RUHR-UNIVERSITÄT BOCHUM

Breaking ecc2-113: Efficient Implementation of an Optimized Attack on a Reconfigurable Hardware Cluster

Susanne Engels

Master's Thesis. February 22, 2014.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Ralf Zimmermann

Abstract

Elliptic curves have become widespread in cryptographic applications since they offer the same cryptographic functionality as public-key cryptosystems designed over integer rings while needing a much shorter bitlength. The resulting speedup in computation as well as the smaller storage needed for the keys, are reasons to favor elliptic curves. Nowadays, elliptic curves are employed in scenarios which affect the majority of people, such as protecting sensitive data on passports or securing the network communication used, for example, in online banking applications. This works analyzes the security of elliptic curves by practically attacking the very basis of its mathematical security — the Elliptic Curve Discrete Logarithm Problem (ECDLP) — of a binary field curve with a bitlength of 113. As our implementation platform, we choose the RIVYERA hardware consisting of multiple Field Programmable Gate Arrays (FPGAs) which will be united in order to perform the strongest attack known in literature to defeat generic curves: the parallel Pollard's rho algorithm. Each FPGA will individually perform a what is called additive random walk until two of the walks collide, enabling us to recover the solution of the ECDLP in practice. We detail on our optimized VHDL implementation of dedicated parallel Pollard's rho processing units with which we equip the individual FPGAs of our hardware cluster. The basic design criterion is to build a compact implementation where the amount of idling units — which deplete resources of the FPGA but contribute in only a fraction of the computations — is reduced to a minimum. As a result, we develop an efficient core which we expect to be able to practically solve the ECDLP on the targeted 113-bit binary curve. Besides offering the mere numbers of the design, we solve the ECDLP over a smaller subgroup of our original target, with a bitlength of only 60 bits, in a first test run. Afterward, we estimate the pending for the attack on the full bitlength.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Susanne Engels

Contents

1 Introduction

For thousands of years, mankind has developed various methods to send secret messages – illegible to everyone but the dedicated receiver. The principle of encrypting is as follows: both sender and receiver share a secret, commonly known as a key, which is then combined with the plain message in order to create a ciphertext. Nowadays this "combination" is generally achieved by some mathematical algorithm. Once the receiver obtains said message, he can reverse the encryption – using the secret key – and recover the original message. This form of encryption is referred to as *symmetric encryption*. One drawback is that the two parties somehow have to exchange the secret key beforehand, ideally such that nobody else can get hold of it, for example by meeting in person. Nowadays, the other party of the conversation, for example the vendor of an online shopping platform, might not even be known to you, hence, meeting in person is not an option – so how to share the secret over an insecured channel?

At the end of the 1970s, the first *asymmetric encryption schemes* have been introduced. The innovation is that here two different keys exist: one private key which is known only to the individual owner of the key and must not be shared, and a public key which can be distributed over the Internet. Sending an encrypted message to someone now works such that the public key is taken in order to encipher the message whereas only the private key is able to decipher it. Thus, the key distribution problem is solved.

Still, one drawback of asymmetric schemes is the considerable bigger key length: the RSA encryption scheme [RSA78] proposed in 1977 by Rivest, Shamir and Adleman needs a key of bitlength 2048 (recommendation by the German "Bundesnetzagentur" as published in their "Algorithmenkatalog" from January 2014 [Bun]), compared to a recommended bitlength of 128 bit for the symmetric AES encryption scheme. The reason behind this is that most asymmetric schemes rely on a mathematically hard problem such as *integer factorization* or solving the *discrete logarithm problem*. As these problems have been investigated by mathematicians even before they played any role in cryptography, algorithms such as *Pollard's Rho Algorithm* or *Pohlig-Hellman Algorithm* exist whose run-time depends on the size of the numbers – hence, the longer, the better. Then again, size also does matter for the run-time of the asymmetric encryption algorithms, making them much slower than symmetric key cryptosystems. In 1985, elliptic curves have been introduced in cryptography, establishing the new field of Elliptic Curve Cryptography (ECC): based on the Elliptic Curve Discrete Logarithm Problem (ECDLP), ECC offers the same asymmetric functionality as, for example, RSA, while demanding for a significantly smaller key size [1]. Hence, cryptographic schemes based on elliptic curves such as the Elliptic Curve Digital Signature Algorithm (ECDSA)

[1] Note that solving the DLP for elliptic curves is far more complex than in standard groups

or Elliptic Curve Diffie Hellman (ECDH) enjoy great popularity.

1.1 Motivation

As ECC is for example used in the new German passport (neuer Personalausweis) in order to secure sensitive data from being accessed unauthorizedly or in the Secure Shell (SSH) protocol [IETa] or Transport Layer Security (TLS) protocol [IETb] which enable for establishing secure network connections , analyzing the security of ECC is of great importance: without breaking cryptographic systems in practice, only theoretical estimations on whether a scheme is secure or not exist. Hence, cryptoanalysts strive after finding flaws or verifying recommended bitlengths such that insufficiency is discovered and fixed as quickly as possible.

Therefore, this thesis aims for practically solving the most complex ECDLP for a curve in $\mathbb{GF}(2^m)$ currently known in the literature. Although the targeted bitsize of $m = 113$ is not recommended to be used for ECC anymore, our attack helps to estimate the security or insecurity of bigger curves: depending on the run-time required to break a small curve in practice, we can deduce the threat for longer bitsizes. Although not within the scope of this thesis, adjusting the resulting design in order to attack further (more complex) curves one step at a time might give practical results for the raise of the complexity of the ECDLP.

Solving the ECDLP of a 113-bit binary curve with help of a reconfigurable hardware cluster has another interesting side effect: as a rule of thumb, the complexity of brute-forcing an Elliptic Curve (EC) cryptosystem is said to be twice as hard as for symmetric block ciphers, leading to a recommended bitsize that is twice as long. Our attack will be able to verify or adjust said assumption: our 113-bit target curve has twice the bitsize of Data Encryption Standard (DES) which can be attacked within on average 6.4 days on COPACOBANA [GKN+08](the predecessor of the hardware cluster used in this thesis).

1.2 Related Work

Prior to this thesis, other works aiming to solve the Discrete Logarithm Problem (DLP) on elliptic curves have been presented, motivated (among others) by the Certicom Challenge issued in Nov 1997, the latest update can be found in [Cer]. The document also states that the so-far biggest binary curve DLP was solved in 2004 (organized by Chris Monico) in 17 months of calendar time, with a group of 2600 users who collaborated in performing the best known attack, i.e., the parallel Pollard's rho algorithm. The current challenge to be solved according to the document is ECC2K-130, as further detailed in [BBB+09].

In 2006, Bulens, de Dormale, and Quisquater present the idea to use special purpose hardware, such as Field Programmable Gate Arrays (FPGAs), for attacking the ECDLP on binary curves [DjQL06]. Implementation results for ECC2–79 and ECC2–163 are given, but since the practical attack was not yet finished, the authors provide an estimated running time for both problems, stating that a collision for ECC2–79 can occur in three hours, with a distinguished point criteria of 26 bits. For ECC2–163, the estimation

comes to a duration of $700 \cdot 10^6$ years using a distinguished point property[2]of 54 bits. In a successive paper published by the same authors in 2007 [dDBQ07], de Normale et al. give a more detailed look into power consumption and performance to provide an estimation for the run-time and cost of their implementation. For this reason, they analyze four different processor sizes (tiny, small, medium, large), in order to offer a solution depending on the available hardware. The medium-size elliptic curve processors, which was implemented, is said to solve the ECDLP for a curve over $\mathbb{GF}(2^{113})$ in six months, using the hardware cluster COPACOBANA. Other estimations for $\mathbb{GF}(2^{131})$ and $\mathbb{GF}(2^{163})$ are also provided but so far, their attack has not been actually run on the cluster.

Bos et al. solved the DLP for the biggest prime-field curve in [BKK+12] in 2012. Their paper provides algorithms for a single instruction multiple data (SIMD) architecture which can be efficiently implemented on a Cell processor and therefore, can be run on a cluster of Playstation 3 consoles. The main idea is to implement the parallel form of Pollard's rho algorithm in combination with what is called "sloppy reduction", where instead of the reduction modulo p, a faster reduction modulo p is used, which can produce incorrect results and hence is not recommended for actual cryptographic computations such as encryption. For the purpose of cryptanalysis, i.e., collecting many distinguished points until two of them collide in this scenario, a wrong result can be tolerated. The implementation in its unoptimized form took roughly six months to solve the ECDLP, collecting approximately 5 billion distinguished points.

Inspired by their work in [BKK+12] described above, Bos, Kleinjung, and Lenstra investigate in [BKL10] whether using the negation map, see Sect. 3.2.3, for solving ECDLPs can be recommended or not. As the main problem of the negation map is getting trapped in fruitless cycles, the authors implement all up to that point suggested methods in order to escape said cycles. Their analysis shows that the expected speedup by a factor of $\sqrt{2}$ can not be achieved with their current implementations. Still, in general, a recommendation for the use of the negation map for solving prime field ECDLPs is given, with the exception of implementations running on SIMD-environments. Based on their findings, Bernstein, Lange, and Schwabe published an improved variant of the use of the negation map in [BLS11] which is presented in more detail in Sect. 3.2.3.

For further related work that presents background information on the Pollard's rho algorithm and its optimizations, the reader is referred to Chap. 3.

1.3 Organization of this Thesis

The thesis is structured as follows: in Chap. 2, we give the relevant background information: first, we present the mathematical foundations such as groups and fields, followed by an introduction to EC and ECC. The chapter closes with an overview over implementation platforms with a detailed look into the structure of an FPGA. Chapter 3 introduces different algorithms to solve the DLP, with focus on Pollard's rho algorithm

[2]The distinguished point property is a predefined bit pattern to select points. For further reference, see Sect. 3.2.2.

and its optimizations. The implementation of our dedicated Pollard's rho processor is then further detailed in Chap. 4. The practical results and estimations on the run-time of our attack are summarized in Chap. 5 while Chap. 6 concludes the thesis.

2 Theoretical Background

This chapter presents the foundations needed in the remainder of this thesis. First, the underlying mathematical structures and arithmetic are described, followed by an overview of EC in general and the binary curves used in this thesis. In addition, we give an introduction to ECC, how it can be used and what the benefits are compared to other cryptographic schemes. The chapter closes with an overview over different platforms for implementations, with a more detailed look into FPGAs and hardware clusters.

2.1 Mathematical Background

In the following sections, we first give a brief introduction to group and field theory and then concentrate on finite fields and their arithmetic. Here, the main focus is on binary fields as they serve as the basis for the ECs relevant to this thesis. Definitions and formulae are based on [Paa10] and [HMV03a].

2.1.1 Finite Field Arithmetic

Finite Fields, as their name implies, are a special form of *fields*, whereas a field relies on the definition of the mathetical structure called *group*. The definition of these structures and their characteristics are detailed in this section.

Groups and Fields

A *group* consists of a set of elements and a so-called group operation, e.g., an addition. If the group operation is now applied to any two elements of the group, the result is also an element of the group. To fulfill the definition of a group, three further terms have to be met, given three group elements a,b, and c, and the group operation $\circ$:

- **Associativity** Computing $(a \circ b) \circ c$ leads to the same result as $a \circ (b \circ c)$.
- **Identity Element** For each group element a exists an identity element 1 such that $a \circ 1 = 1 \circ a = a$.
- **Inverse Element** For each group element a exists an inverse element a^{-1} such that $a \circ a^{-1} = a^{-1} \circ a = 1$.

Furthermore, if the group satisfies a fourth condition, it is called *abelian* or *commutative*:

- **Commutativity** For all group elements a, b it holds that $a \circ b = b \circ a$.

A common example is the set of integers, using addition as group operation.

While the elements of a group can be computed using only one group operation and its corresponding inverse operation, e.g., addition and subtraction, a *field* consists of an additive group (addition, subtraction) and a multiplicative group (multiplication, division), i.e., all elements of a field can be combined using the four basic arithmetic operations. The following requirements have to be met by a field:

- **Additive group** All field elements form an additive group, i.e., group operation + with identity element 0.
- **Multiplicative group** All field elements form a multiplicative group, i.e., group operation $\times$ with identity element 1.
- **Distributivity** For all field elements it holds that $a \times (b + c) = (a \times b) + (a \times c)$.

Finite Fields

A field is defined over a number of field elements, however, depending on the underlying additive and multiplicative groups, the amount of elements can be infinite. As the name already indicates, *Finite Fields* (or Galois Fields) consist of a finite number of field elements. These fields are characterized by two factors: a prime number p and a positive integer n, which define the finite field $\mathbb{GF}(p^n)$. p^n also determines the number of elements, i.e., the order, of the finite field. For $n = 1$, the finite field consists of the elements $0..p-1$ and is also denoted as prime field. For $n > 1$, the elements of the finite field $\mathbb{GF}(p^n)$ are often represented as polynomials of the form $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0$ where the maximum degree of the polynomials is $n-1$ and the coefficients are taken from the prime field defined by p. To allow multiplication of two field elements, an irreducible polynomial used to reduce results which exceed the group, is also part of the definition of a finite field.

Binary Field Arithmetic

Finite Fields with prime number 2 are called *Binary Fields*. In the remainder of this thesis, we focus on binary fields and their arithmetic, as later we define ECs over this type of fields. The polynomials representing the elements of a prime field (see above) have coefficients in $\mathbb{GF}(2)$ which are commonly denoted as a simple binary string.

Addition The sum of two field elements is computed by adding both elements in their polynomial representation, i.e., adding the coefficients belonging to the same degree of x (modulo prime p). Therefore, for binary fields it can easily be seen that both addition and subtraction (modulo 2) are equal and can be computed as exclusive-or (xor) of the coefficients, hence, finding the additive inverse is trivial.

Multiplication In order to multiply two field elements $A(x)$ and $B(x)$, their polynomials are multiplied according to standard polynomial multiplication:

$$\begin{aligned} A(x) \cdot B(x) &= (a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0x^0) \cdot (b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_0x^0) \\ &= (a_{n-1}b_{n-1})x^{2n-2} + (a_{n-1}b_{n-2} + a_{n-2}b_{n-1})x^{2n-3} + \cdots + a_0b_0x^0 \mod c(x) \end{aligned}$$

Again, the coefficients of the result are elements of $\mathbb{GF}(2)$, i.e., are computed modulo 2. Whenever the result of the multiplication has a higher degree than $n-1$ — which is the general case, as the highest possible degree is $2n-2$ as seen above — it cannot be considered as an element of the finite field just like that. Rather, the result has to be reduced using an irreducible polynom P(x) of degree n for example by polynomial division.

The above described method is a so-called basic schoolbook multiplication, other (more advanced) multiplication techniques in the literature are for example the Karatsuba algorithm [KO63] which operates according to a recursive divide-and-conquer approach. In Sect. 4.2, we argue which multiplier to implement.

Inversion Finding the multiplicative inverse $A^{-1}(x)$ of a non-zero element in $\mathbb{GF}(2^m)$ as opposed to the additive inverse is more difficult. The task is to find the element $A^{-1}(x)$ such that

$$A^{-1}(x) \cdot A(x) = 1 \mod P(x).$$

There exist different algorithms to efficiently compute $A^{-1}(x)$ such as the extended Euclidean algorithm (EEA) for polynomials as described in [HMV03b] or Fermat's Little Theorem as stated in [MVO96a]. Both algorithms can efficiently compute the multiplicative inverse of any multiplicative group. If we want to compute the inverse in a specific group, the use of an addition chain might come in handy, which will be further detailed in Sect. 4.2.

2.1.2 Elliptic Curves

In this section, an introduction to elliptic curves and their mathematical relevance is given. Additionally, we illustrate how computations on elliptic curves work and how their characteristics can be used in the cryptographic world. Again, definitions and formulae are taken from [HMV03a], [Paa10], and [CFA+12].

Introduction to Elliptic Curves

Elliptic curves have been studied by mathematicians for many years and have been used to analyze different mathematical theorems, e.g., Fermat's Last Theorem which was proven to be correct by Andrew Wiles [Wil95]. Generally speaking, an elliptic curve, together with a point $\mathcal{O}$, can be considered as a group, hence it is defined over an (infinite)

set of elements. $\mathcal{O}$ serves as the identity element and is the "point at infinity", i.e., an imaginary point in the y-plane of a coordinate system. Another characteristic of an elliptic curve is the field K it is defined over, e.g., the set of real numbers or some binary field $\mathbb{GF}(2^m)$. An elliptic curve is defined by the so-called Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, \ldots, a_6$ are constants in the underlying field K and the discriminant $\Delta \neq 0$, defined by

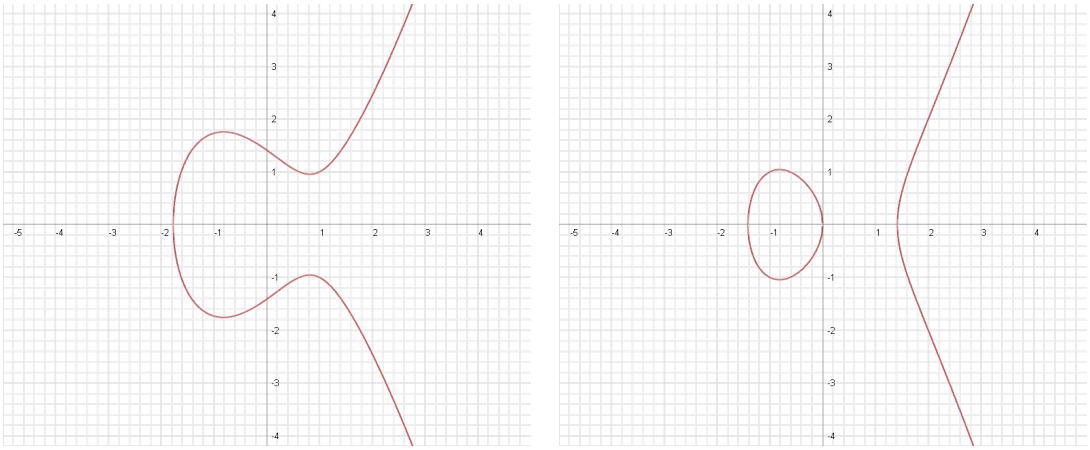
$$\begin{aligned}
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2.
\end{aligned}$$

In the literature, we often find either $E : y^2 = x^3 + ax^2 + b$ with $\Delta = -16(4a^3 + 27b^2)$ or $E : y^2 + xy = x^3 + ax^2 + b$ and $\Delta = b$, with a and b as the only remaining constants. These simplified Weierstrass equations can be obtained by transforming the original equation and hold, if the characteristic of K is either not equal to 2 or 3 in the first case, or equal to 2 with $a_1 \neq 0$ for the second case. Curves of the second type are also known as *non-supersingular*. Every combination of x, y which fulfills these equations is a valid point on the elliptic curve. Describing a point by its x- and y-coordinate, one uses the so-called *affine coordinates*. In the remainder of this thesis, we use affine coordinates and the corresponding formulae for point arithmetic. Other representations of an elliptic curve point exists which use an additional third z-coordinate. An affine point (x,y) can be defined by the triple (X,Y,Z) in *projective coordinates* where $x = X/Z$ and $y = Y/Z$. A slightly different version are *Jacobian coordinates* where $x = X/Z^2$ and $y = Y/Z^3$. Both representations eliminate costly inversion operations when computing a point addition or doubling.

Arithmetics on Elliptic Curves

Elliptic curves can be defined over arbitrary groups. In illustrations, elliptic curves over the real numbers are chosen since here all possible x-coordinates are defined and a corresponding y-coordinate satisfying the curve equation can be found. Plotting all points which belong to the curve results in the graphs shown in Fig. 2.1.

The group law for elliptic curves is referred to as *addition* and works as follows: given two curve points $P = (x_p, y_p)$ and $Q = (x_q, y_q)$, a third point R which is also an element of the curve can be uniquely derived from P and Q as $R = (x_r, y_r) = P + Q$. The resulting coordinates of R are not simply computed by adding the coordinates of P and Q, i.e., $x_r \neq x_p + x_q$ and $y_r \neq y_p + y_q$, but can be derived geometrically: Figure 2.2 depicts the group operation for a curve over the real numbers. Two different cases have to be distinguished.



(a) $y^2 = x^3 - 2x + 2$ (b) $y^2 = x^3 - 2x$

Figure 2.1: Two elliptic curves defined over the real numbers.

Point Addition If P and Q are two distinct points of the curve, the resulting point R is constructed as follows. A line drawn through P and Q intersects the elliptic curve at some other point I. Mirroring this point at the x-axis leads to a new point which is then defined to be the result $R = P + Q$.

Point Doubling If P and Q are the same point of the curve, the construction of R as described above needs to be adapted: now a tangent line through P is drawn. Then, the point at the intersection I with the elliptic curve is again reflected about the x-axis to obtain the result $R = P + Q = 2P$.

An algebraic description of point addition and point doubling for an elliptic curve of the form $y^2 = x^3 + ax + b$ is

$$x_r = \lambda^2 - x_p - x_q \quad \text{and} \quad y_r = \lambda(x_p - x_r) - y_p$$

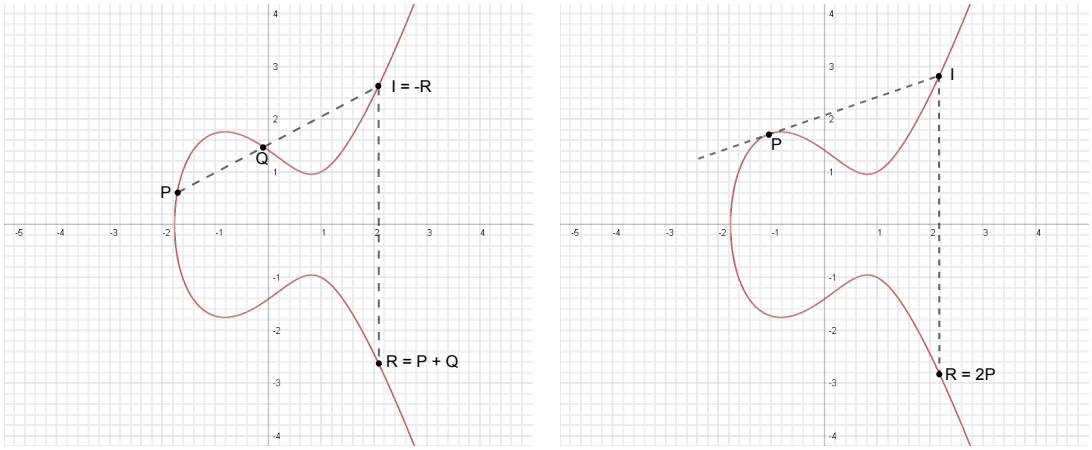
where

$$\lambda = \begin{cases} \frac{y_q - y_p}{x_q - x_p} & \text{(if P} \neq \text{Q)} \\ \\ \frac{3x_p^2 + a}{2y_p} & \text{(if P} = \text{Q).} \end{cases}$$

For binary curves in Galois fields $\mathbb{GF}(2^m)$ of the form $y^2 + xy = x^3 + ax^2 + b$ as used in this thesis, these formulae differ slightly: as explained in Sect. 2.1.1, the subtraction and addition of field elements is the same (and can be performed by xor operations). Therefore, the corresponding formulae are

$$x_r = \lambda^2 + \lambda + x_p + x_q + a \quad \text{and} \quad y_r = \lambda(x_p + x_r) + x_r y_p$$

where



(a) Point Addition (b) Point Doubling

Figure 2.2: Geometric construction of $R = P + Q$ resp. $R = 2P$.

$$\lambda = \begin{cases} \frac{y_p + y_q}{x_p + x_q} & \text{(if P} \neq \text{Q)} \\ x_p + \frac{y_p}{x_p} & \text{(if P} = \text{Q).} \end{cases}$$

Negatives Determining the corresponding negative $-P$ for a given point $P = (x_p, y_p)$, such that $P + -P = \infty$, depends on the underlying group: for $\mathbb{GF}(p)$, the negative is $-P = (x_p, -y_p)$, i.e., the point mirrored about the (shifted) x-axis, whereas for $\mathbb{GF}(2^m)$, we have $-P = (x_p, x_p + y_p)$. Note that the negative of each point is also a point of the elliptic curve.

Elliptic Curve Cryptography

In the field of modern cryptography, two forms of cryptosystems can be distinguished: symmetric or private-key and asymmetric resp. public-key cryptosystems. Both forms use secret keys and (ideally) publicly known algorithms in order to encipher information that needs to remain secret to everyone else except the legitimate persons, commonly conversation partners. Where symmetric encryption uses one shared secret key, asymmetric cryptosystems use a key pair consisting of a public and private key. Asymmetric systems generally need a comparably much longer key length, here, elliptic curves play an important role since they enable the same level of security using less key bits than for example RSA.

Asymmetric cryptosystems usually rely on some mathematically hard problem, i.e., it is infeasible to compute a solution to said problem, even with a lot of computational power[1]. One example for such a problem is the DLP: we have a finite cyclic group with

[1] At least for the computational power of today's conventional computer systems — the invention of

a prime p and a primitive element α, i.e., a generator of this group. Given a second element β, the problem is to find any integer x such that $\alpha^x \equiv \beta \mod p$. The DLP is for example used by the ElGamal encryption system: both communicating parties choose an (individual) integer i as secret key and then compute the exponentiation α^i using a publicly known generator α. The result of the exponentiation is then published as public key. Recovering the secret message is only possible for a party that is in possession of the secret exponent, otherwise the DLP needs to be solved.

One drawback of the DLP is its considerable long bit length: the security is linked to the size of the prime p which determines the number of elements in the group. Chosen too small, it is possible to recover the secret exponent by performing an exhaustive key search, i.e., try all exponents x until $\alpha^x = \beta$. Currently, the recommended key length for ElGamal is 2048 bits. Reducing this bit length without weakening the security is possible using a DLP over elliptic curves, i.e., Elliptic Curve Discrete Logarithm Problem.

The definition of the ECDLP slightly differs from the "original": given is an elliptic curve and a primitive element P, i.e., a point of the curve. Given a second curve element T, the problem is to find the integer k such that $P+P+\cdots+P = kP = T$. Note that the points of an elliptic curve (including the point at infinity) form cyclic (sub)groups, hence, the complexity of the ECDLP is connected to the amount of points the curve consists of. Hasse's theorem helps to find an estimation of curve points for an elliptic curve, stating it is approximately[2] equal to the field the curve is defined over, i.e., we expect 2^{113} curve elements for an elliptic curve over $\mathbb{GF}(2^{113})$. Therefore, the recommended key length can be significantly smaller, for example 224 bits offer roughly the same level of security as 2048 bits of RSA or ElGamal. The higher the desired strength is, the more the use of elliptic curves pays off: the security per bit increases much quicker than for the other public key systems.

2.2 Implementation Platforms

In order to test the strength of public-key cryptosystems, one approach is trying to solve the underlying mathematically hard problem in practice. There are different platforms available which offer the needed computational power, four of them are presented in this section. One approach, the reconfigurable hardware cluster as employed in this thesis, is introduced in detail.

2.2.1 Overview on Computing Platforms

This section introduces four different computing platforms and lists their advantages and disadvantages. Choosing the appropriate platform strongly depends on the requirements of the task to be executed.

quantum computers will have a massive impact on current cryptosystems as they will be able to, for example, factorize large numbers and therefore break RSA encryptions.

[2]Note that the amount of curve points must not be equal to the order of the finite field, as the ECDLP of curves with this property can be solved in linear time as shown in [SS99], and also in [SA98] and [Sem98].

CPU

A commonly used hardware to implement almost any kind of program in software is a central processing unit (CPU). Generally, a basic version consists of an arithmetic logic unit (ALU), a control unit (CU), and (general-purpose) registers. Additional units include an i/o unit responsible for the interaction with the outside world and an interrupt unit for interrupting the current program flow in order to do something else. Other special purpose units such as dedicated encryption modules as applied in recent CPUs also exist.

The ALU is a circuit which computes basic arithmetic based on integers, such as addition, substraction and multiplication, as well as logical operations like xoring or shifting. More advanced variants also offer for example a floating point unit. The CU is responsible for the sequence of operations performed by the CPU, i.e., depending on the current program, the CU determines which step to do next by decoding the current instruction. The program itself is stored in the program memory, often a read-only memory (ROM) from where the CU fetches the next instruction to be carried out, in a serial fashion. Each type of CPU has its own instruction set which includes all operations the CPU is able to execute. Depending on the project it is possible that the CPU offers a bigger variety of instructions than actual needed, but it also offers for easy adjustments and optimizations. The computational power of several CPUs can be combined in a multiprocessing platform. CPUs often provide fast access to large volatile memories (SRAM), that contain data to be processed, via a dedicated data bus. Since CPUs are manufactured in large quantities, they can be a cost-efficient solution for a given problem with a fast time-to-market.

ASIC

As opposed to a CPU, an application-specific integrated circuit (ASIC) is designed to fulfill one special task and cannot be altered later, i.e., it is manufactured for one particular purpose, while a CPU can be re-programmed by changing the program memory. ASICs can be divided in groups depending on their level of design complexity: some ASICs consists of standards cells and macro cells, i.e., smaller circuits which perform one low-level basic function. Some examples for standard cells are Boolean logic functions, e.g., AND, OR, XOR, or they serve as a storage unit, e.g., flipflop or latch. Macro cells are more complex units like an integer adder, a multiplier, or complete registers. These cells can be placed and wired such that they function efficiently and as needed for the desired task. One advantage over CPUs is that only cells which are actually of use for the design are employed. The most drastic approach is to design an ASIC from scratch, i.e., placing every transistor on the exact required position, and thereby optimizing amongst others the run-time, power dissipation, and area of the chip. This design choice is called full-custom design. The downside is that creating and designing a specialized chip is far more cost-intensive and time-consuming than buying any off-the-shelf processing unit, therefore it is only profitable if a huge amount of this type of ASIC is needed.

GPU

The graphics processing unit (GPU) is a specialized circuit designed to compute real-time 3D computer graphics, i.e., images for the output to a display, very efficiently. These graphics are based on matrix and vector calculations, which means that large blocks of data need to be processed in parallel. Therefore, a GPU offers an inherent parallel structure, consisting of thousands of small cores, as opposed to CPUs which are optimized for serial assignments. Recently, GPUs are also used for non-graphic applications which require many parallel computations or which are computationally intensive, e.g., some of the CPU's workload is outsourced to the GPU.

In cryptography, GPUs can be used to accelerate time-consuming operations such as modular exponentiation used in RSA cryptosystems or an efficient point multiplication as used for ECC-based cryptography for asymmetric cryptography, cp. [SG08], as well as for an optimization of AES as described in [MV08] and [HW07]. Still, it is difficult in practice to map a given algorithm efficiently to the parallel cell architecture of a GPU, for example GPUs are not as flexible as FPGAs when it comes to meeting requirements like low latency.

FPGA

Other than CPUs, an FPGA does not offer a fixed instruction set, but only offers very basic Boolean logic functions, e.g., OR, AND, and XOR. Multiplexers allow for various ways of routing digital signals inside the FPGA. Compared to an ASIC which is also built from low-level cells, an FPGA can be reconfigured, i.e., (almost) each cell of the FPGA can be configured to act according to the function chosen by the programmer. These transformable cells are called configurable logic cells (CLBs) for Xilinx FPGAs and are explained in more detail in Sect. 2.2.4. Apart from the CLBs, the FPGA also offers special purpose unit like BRAMs or DSPs. Anything else, e.g., adder, multiplier, and others, needs to be written from scratch — which means it is possible to design an ALU well adapted for the design and update its implementation in case that any requirements change. This flexibility makes FPGAs not as efficient as dedicated hardware but still more customizable than a CPU.

2.2.2 Software vs. Hardware Implementation

Given the overview over different implementation platforms, we have to decide which architecture serves our purpose best. As we aim to optimize an exhaustive key search to solve the ECDLP using Pollard's rho algorithm, a parallelized approach enables for running many independent instances of the key search at the same time.

A hardware implementation suits the binary target curve much better than software does: an FPGA can compute results of basic logic blocks like or-, and- or xor-gates with bit strings of arbitrary length. Hence, it is possible to build a unit which handles the for our target curve required bitlength of 113 bits in a single register. In contrast, a microcontroller is designed to operate on a fixed register size, e.g., 8 or 32 bits, thus computations involving elements with a size of 113 bits are inefficient. Comparing ASIC

and FPGA, an implementation on an ASIC would offer the best result regarding, amongst others, computation time and area consumption. But, due to the long time-to-market in combination with high development costs, using ASICs is out of scope for this thesis. Although because of its inherent parallel structure, a GPU might appear to be an attractive target platform, the toolchain of GPUs is currently not as technically mature as the toolchain offered for FPGAs by, for example, Xilinx.

Apart from practically solving the ECDLP for biggest binary elliptic curve known in the literature, implementing our attack on the hardware cluster RIVYERA — consisting of several FPGAs — enables us to revise a common rule of thumb used in the cryptographic world: to achieve the same level of security for an EC cryptosystem in comparison to a symmetric cryptosystem, the keylength of the EC cryptosystem needs twice the size of the symmetric one. Hence, our target curve using 113 bits equals 56 bits of symmetric key, which is exactly the key length of the Data Encryption Standard. Thus, we can compare the complexity of both attacks. Note that DES has been brute-forced with help of a hardware cluster similar to the RIVYERA in 6.4 days [GKN+08].

2.2.3 A Closer Look into FPGAs

Given our decision on the target platform, in this section we present a more detailed look into FPGAs. Generally, the structure of an FPGA is as follows[3]: the essential component are CLBs which are arranged in a matrix. In between these blocks, there are programmable interconnections, also known as switch matrix, which allow for free linking of different CLBs. Columns of block RAM (BRAM) and digital signal processor (DSP) resp. dedicated multiplier units are injected into the arrangement of CLBs. Surrounding the matrix of CLBs, we find input/output blocks (IOBs) to connect the FPGA to the outside world and digital clock managers (DCMs) to configure the input clock. Figure 2.3 depicts the components of an FPGA.

Each CLB consists of four slices[4] which are directly wired to each other (and the adjacent CLBs) using local routing paths and also connected to the switch matrix in order to allow for connections to distant CLBs. Inside each slice, we find — depending on the series of FPGA — two or four look-up tables (LUTs) and two resp. four storing elements. These storing elements can be programmed either as common D flip-flops (FFs) or as latches, and accept an input either directly from the CLB input or as result of the LUT. LUTs can be programmed as truth tables which store the result of a combinatorial logic function. The complexity is bound by the number of inputs into the LUT: currently, either 4 or 6 input-LUTs are common. Additionally, they can be configured as shift registers or distributed RAM in order to save FFs. Now, although each slice consists of LUTs and FFs, they can be programmed as either only logic function, only storage, or logic function with subsequent storing of the result.

[3] Note that the vocabulary is used according to Xilinx FPGAs.
[4] for the Xilinx Spartan series.

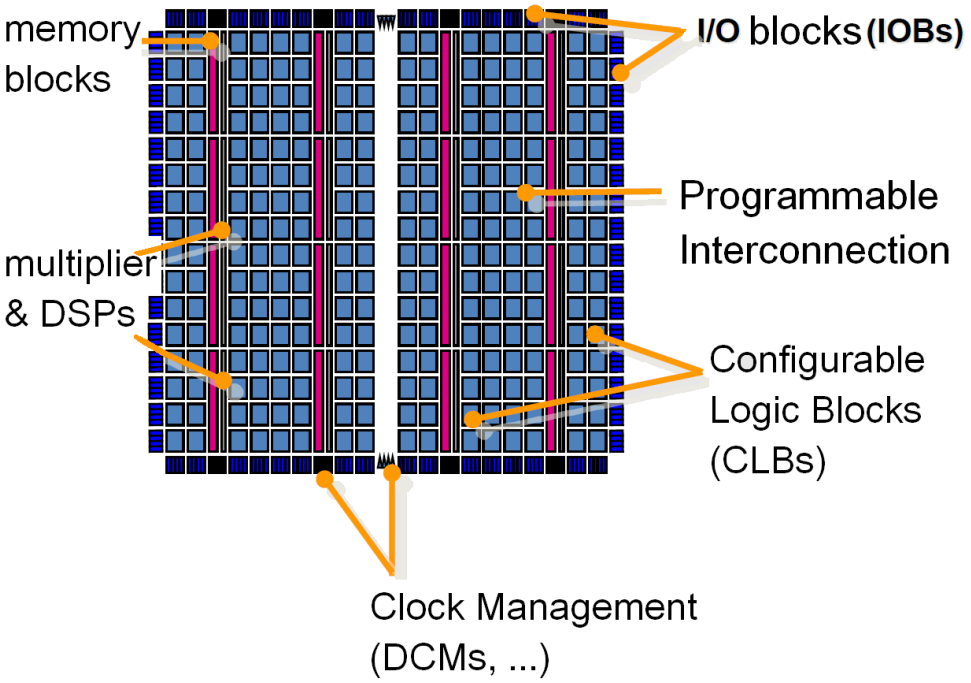


Figure 2.3: Structure of an FPGA (adapted from [GÏ1])

Critical Path

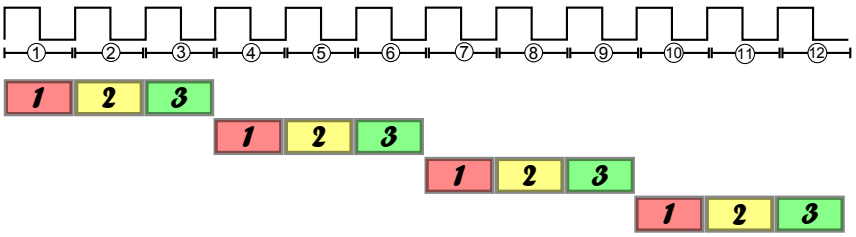
For a stand-alone LUT the output changes almost immediately — apart from the propagation delay — whenever any input values changes. This propagation delay becomes more important when we consider a path through more than one LUT in order to compute the result of a complex logical function: after the input changes at the first LUT, the propagation delay determines when the corresponding correct input for the second LUT, i.e., the output of the first LUT arrives, and so on. The final result is valid after the input values have passed through all involved LUTs, i.e., the delay among the path is defined by the sum of propagation delays of the individual LUTs (and wiring). The *critical path* is then defined as the longest path between two FFs, and determines the maximum frequency allowed for the design. Hence, reducing the length of the critical path by adding register stages into long logical paths results in a higher maximum clock frequency and often a faster design[5]. On the other hand, with additional register stages, the design uses more area of the FPGA. Accordingly, it is possible to trade area for execution time and vice versa. Thus, an important design criterion is to find the best trade-off between area and execution time for the respective project.

[5] If not the additional clock cycles resulting from the register stages outweigh the benefit of the faster clock frequency.

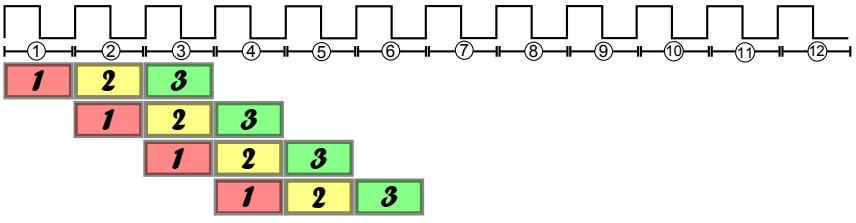
Pipelining

Inserting additional levels of FFs into the design naturally costs more slices, but also allows for a technique called *pipelining*. A pipelined structure can accelerate the throughput, if the same computation needs to be repeated over and over again and each computation is a single instance, i.e., is independent from other intermediate results. Figure 2.4 illustrates how a pipeline works: for a non-pipelined design, the processing of the next input starts after the previous computation is completed. Hence, after all three Steps $1, 2, 3$ are processed for the first input, the computations start for the next input, again through Step 1 through 3 and so on. Note that the individual steps always start at the same defined clock signal, e.g., the rising edge of a clock cycle. Our example of the sequential workflow without pipelining takes twelve clock cycles to process four inputs. Note how the units for processing Step 1-3 are only busy in every third clock cycle and are idle for two thirds of the run-time.

A pipelined architecture consists of several stages, such that the workflow can be separated into the individual steps. Whenever the computation of one of these stages finishes, the next input is instantly processed. That means that if we consider a register stage after Step 1, the processing of the next input can start while for the first input Step 2 is executed. Likewise, while the third step for the first input is processed, and Step 2 is performed for the second input, the third input is taken for processing Step 1. Hence, all four inputs depicted in Fig. 2.4 are finished already after only six clock cycles. For an FPGA implementation, the pipeline stages can be defined by the register stages, therefore, a new computation begins with each new clock cycle. The big advantage of a pipelined design is that once the pipeline is fully filled, i.e., the first computation is finished, a valid result is output in every clock cycle.



(a) Sequential workflow without pipeline



(b) Parallel workflow in a pipelined design

Figure 2.4: Processing of four inputs without pipelining (top) and with pipeling (bottom).

Design Techniques

There are two different approaches to implement an extensive hardware project: either a top-down design, where the designer concentrates on defining the overall architecture of the system where details are ignored at first. Then, once the design is completely defined, it is split up into smaller parts which are then described in more detail and finally implemented.

The other possibility is to start with the implementation of individual low-level modules, such as a simple multiplication, test them and afterward wire everything up to the final project. The latter approach called buttom-up design is the one we decided to use for our project.

2.2.4 The Reconfigurable Hardware Cluster RIVYERA

The RIVYERA[6] is a reconfigurable hardware cluster family, where each member consists of various types and numbers of Xilinx FPGAs [Sci]. The different models of RIVYERA clusters range from Spartan-3 5000 boards as a basic version over Spartan-6 LX150 boards currently up to Virtex-7 2000T. The FPGAs are interconnected by a high-throughput bus system while PCIe controller cards allow for connection to a PC, e.g., as a data base. The RIVYERA cluster with its highly parallel structure offers performance comparable to supercomputers and is, amongst others, suitable for the exhaustive search of cryptographic keys. Its predecessor, the COPACOBANA (Cost-Optimized Parallel Codebreaker) was used in 2006 to break for example DES in less than nine days[KPP+06]. In this thesis, we employ the RIVYERA with 64 Spartan-6 LX150 FPGAs.

[6] Redesign of the Increadibly Versatile Yet Energy-efficient, Reconfigurable Architecture

3 Algorithms for Solving (EC)DLPs

In Sect. 2.1.2, the relevance of ECs in the cryptographic world has been explained. In this chapter we show methods to evaluate the security of such a system, i.e., how and if it is feasible to recover the secret key. From a mathematical point of view, in our case this means solving the ECDLP. As a highly efficient method to tackle this problem, we intro we introduce the standard Pollard Rho algorithm along with various additions to make the algorithm even more effective. In Sect. 2.1.2, the relevance of ECs in the cryptographic world has been explained. In this chapter we show methods in order to evaluate the security of such a system, i.e., how and if it is feasible to recover the secret key. From a mathematical point of view, this means solving the ECDLP, therefore we introduce the standard Pollard Rho algorithm along with various additions to further optimize the algorithm.

3.1 Generic Algorithms

This section introduces generic algorithms to solve the generalized DLP $x = log_{\alpha}\beta$, as described in Sect. 2.1.2, which differ for example in execution time or necessary memory needed to store intermediate results. Generic means that they can be used on any group since they do not exploit any particular group-related properties. We summarize the *Baby-Step Giant-Step algorithm*, *Pohlig-Hellman Algorithm*, as well as the naive *brute-force search*, as described in [Paa10]. Pollard's Rho algorithm is explained in more detail in Sect. 3.2.

3.1.1 Brute-Force Search

The most naive method to find the discrete logarithm is to simply try all possible powers α^x until we reach the desired result β, generally starting at $x = 1$ and testing all following integers in ascending order. This method is known as brute-force search or exhaustive search, although usually we do not really have to test *all* possible exponents, since we stop once we reach the correct one. The correct x is expected to be found after testing on average half of all possible exponents, therefore, the execution time depends on the amount of group elements, i.e., the complexity of this method is $\mathcal{O}(\#group\ elements)$. On the plus-side, no storage is needed.

3.1.2 Baby-Step Giant-Step Algorithm

Opposed to needing no storage at all, Shank's Baby-Step Giant-Step Algorithm [Sha71] requires as many computations as it needs memory locations. As the name indicates,

the algorithm consists of two steps: the *baby-step* phase and the *giant-step* phase. The basic idea is to split the exponent such that $x = x_{giant} \cdot m + x_{baby}$. Then, the discrete logarithm can be rewritten as follows:

$$\begin{aligned} \beta = \alpha^x &= \alpha^{x_{giant} \cdot m \ + \ x_{baby}} \\ \beta \cdot (\alpha^{-m})^{x_{giant}} &= \alpha^{x_{baby}} \end{aligned}$$

Now it is possible to compute both sides of the equation separately. First, during the baby-step, a list for all possible results of the right side of the equation is stored, i.e., $\alpha^{x_{baby}}$ is calculated for all x_{baby} in $[0, m[$. Hence, m exponentiations and m storage positions are needed to complete the first phase. The giant-step then computes the left side of the equation, i.e., $\beta \cdot (\alpha^{-m})^{x_{giant}}$ with x_{giant} in $[0, m[$, and compares whether the result is equal to one of the previously stored results of the baby-step. If so, the discrete logarithm is equal to the linear combination of the corresponding $x_{baby,hit}$ and $x_{giant,hit}$, i.e., the sought-after exponent is $x = x_{giant,hit} \cdot m \ + \ x_{baby,hit}$. Note that m should be approximately the square root of the group order, $m = \lceil \sqrt{(|G|)} \rceil$, so that the exponent is halved, i.e., the baby-step giant-step algorithm can be regarded as a meet-in-the-middle approach. As a summary, this algorithm reduces the complexity of the DLP by the square root of the group order, but requires as many storage locations as computations.

3.1.3 Pohlig-Hellman Algorithm

For solving the DLP by means of the Pohlig-Hellman algorithm [PH78], the first step is to find the prime factorization of the group order, i.e., $|G| = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l}$ where p_i are the different prime factors and e_i is the corresponding quantity. The goal is now, instead of directly finding a solution for $x = \log_\alpha \beta$, to solve the DLP in the smaller subgroups of order $p_i^{e_i}$ with some $x_i \equiv x \mod p_i^{e_i}$. Once all small DLP are solved — using for example the baby-step giant-step algorithm — the Chinese Remainder Theorem allows for deducing the solution of the original DLP. Though more than one (smaller) DLP needs to be solved, it becomes computationally feasible to solve any DLP, if the prime factors of the group order are too small. In consequence, bigger groups are only as strong as their largest prime factor. The Pohlig-Hellman algorithm relies

3.2 Pollard's Rho Algorithm and Optimizations

The Pollard's rho algorithm for solving the DLP was invented by John Pollard in 1978 [Pol78]. It relies on the *birthday paradox* which states that surprisingly a group of only 70 people is needed to find a pair sharing the same birthday with a probability of 99.9 %, and 23 people[1] suffice for a 50 % probability [MVO96b]. This is a striking descent compared to the 367 people which are needed to have a guaranteed birthday collision

[1] With $n = 366$ we get $23 = \sqrt{\pi \cdot n/2}$, which is an approximation for a success rate of 50%.

(as there are 366 different birthdays available). The reason why intuitively we expect that more people are needed is that we ask ourselves how likely it is to meet someone who shares our own birthday. Note that this is a different question since one variable — our date of birth — is already fixed, i.e., the collision is restricted to one specific day and therefore indeed a lot less likely, for a probability over 50 % at least 253 people are needed.

This section introduces how Pollard's rho algorithm can be used to find a solution for the DLP over elliptic curves $P \cdot k = Q$. The general Pollard's rho attack including the algorithm is summarized as described in [HMV03c] and [Paa10], whereas the parallelized version was presented in [Tes00], [HMV03d], and [OW94]. As an optimization, [WZ98], [BLS11], and [WZ] describe the idea of using negation maps presented in Sect. 3.2.3.

3.2.1 Pollard's Rho Attack on Elliptic Curves

Using Pollard's rho algorithm for solving the ECDLP $P \cdot k = Q$, the idea is to find two distinct linear combinations of the given points Q and P such that they result in the same (new) point:

$$c\ P + d\ Q = c'\ P + d'\ Q$$

The coefficients c, c', d, d' can be chosen as (random) integers in the range of $[0..n-1]$, with $n = 2^m$ for binary curves over $\mathbb{GF}(2^m)$. Then, when a collision is found, rewriting the equation above shows how the desired factor k can be recovered: since $Q = kP$, we replace Q and get

$$\begin{aligned} c\ P + d\ k\ P &= c'\ P + d'\ k\ P \\ (c - c')\ P &= (d' - d)\ k\ P \end{aligned}$$

Solving this equation for k, we obtain the solution for the ECDLP (with high probability) as

$$k = (c - c')/(d' - d) \mod n.$$

Note that we need to keep track of the coefficients c, d for each linear combination in order to be able to compute k, hence, one (naive) possibility is to generate a table where the new point (resulting from the linear combination) as well the corresponding coefficients c, d are stored. This table is updated until the same point is reached twice, allowing for computing k. The table requires a comparatively big amount of storage, especially since three values need to be stored per iteration. The more sophisticated Pollard's rho method — which reduces said memory overhead significantly — defines

the coefficients c, d for the next test candidates not purely random but such that they can be derived from the previous point. A suitable *iterating function* f still behaves like a random function, but is periodically recurrent after the first collision is found. This form of periodical recurrence is eponymous for the algorithm: starting at some randomly chosen starting point X_0, the first series of subsequent points X_1 to X_{m-1} form just a straight line until we hit the point X_m which is the first collision. Then, every following point X_{m+i} is repeated periodically, depending on the length of the cycle i. The resulting figure with a cycle length of 8 is depicted in Fig. 3.1 and resembles the greek letter ρ.

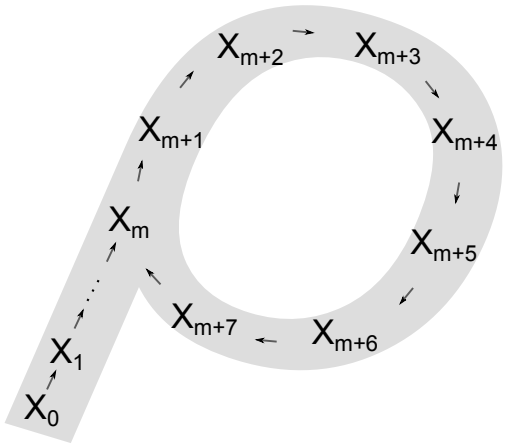


Figure 3.1: Pollard's rho algorithm forming the greek letter rho.

One example of an iterating function f is as follows: we divide the amount of curve elements, i.e., the group order, into r different subsets R_0 to R_{r-1} of approximately the same size. Then, we define a mapping criterion that decides to which subset the current input belongs. A possible definition could be to evaluate the least significant bits of the current x-coordinate, i.e., if 32 different sets exist, the last 5 bits of the x-coordinate are decisive. Each subset R_j has its own precomputed linear combination $a_j\ P + b_j\ Q$ of P and Q with randomly chosen values for a_j and b_j. The computations start with some point $X_0 = c_0\ P + d_0\ Q$, where c_0 and d_0 are again random values from $[0, n-1]$. The next point X_1 is calculated by $X_1 = f(X_0) = X_0 + a_j\ P + b_j\ Q$, i.e., the precomputed value $a_j\ P + b_j\ Q$ belonging to the subset R_j — where j is derived from the current value X_0 — is added to the current point X_0 in order to generate the subsequent point X_1. This method is referred to as *additive walk* and was initially introduced by Teske in [Tes00]. Since X_1 now has been computed without actually touching the (original) coefficients c and d, they need to be updated in a second step to $c_1 = (c_0 + a_j) \mod n$ and $d_1 = (d_0 + b_j) \mod n$, as c and d are needed for solving the DLP. Then the same steps are performed using $X_1, X_2, \ldots$ as point to define the subset R_j. The choice of the different subsets R_j in turn defines a *random walk* over the elliptic curve points, until

finally we encounter a point which has been met before, i.e., $X_i = X_j$, which allows for computing the desired value k as described above. The use of the predefined subsets R_j makes it clear that once we find the first collision, all subsequent points can only be the points which have been calculated before, since the linear combination we add according to the chosen subset stays the same. Hence, after the first collision, the points in involved in the computations stay in a cycle and form the shape of ρ.

Pollard's rho algorithm uses *Floyd's cycle-finding algorithm*, also known as "tortoise and the hare" algorithm, which needs to store only two points and their corresponding coefficients at the same time, thereby, reducing the memory overhead drastically. The idea is to compute the pair of points X_i, X_{2i} until $X_i = X_{2i}$, where the values X_i simply perform the (pseudo)random walk as described above, and by calculating X_{2i} we sneak a peek into future values. Once the computations enter a cycle, sooner or later the second value X_{2i}, which always hops several steps ahead, will again reach the slower one X_i. Floyd's algorithm states that the maximum amount of pairs needed to find the collision $X_i = X_{2i}$ is equal to the length of the cycle. As we most likely require some previous steps before entering the cycle, the length of the "tail" of rho has to be added to the length of the cycle. As a worst-case estimation, in [HMV03c] the authors estimate a number of $3\sqrt{n}$ elliptic curve group operations until a collision of two points is encountered. Algorithm 3.2.1 — taken from [HMV03c] — lists the complete Pollard's rho algorithm including Floyd's cycle-detection.

3.2.2 Parallelized Pollard Rho

The runtime of Pollard's rho algorithm, i.e., the time until we eventually find the solution for the ECDLP, depends, amongst others, on the starting point we choose: imagine we choose — luckily — a starting point which is only one step of the algorithm away from entering the cycle, or which is generating a cycle of small length. Then in this scenario we succeed faster than actually expected. If now we had the opportunity to run more than one instance of Pollard's rho at the same time, i.e., on many processing units, one idea could be to run independent computations and hope that for one of them we choose a lucky starting point. On average, finding such a starting point is quite unlikely, hence, this setup leads to a speedup of only $\sqrt{M}$, as the authors state in [HMV03d].

A different approach is presented by van Oorshot and Wiener in [OW94] and [OW99]: again, each processor starts with its own starting point and generates its own random walk, similarly to the single processor version of Pollard's rho. But, instead of using Floyd's cycle algorithm to check whether a collision is found in the walk of an independent unit, we check whether any of the processors reached a point which has already been visited by another processor. After this collision has occurred, the walks of both processors continue identically. Figure 3.2 depicts this behavior: instead of drawing a ρ, the walks form the greek letter λ once they collide[2].

In order to be able to easily check whether a collision has happened, van Oorshot and Wiener introduce the concept of *distinguished points*: during the computations, we

[2] As Pollard's lambda method for solving discrete logarithms already exists and describes a different idea, this method is still known to be Pollard's rho attack.

Algorithm 3.2.1: POLLARD'S RHO ALGORITHM FOR THE ECDLP

INPUT : $P \in$ elliptic curve E, with P of order n, $Q \in \langle P \rangle$
OUTPUT: $k = \log_P Q$

begin
  Select amount r of subsets R_j.
  Select sorting criteria h for subsets (e.g., least significant bits of current x-coordinate).
  for $j = 1$ **to** r **do**
    Select $a_j, b_j \in_R [0, n-1]$
    $R_j = a_j P + b_j Q$
  Select $c, d \in_R [0, n-1]$
  $X \leftarrow X_0 = c\ P + d\ Q$
  $X' \leftarrow X, \quad c' \leftarrow c, \quad d' \leftarrow d$
  repeat `/*Floyd's cycle finding algorithm*/`
    $j = h(X)$.
    $X \leftarrow X + R_j, \quad c \leftarrow c + a_j \mod n, \quad d \leftarrow d + b_j \mod n.$
    for $i = 1$ **to** 2 **do**
      $j = h(X')$.
      $X' \leftarrow X' + R_j, \quad c' \leftarrow c' + a_j \mod n, \quad d' \leftarrow d' + b_j \mod n.$
  until $X = X'$.
  if $d = d'$ **then**
    return *"failure"*
  else
    Compute $k = (c - c')(d' - d)^{-1} \mod n$ and **return** k

look for points which fulfill a predefined property. This DP property should be easy to test without causing much additional overhead. One possible definition could be a fixed amount of leading zeros in the x-coordinate. Once a DP is encountered, it is printed to one central list collecting the DPs of all processors. The list can be stored on a server which then also looks out for a "golden" collision, cp. Fig. 3.2, among the DPs: as soon as any DP is met for a second time, we can try to solve the ECDLP using the same formula $k = (c - c')(d' - d)^{-1} \mod n$ as for the single Pollard's rho algorithm. Hence, the central list also needs to store the corresponding linear combination, i.e., coefficients c, d, for each DP. Whenever a processor hits a DP, there are two possibilities how to proceed: either we just continue with the same walk, or reset the processor so that it starts over using a new starting point. The latter option is preferable since it avoids that a single processor runs into a cycle ("ρ") and starts outputting the same DPs over and over.

The following problems might arise in this setup: since we are no longer interested in cycle, i.e., *any* point reoccuring during one walk, but scanning only for DPs, it can happen that a walk runs into a cycle which contains no DP — which means that practically, this

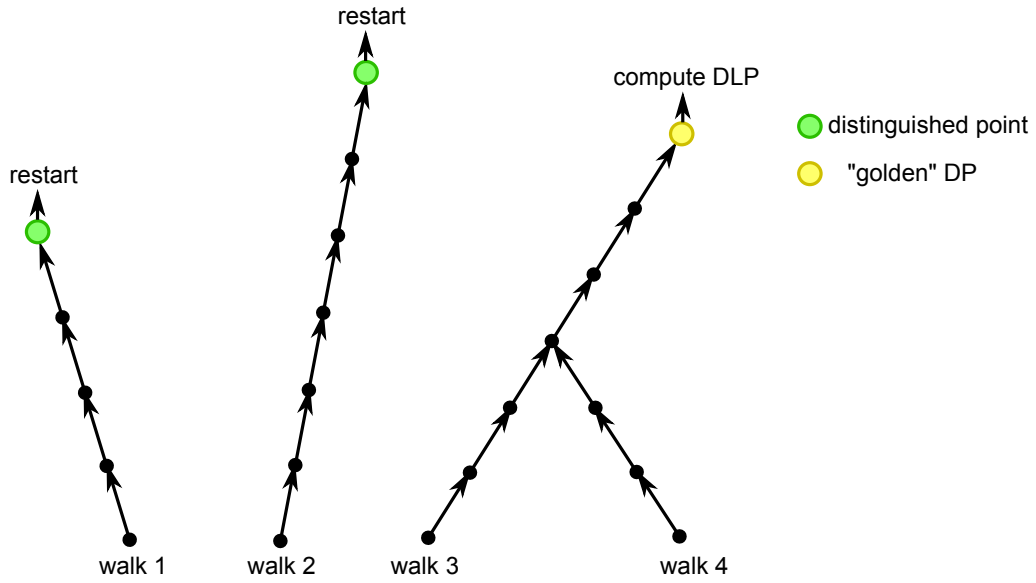


Figure 3.2: Parallel Pollard's rho: different walks look for DPs, until a collision of two walks is found.

processor no longer participates in a for the attack meaningful fashion. In order to avoid to diminish the computational power by that scenario, a maximum length of the walk can be defined, i.e., after a fixed amount of steps, the old walk will be abandoned and a new walk starts. The frequency of these *fruitless cycles* also depends on the DP criteria: if we define an easily met criteria, e.g., only 3 leading zeros instead of 10, the walks encounter DPs more often and are less likely to run into cycles. Naturally, more DPs need more storage (if we want to store all of them) and a more frequent communication with the server, hence, the DP criterion has to be chosen with care. During the computations, it might also happen that the walk of one processor hits the starting point of another processor, i.e., these two walks are not able to collide anymore (only if they run into a cycle, which we try to avoid in the first place), since the one walk will always chase the other (but never capture it). Generally, the occurrence of these "Robin Hoods" — as Wiener and van Oorshot refer to them — is negligible and, if a new random starting point is generated once we hit a DP, the second walk is independent again. Algorithm 3.2.2 gives the algorithmic representation of the parallel Pollard's rho attack on the basis of [HMV03d], again using an additive walk.

3.2.3 Using the Negation Map to Solve ECDLPs

The concept of using the negation map to further accelerate Pollard's rho algorithm has first been introduced in 1998 by Wiener and Zuccherato [WZ98]: their idea is based on the fact that for any point on an elliptic curve over $\mathbb{GF}(2^m)$ as well as $\mathbb{GF}(p)$ it is trivial to find its corresponding negative, see Sect. 2.1.2. Negation is a group automorphism of order 2 if the selecting function h is defined such that $j = h(X_i) = h(-X_i)$, i.e., the same

Algorithm 3.2.2: Parallelized Pollard's rho algorithm for the ECDLP

INPUT : $P \in$ elliptic curve E, with P of order n, $Q \in \langle P \rangle$
OUTPUT: $k = \log_P Q$

```
1  begin
2      Select amount r of subsets R_j.
3      Select sorting criteria h for subsets (e.g., least significant bits of current
       x-coordinate).
4      Select a DP property (e.g., fixed amount of leading zeros).
5      for j = 1 to r do
6          Select a_j, b_j ∈_R [0, n − 1]
7          R_j = a_j P + b_j T
       /* Each processor computes the following:                              */
8      Select c, d ∈_R [0, n − 1]
9      X ← c P + d Q
10     repeat
11     until Same DP Y is encountered for the second time.
12     if X is DP then
13         Send (c, d, X) to central server.
14     j = h(X).
15     X ← X + R_j,  c ← c + a_j mod n,  d ← d + b_j mod n.
       /* With c, d and c', d' as coefficients for the same Y:               */
16     if d = d' then
17         return "failure"
18     else
19         Compute k = (c − c')(d' − d)^{-1} mod n and return k
```

subset $R_j = R_{h(X_i)}$ is chosen for any point and its negative. Since the iterating function f, cp. Sect. 3.2.1, is chosen by the designers in the first place, it poses no problem to design it accordingly. Hence, using the equivalence classes instead of all possible points halves the space to be searched and thereby reduces the amount of steps needed to encounter a collision by a factor $\sqrt{2}$.

Compared to the parallel Pollard's rho algorithm using an additive walk, now in each step we additionally have to generate the negatives $-X_i$[3]. Then, we have to make a canonical decision $|X_i|$ whether to continue the computations with X_i or $-X_i$, for example by choosing the point with the lexicographical smaller y-coordinate (as the x-coordinate is by definition identical). After determining which point to use, the (negating) walk itself is performed similarly to the additive walk, i.e., the linear combination $a_j\ P + b_j\ Q$ of the corresponding subset is added and once we encounter any DP, it gets sent to a

[3]Therefore, in order to minimize the computational overhead, an easily computable automorphism should be chosen.

central server.

However, taking a closer look at the points which are calculated, it becomes clear that due to the negating property, more trivial fruitless cycles than before can occur. In each step, with a probability of $\frac{1}{2}$, the negative point $-X_i$ is chosen. Also, with a probability of $\frac{1}{r}$, the next point belongs into the same subset as its predecessor. If both conditions are met, i.e., with a probability of $\frac{1}{2r}$, the walk runs into trivial fruitless cycles of length 2: starting with some X_i, we have $|X_{i+1}| = -X_{i+1}$ and $h(X_i) = h(X_{i+1})$. The calculation for X_{i+2} is

$$X_{i+2} = f(X_{i+1}) = -X_{i+1} + R_{h(X_{i+1})}$$

Replacing X_{i+1} with $X_{i+1} = |X_i| + R_{h(X_i)}$ and $h(X_i) = h(X_{i+1})$ this leads to

$$X_{i+2} = -(|X_i| + R_{h(X_i)}) + R_{h(X_i)} = -|X_i|.$$

It follows that $|X_{i+2}| = |X_i|$, hence, the walk will continue to compute the same points over and over again, never reaching any distinguished points (again). A similar effect can be witnessed (less frequently) with cycles of length $4, 6, 8, \ldots$.

Wiener and Zuccherato propose following look-ahead technique to escape fruitless cycles [WZ98]: if the subsequent point X_{i+1} — computed according to the function $X_{i+1} = f(X_i) = |X_i| + R_{h(X_i)}$ — falls into the same subset R_j as X_i, i.e., one of the two criteria for fruitless cycles is valid, this X_{i+1} is discarded. Instead, a new X_{i+1} is computed, under the assumption that X_i belongs to the subset R_{j+1} (with $j+1$ reduced modulo #amount of subsets, in the paper Wiener and Zuccherato propose 20). Then, we check whether the resulting new X_{i+1} and the X_i it is derived from have the same subset. If yes, X_{i+1} is again discarded and the procedure begins anew, until we find one subset R_j where the subsets of X_i and X_{i+1} are distinct. If all subsets have been used for X_i and still each corresponding following X_{i+1} uses the same subset — which happens only with a very low probability — we just use the original subset R_j and hopefully the second requirement for fruitless cycles fails. Hence, the occurrence of fruitless cycles of length 2 is still possible, but has become rather unlikely.

Bos, Kleinjung, and Lenstra investigate this proposal for escaping cycles in [BKL10], and state that the method introduces new "2 cycles". Instead, Bos et al. introduce alternative approaches where a point doubling instead of the usual point addition is performed when subsequent points belong to the same subset. This method can also be used to avoid cycles of length $4, 6, \ldots$. Another idea — which is also used by Bernstein et al. in [BLS11] — is to occasionally keep track of the next m points and check whether the m-th point is equal to the start point. If they are equal, i.e., the computations are stuck in a cycle, we need to choose one of the recorded points as an escape point which will then be doubled and afterward the walk continues. The escape point needs to be chosen

deterministically, e.g., the minimum of all points recorded. The value for k determines which cycles can be detected, 12 will detect the most common cycles of lengths $2, 4$, and 6 while the amount of storage needed for finding a suitable escape is manageable.

3.3 Attacking ecc2-113

This section describes the actual attack approach as implemented in this thesis. We discuss a reasonable target curve and other design criteria, such as which constants to choose, for example, for the amount of subsets in an additive walk.

3.3.1 Target Curve

The two hardest ECDLPs known to be solved at present are of bitlength 112 for prime field curves in [BKK+12] and 109 bits for the binary field case, see Sect. 1.2. Additionally, as already mentioned in Sect. 1.2, a challenge to solve a 130-bit binary curve ECDLP is currently running, but has not been successful so far [BBB+09]. Hence, in this work, we intentionally only move a small step towards the unsolved 130-bit challenge in order to find the next possible bitlength to be practically broken in reasonable time. Considering the two solved bitlengths, we decide to go for a binary elliptic curve with a bitlength of 113 bits.

For the target curve, there exist two possibilities: either designing an own curve fitting our ambitions, or find some existing curve to attack. We prefer the latter option — taking a curve which has already been used and hence been recommended for usage — in order to make sure our target can be considered a real-world target instead of just being created to be attacked.

In 2000, the Certicom Corporation published a document called "SEC 2: Recommended Elliptic Curve Domain Parameters" [Res00], which enlists elliptic curve parameters for different security levels needed for various cryptographic standards. From this document, we select our target curve as the recommended binary curve of the form $E : y^2 + xy = x^3 + ax^2 + b$ over the finite field $\mathbb{F}_{2^{113}}$. The irreducible polynom is the trinom $f(x) = x^{113} + x^9 + 1$. To simplify computations, we transform the curve such that the parameter a is set to 1. Note that transforming the curve does not change the DLP, therefore the complexity of the problem is still the same as of the original curve.

3.3.2 Setup for Our Attack

For our attack, we want to perform an additive version of the parallel Pollard's rho algorithm which can easily be extended to include the negation map, which itself is not in the scope of this thesis. In a first run, we modify our target curve in a way that it falls into smaller subgroups where the ECDLP to be solved does not have the full bitlength of 113, i.e., can be solved much faster. These toy examples support the verification of our design and enable to make an estimation about the run-time of the attack on the real target curve. The results for the complete bitlength are expected to be produced only after this thesis is due.

We follow the ideas presented in [BLS11]: in order to reduce the needed storage as well as the communication overhead, we introduce *seeds* in order to keep track of the current linear combination. The idea is that instead of updating the coefficients c, d, see Sect. 3.2.1, and after every step and reducing the needed storage, each walk begins at a (different) fixed seed which is stored and, upon finishing the walk, output along with the DP. Once a collision among the DPs is found, the two corresponding walks need to be reconstructed in software to derive the coefficients for solving the ECDLP $k = \log_P Q$ with $k = (c - c')(d' - d)^{-1} \mod n$ using the respective seed.

We choose the seed such that each walk starts with a multiple of Q, i.e., $Q, 2Q, 3Q, \ldots$. Following this, the second difference to the additive walk as described in Sect. 3.2.1 is how to create the subsets R_j: since we do not care for the coefficients c, d during the walks, we do not need the linear combination for the subsets either. Instead, the value which is added in each step is some multiple of the base point P, depending on the current point processed.

Accordingly, we generate a large table consisting of values $\texttt{AES}(0) \cdot P$, $\texttt{AES}(1) \cdot P$, $\texttt{AES}(2) \cdot P$, ..., where AES(x) is the `AES` encryption of the corresponding integer with an all-zero key. Each row of the table represents one subset R_j. The table index j is derived from the least significant bits (LSBs) of the current x-coordinate (ignoring the very last bit, since it is a constant 1), e.g., for 256 subsets we need the last 8 but one LSBs. Since storage for the precomputed points is naturally a limiting factor, we decide to use 1024 subsets, i.e., the last but one 10 bits decide the subset.

We define two different DP criteria for our attack: in a first run, we set a likely property, i.e., the leading 4 bits of the x-coordinate equal to zero. This helps us to quickly find DPs and results in shorter walks where cycles are less likely. The drawback is that since significantly more DPs are found, there will be more communication overhead and inputs to be processed on the server side, points need to be compared, stored if necessary, and so on. Hence, this DP property is only applicable for the small, "toy" ECDLPs to be solved. For the real target, we choose a DP criterion of 25 leading zeros on the x-coordinate.

In order to prove that we successfully solved the ECDLP and not simply generated Q as a known multiple of base point P, we use the outputs of a hash function as starting points: with probability $\frac{1}{2}$, the result is equal to the x-coordinate of some point of the curve, i.e., we can find the corresponding y-coordinate and use it as Q. For determining P, additionally we have to verify that the point is a generator, i.e., has maximum order. Finally, we double both points in order to assure that we enter the needed prime order subgroup and have verifiable created an ECDLP without knowing the multiple k.

4 Implementation of the Pollard's Rho Processor

In this chapter, we present the implementation details of the attack. First, we briefly introduce the used toolchain, followed by the main part of this chapter. Here, we give detailed information on the implemented base components as well as their optimization and finally depict the complete design as used for our attack.

4.1 Toolchain

To program the FPGA, we used the design tools of Xilinx[Xil]. The VHDL code is written and synthesized with the help of Xilinx ISE Design Suite Version 14.3, while the corresponding ISim simulator is chosen to test the behavior of the code. The software runs on a Windows 7 64-bit machine with multiple cores and sufficient RAM to speed up the synthesis and place-and-route process. Xilinx ISE allows for generating the bitstream which is programmed into the FPGA. The following steps need to be executed to generate the bitstream: first, we run the *synthesis* of the design, i.e., verifying the code against logic errors, such as undefined signals or syntax errors, and providing a netlist of the design. Next, the *mapping* operation checks the design rules and actually fits the generated netlist into the available resources of the FPGA. Afterward, it is possible to run the *place-and-route* process which verifies the design against the chosen timing constraints and enables for finally generating the desired bitstream. In order to verify single modules, i.e., create test vectors in software and compare them to the result of the ISim simulation, we used the computational algebra system Magma, Version V2.18-8[Com]. Magma — opposed to other algebra systems — allows to create binary fields with the desired bitlength of 113 bits, which is, amongst others, needed for testing the multiplication of field elements or addition of two elliptic curve points.

4.2 Low Level Arithmetics

Looking at Pollard's rho algorithm, we can see that the main operation is the point addition on the underlying elliptic curve. Therefore, we need to implement the formulas as described in Sect. 2.1.2, i.e., modules for addition and multiplication of two 113-bit field elements as well as squaring or inverting them, and finally the reduction according to the reduction polynomial of our finite field.

4.2.1 Addition

The addition of two 113-bit binary field elements is technically for free: cost with respect to time and area of an xor of two elements is negligible compared to the other implemented arithmetic operations. Computing the result of basic logic functions such as *and*, *or*, and *xor* on an FPGAs is straightforward: the structure of an FPGA allows for efficiently computing binary operations. Since the result of the bitwise xor is still an element of the field, no modular reduction is required.

4.2.2 Multiplication

Multiplying two 113-bit binary field elements in hardware is — compared to the simple addition described above — more complex, since the FPGA does not offer multipliers for binary fields[1]. The simplest architecture for multiplying elements in binary fields is a bit-serial multiplier. Similar to the shift-and-add algorithm in Software [HMV03e], it processes the bits of one element sequentially and thus requires 113 clock cycles for a multiplication. As our target platform RIVYERA consists of FPGAs with 4-input LUTs, an algorithm which processes only one bit at a time does not exploit the hardware in an optimal way.

Instead, our multiplier of choice is the *digit-serial-multiplier*. The difference is that the digit-serial architecture in each clock cycles multiplies a full-sized factor by a *digit* of the second factor, where a digit is defined as a fixed amount of bits. As we do not know in advance which digitsize is optimal, we design the multiplier in a variable way, i.e., the chosen digitsize is one of the multiplier's input vectors. The algorithm of the digit-serial multiplier as described in [HMV03a] is shown in Alg. 4.2.1, where k denotes the digitsize and B_i a digit of bit length k.

Algorithm 4.2.1: DIGIT-SERIAL MULTIPLIER FOR $\mathbb{GF}(2^m)$

INPUT : $a = \Sigma_{i=0}^{m-1} a_i z^i, b = \Sigma_{i=0}^{l-1} B_i z^{ki} \in \mathbb{GF}(2^m)$, reduction polynomial $f(z)$
OUTPUT: $c = a \cdot b \mod f(z)$

```
1 begin
2     c ← 0
3     for i = 0 to l − 1 do
4         c ← c + B_i a
5         a ← a · z^k mod f(z)
6     return c mod f(z)
```

Taking a look at the multiplication algorithm, we can identify the necessary low-level hardware modules for a multiplication unit: first of all, we notice a looped function which, naturally for a multiplication, has two inputs and one output. Since both the input a and the output c are updated throughout the algorithm, we need to store them in registers. At the beginning of the computation, the register used to store a is loaded with the

[1] For efficiently multiplying integer values, the DSPs can be used.

coefficients a_i of input a while the one used to store c is initialized with 0. The final result $c = a \cdot b$ will be stored in register c. For input b, a static register is not the optimal choice: in each round, we need yet another chunk of the input vector b, consecutively parsing one digit after another. Instead, input b is stored in a shift register which outputs the bits of the current digit in each round.

Now, in order to get from an "empty" output register c to the correct product $c = a \cdot b$, the same round function is repeatedly executed l times, with $l = \lceil \frac{m}{k} \rceil$ being the amount of digits that input b consists of. Thus, the multiplication completes after l clock cycles.

We implement the round function as a single module by itself which is supplied with the correct inputs in each round. In each clock cycle, three main steps are performed in the round function. First, factor a is multiplied with the current digit B_i. The result is accumulated in register c. Note that due to the multiplication of B_i with length k by the factor a with length m, the register c accumulating the intermediate result must have a width of $k + m - 1$. As the third and final step of the round function, the content of register a is shifted k times and at the same time reduced with the irreducible polynomial $f(z)$. Accordingly, for register a a width of m bits suffices. As a final reduction step, after l clock cycles the content of register c is reduced by $f(z)$ to obtain the desired result of the multiplication with length m.

The input b is parsed from right to left, hence, the first round starts with the least significant digit. Accordingly, the architecture is termed least significant digit (LSD) first multiplier.

To implement the multiplication of a by B_i, we create an array with k output vectors with a size of $|a| + |B_i|$. Then, we parse digit B_i bit wise: whenever the bit is 0, we store a vector of 0 inside our array. If the processed bit is 1, we store the current value of register a surrounded by zeroes to shift it to its correct position. Thinking back to schoolbook multiplication, the zeroes on the right indicate the decimal place our result belongs to: if, for example, we actually multiply by 300, we pretend to multiply by 3 and set the two lower positions to zero. In our array all bitvectors are defined by the same bitlength $|a| + |B_i|$, hence, we need to pad the left side of the array vector with zeros for every bit of B_i but the most significant bit (MSB).[2] Once the array is filled, all lines are added up and finally added to the content of register c. Figure 4.1 depicts the structure one round module.

To complete the digit-serial multiplier, a counter is implemented to keep track which digits have been processed yet and to trigger the final reduction of the last output. Since the multiplier is part of a bigger design and might need to notify the next function once the computation is finished, the unit also gets a "done" signal as second output.

4.2.3 Squaring

As squaring is basically the same as multiplying a value with itself, one might think that implementing a dedicated squaring unit is needless. In case of a binary field squaring,

[2] Actually, we spend another zero bit at both MSB and LSB position of the array which is removed later. This is because we aim for a generic approach and it is not possible to define a non-existing zero which is needed when we multiply with MSB or LSB of our digit in VHDL.

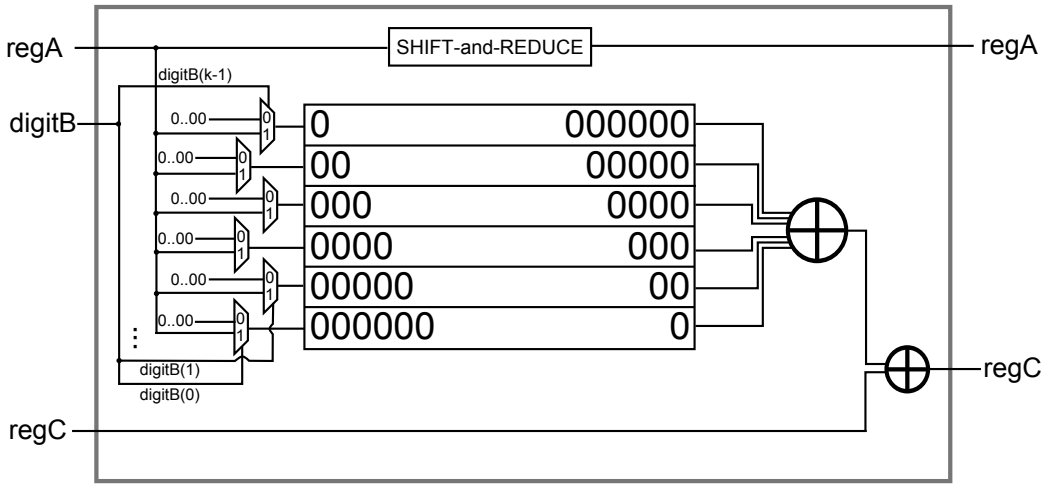


Figure 4.1: Round function of the digit-serial multiplier

multiplication is far more complex, hence, it pays off to sacrifice a little more area for a separated squaring module. Squaring a binary element is equivalent to inserting zero bits between the bits of the element to be squared, as depicted in Fig. 4.2.

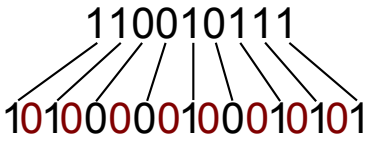


Figure 4.2: Squaring of a binary field element

In hardware, all that is needed is a bitvector twice as big as the initial value. Then, we simply assign a bit of the initial value to every even bit position, and a zero to every odd bit position, respectively. So far, the complete squaring has been achieved only by appropriately wiring the bits of registers, without the usage of extra resources. But, since we need to square binary field elements, the squared element is not part of our finite field anymore, hence, we need to reduce it accordingly which costs additional logic as described later.

4.2.4 Inversion

Taking a look at the formulas for point addition and point doubling (see Sect. 2.1.2), we notice that we need a division unit as well, with division naturally being the inverse of a multiplication. There exist different algorithms in order to find the multiplicative inverse of an arbitrary field element, such as the EEA in Sect. 2.1.1. An algorithm like this could be implemented as a single new module to provide a general inversion method, but generally inversion is a costly operation and a dedicated inverting unit might consume a lot of space of the hardware. Instead, we want to reuse modules which need to be

implemented anyway, such as the multiplier.

One idea is to implement Fermat's Little Theorem which utilizes the fact that for any field element $a \in \mathbb{GF}(2^m)$ it holds that $a^{-1} = a^{2^m-2} = a^{2(2^{m-1}-1)} \mod f(z)$. Using this equation, the multiplicative inverse can be computed using only squarings and multiplications, two units, which we implement anyway. The drawback of employing the "Square and Multiply" algorithm is that the amount of multiplications and squarings is almost equal to m, which, in our case of $m = 113$, takes too much time. As mentioned above, the time-consuming part are the multiplications, hence, we want to save as much of them as possible.

One possibility to reduce the number of multiplications is to find a so-called addition chain, which computes the same exponentiation for a fixed exponent with a minimal number of operations. Since in this thesis we operate on a fixed field in $\mathbb{GF}(2^{113})$, the prerequisite for using an addition chain is given. The optimal addition chain for computing a field element to the power of $2^{113} - 1$ is shown in Tab. 4.1. Note that the column "Computation to Generate Exponent" shows which operation needs to be done on previous variables in order to achieve the current result. It can be seen that two times an intermediate value, here c, d, need to be stored. Altogether, the efforts for the inversion are thus reduced to 8 multiplications and 128 squarings. Now, whenever the multiplicative inverse is needed, the squaring and multiplication unit are used in this particular order, hence, the additional space for realizing the inverter is kept as small as possible.

4.2.5 Reduction

There exist various possibilities how to implement the reduction of a finite field element. Usually, to speed the reduction up, the reduction polynomial is chosen either to be a trinomial $x^m + x^k + 1$ or a pentanomial $x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1$. The underlying field of the target curve is defined by the reduction polynomial $x^{113} + x^9 + 1$, which means reducing each bit with a degree of 113 or higher is done by two simple xor operations, e.g., $x^{113} = x^9 + 1, x^{114} = x^{10} + x$ and so on.

Another advantage is that the highest degree of the resulting polynomial of either squaring or multiplication (addition is trivial since the polynomial will never leave the field) is x^{224}, i.e., maximum double bitlength. In our implementation, only the squaring unit really produces an intermediate result of such size, because the digit multiplication results in a polynomial with maximum degree $k + m - 2$ (see Sect. 4.2.2).

In [PLP07], a method to reduce binary field elements with a trinomial or pentanomial reduction polynomial is explained. In their paper, the authors design a hardware unit which is able to adjust the reduction flexibly to any elliptic curve (up to a certain length), resulting in building a module for elliptic curve cryptography which is not restricted to a single curve. Although this thesis focuses only on one curve, the presented reduction offers a smart way for a full size reduction using a trinom as reduction polynomial. Figure 4.3 is taken from the paper and depicts the reduction method for a pentanom.

The reduction takes the unreduced — full size — result (of, for example, a previous squaring) as input and adds to it shifted versions of the upper half in the first step. A

Table 4.1: Addition Chain to compute the multiplicative inverse by means of Fermat's Little Theorem.

Actual Exponentiation	Computation to Generate Exponent	squarings	multiplications
$a \leftarrow x^{2^0}$	a	0	0
$a \leftarrow x^{2^1}$	a^2	1	0
$a \leftarrow x^{2^2}$	a^2	2	0
$b \leftarrow x^{2^3} - x^{2^1}$	$b \cdot a$	2	1
$a \leftarrow x^{2^4} - x^{2^2}$	b^2	3	1
$a \leftarrow x^{2^5} - x^{2^3}$	a^2	4	1
$b \leftarrow x^{2^5} - x^{2^1}$	$b \cdot a$	4	2
$a \leftarrow x^{2^6} - x^{2^2}$	b^2	5	2
$a \leftarrow x^{2^7} - x^{2^3}$	a^2	6	2
$a \leftarrow x^{2^8} - x^{2^4}$	a^2	7	2
$a \leftarrow x^{2^9} - x^{2^5}$	a^2	8	2
$b \leftarrow x^{2^9} - x^{2^1}$	$b \cdot a$	8	3
$a \leftarrow x^{2^{10}} - x^{2^2}$	b^2	9	3
$a \leftarrow x^{2^{11}} - x^{2^3}$	a^2	10	3
$a \leftarrow x^{2^{12}} - x^{2^4}$	a^2	11	3
$a \leftarrow x^{2^{13}} - x^{2^5}$	a^2	12	3
$a \leftarrow x^{2^{14}} - x^{2^6}$	a^2	13	3
$a \leftarrow x^{2^{15}} - x^{2^7}$	a^2	14	3
$a \leftarrow x^{2^{16}} - x^{2^8}$	a^2	15	3
$a \leftarrow x^{2^{17}} - x^{2^9}$	a^2	16	3
$c \leftarrow x^{2^{17}} - x^{2^1}$	$b \cdot a$	16	4
$a \leftarrow x^{2^{18}} - x^{2^2}$	c^2	17	4
$a \leftarrow x^{2^{19}} - x^{2^3}$	a^2	18	4
...	...	...	...
$a \leftarrow x^{2^{33}} - x^{2^{17}}$	a^2	32	4
$d \leftarrow x^{2^{33}} - x^{2^1}$	$c \cdot a$	32	5
$a \leftarrow x^{2^{33}} - x^{2^2}$	d^2	33	5
...	...	...	...
$a \leftarrow x^{2^{65}} - x^{2^{33}}$	a^2	64	5
$b \leftarrow x^{2^{65}} - x^{2^1}$	$d \cdot a$	64	6
$a \leftarrow x^{2^{66}} - x^{2^2}$	b^2	65	6
...	...	...	...
$a \leftarrow x^{2^{97}} - x^{2^{33}}$	a^2	96	6
$b \leftarrow x^{2^{97}} - x^{2^1}$	$d \cdot a$	96	7
$a \leftarrow x^{2^{97}} - x^{2^2}$	b^2	97	7
...	...	...	...
$a \leftarrow x^{2^{97}} - x^{2^{33}}$	a^2	128	7
$b \leftarrow x^{2^{97}} - x^{2^1}$	$c \cdot a$	128	8

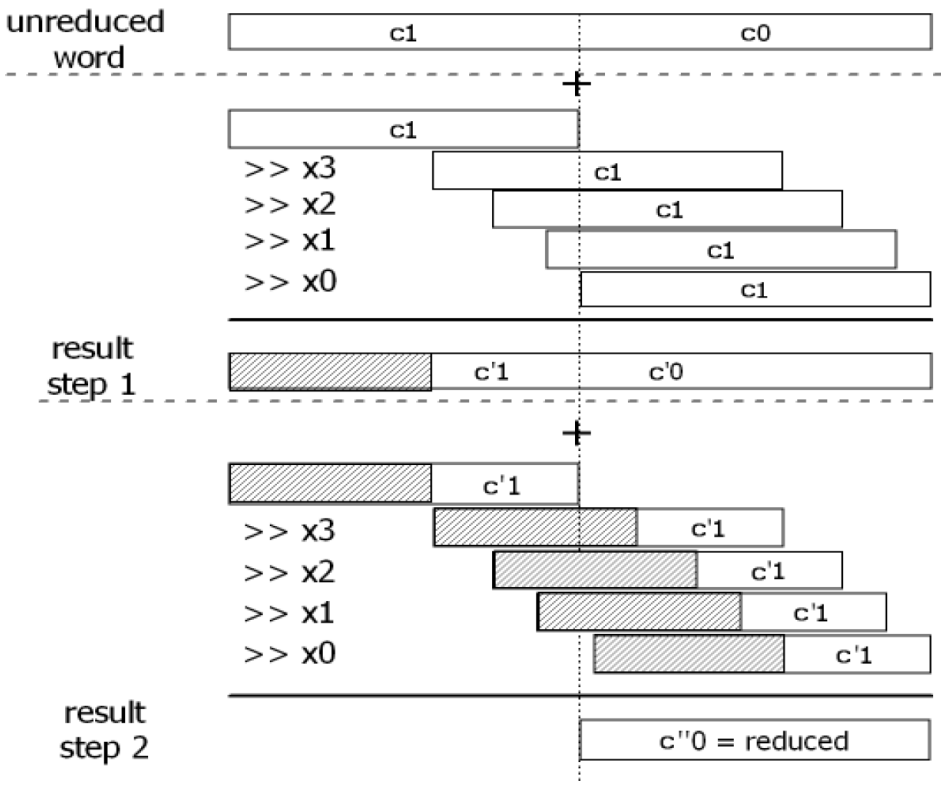


Figure 4.3: Full size reduction of a binary field element with a pentanomial as depicted in [PLP07]

second step repeats the same operations with the result of the first step. The shifts are defined by the reduction polynomial: Figure 4.3 uses a pentanomial, hence, four different shifting values can be seen. Using the trinomial $f(z) = x^{113} + x^9 + 1$, the shifting values are derived the following way: the highest degree value x^{113} is ignored, x^9 generates the first shifting value to $113 - 9 = 104$ and $1 = x^0$ gives a second shifting value as $113 - 0 = 113$. (In the picture, the remaining two shifting values are derived from the remaining two summands of the pentanomial.) Hence, for the trinomial, the reduction comprises four shifts of the value to be reduced as well as two xor operations. When implementing in hardware, it becomes clear that in our case, using a fixed reduction polynomial, the shifts are just bitvectors with a fixed amount of zeroes surrounding the upper half of the input value. The only drawback is the sequential computation: the second part can only be computed once the result of the first xor is finished. Apart from that, the reduction unit only needs the logic necessary for two (relatively big) xor operations.

4.3 Optimization of Base Components and Critical Path

Having implemented the needed arithmetic for binary 113-bit operations in a basic form, we now have a closer look at possible optimizations in order to get a fast and small design. As a fast design also requires a fast clock frequency, we need to minimize the critical path by adding register stages for intermediate results (compare with Sect. 2.2.3). The presented implementation of the digit-serial multiplier already includes registers after the computation of every round, but both addition and squaring unit are so far nothing more than a signal. One idea would be to simply surround them by registers but as they only need one respectively two (for the squaring unit combined with following reduction) levels of logic, the critical path build of one module is very short. Hence, including a register stage after each single instance would be a total waste of space. But, as shown in Tab. 4.1, for the addition chain, multiple squarings are computed consecutively, 32 at most. A good solution is, thus, to build a new module consisting of several individual squaring units and surround this one by registers. Note that the amount of square operations performed by the multiple squaring module should be a divisor of 32 so that it can be used for the addition chain at optimal cost. Testing shows that four sequent squarings offer a good trade-off between length of the critical path and space for registers. Now, with this four-times squaring unit, an overall squaring unit can be built for the addition chain where different amounts of consecutive squaring operations are needed. Figure 4.4 illustrates this module which is controlled by an input value `sq amount` that enables choosing any number of squarings needed for the addition chain (or at least to compute the input to a power of 2^{16}, to be precise, the reason is given in Sect. 4.4).

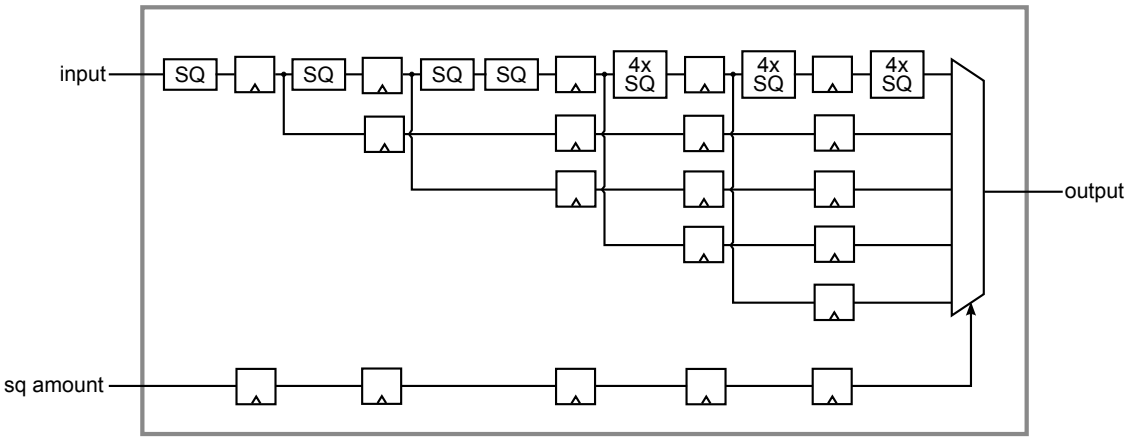


Figure 4.4: Module to compute an input value to a flexible power of 2^{16}.

If we compare the separate modules which will be wired to become the core of our Pollard's rho processor, i.e., the point addition, we can see that the multiplier consumes both most space and time to finish its computations. Hence, it would be most convenient to keep it busy at all times. The formulas for point addition, see Sect. 2.1.2, show that the multiplication and other operations, such as addition and squaring, alternate. Moreover,

all computations need to be performed in a strictly sequential way — which is most unfavorable for a hardware implementation because whenever a module is implemented it consumes space, whether it is idling or not. Thus, to occupy the multiplier as well as the other modules throughout the whole point addition, we decide to build a pipelined design. In hardware designs, a pipeline means that we perform several computations, here point additions, at the same time, and in every clock cycle each module finishes one computation, hands the results over to the next unit and simultaneously gets new input from its preceding unit. The pipeline stages are defined by the register stages, as explained in Sect. 2.2.3, here the previous result can be stored and therefore the computation split. In order to use our multiplication unit in a pipelined design, it needs slight adjustments: instead of building the multiplication around one single round function module, we now want to employ as many round functions as we have rounds respectively digits. Consequently, every round function module needs its own registers for input a and output c. Another important change to the old multiplier is that the module now receives a new input with every new clock cycle — since the multiplication itself needs more than one cycle, we need to make sure that no input needed later during the multiplication gets overwritten. As input a is only needed once at the beginning of the computation in round one, we do not have to worry about storing it.

In contrast, the digits of input b are needed throughout the multiplication, a new one in every round. Here, an update of the input in every clock cycle is fatal. As a solution, each one-round module has an shift register of different length: for the first round, the first digit of input b can directly be used and then discarded. For the following rounds, each shift register offers storage for one more digit than the previous one, and outputs one digit in every cycle. Thereby, the digit needed for the last round is shifted through its shift register until the rest of the data finally arrives at the last round function instantiation to finish the multiplication. Figure 4.5 depicts the pipelined design of the multiplication. Note that the round function module is the one depicted in Fig. 4.1.

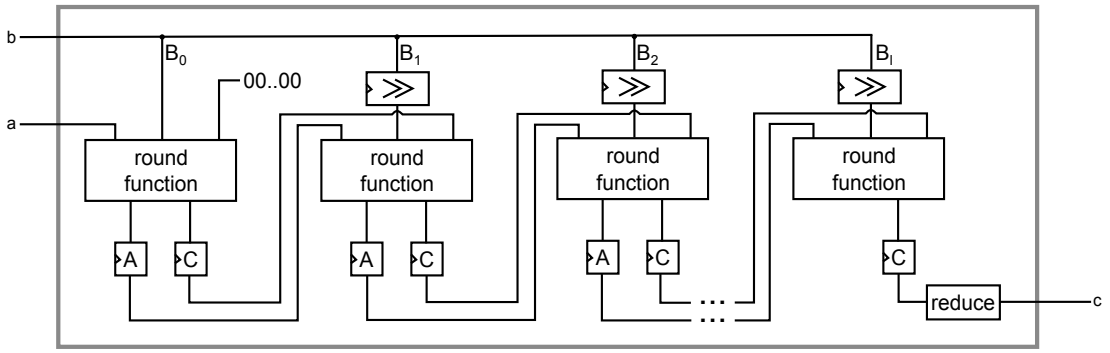


Figure 4.5: Pipelined digit-serial multiplier.

What is missing in the picture is the state machine and corresponding counter which control the multiplier and output a “done” signal once the first valid result of the multiplication is computed. This is needed since the unit outputs random data for the first rounds until the pipeline is completely filled.

4.4 Towards the Complete Design

As already mentioned above, the complete design is going to be organized as a big pipeline which can compute the needed point arithmetic. In "normal" parallelized Pollard's rho mode, only the point addition is needed, if we go for the approach including negation map, the point doubling needs to be implemented to enable leaving fruitless cycles. Recalling the formulas for point addition resp. doubling in Sect. 2.1.2, one can see that they only differ slightly:

$$x_r = \lambda^2 + \lambda + x_p + x_q + a \quad \text{and} \quad y_r = \lambda(x_p + x_r) + x_r y_p$$

with

$$\lambda = \begin{cases} \frac{y_p + y_q}{x_p + x_q} & \text{(if P} \neq \text{Q)} \\ \\ x_p + \frac{y_p}{x_p} & \text{(if P} = \text{Q).} \end{cases}$$

That suggests that in order to save space and avoid idling units, the best idea would be to build a core which is able to both compute point addition and doubling. Both operations first need to compute the new x-coordinate x_r and derive y_r from it. Computing x_r then starts by calculating λ: λ is again slightly different for addition and doubling, but both have in common that the addition chain has to be computed in order to find the multiplicative inverse. For the implementation, this means that a first step is to surround the multiplication unit by the squaring element presented in Fig. 4.4. Also, some addition units are needed to generate the correct input for the addition chain. Once λ is computed successfully, a few additional additions are needed to get x_r. y_r is characterized by one multiplication and more additions (and another squaring operation needed for the y_r of the point doubling).

The Data Path

Figure 4.6 shows the data path of the complete design. Note that the depicted units do not show the correct area occupation, the multiplier consumes far more area than a simple squaring unit. The data path can be divided by register stages depicted by the flip flop symbol. Some of these register stages are also hidden inside the arithmetic module, indicated by the clock input: multiplier as well as multiple squaring unit both already implement flip flops to store intermediate results, these also count for the pipeline. Hence, the amount of pipeline stages highly depends on the digit size resp. amount of rounds chosen for the multiplication unit. On the left side the data path starts with a level of BRAMs. The Pollard's rho processor needs to interact with the outside world — for example to reload points — and also requires additional storage for precomputed points (BRAM P) or initial input points to fill the pipeline with (BRAM Q). Using flip flops to store this overhead would go beyond the possible area of the FPGA, but to enable storing of a huge amount of data, the FPGA provides BRAMs. Also, during the point addition, there are intermediate values which are needed later on, for example λ. Due to

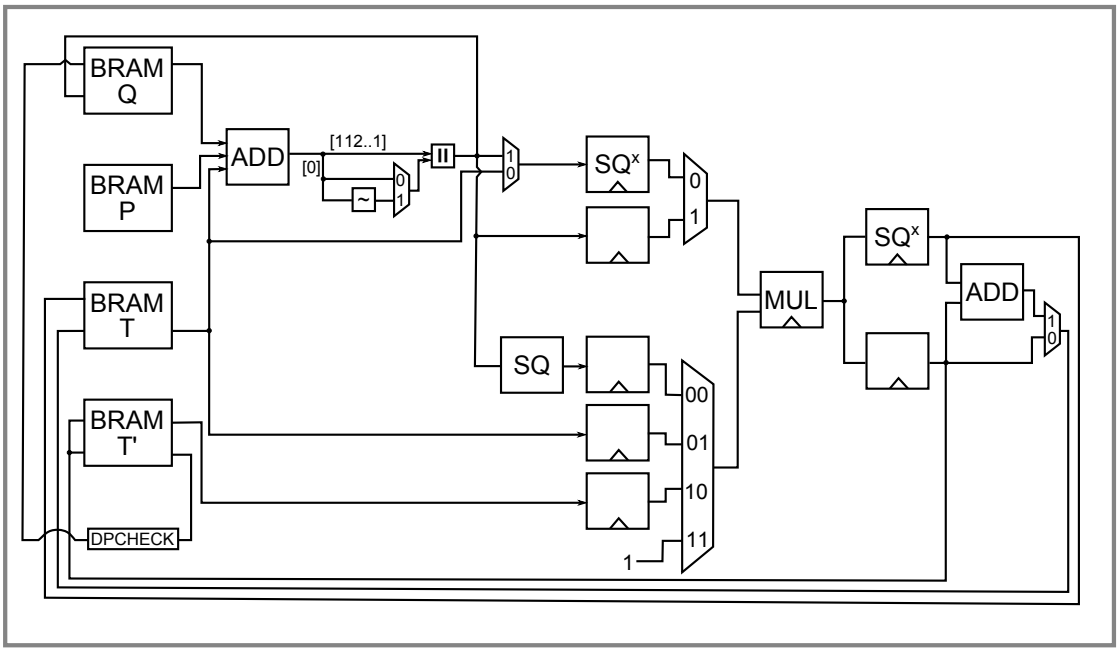


Figure 4.6: Internal data path of the design.

the pipeline, in every clock cycle a new λ is calculated, as many as pipeline stages exist — hence, more storage is needed which is provided by BRAM T and T'. The BRAMs are implemented as dual core instances, that means that both ports are enabled. Note that BRAM P is used to store the precomputed points, thus, it is never updated and needs no input ports.

Most parts of Fig. 4.6 are self-explaining and their necessity can be verified once one tries to figure out how to calculate for example the addition chain, i.e., which paths to take and where to store intermediate results. Some decisions are not as clear to understand and will be clarified in the following.

Starting on the left side of the module, three outputs of BRAM Q, P, and T serve as inputs to an addition unit. Note that the BRAM offers a reset option, if enabled the reset value is output instead of any internal value stored in the BRAM. Setting the reset value to zero allows for adding only two values of these BRAMs when needed. Subsequent to the adder follows a construct which executes the addition of curve parameter a: since we transformed the target curve such that $a = 1$, it is possible that instead of using another full-size 113 bit adder, only the LSB of the bit string is inverted and then again appended to the remaining — unchanged — bit string. Next follows the digit-serial multiplier, here different input values are chosen by two multiplexers. Because of the pipelined design, the data path always leads through the multiplication unit, even when no multiplication is needed. Thus, one of the input options is a constant "1" to be multiplied with the other input which is supposed to pass the multiplier unchanged — this is a drawback because clock cycles are wasted but it also saves further (113 bit) multiplexers. One exception is

the short cut leading back into BRAM Q: here, a result of the first adder can already be stored as intermediate value while it is also fed further through the pipeline. The logic following the multiplication unit goes without explanation, results can be stored both in BRAM T or T'.

The Instruction Set

So far, the control logic of the design has been ignored completely: in Fig. 4.6 several multiplexers can be seen, and also some of the arithmetic units or the BRAMs need additional control bits, such as the multiple squaring module which needs the amount of squarings to be performed or the memory which needs a write enable signal (amongst other signals which will be explained later on). These control bits basically decide which functionality the pipeline currently has, hence, they form the instruction set of the Pollard's rho processor. Then again a sequence of these instructions build the complete calculation of a point addition resp. doubling, or the reloading operation of new points. The instruction set can be organized as a ROM because all possible and required instructions (as not every variation of the control bits is useful) are known beforehand and thus can be hard-coded. Additionally, six necessary "higher-level" instructions, also known as opmode, can be identified: IDLE, INIT, RELOAD, POINTADD, POINTDOUBLE, and FRUITLESSCHECK which are implemented as pointers pointing to the beginning of each sequence of instructions, respectively. Figure 4.7 illustrates the organization of the instruction set and mode of operations (opmodes).

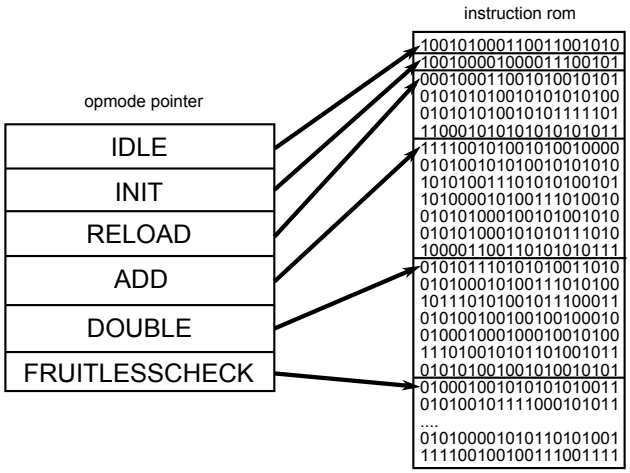


Figure 4.7: Mapping from instruction set to opmode

Most bits of the instructions derive from the BRAM: as BRAMs are one of the complex structures already part of the FPGA, they come along with predefined control bits. Figure 4.8 gives a more detailed look on the four BRAMs as depicted in Fig. 4.6. Also, an additional BRAM is needed in order to decide which one of the precomputed points stored in BRAM P to add next in the random walk: dependent on the amount of precomputed points an adapted amount of LSBs of x_r are used to address BRAM

P in the next round. The address line of the other BRAMs is controlled by a counter which counts up to the number of pipeline stages — which is equal to the amount of handled points resp. simultaneous calculations and therefore also decides when to load the next instruction. Two consecutive addresses of the BRAM — belonging to the same counter value — need to be used to store one point, since each point consists of both x and y coordinate. Another control bit put at the LSB end of the counter value decides on which position to store the result, and this bit string is then wired to the address line. The other bits are directly taken from the current instruction. Last but not least we also need to decide when to stop the algorithm, which means deciding whether we have encountered a distinguished point or not. To do that, at the end of each point addition the x coordinate of all points is compared to the distinguished point criterion and output resp. exchanged according to this test. The criterion is for example a fixed amount of leading zeros in the x coordinate and is implemented as a huge comparator. In order to know which points passed the test, a "distinguished point" flag is set in an additional register and later on the relevant points, or their x coordinates, to be precise, are sent to the database on the software side.

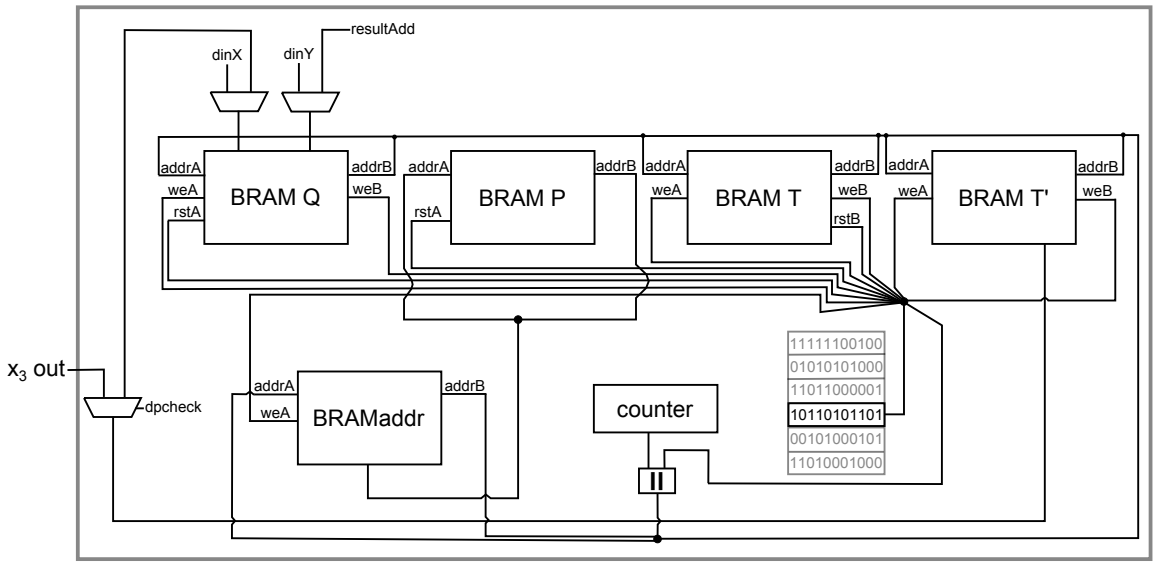


Figure 4.8: A more detailed look into the implementation of BRAMs

4.4.1 Extension Options for Future Work

The current implementation is able to perform the parallelized Pollard's rho algorithm as explained in Sect. 3.2.2, i.e., a random walk where a precomputed random point chosen according to the input point is added until some distinguished point criterion is met. Apart from the needed point addition and distinguished point check, some functionality for the negating random walk as described in Sect. 3.2.3 is already prepared for further implementation: first of all, the Pollard's rho processor is designed such that it is also able

to calculate the point doubling operation which is needed to jump out of a fruitless cycle. Since no further point additions take place until the fruitless cycle check is performed, it makes sense to use the same logic for addition and doubling. Furthermore, the negating random walk requires significantly more precomputed points than the standard one — more precomputed points lead to a smaller chance to end up in a fruitless cycle, hence less frequent fruitless cycle checks. The implementation as is already offers enough space to store a sufficient amount of precomputed points.

Mainly, two things are still missing to enable the processor to execute the negating random walk: first, we need to implement a check for the negation criteria so that depending on the previous x-coordinate we continue either with the resulting point (x, y) as is or instead use the additive inverse $(x, x + y)$, cp. Sect. 3.2.3. The negation of the point is simple, nothing more than another adder unit and an additional multiplexer as well as more wiring is needed. The other thing is to decide whether the computations entered a fruitless cycle, which is done by occasionally checking, if one of the previous results is equal to the current one. Again, the additional overhead and work load can be considered to be small. Once this is added, the processing unit can perform the doubling operation in order to escape a fruitless cycle when needed.

5 Results

This chapter summarizes the results of the implementation described in Chap. 4. Also, we show how the single instances are integrated into the hardware cluster.

5.1 Area Consumption and Timing Results of the Base Components

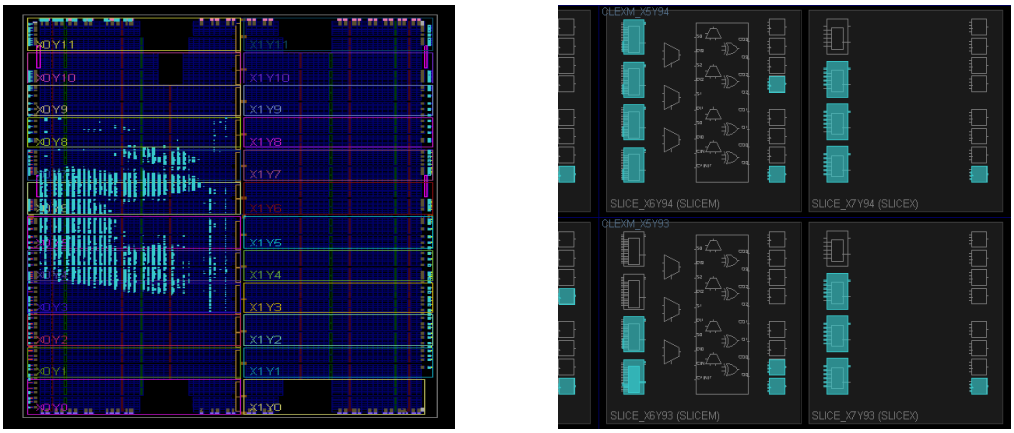
Table 4.6 shows the synthesis and place-and-route results of the (clocked) main components of the random walk unit as depicted in Fig. 4.6 and 4.8. It becomes clear that compared to counter and multiple squaring unit, the digit-serial multiplier occupies much more space on the FPGA. Figure 5.1 depicts the floor plan of the FPGA, i.e., the layout of useable logic and memory components on the chip. The light blue highlighted areas are used by our implementation of the digit-serial multiplier. Note that the allocation is done by the place-and-route tools and might look a little random and not specifically space-saving. The closeup of the slices shows that they are not completely used, some LUTs or flip flop might stay idle.

Table 5.1: Results after synthesis/place-and-route for the base components

	digit-serial multiplier	multiple squaring unit	counter
results after synthesis			
#slice registers	7481	1374	4
#slice LUTs	9887	1849	6
#fully-used LUT-FF pairs	6619	1271	0
critical path (in ns)	2.626	2.738	2.418
max. frequency (in MHz)	380.865	365.257	413.548
results after place-and-route			
#slice registers	7481	1374	4
#slice LUTs	9887	1849	6
critical path (in ns)	5.897	3.828	2.239
#occupied slices	3248	396	2

The table also emphasizes the difference between synthesis and place-and-routed results: the synthesis estimations for slice registers are accurate, as normally no further optimization can take place. Interestingly, for the number of slice LUTs the post place-and-route result is for some parts, e.g., for the multiple squaring unit, significantly smaller than the synthesis. The real critical path on the other hand is clearly bigger for multiplication and squaring unit, for the counter it is marginally smaller. Naturally,

these results depend on the design goal chosen in ISE, here, the *Balanced* optimization is chosen. One might also consider optimizing the design for either space (*Area Reduction* or *Minimum Runtime*).



(a) Placement and area of the multiplier

(b) Closeup on the utilization of the slices

Figure 5.1: Floor plan of the multiplier created after place-and-route.

The digit-serial multiplier has been designed such that the digit length can be adapted. In order to determine the optimal digit length for the given design, the random walk has been synthesized for different digit lengths. Table 5.2 lists the number of slices, flip flops and LUTs as computed by the synthesis tool as well as estimations of the effectiveness: the column "cycles" states how many rounds resp. pipeline stages are needed while "cores" holds an estimation of how many Pollard's rho processor modules fit onto one FPGA (assuming the multiplier composes half of the processor). Note that the column for clock frequency shows the result if only one digit-serial multiplier occupies the whole FPGA, i.e., it can be routed perfectly — if we fill the FPGA, this frequency will drop noticeable. The digitlength k for the digit-serial multiplier is chosen to be 9 since for this value, five cores fit onto one FPGA. Although the maximum clock frequency drops, another processing unit which computes random walks leads to a higher throughput. Note that, due to the pipelined design, the bigger the digitsize the less clock cycles are required for a multiplication and accordingly less extra register stages are needed to buffer intermediate results.

5.2 Complete Design

Analyzing the synthesis and place-and-route results of the complete design, it becomes clear that — as already assumed — the multiplication unit represents a significant part of the Pollard's rho processor core, even more than half it. Table 5.4 extends Tab. 5.1 by these results, listed for different optimization goals. Note that the optimization goal "minimum run-time" already results in a critical path of over 10 ns for one core, a result

Table 5.2: Synthesis results for a pipelined digit-serial multiplier for varying digitlengths k.

digitlength	#Slice Registers	#LUTs	#cycles	#cores	clock frequency (in MHz)
2	13387	13100	57	3	393.623
3	9394	12893	38	3	389.181
4	7481	9887	29	4	380.865
5	6043	10283	23	4	368.616
6	5295	8745	19	4	369.905
7	4807	7660	17	4	353.033
8	4383	8274	15	4	348.578
9	3784	7407	13	5	340.038

which is not desirable since the maximum frequency will be below 100 MHz, our target frequency.

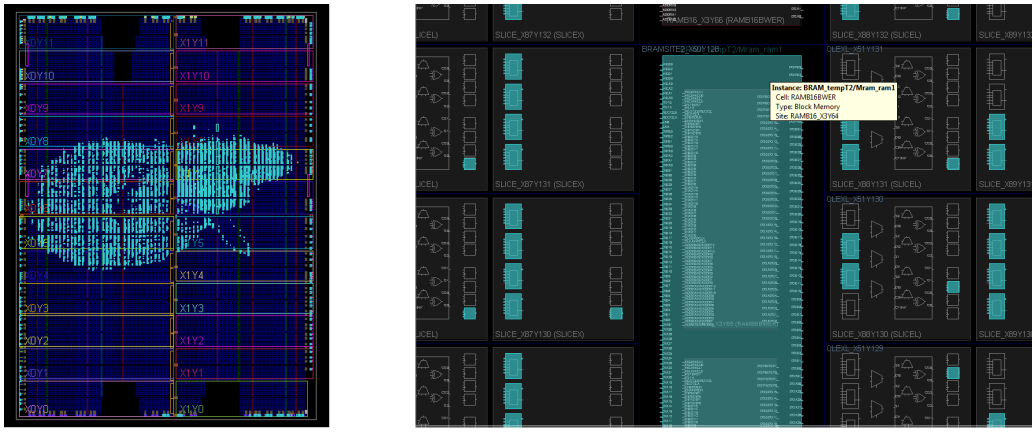
Table 5.3: Results after synthesis/place-and-route for the complete design using different optimization strategies.

	Balanced	Area Reduction	Minimum Runtime
results after synthesis			
#slice registers	9790	10115	10115
#slice LUTs	14403	15037	15037
#fully-used LUT-FF pairs	9610	10106	10106
critical path (in ns)	5.794	7.487	7.487
frequency (in MHz)	172.598	133.568	133.568
results after place-and-route			
#slice registers	9790	10115	10115
#slice LUTs	11278	11526	11526
critical path (in ns)	7.656	7.656	10.811
#occupied slices	4414	3401	3410

As one of the goals is to fit as many Pollard's rho processor units on one FPGA as possible, the bottleneck for doing so has to be identified: in our case, this is given by the amount of LUTs needed. Depending on the optimization, the random walk unit occupies 12 to 14% of the provided LUTs. Since we still need to reserve space for the communication interface and additional features, as described in Sect. 4.4.1, we estimate to put five cores to one FPGA, knowing that with further optimization (for example switching to another multiplication unit), six cores may be within reach.

Another limiting factor for our design is the amount of available BRAMs: for the negating random walk the more precomputed points can be stored the better. The balanced design occupies 26 BRAMs for 1024 precomputed points, hence, five cores simultaneously implemented use 130 BRAMs. The Spartan 6 offers 268 BRAM (of 18 kB in size, or 536 if used in 9 kB variant), thus we have some unused BRAMs left which can

be used for the communication interface. Figure 5.2 depicts how the complete random walk unit core is mapped onto the FPGA, as well as a zoom onto one of the BRAMs.



(a) Placement and area of the random walk unit

(b) Closeup on the utilization of the BRAM next to some slices

Figure 5.2: Floor plan of the random walk core created after place-and-route.

For the instruction set, a static memory is used and addressed by another memory which stores pointers to the corresponding first instruction of the sequences which build the operations, see Fig. 4.7. Currently, each instruction consists of 39 control bits, and a total of 38 instructions builds the instruction ROM. Note that the amount of control bits can be reduced in a further step since some of them have a static value throughout the instructions, but for now the storage of 1482 bit is not a limiting factor. The six different operations are encoded in the opmode ROM which uses another 30 bits.

Finally, another interesting benchmark is how long one point addition takes. In our design, this depends on different factors: one determinant is the size of the pipeline, another one is the amount of instructions until the point addition is completed. Using a digit-serial multiplier with a digitlength of 9, the random walk unit has a pipeline of 13 stages. Ignoring the initialization phase where the first round of points is loaded into BRAM Q (which is happening only once), each point addition consists of twelve consecutive instructions: eight of them are needed for the addition chain, another four for the remaining operations. Afterward follows the reload operation — one instruction — where distinguished points are output and exchanged by new points before starting the next point addition. As this instruction can be easily integrated into one of the other twelve instructions, we continue with a total amount of twelve instructions per point operation. Each instruction takes 13 clock cycles, the 13 stages of the pipeline. Doing the math, each point addition takes 136 clock cycles. Now, the benefit of the pipeline is that after these 136 clock cycles, we have calculated not only one but 13 point additions simultaneously. Hence, every individual point addition takes 12 clock cycles. Note that the amount of clock cycles does not depend on the target, i.e., whether we attack a 60-bit binary curve or a 113-bit binary curve.

To compute how many clock cycles are needed until we find the "golden" collision amongst the DPs, we need to come back to the birthday paradox introduced in Sect. 3.2. We know that — with a success rate of 50% — after $\sqrt{\pi \cdot n/2}$ steps we encounter a previously computed point for a second time. Adding the probability that we meet a DP, we get $\sqrt{\pi \cdot n/2} + \frac{1}{\omega}$, where ω is derived from the DP property: if we define the DPs to have 25 leading zeros (which is our definition for the real 113-bit target), we find the next DP after $\omega \approx \frac{1}{2^{25}}$ iterations. Hence, for one core the resulting number of clock cycles until we can solve the ECDLP of the 113-bit curve can be computed by

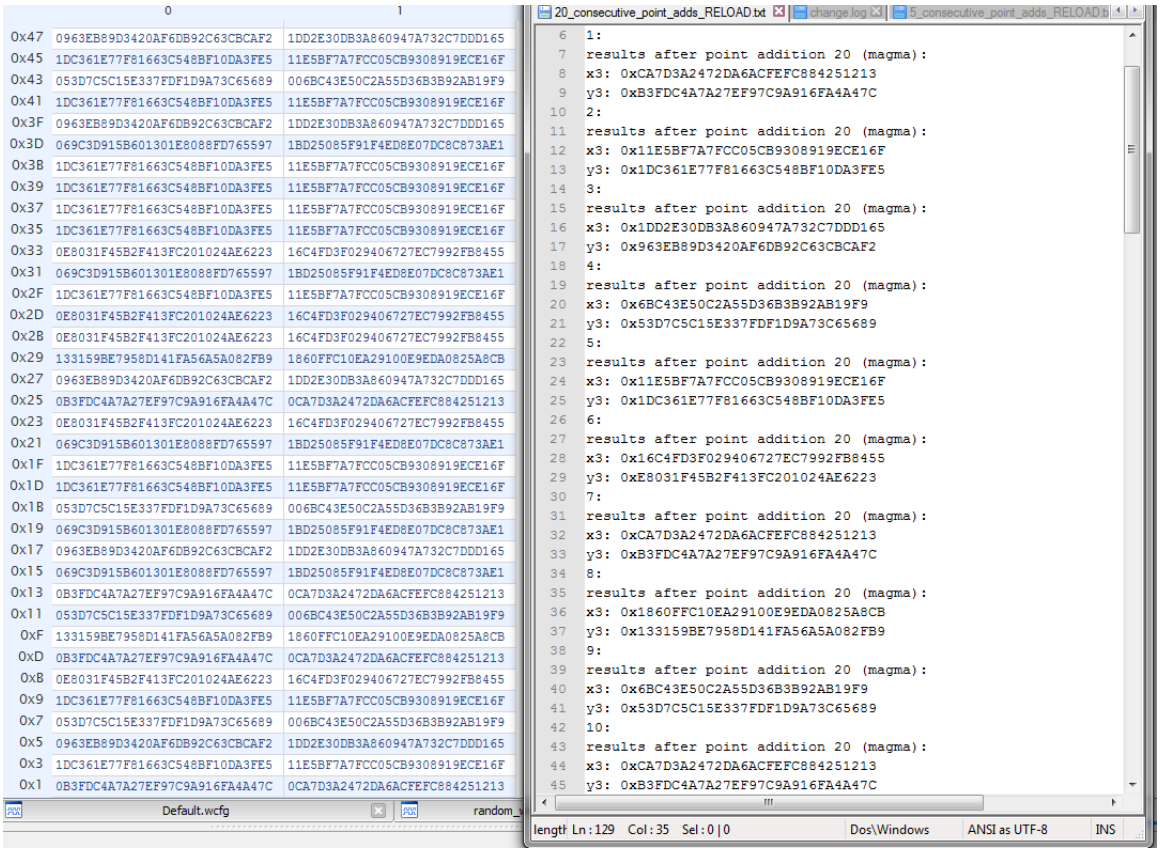
$$\begin{aligned} \#\text{clock cycles} &= 12 \cdot \sqrt{\pi \cdot \frac{n}{2}} + \frac{1}{\omega} \\ &= 12 \cdot \sqrt{\pi \cdot \frac{2^{113}}{2}} + 2^{25} \\ &= 1.533 \cdot 10^{18} \end{aligned}$$

5.3 Verification of the Design

In order to verify the correct behavior of base components as well as the complete Pollard's rho processor, Xilinx offers the possibility to use its integrated simulator ISim in combination with *test benches*. A test bench is another form of VHDL code where the designer can define test cases, i.e., the expected results the code should produce, and define outputs such as reports or failures while testing. We used the computational algebra system Magma, see paragraph Toolchain in Sect. 2.2.2, for computing test vectors in software.

Once the test bench is programmed, ISim can be launched. ISim tries to simulate the behavior of the corresponding VHDL code as if it was loaded onto the FPGA. All signals defined in the different instances can be added — in a chosen color — to the simulation waveform and hence it is possible to retrace the changes of a single signal throughout all clock cycles, see Fig. 5.3. Also, the current content of the BRAMs can be checked, although only at the corresponding point in time during the simulation, which allows us for example to verify the results of each point addition, even if they are not output as distinguished points, as shown in Fig. 5.3. Our tests show that for a number of 256 precomputed points and an input vector of 40 points, ISim correctly simulates 20 subsequent point additions according to the additive random walk: distinguished points (with a DP property of four leading zeroes on the x-coordinate) are output and exchanged by a new starting point and the value to be added is decided by the LSBs of the current x-coordinate. Having successfully verified the mathematical correctness of our design, the next step is to build a bitstream for the RIVYERA and continue testing on the hardware.

(a) Waveform of the ISim simulation of the complete design.



(b) Comparison of the contents of the BRAM with precomputed Magma results.

Figure 5.3: Two resulting screenshots of the successful simulation of the design using ISim.

5.4 RIVYERA Result

In this section, we present the results of the overall implementation executable on the hardware cluster RIVYERA. Additionally, the results of an attack on a target with a 60-bit ECDLP are given. Conclusively, we estimate the run-time our design will take to find a solution for our target curve with full bitlength of 113 bits.

Communication Interface and FPGA Utilization

After successfully verifying the behavior of a single core, the next step is to integrate our implementation into a framework which allows communication to and from the hardware cluster RIVYERA. Figure 5.4 illustrates how each of the resulting FPGAs looks like: the API receives new input points from the central server, and fills them into an *input FIFO*. Because the application programming interface (API) can receive only 64-bit blocks in each clock cycle while the FPGA expects the whole input in one clock cycle, the FIFO is needed to collect the data blocks for the complete point, i.e., a 113-bit x-coordinate and a 113-bit y-coordinate (both padded to 128 bit, i.e., two data blocks), and the seed (another 64-bit data block) belonging to the point before transferring it to one of the cores. Hence, we introduce three additional registers `X`, `Y`, and `seed`, where the next point to be processed waits until any of the cores needs to reload a new point. Additional logic makes sure that the others cores do not reload a new point until the register is again filled by the input first in, first out (FIFO). In order to output distinguished points to the central server, each core needs to be connected to the API via an output FIFO.

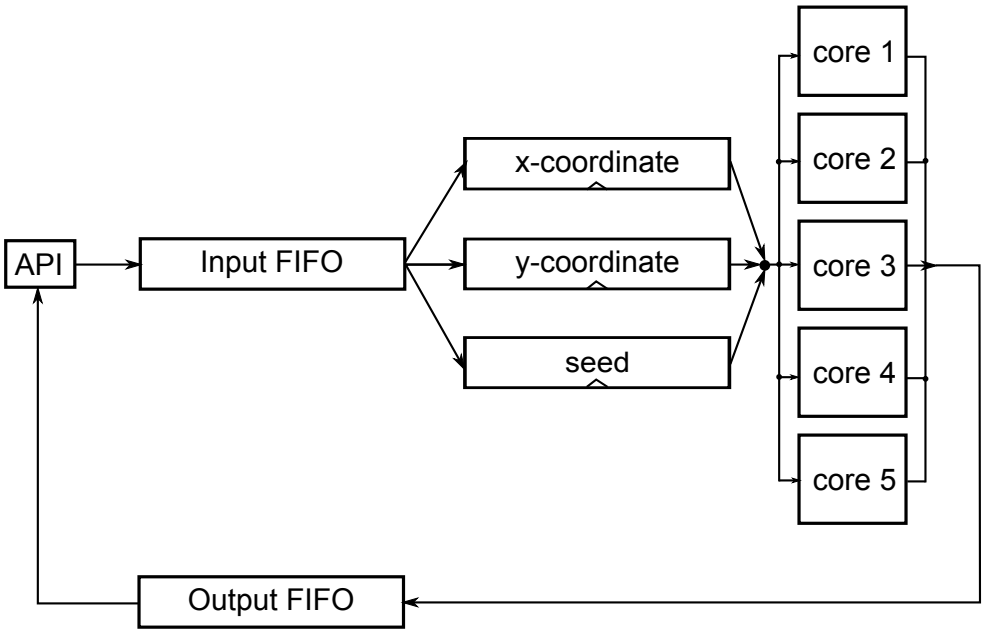


Figure 5.4: Overview on the communication of cores and API.

Table 5.5 lists the resources our design occupies on a single FPGA compared to the

resources available: as expected, with a digitsize of 9 bits, we can fit 5 Pollard's rho processors onto a single FPGA. As the column "Occupation in %" induces, adding further cores is possible with respect to the available resources, but with one major drawback: as the FPGA is filled, the routing of the single units gets more and more complicated and signals might have to cross a longer distance before they arrive at their destination. Hence, the critical path increases, leading to a smaller clock frequency. The implementation with five parallel cores allows for our favored clock frequency of 100 MHz.

Table 5.4: Utilization of a single FPGA with five Pollard's rho processing units (after Place-and-Route).

	5 cores (Placed-and-Routed)	Occupation in %
#slice registers	40876 of 184304	22.2%
#slice LUTs	55829 of 92152	60.6%
#occupied slices	21043 of 23038	91.3%
#BRAM 16B	122 of 268	45.5%
#BRAM 8B	42 of 536	7.8 %

Experimental Results using a Small ECDLP

In order to test the functionality of our implementation on the RIVYERA, we generate a smaller target curve where the complexity of the ECDLP is only 60 bits, cp. Sect. 3.3. The setup is as follows: each FPGA of the RIVYERA Formica (8 Xilinx Spartan-6 LX150) FPGAs is equipped with only one Pollard's rho processor. The amount of precomputed points, i.e., subsets, is set to 1024 and a curve point is a DP when the leading 25 bits equal zero. After approximately 15 minutes run-time, we found that the walks starting with seed 24 and seed 117 collided, i.e., all subsequent computations of the walks led to the same points. The results were send to the central server which recomputes the coefficients c, d to allow for solving the ECDLP $k = (c - c')/(d' - d) \mod n$. Computing the product $k \cdot Q \stackrel{?}{=} P$ in software verified that we successfully found the correct solution.

Estimations for the 113-bit Target

For attacking the 113-bit target, we upgrade the FPGAs to run with maximum power, i.e,. we program each FPGA with our 5-core implementation. The amount of precomputed points as well as the distinguished point property remain 1024 subsets resp. 25 leading zeros. Since performing the actual attack is still pending, for this work we will estimate the run-time of the implementation on the RIVYERA until we expect to find a collision among the DPs.

In Sect. 5.2, we already estimated that on average $1.533 \cdot 10^{18}$ clock cycles are required to solve the given ECDLP. This result is derived from the statical likelihood to find a collision, in practice, the run-time naturally also depends on other values such as number of parallel computations on the FPGAs and clock frequency. The used clock frequency is limited by the implementation to be 100 MHz, i.e., each clock cycle takes

10 ns. The amount of parallel computations is decided by the implementation — five cores per FPGA — as well as the actual amount of FPGAs in the cluster. We compare the RIVYERA Formica, consisting of the 8 FPGAs used for the small target curve, and the RIVYERA S6-LX150 hardware cluster with 64 FPGAs. Hence, the complete run-time can be computed by

$$\begin{aligned}\text{expected run-time} &= \frac{\#\text{clock cycles } \cdot \text{ cycle duration}}{\#\text{cores per FPGA} \cdot \#\text{parallel FPGAs}} \\ &= \frac{(12 \cdot \sqrt{\pi \cdot \frac{2^{113}}{2}} + 2^{25}) \cdot 10 \cdot 10^{-9} s}{5} \cdot \frac{1}{\#\text{parallel FPGAs}} \\ &= 3065250241\ s \cdot \frac{1}{\#\text{parallel FPGAs}}\end{aligned}$$

which results in the expected run-time, depending on the platform, as shown in Tab. 5.6.

Table 5.5: Expected run-time for the attack on different FPGA clusters.

RIVYERA Formica	RIVERYA Spartan S6-LX150
$\approx$ 12 years	$\approx$ 1.5 years

We expect the run-time to decrease once all optimizations, for example the negation map, are included and estimate a final run-time of approximately six months on the RIVYERA Spartan S6-LX150 with 64 FPGAs.

6 Conclusion

In this chapter, we summarize the results produced in this work and conclude their impact on cryptographic applications. We end the chapter with an overview on how the results of this thesis can be used for future work.

6.1 Summary and Further Implications

The goal of this thesis was to implement a highly-efficient processing unit to solve the ECDLP of a curve over $\mathbb{GF}(2^{113})$. Our target curve was chosen such that we can provide an attack for the full bitlength of 113, a bitlength, which has not yet been successfully attacked in practice, as of today. Instead of designing the target ourselves, we picked a curve (formerly) recommended by the Certicom Corporation, which allows for the deduction that this curve satisfied the requirements on the complexity of its ECDLP in the best possible way.

The large operands involved in the parallel computations can be best mapped to a flexible hardware architecture,thus we chose an FPGA as implementation platform, or, more precisely, the hardware cluster RIVYERA consisting of several FPGAs. For the attack, the best known algorithm for generic curves, the Pollard's rho algorithm, was implemented in its parallel variant, i.e., performing an additive random walk until a distinguished point is hit. The collision of two DPs then enables us for computing the discrete logarithm.

We developed a design which is able to compute point additions as the core operation of an additive random walk, as well as point doublings. This enables for modifying the design such that it also supports negating random walks over the underlying curve. Since both operations are inherently sequential — a property which contradicts the parallel structure of an FPGA — we developed a pipelined approach which performs the computations of 40 point operations in parallel in every clock cycle. In order to save space and fit as many Pollard's rho processing units as possible onto each FPGA in the cluster, we tried to reduce the amount of idling parts of the design. As a result, our core is built around the biggest element of the design, the digit-serial multiplication unit, which is needed for computing the inversion with help of an addition chain. The multiplier is also needed for several further steps inside the point operations, and can be used in every step of the computation.

In order to verify our design, we solved the ECDLP over a subgroup of $\mathbb{GF}(2^{113})$ which provides a security level of only 60 bits, and which could be computed experimentially in approximately 15 minutes. In total, around 115 DPs were needed until we found the "golden" collision among the list of DPs. For the attack on the real target, we estimate a

duration of 18 months, until two DPs collide and we can successfully derive the solution for the DLP. Further optimizations are expected to lead to a duration of six months.

The implications of our attack can be summarized as follows: first of all, for the time being, we can safely assume that the currently recommended minimum bitlength for ECC, 256 bits, is still secure, and also breaking elliptic curve keys of 160 bits can be considered out of reach for now. Secondly, the commonly assumed "rule of thumb" regarding the level of security of elliptic curve cryptosystems and symmetric cryptosystems, e.g., the Advanced Encryption Standard (AES) block cipher, could be updated: now, one assumes that if we want to secure 128 AES keys, a bitlength of 256 should be used for elliptic curves, i.e., the bitlength should be doubled (if we consider only bruteforce attacks). Since our 113 bit target has twice as many bits as the implementation of the block cipher DES which has been successfully bruteforced using a similar hardware cluster, we can compare the running time for both attacks: in our current version, our attack needs notedly more time than the 6.4 days which were needed to break DES on the predecessor of RIVYERA, the COPABOBANA. Since other implementations offer similar estimations, we can deduce that elliptic curves might offer even more security per bit that currently assumed.

6.2 Future Work

In order to further optimize the design, preparations to perform a negating random walk and thus speeding the attack up to find a distinguished point in less time are already done. Speeding the computations up by a factor of $\sqrt{2}$ is especially interesting when aiming to adjust the implementation for breaking the next higher level of security. Since we implemented the base components in a generic fashion, a new target could be attacked with manageable overhead.

Another optimization with regard to attacking curves with a longer bitlength is to exchange the current multiplier by a more efficient variant, for example the Karatsuba algorithm [KO63] which performs noticeably faster than schoolbook multiplication. Since the current processor is designed such that the individual modules can be exchanged, the work is limited to implementing the multiplication algorithm in VHDL.

In combination with exchanging the multiplier, another idea could be to replace the current inversion employing the addition chain by some other inverting algorithm suggested in the literature, such as Montgomery inversion [dDBQ04]. Opposed to the relatively easy exchange of the multiplier, implementing Montgomery inversion significantly changes the current design: as for now, we make a point of reducing idling units. A dedicated inverter would be used once during the computation of a single point addition and afterward wait for the next one to start. An estimation of the benefit of Montgomery inversion, i.e., computing many inversions at the same time, is left for future work.

Finally, one could imagine to fully unroll the design. This approach would probably reduce the amount of Pollard's rho processors per FPGA but might offer a much better throughput especially in combination with Montgomery inversion.

A Acronyms

ECC Elliptic Curve Cryptography

EC Elliptic Curve

ECDLP Elliptic Curve Discrete Logarithm Problem

DLP Discrete Logarithm Problem

FPGA Field Programmable Gate Array

ECDSA Elliptic Curve Digital Signature Algorithm

xor exclusive-or

DES Data Encryption Standard

AES Advanced Encryption Standard

LUT look-up table

MSB most significant bit

LSB least significant bit

BRAM block RAM

RAM random access memory

ROM read-only memory

opmode mode of operation

ECDH Elliptic Curve Diffie Hellman

nPa neuer Personalausweis

CPU central processing unit

ALU arithmetic logic unit

CU control unit

i/o unit input/output unit

ASIC application-specific integrated circuit

GPU graphics processing unit

DSP digital signal processor

CLB configurable logic cell

IOB input/output block

FF flip-flop

DCM digital clock manager

DP distinguished point

SIMD single instruction multiple data

EEA extended Euclidean algorithm

TLS Transport Layer Security

SSH Secure Shell

API application programming interface

FIFO first in, first out

LSD least significant digit

List of Figures

List of Tables

List of Algorithms

Bibliography

[BBB+09] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/`.

[BKK+12] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. *IJACT*, (3):212–228, 2012.

[BKL10] Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra. On the Use of the Negation Map in the Pollard Rho Method. In *ANTS*, pages 66–82, 2010.

[BLS11] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the Correct Use of the Negation Map in the Pollard Rho Method. *IACR Cryptology ePrint Archive*, 2011:3, 2011.

[Bun] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen H. `http://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/QES/Veroeffentlichungen/Algorithmen/2014Algorithmenkatalog.pdf;jsessionid=6788559446C0A8CB9DE8535BF8F1E37A?__blob=publicationFile&v=1`, as of February 22, 2014.

[Cer] Certicom Corporate. `http://www.certicom.com/images/pdfs/challenge-2009.pdf`, as of February 22, 2014.

[CFA+12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2012.

[Com] Computational Algebra Group. Magma Computational Algebra System. `http://magma.maths.usyd.edu.au/magma/`, as of February 22, 2014.

[dDBQ04] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater. An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA. In *FPT*, pages 441–444, 2004.

[dDBQ07] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *CHES*, pages 378–393, 2007.

[DjQL06] Guerric Meurice De Dormale, Jean jacques Quisquater, and Université Catholique De Louvain. Hardware for Collision Search on Elliptic Curve over GF(2^m). In *In "Special-purpose Hardware for Attacking Cryptographic Systems — SHARCS'06*, pages 03–04, 2006.

[GÏ1] Tim Güneysu. in: Lecture Notes "Kryptographie auf programmierbarer Hardware", 2011.

[GKN+08] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.

[HMV03a] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HMV03b] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*, chapter Finite Field Arithmetic, pages 57 – 58. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HMV03c] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*, chapter Cryptographic Protocols, pages 157 – 159. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HMV03d] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*, chapter Cryptographic Protocols, pages 160 – 161. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HMV03e] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*, chapter Finite Field Arithmetic, page 48. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HW07] Owen Harrison and John Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *CHES*, pages 209–226, 2007.

[IETa] IETF Trust, D. Stebila, J. Green. `http://tools.ietf.org/html/rfc5656`, as of February 22, 2014.

[IETb] IETF Trust, S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, B. Moeller. `http://tools.ietf.org/html/rfc4492`, as of February 22, 2014.

[KO63] Anatolii Karatsuba and Yuri Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[KPP+06] Sandeep S. Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In *CHES*, pages 101–118, 2006.

[MV08] Svetlin Manavski and Giorgio Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9(S-2), 2008.

[MVO96a] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*, page 69. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[MVO96b] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*, page 53. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[OW94] Paul C. Van Oorschot and Michael J. Wiener. Parallel Collision Search with Application to Hash Functions and Discrete Logarithms. In *In ACM CCS 94*, pages 210–218. ACM Press, 1994.

[OW99] Paul C. Van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12:1–28, 1999.

[Paa10] Paar, C. and Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag New York Inc, 2010.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. 24(1):106–110, 1978.

[PLP07] Steffen Peter, Peter Langendörfer, and Krzysztof Piotrowski. Flexible Hardware Reduction for Elliptic Curve Cryptography in gf(2^m). 2007.

[Pol78] J. Pollard. Monte Carlo methods for Index Computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[Res00] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. In *Standards for Efficient Cryptography*, 2000.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SA98] Takakazu Satoh and Kiyomichi Araki. Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves. *Commentarii mathematici Universitatis Sancti Pauli*, 47(1):81–92, jun 1998.

[Sci] SciEngines GmbH. `http://www.sciengines.com/products/`, as of February 22, 2014.

[Sem98] I. A. Semaev. Evaluation of Discrete Logarithms on Some Elliptic Curves. *Mathematics of Computation*, 67:353–356, 1998.

[SG08] Robert Szerwinski and Tim Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *CHES*, pages 79–99, 2008.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proceedings of Symposia in Pure Mathematics*, pages 415–440. American Mathematical Society, 1971.

[SS99] Nigel P. Smart and N. P. Smart. The Discrete Logarithm Problem on Elliptic Curves of Trace One. *Journal of Cryptology*, 12:193–196, 1999.

[Tes00] Edlyn Teske. On Random Walks For Pollard's Rho Method. *Mathematics of Computation*, 70:809–825, 2000.

[Wil95] Andrew Wiles. *Annals of Mathematics*, chapter Modular Elliptic Curves and Fermat's Last Theorem. Department of Mathematics, Princeton University, 1995.

[WZ] Ping Wang and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with the Negation Map.

[WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *Selected Areas in Cryptography, LNCS 1556*, pages 190–200. Springer-Verlag, 1998.

[Xil] Xilinx Inc. `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_3---14_3.html`, as of February 22, 2014.