC-Breaker V2` on these FPGAs are shown in Table 5.8. The prices are again taken for development boards from www.avnet.com [1] and do not contain taxes Fortunately, the results show that the ECC-Breaker design is ideally suited for the latest Xilinx FPGAs. The best results in terms of cost effectiveness are achieved for the Kintex-7 FPGA, as this FPGA offers enough slices for two instances of the design. Unfortunately, the amount of slices on a Virtex-7 FPGA is just under the required amount for three instances, which makes this FPGA impractical in terms of cost effectiveness. Except for the Virtex-7 FPGA the more recent Xilinx FPGAs offer a higher price-performance ratio compared to older ones.

5.6.3 Expected Runtime for larger curves

The gathered results are promising and present an opportunity for attacking larger elliptic curves. Table 5.9 show runtime estimations for ECDLPs of higher order. These are best

Table 5.8: Results of `ECC-Breaker V2` on several different FPGAs.

FPGA Series	Part Number	Price [$]	Frequency [MHz]	Slices [%]
Virtex-6	XC6VLX240T	2,495	291	57
Spartan-6	XC6SLX150T	995	161	89
Artix-7	XC7A200T	999	293	66
Kintex-7	XC7K325T	1,695	372	43
Virtex-7	XC7VX485T	3,495	372	34

case estimations, as they are calculated using post synthesis data and assume some FPGAs to implement multiple instances. One setup (18 Virtex-6 FPGAs) was verified in practice and is able to solve an ECDLP on ECC2K-112 within 19 days. 256 Spartan-6 FPGAs, as offered by SciEngines RIVYERA's S6-LX150 FPGA cluster [2], even only need 2 days, and only 36 days for an ECC2-112 challenge. The Certicom ECC2K-130 challenge is estimated to be solved using two of these clusters within 401 days. However, a Kintex-7 cluster of 256 FPGAs is cheaper and needs approximately only 174 days. Assuming almost an astronomical financial budget, even the Certicom ECC2-131 challenge is within reach. The currently standardized curves ECC2K-162 and ECC2-163 are to date not under thread, not even when attacked by certain agencies having incredible resources.

5.6.4 Solution to the posed ECDLP

We are proud to present having broken ECC2K-112. To date (May 2014) this is the largest solved binary ECDLP. The attack was conducted using up to 19 FPGAs (18 Virtex-6 and 1 Spartan-6) calculating for 46 days. At the starting time `ECC-Breaker V2` was still under development, so the Virtex-6 FPGAs were running at 165 MHz with the `ECC-Breaker V1` design, and the Spartan-6 FPGA was added later to the cluster running at 80 MHz with `ECC-Breaker V2`.

The FPGAs were connected to three different PCs. Three FGPAs (2 Virtex-6 and 1 Spartan-6) were connected to the first client, the other two clients were connected to up to eight Virtex-6 FPGAs each. The attack was started on 03/04/2014, the matching triple was found on 04/19/2014. Altogether the database contained 6,064,260 triples, which are about 2 million fewer than expected. Figure 5.13 shows the database's growth over time. The growth rate's variation is due to the fluctuating amount of FPGAs used. All 19 FPGAs together calculated about 250,000 distinguished triples per day, which means that the challenge would have been solved within 24 days, if we constantly had used all resources.

The matching triples and the resulting scalar can be found in Table 5.10.

Table 5.9: Best-case runtime and cost estimations for different challenges and FPGAs.

FPGA	**Challenge**	**Instances**	**Throughput** [10^6 IPS]	**FPGAs**	**Costs** [10^3 $]	**Iterations** [10^{15}]	**Runtime** [days]
Virtex-6	ECC2K-112	1	291	18	45	8.496	19
Spartan-6	ECC2K-112	1	161	256	255	8.496	2
Virtex-6	ECC2-112	1	291	18	45	90,3	200
Spartan-6	ECC2-112	1	161	256	255	90.3	25
Spartan-6	ECC2K-130	2	161	512	510	2,857	401
Kintex-7	ECC2K-130	2	744	256	434	2,857	174
Kintex-7	ECC2-131	2	744	1,000	1,695	46,239	719
Kintex-7	ECC2K-162	2	744	100,000	169,500	$237 \cdot 10^6$	$37 \cdot 10^3$
Kintex-7	ECC2-163	2	744	100,000	169,500	$3,030 \cdot 10^6$	$471 \cdot 10^3$

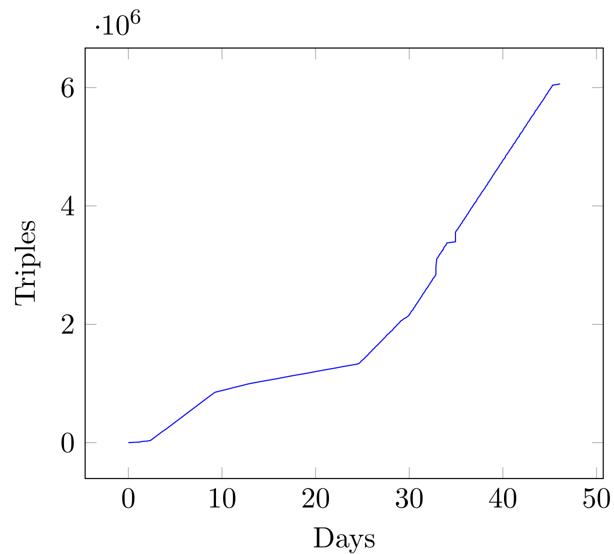

Figure 5.13: Growth of the database containing distinguished triples.

Table 5.10: Matching Triple and calculated solution to the posed 113-bit binary ECDLP.

Parameter	**Value**
$R.x$	`0x0000000052c1d359e3d293da92097`
$R.y$	`0x1f3504dd1deb743fe03e451972dc7`
c_1	`0x053f3f42abc6b0149bc75b0e90551`
d_1	`0x084613bfcdae50d1b760cf6d6129b`
c_2	`0x030bfb2cafee83dd35ee9132be301`
d_2	`0x0b022d4b25c4f3c3d277017cfd075`
k	`0x0276c233740d817000b80478fde46`

Chapter 6

Conclusions

Having spent over one year development time, we accomplished what we were looking for. A new record. More precisely, a new binary ECDLP record on a 113-bit Koblitz curve. This is the result of a long process, started by studying Pollard's rho attack. We closely examined different iteration functions in order to choose the most efficient one for implementing in hardware. We evaluated many different iteration functions using a C reference implementation. The mathematical foundation of these iteration functions is fairly complex and required a deep understanding of basic algebraic structures. Not carefully enough implemented improvements, sometimes even lead to incorrect results.

Once we had decided which iteration function to use, we had to find a target platform. We use an FPGA based design. Compared to many previous work, we use a state-of-the-art FPGA, a Virtex-6, and emphasize the advantages of this decision. The crucial part of a hardware implementation of Pollard's rho algorithm is the finite-field multiplier. We evaluated several multipliers and carefully modified a binary Karatsuba multiplier to meet our requirements of maximal throughput per area.

The resulting hardware design, the ECC-Breaker, is a fully autonomous, cyclic, piplined iteration function, producing one result each cycle. Except for the RS232 interface, which is used for communication, the hardware design approaches the FPGA's full capacity, even exceeded it. We encountered power problems, when running the design at the maximal achievable post place-and-route frequency. Sudden sporadic emergency switch-offs of the power controller, resulted from an average current of about 12 Ampere at the internal power supply. Although the FPGA board's power supply is designed to support current flows of up to 20 Ampere at the internal power supply, the only way to solve this issue was to reduce the power consumption. This was done by using the normal basis representation of binary polynomials in the automorphism module, which performs 224 squaring operations each cycle. As squarings are free in normal basis representation, we could significantly reduce the current flow to about 5 Ampere.

A further hard lesson we had to learn, was in regards to the maximum achievable frequency. As the ECC-Breaker design is rather complex, the estimated post-synthesis frequency cannot be achieved by the place-and-route step. Adding new pipline stages always increased the estimated maximum frequency, however sometimes the additional required register have lead to the opposite effect on the post place-and-route frequency. Additionally the tools used for implementing the design in real hardware offer a tremendous amount of

configuration possibilities, we still potentially have not exhausted. Nevertheless, we were able to successfully route a design with 200 MHz, which is 69% of the estimated maximum frequency.

When we tried to implement the first version of ECC-Breaker on different FPGAs, we realized that many FPGAs do not offer enough DSP-slices for two 112-bit prime field multipliers. We worked around this issue, by modifying our iteration function to avoid the prime field multiplication by a using additional adders. This modification allowed us to implement the design on smaller FPGAs, as the Spartan-6 and even enabled better results on our target FPGA, the Virtex-6.

The actual attack of ECC2K-112 was started in March 2014, and provided the solution in April 2014. The attack was conducted using up to 18 Virtex-6 FPGAs and 1 Spartan-6 FPGA, resulting in a combined throughput of about 3 billion iterations per second. Besides the new record, the results, both theoretical and practical, are of high value regarding security estimations of current and future elliptic curve standards. ECC-Breaker can be modified by minor changes to attack ECDLPs of higher order. Our estimations show that Certicom's ECC2K-130 is definitely computationally feasible using more instances of ECC-Breaker.

Appendix A

Parameter

This section summarizes the parameters of the curve we solved an ECDLP on, namely the parameters of a 113-bit Koblitz curve. The curve and its parameters are generated using Sage [4]. Table A.1 contains the generated parameters. All countermeasures to avoid attacks described in Section 4.2 are taken into account. We choose ECC2K-112 to really provide a solution to an ECDLP within a few days and thereby prove the functionality and power of our hardware design.

In order to prove that an ECDLP was solved for real and the answer was not known in advance, we used Sage to comprehensibly, pseudo-randomly generate two points. Listing A.1 contains the Sage-code used for this purpose. The functions `polyvec_to_str` and `int_to_poly` are used to generate human readable strings of 113-bit polynomials alternatively to do it the other way around. The points are generated using a SHA256 [5] digest. The x-coordinate is directly derived from the digest, the y-coordinate is calculated by solving the elliptic curve equation.

Table A.1: Curve parameters of targeted 113-bit elliptic curve.

Property	Value
challenge name	ECC2K-112
m	113
irreducible polynomial	$x^{113} + x^5 + x^3 + x^2 + 1$
irreducible polynomial	0x2000000000000000000000000002d
elliptic curve E	$y^2 + xy = x^3 + x^2 + 1$
order n	0xfffffffffffffffdbf91af6dea73
$\mathrm{ld}(n)$	112
cofactor h	2
base Point $P.x$	0x0a27644cfced9667d2084f8be061c
base Point $P.y$	0x0d5acd887d5585dd75c5d07165699
public Point $Q.x$	0x189037f88aed8e32400b16d2b1a6e
public Point $Q.x$	0x00e4718fb1e9f50f845ff162ff59c
scalar $k = log_Q P$	0x0276c233740d817000b80478fde46

Listing A.1: Sage code used to pseudo-randomly generate point targets.

```
def polyvec_to_str(x):
    string = ""
    for i in range(0,len(x)):
        string = string + str(x[len(x) - i - 1])
    return hex(Integer(string, base=2))

def int_to_poly(s):
    poly = K.gen()
    poly = 0;
    for i in range (0, len(s.bits())):
        if i < 113:
            if s.bits()[i] == 1 :
              poly = poly + x^i
    return poly;

FF = sage.rings.finite_rings.finite_field_ext_pari.FiniteField_ext_pari;
K = FF(2^113,'x', 0x20000000000000000000000000002D.bits());
x = K.gen()
E = EllipticCurve(K, [1,1,0,0,1])
cofactor = (factor(E.order()))[0][0]

import hashlib
points = 0
for i in range(0,10):
     H  = hashlib.sha256(str(i))
     PX = int_to_poly(Integer(H.hexdigest(), base=16))
     PY = PolynomialRing(K, 'PY').gen()
     ROOTS = (PY^2+PX*PY+PX^3+1*PX^2+1).roots()
     if len(ROOTS) > 0:
         PY = ROOTS [0][0]
         P = E([PX, PY])
         P = P * cofactor
         print 'x: ' + polyvec_to_str(P[0]._vector_())
         print 'y: ' + polyvec_to_str(P[1]._vector_())
         print '\n'
         points = points + 1
         if points == 2:
             print 'points successfully generated'
             break
```

Appendix B

Abbreviations

ABC	Anomalous Binary Curves
AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
BBE	Batch Binary Edwards
CPU	Central Processing Unit
DES	Data Encryption Standard
DLP	Discrete Logarithm Problem
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECRYPT II	European Network of Excellence in Cryptology II
EEA	Extended Euclidian Algorithm
FPGA	Field Programmable Gate Array
gcd	Greatest Common Divisor
GPU	Graphics Processing Unit
IPS	Instructions Per Second
LUT	Look-Up-Table
NB	Normal Basis
NIST	National Institute of Standards and Technology
NSA	National Security Agency
ONB	Optimal Normal Basis
OTP	One-Time-Pad
PB	Polynomial Basis
RFID	Ratio Frequency Identification
Reg	Register
VHDL	Very high speed integrated circuit Hardware Description Language

Bibliography

[1] Avnet Inc. http://www.avnet.com/, Feb 2014.

[2] SciEngines GmbH. http://www.sciengines.com/, Feb 2014.

[3] Patentschrift Chiffrierapparat DRP Nr. 416 219. Available online at http://www.cdvandt.org/Enigma%20DE416219C1.pdf, April 2014.

[4] Sage. Available online at http://www.sagemath.org/index.html, May 2014.

[5] SHA-224 and SHA-256. Available online at https://tools.ietf.org/html/rfc4634#section-5.1, May 2014.

[6] Avnet Inc. Xilinx Spartan-6 FPGA LX150T Development Kit. Available online at http://www.em.avnet.com/en-us/design/drc/Pages/Xilinx-Spartan-6-FPGA-LX150T-Development-Kit.aspx, May 2014.

[7] S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012). Available online at http://www.ecrypt.eu.org/, Sep 2012.

[8] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. V. Herrewege, and B.-Y. Yang. Breaking ECC2K-130. IACR Cryptology ePrint Archive, Report 2009/541, 2009.

[9] E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication*, 800-133, 2012.

[10] E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication*, 800:133, 2012.

[11] D. J. Bernstein. Batch Binary Edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.

[12] Bernstein, Daniel J. Binary Batch Edwards 113-bit Multiplier. Available online at http://binary.cr.yp.to/bbe251/113.gz, May 2009.

[13] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. *International Journal of Applied Cryptography*, 2 (3):212, 2012. ISSN 1753-0563. doi: 10.1504/IJACT.2012.045590. URL http://www.inderscience.com/link.php?id=45590.

[14] E. Brown and B. Myers. Elliptic curves from Mordell to Diophantus and back. *American Mathematical Monthly*, (September):639–649, 2002. URL http://www.jstor.org/stable/3072428.

[15] Certicom Research. The Certicom ECC Challenge. Available online at https://www.certicom.com/index.php/the-certicom-ecc-challenge, Nov 1997.

[16] Certicom Research. Certicom Announces Elliptic Curve Cryptosystem (ECC) Challenge Winner. Available online at http://www.certicom.com/index.php/2002-press-releases/340-notre-dame-mathematician-solves-eccp-109-encryption-key-problem-issued-in-1997, Nov 2002.

[17] Certicom Research. Certicom Announces Elliptic Curve Cryptography Challenge Winner. Available online at http://www.certicom.com/index.php/2004-press-releases/300-solution-required-team-of-mathematicians-2600-computers-and-17-months-, April 2004.

[18] R. Churchhouse. *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet.* Cambridge University Press, 2002. ISBN 9780521008907.

[19] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Discrete Mathematics and its Applications. Chapman & Hall/CRC, Boca Raton, FL, 2006. ISBN 978-1-58488-518-4; 1-58488-518-1.

[20] Cohen, Fred. A Short History of Cryptography. Available online at http://all.net/edu/curr/ip/Chap2-1.html, 1996.

[21] Cypher Research Laboratories. A Brief History of Cryptography . Available online at http://www.cypher.com.au/crypto_history.htm, 2006.

[22] P. de Fermat, P. Tannery, C. Henry, and F. M. de l'éducation nationale. *OEuvres de Fermat.* Number Bd. 1 in OEuvres de Fermat. Gauthier-Villars et fils, 1891.

[23] ECRYPT II. ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012). Available online at http://www.ecrypt.eu.org/documents/D.SPA.20.pdf, sep 2012.

[24] T. ElGamal. A PUBLIC KEY CRYPTOSYSTEM AND A SIGNATURE SCHEME BASED ON DISCRETE LOGARITHMS. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[25] J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede. Breaking Elliptic Curve Cryptosystems Using Reconfigurable Hardware. In *Field Programmable Logic and Applications (FPL)*, pages 133–138. IEEE, 2010.

[26] L. Fibonacci and L. Sigler. *The Book of Squares.* Academic Press, 1987. ISBN 9780126431308.

[27] P. FIPS. 46-3: Data Encryption Standard (DES). *National Institute of Standards and Technology*, 25(10), 1999.

[28] G. Frey and H.-G. Rück. A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. *Mathematics of Computation*, 62 (206):865–874, 1994.

[29] R. Gallant, R. Lambert, and S. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation of the American Mathematical Society*, 69(232):1699–1705, 2000.

[30] S. Gao and H. W. Lenstra Jr. Optimal normal bases. *Designs, Codes and Cryptography*, 2:315–323, 1992. URL `http://link.springer.com/article/10.1007/BF00125200`.

[31] C. F. Gauß. Disquisitiones Arithmeticae, 1801. English translation by Arthur A. Clarke, 1986.

[32] T. Güneysu, C. Paar, and J. Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *FPGA*, page 207. ACM Press, 2007.

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

[34] R. Harley. Elliptic Curve Discrete Logarithms: ECC2K-108, 2000. `http://cristal.inria.fr/~harley/ecdl7/readMe.html`.

[35] T. Heath. *Diophantus of Alexandria - A Study in the History of Greek Algebra.* Read Books, 2007. ISBN 9781406763140.

[36] Intel Corporation. Intel ® Core™ i7-3537U Processor. Available online at `http://ark.intel.com/de/products/72054/Intel-Core-i7-3537U-Processor-4M-Cache-up-to-3_10-GHz`, May 2014.

[37] Intel Corporation. Intel ® Core™2 Extreme Processor QX6850. Available online at `http://ark.intel.com/products/30789/Intel-Core2-Extreme-Processor-QX6850-8M-Cache-3_00-GHz-1333-MHz-FSB`, May 2014.

[38] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[39] L. Judge and P. Schaumont. A Flexible Hardware ECDLP Engine in Bluespec. In *Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS)*, 2012.

[40] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, 1996. ISBN 9781439103555.

[41] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[42] T. Kelly. THE MYTH OF THE SKYTALE. *Cryptologia*, 22(3):244–260, 1998.

[43] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177): 203, Jan. 1987. ISSN 00255718. doi: 10.2307/2007884. URL `http://www.jstor.org/stable/2007884?origin=crossref`.

[44] A. K. Lenstra and H. W. L. Jr. Algorithms in Number Theory. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 673–716. 1990.

[45] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. In *Public Key Cryptography*, pages 446–465. Springer, 2000.

[46] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[47] H. Liddell and R. Scott. *Liddell and Scott's Greek-English Lexicon, Abridged: The Little Liddell*. Simon Wallenberg Press, 2007. ISBN 9781843560265.

[48] R. Lidl. *Introduction to finite fields and their applications*. 1994. ISBN 0521307066.

[49] S. Mane, L. Judge, and P. Schaumont. An Integrated Prime-Field ECDLP Hardware Accelerator with High-Performance Modular Arithmetic Units. In *Reconfigurable Computing and FPGAs*, pages 198–203. IEEE, Nov. 2011. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6128577`.

[50] E. D. Mastrovito. VLSI Designs for Multiplication over Finite Fields GF(2^m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 297–309, 1988.

[51] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. *Transactions on Information Theory*, 39(5):1639–1646, 1993.

[52] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[53] G. Meurice de Dormale, P. Bulens, and J.-J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *CHES*, pages 378–393. Springer, 2007.

[54] A. R. Miller. The Cryptographic Mathematics of Enigma. *Cryptologia*, 19(1):65–80, 1995.

[55] V. Miller. Use of elliptic curves in cryptography. *Advances in Cryptology—CRYPTO'85 Proceedings*, pages 417–426, 1986. URL `http://link.springer.com/chapter/10.1007/3-540-39799-X_31`.

[56] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985.

[57] L. J. Mordell. On the rational solutions of the indeterminate equations of the third and fourth degrees. In *Proc. Cambridge Philos. Soc*, volume 21, pages 179–192, 1922.

[58] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal normal bases in GF(p^n). *Discrete Applied Mathematics*, 22(2):149–161, 1989.

[59] I. Newton. De resolutione quaestionum circa numeros. *Mathematical Papers*, 4:110–115.

[60] NVIDIA Corporation. GeForce GTX 295. Available online at `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295`, May 2014.

[61] Oracle. MySQL. Available online at `http://www.oracle.com/de/products/mysql/overview/index.html`, April 2014.

[62] Plutarchus and B. Perrin. *Plutarch's Lives: Alcibiades and Coriolanus Lysander and Sulla*. Lcl, 80. Harvard University Press, 2000. ISBN 9780674990890.

[63] S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms over $GF(p)$ and Its Cryptographic Significance. *Transactions on Information Theory*, 24(1):106–110, 1978.

[64] J. M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[65] N. F. Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.

[66] C. Research. SEC 2: Recommended Elliptic Curve Domain Parameters. Available online at `http://www.secg.org/collateral/sec2_final.pdf`, September 2000.

[67] A. Rice and E. Brown. Why Ellipses Are Not Elliptic Curves. *Mathematics Magazine*, 85(3):163–176, 2012.

[68] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[69] F. Rodrıguez-Henrıquez and Ç. Koç. On Fully Parallel Karatsuba Multipliers for GF(2^m). In *Computer Science and Technology*, pages 405–410, 2003.

[70] T. Satoh. Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves. *Commentarii Math. Univ. Sancti Pauli*, 47(1):81–92, 1998.

[71] I. Semaev. EVALUATION OF DISCRETE LOGARITHMS IN A GROUP OF p-TORSION POINTS OF AN ELLIPTIC CURVE IN CHARACTERISTIC p. *Mathematics of Computation of the American Mathematical Society*, 67(221):353–356, 1998.

[72] D. Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.

[73] C. E. Shannon. Communication Theory of Secrecy Systems. *Bell system technical journal*, 28(4):656–715, 1949.

[74] S. Singh. *Geheime Botschaften: die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet.* Dtv-Taschenbücher. Hanser, 2000. ISBN 9783446198739.

[75] N. P. Smart. The Discrete Logarithm Problem on Elliptic Curves of Trace One. *Journal of Cryptology*, 12(3):193–196, 1999.

[76] Sony Computer Entertainment Europe. Offizielle PlayStation-Website: PlayStation 3, PS3. Available online at `at.playstation.com/ps3/`, May 2014.

[77] B. Sunar and C. Koç. An efficient optimal normal basis type II multiplier. *IEEE Transactions on Computers*, 50(January):83–87, 2001. URL `http://pdf.aminer.org/000/292/234/efficient_inversion_algorithm_for_optimal_normal_bases_type_ii.pdf`.

[78] R. Taschner. *Die Zahl, die aus der Kälte kam: Wenn Mathematik zum Abenteuer wird*. Hanser, Carl GmbH, 2013. ISBN 9783446436831.

[79] E. Teske. Speeding up Pollard's Rho Method for Computing Discrete Logarithms. In *Algorithmic Number Theory*, volume 1423, pages 541–554. Springer, 1998.

[80] C. S. Tranquillus. *Suetonius: The Lives of the Twelve Caesars; An English Translation, Augmented with the Biographies of Contemporary Statesmen, Orators, Poets, and Other Associates*. Gebbie & Co., Medford, MA, 1889.

[81] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[82] von Lieven, A. *Grundriss des Laufes der Sterne: das sogenannte Nutbuch*. Number Bd. 1 in CNI Publications: Carlsberg Papyri 8. Carsten Niebuhr Institute of Near Eastern Studies, University of Copenhagen, 2007. ISBN 9788763504065.

[83] A. Weil. L'arithmétique sur les courbes algébriques. *Acta mathematica*, 52(1):281–315, 1929.

[84] M. J. Wiener and R. J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *Selected Areas in Cryptography*, pages 190–200. Springer, 1999.

[85] F. Wrixon. *Codes, Chiffren und andere Geheimsprachen.: Von den ägyptischen Hieroglyphen bis zur Computerkryptologie.* Könemann im Tandem, 2000. ISBN 9783829038881.

[86] Xilinx. Artix-7 FPGA Family. Available online at `http://www.xilinx.com/products/silicon-devices/fpga/artix-7/`, May 2014.

[87] Xilinx. ISE Design Suite. Available online at `http://www.xilinx.com/products/design-tools/ise-design-suite/`, May 2014.

[88] Xilinx. Kintex-7 FPGA Family. Available online at `http://www.xilinx.com/products/silicon-devices/fpga/kintex-7/`, May 2014.

[89] Xilinx. Virtex-6 FPGA ML605 Evaluation Kit. Available online at `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm`, April 2014.

[90] Xilinx. Spartan-3 FPGA Family. Available online at `http://www.xilinx.com/products/silicon-devices/fpga/spartan-3.html`, April 2014.

[91] Xilinx. Virtex-5 Boards and Kits. Available online at http://www.xilinx.com/products/boards_kits/virtex5.htm, May 2014.

[92] Xilinx. Virtex-7 FPGA Family. Available online at http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/, May 2014.