Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelor thesis

Amin Bakhtizin

Simple Power Analysis of Binary Field Elliptic Curves

*Fakultät Technik und Informatik*
*Department Informations- und Elektrotechnik*

*Faculty of Engineering and Computer Science*
*Department of Information and Electrical Engineering*

Amin Bakhtizin

# Simple Power Analysis of Binary Field Elliptic Curves

Bachelor thesis based on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Applied Sciences

Supervising examiner : Prof. Dr. Heike Neumann
Second examiner : Prof. Dr. -Ing. Lutz Leutelt

Day of delivery August 11, 2017

**Amin Bakhtizin**

## Title of the Bachelor thesis

Simple Power Analysis of Binary Field Elliptic Curves

## Keywords

Elliptic Curve Cryptography, ECC, analysis, side channel, micro-controller

## Abstract

This paper describes an implementation of ECC algorithm over binary field and its simple side channel power analysis.

**Amin Bakhtizin**

## Titel der Arbeit

Einfache Leistungsanalyse von Binärkörper Elliptischen Kurven

## Stichworte

Elliptische-Kurven-Kryptografie, ECC, Analyse, Seitenkanal, Mikrocontroller

## Kurzzusammenfassung

Diese Arbeit beschreibt die Implementierung von ECC Algorithmus über dem Binärkörper und dessen einfache Seitenkanalanalyse

# Contents

# List of Tables

# List of Figures

# 1. Introduction

Communication (from Latin communicatio(n-), from the verb communicare "to share"[15]) can not be overrated. The ability of competent collaboration enforces the quality and efficiency of human cooperation. The result of such a workmanship reflects in todays ever-changing world deeper and faster than ever before, mainly because of the breakthroughs in several fields of science. It is difficult to imagine our everyday lives without the devices like laptops, smart-phones, smart-watches, printers, smart-TVs, etc. Almost all of these gadgets/devices in their operation modus include information exchange in a very vague meaning of this process. It can be as simple as a device sending its status over the local network or more complex which involves human interaction such as email delivery, fast message exchange (any popular messenger). The confidentiality of message exchange always was and still is one of the important aspects of communication. The history of various message conversion techniques dates back long before the common era. Hence, athirst secrecy and constantly evolving communication technologies urged the need of new field in science referred as Cryptography.

This thesis examines one of the practical implementations of public-key cryptography known as Elliptic Curve Cryptosystem[1]. It describes how ECC can be realized and verified to be functional. Furthermore, information about conduction of side channel attack (i.e. simple power analysis attack) is conveyed. Prior to implementation of an elliptic curve system, several aspects concerning realization need to be clarified, such as:

- a finite field, field element representation, algorithms performing field arithmetics
- elliptic curves, curve points representation, algorithms performing curve arithmetics

The outlined concerns were partially defined by the requirements. Hence, finite field has to be field of characteristic 2, whereas its elements are represented as binary polynomials. Exploited curves need to be named curves, which means that they have to comply with NIST[2]/SECG[3] or Brainpool[4] standards. The last stipulation in reference to implementation

[1]It was discovered in 1985 by Neal Koblitz and Victor Miller [8] and the security of its schemes is based on elliptic curve discrete logarithm problem. Known acronym: ECC

[2]National Institute of Standards and Technology is an agency of the United States Department of Commerce

[3]Standards for Efficient Cryptography Group (SECG), an industry consortium, that facilitate the adoption of efficient cryptography

[4]Standard specification for ECC Curves

was the computing platform being a microcontroller. Therefore, successful realization of the task implies implementation of algorithms performing both field and curve arithmetics coupled with verification of calculations as well as accomplishment of side channel analysis.

This thesis is structured in such a manner, that in every chapter the investigation of a single major aspect of the final accomplishment is unfolded. That being so yields the following constitution. Chapter 2 provides succinct introduction to finite fields, field arithmetics and necessary concepts for elliptic curve system. Chapter 3 reports on decisions taken in regards to tools facilitating the development process and presents the software used for verification of field and curve arithmetics implementation. It specifies the target device and describes how it simplifies execution of simple power analysis. Furthermore, it explains the actual implementation and illuminates existing intricacies of realized functions. Thorough clarification is given on employed representations of field elements and curve points. Lastly, it elucidates the validation and verification process. Chapter 4 reviews the execution of side channel attack by examining and evaluating the gathered measurements. Chapter 5 delivers a summary of the task accomplishments and suggests possible improvements. Finally, the contents of the attached CD are listed in the Appendices.

# 2. Mathematical Background

One of the main concerns of the implementation was deciding how the underlying field arithmetics for an elliptic curve system was going to be implemented. Thus, this chapter walks through certain mathematical concepts of particular interest for this work, describing in details how the filed and curve arithmetics is carried out.

## 2.1. Finite Binary Field Arithmetics

This section provides the necessary mathematical background of binary extension field, introduces the field element representation and basics of field arithmetics. For a more elaborate explanation of mentioned terms and detailed mathematical description reader can refer to literature such as [11].

### 2.1.1. Important Mathematical Concepts

Understanding binary field arithmetics is crucial, since it is the core for the techniques used in the implementation. Before defining the operations, some preliminary mathematical concepts are important to mention.

**Definition 1.** *A* ***group*** *is a set $G$ together with a binary operation $\circ$ on $G$ and it's called* ***abelian*** *if the following four properties hold:*

1. *$\circ$ is* ***associative****; that is for any $a, b, c \in G$,*

$$a \circ (b \circ c) = (a \circ b) \circ c.$$

2. *There is an* ***identity*** *(or* ***unity****)* ***element e*** *in $G$ such that for all $a \in G$,*

$$a \circ e = e \circ a = a.$$

3. *For each $a \in G$, there exists an* ***inverse element*** *$a^{-1} \in G$ such that*

$$a \circ a^{-1} = a^{-1} \circ a = e.$$

4. *The group also satisfies for all $a, b \in G$,*

$$a \circ b = b \circ a.$$

**Definition 2.** *A **field** is a set $F$ on which two binary operations, addition and multiplication are defined and which contains two distinguished elements $0$ and $e$ with $0 \neq e$. $F$, furthermore, is an abelian group with respect to addition having $0$ as the identity element, and the elements of $F$ that are $\neq 0$ form an abelian group with respect to multiplication having $e$ as the identity element. Both of the operations, namely addition and multiplication, are linked by the distributivity law*

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

*where $a, b, c \in F$ and the element $0$ is called **zero element** and $e$ called the **multiplicative identity element**.*

**Definition 3.** *A field $\mathbf{F}$ is called **finite** if there are limited number of elements in $\mathbf{F}$. Finite field or **Galois Field** has an order which is always a power of a prime $p^m$, where the prime $p$ is called the **characteristics** and positive integer $m \in \mathbf{N}$.*

A finite field satisfying the Definition (2) with prime $p$ being equal to 2 is called finite field of characteristic 2 or binary extension field. For the sake of clarification and distinction one can denote it as $GF(2^m)$ instead of $\mathbf{F}_{2^m}$.

**Definition 4.** *Binary extension field is a field $GF(2^m)$ with $m \geq 2$ and order $2^m$:*

$$GF(2^m) = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_0 | a_{0 \geq i \geq m-1} \in GF(2)\} \tag{2.1}$$

*and an element of this field is compactly represented as*

$$a(x) = \sum_{i=0}^{m-1} a_i x^i \tag{2.2}$$

### 2.1.2. Arithmetics in Binary Extension Field

Here the definitions of arithmetical operations over $GF(2^m)$ are given. It is important to notice that these definitions are theoretical and they were not directly implemented in realization. Detailed description of actual algorithms are given in corresponding chapter (see Section 3.2.1) of this thesis.

#### Addition

Addition and subtraction are the same in $GF(2^m)$, the addition of $a(x), b(x) \in GF(2^m)$ can be written in form of

$$a(x) \pm b(x) = \sum_{i=0}^{m-1} a_i x^i \pm \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) x^i, \tag{2.3}$$

where $\oplus$ stands for binary exclusive $OR$ operation, known as $XOR$.

#### Multiplication

Multiplication of two elements in binary extension field is carried out as multiplication modulo irreducible polynomial $f(x)$, which is denoted exactly as any element of the field $GF(2^m)$. Having two elements in $GF(2^m)$ such as $a(x), b(x)$, then multiplication is defined as

$$a(x) \cdot b(x) = \sum_{i=0}^{m-1} a_i x^i \cdot \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i x^i \cdot b(x)) mod\, f(x) \tag{2.4}$$

#### Squaring

The square of a field element $a(x) \in GF(2^m)$ can be computed the following way

$$a^2(x) = (\sum_{i=0}^{m-1} a_i x^i)^2 = \sum_{i=0}^{m-1} (a_i x^i)^2 = \sum_{i=0}^{m-1} a_i x^{2i} mod\, f(x) \tag{2.5}$$

This way the squaring in the field becomes a linear operation and is carried out more efficient than simply multiplying an element with itself. Both of the operations (multiplication and squaring) produce a result, which is twice as long as the element or elements of operations, therefore it should be reduced with irreducible polynomial.

### Inversion and Division

Division and inversion are closely related. Taking two elements in $GF(2^m)$ such as $a(x), b(x)$, then division $b(x)/a(x)$ is defined as

$$b(x)/a(x) = b(x) \cdot a^{-1}(x), \tag{2.6}$$

where $a^{-1}(x)$ is the unique element in $GF(2^m)$ such that $a(x) \cdot a^{-1}(x) \equiv 1 \ mod \ f(x)$ called the inverse of $a(x) \in GF(2^m)$. One of the common methods for inversion is based on extended Euclidean algorithm. For in depth description of this method reader can refer to Chapter 2, section 2.3.6 of "Guide to Elliptic Curve Cryptography" [8].

### Reduction

As it was mentioned above, multiplication and squaring operations' results require twice the size of a regular field element and therefore they need to be reduced. One of the favored choices of irreducible polynomial is the form of $f(x) = x^m + g(x)$ where the degree of $g(x)$ is small relative to $m$ and $deg(g) = k$. Thus, it allows a fast modular reduction procedure, however slightly less efficient than the one carried out using low weight irreducibles. Next, by splitting the polynomial $c(x)$, which is the result of aforementioned operations into $c(x) = c_H(x) + c_L(x)$ with $deg(c) = m + t$, where $c_H(x) = \sum_{i=m}^{t} c_i x^i$ and $c_L(x) = \sum_{i=0}^{m-1} c_i x^i$. The reduction of $c(x)$ is then derived as

$$c_{Red}(x) = c_L(x) \oplus c_H(x) \cdot g(x). \tag{2.7}$$

Derived polynomial $c_{Red}(x)$ has degree $deg(c_{Red}) = max(m - 1, t + k)$. The operation is applied recursively until $deg(c_{Red}) < m$.

## 2.2. Introduction to Binary Elliptic Curves

This section provides the introductory information about elliptic curves, familiarizes the reader with arithmetics involving the curve points. There is vast amount of literature on this topic, owing to the fact that elliptic curves come to light in many branches of mathematics. For a throughout introduction to the theory of elliptic curves see the Chapter 3 of Blake and Seroussi's book [1] or refer to the book by Hankerson and J. Menezes [8].

### 2.2.1. General Elliptic Curves

**Definition 5.** *An **elliptic curve** $E$ over a field* **F** *is defined as follows*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.8}$$

*where $a_1, a_2, a_3, a_4, a_6 \in F$ and $\Delta \neq 0$; $\Delta$ is the **discriminant** of $E$ and is defined by equations:*

$$\left.\begin{aligned} \Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 &= a_1^2 + 4a_2 \\ d_4 &= 2a_4 + a_1a_3 \\ d_6 &= a_3^2 + 4a_6 \\ d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{aligned}\right\} \tag{2.9}$$

Equation (2.8) is known as **long Weierstrass form** which describes curve $E$ defined over field **F**; condition $\Delta \neq 0$ enforce that there are no points at which the curve can more than one tangent line. If $G$ is an extension of field **F**, then the set of $G - rational\ points$ on $E$ can be written in the form of

$$E(G) = \{(x, y) \in G \times G : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\mathcal{O}\}$$

where $\mathcal{O}$ is **point at infinity**.

#### Simplified Weierstrass Equation

Weierstrass equation defined in form of Equation (2.8) can be significantly untangled by a transformation which exploits the characteristic of elliptic curve known as **isomorphism**. Mathematically it is described in the following form:

**Definition 6.** *Let $E_1$ and $E_2$ be elliptic curves over* **F** *and defined by Weierstrass equations*

$$\begin{aligned} E_1 &: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \\ E_2 &: y^2 + \bar{a}_1xy + \bar{a}_3y = x^3 + \bar{a}_2x^2 + \bar{a}_4x + \bar{a}_6 \end{aligned}$$

*they are said to be isomorphic over* **F** *if there exist $u, r, s, t \in F, u \neq 0$, such that the change of variables*

$$(x, y) \rightarrow (u^2x + r, u^3y + u^2sx + t) \tag{2.10}$$

*transforms $E_1$ into $E_2$. The transformation (2.10) itself is called **admissible change of variables**.*

Now, let **F** be a finite field of characteristic 2 and $a_1 \neq 0$, then the admissible change of variables

$$(x, y) \rightarrow (a_1^2 x + \frac{a_3}{a_1}, a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3})$$

transforms Equation (2.8) into

$$y^2 + xy = x^3 + ax^2 + b \tag{2.11}$$

where $a, b \in F$. This curve is called to be **non-supersingular** and has a discriminant $\Delta = b$. The case of $a_1 = 0$ is equivalent to the curve being super-singular , this very special type of curve is avoided in cryptography due to MOV[1] attack.

#### Elliptic Curves over $GF(2^m)$

With the background from previous sections, the mathematical definition of elliptic curve over binary extension field can be given. Thus, following holds:

$$E/GF(2^m) = \{(x, y) \in GF(2^m) \times GF(2^m) : y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\} \tag{2.12}$$

where $a, b \in GF(2^m)$ are constants, $b \neq 0$ and $\mathcal{O}$ is the point at infinity. Curves with $a \in \{0, 1\}$ and $b = 1$ are called Koblitz curves, otherwise they are characterized as random curves.

## 2.3. Elliptic Curve Arithmetics

This section provides necessary information regarding fundamental concepts for elliptic curve arithmetics.

### 2.3.1. Group Law

The law can be defined by a simple statement that three points on the curve will sum to zero if and only if they lie on a straight line. Based on this statement explicit algebraic formulas for elliptic points arithmetics can be defined. Before obtaining them the comprehensive denotation of the group law would be appropriate to mention.

[1]The MOV attack is named after Menezes, Okamoto and Vanstone (1993). It is a known algorithm which uses Weil pairing to convert DLP in $E(F_q)$ to one in $F_{q^m}^*$. For brief explanation refer to [13].

**Definition 7.** *Let $E/\mathbf{F}$ then there is a* ***chord-and-tangent rule*** *for two points addition in $E/\mathbf{F}$ to give a third point in $E/\mathbf{F}$. This operation together with the set of points $E/\mathbf{F}$ form an abelian group with $\mathcal{O}$ serving as its identity. To add a point to itself tangent to the curve at this point is taken.*

Operations in Definition (7) are best explained geometrically. Assume that $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are two different points on the curve $E/\mathbf{F}$.

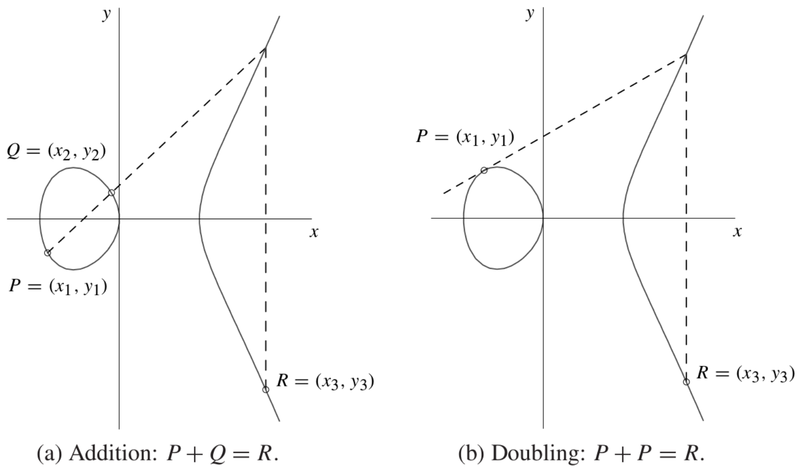


Figure 2.1.: Adding and doubling elliptic curve points ("Guide to Elliptic Curve Cryptography" Chapter 3, page 80 [8])

The sum $R$ of points $P$ and $Q$ can be found by drawing a line through the points. This line intersects the curve at the third point and the reflection of this point about the x-axis will be $R = (x_3, y_3)$. This can be seen in Figure 2.1$a$. Figure 2.1$b$ depicts adding a point to itself, which is performed by drawing a tangent at point $P$. The tangent line crosses the curve at the second point and x-axis reflection of it is the result of operation $R = (x_3, y_3)$.

Thus, the algebraic formulae of the group law for non-supersingular elliptic curve $E$ defined over field $\mathbf{F}_{2^m}$ with $\mathcal{O}$ as identity, i.e. $P + \mathcal{O} = \mathcal{O} + P = P$ for $\forall P \in E/\mathbf{F}_{2^m}$ and with negative of $P$ denoted as $-P$ [2] is defined in the following form for addition

$$\left.\begin{aligned} \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 y_1. \end{aligned}\right\} \tag{2.13}$$

[2] if $P = (x, y) \in E/\mathbf{F}_{2^m}$, then $-P = (x, x + y)$ since $(x, y) + (x, x + y) = \mathcal{O}$

Adding point to itself, or simply **point doubling** $R = 2P = (x_3, y_3)$ is denoted as

$$\left.\begin{aligned} \lambda &= x_1 + \frac{y_1}{x_p} \\ x_3 &= \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \\ y_3 &= x_1^2 + \lambda x_3 + x_3. \end{aligned}\right\} \tag{2.14}$$

Reader can find comprehensive explanation of group law definition and in-depth derivation of Equations (2.13),(2.14) at the Chapter 3, Sections 3-4 of book by I.F. Blake, G. Seroussi and N.P. Smart [1].

### 2.3.2. Discrete Logarithm Problem

The **point** or **scalar multiplication** is the core of cryptosystem based on elliptic curves over $\mathbf{F}$. The point multiplication is an equation of the form

$$Q = [k]P = \underbrace{P + P + \cdots + P}_{k \text{ times}} \tag{2.15}$$

where $k$ is an integer and $P$ is a point on the elliptic curve $E$ with underlying field $\mathbf{F}$. The strength of the system is characterized by the fact that with a known curve, a given point $P$ (studied a priori or arbitrary) and $[k]P$ it is hard to impossible to retrieve $k$. This is known as the elliptic curve discrete logarithmic problem.

There are numerous well-known algorithms to efficiently perform scalar multiplication, in this thesis multiply-and-add algorithm is used for this purpose. These algorithms exploit various aspects of elliptic curves' structure. Moreover, several possibilities of point representation can significantly boost the computational time. For detailed descriptions of such algorithms one can consult dedicated literature [16], [8], [1].

# 3. Realization of Elliptic Curve Cryptography

Several issues arise when one investigates the problems in realization. One such problem is the technique of number representation. The traditional data types offered by C programming language addressing the aforementioned issue are limited either to 32 or 64 bits. Implementation of the underlying logic for memory allocation and deallocation, efficient referencing and bitwise relations in order to implicitly overcome this constraint is cumbersome and immensely elaborate. Therefore, corresponding Section (see 3.1.1) describes a possible solution, which utilizes external library. Further investigation of realization requirements shows that there are other problems to be addressed as well. Thus, Section 3.1.2 covers proposals for concerns related to microcontroller used for realization and third-party tools facilitating the implementation (see 3.1.2). The validation of the implemented system is one of the important aspects of this thesis. Section 3.1.3 reviews the software, which was solely used for this purpose.

Section 3.2 details software realization of ECC by providing the reasoning for methods and algorithms implementing the underlying logic. Therefore, detailed description, corresponding Nassi-Shneidermann diagrams and comprehensive explanation of implementation for each mathematical operation over binary field (e.g. multiplication, division, addition, etc) can be found in chapter 3.2.1. Section 3.2.2 looks into elliptic curve operations. To be specific, it walks through the granular components of point addition and doubling operations, providing an insight into factual constitution of those. Lastly, the dedicated Section 3.3 reports on strategy used for testing the correctness of all calculations, which is a bottom-up approach verifying small pieces of computation to establish a solid foundation for complex elliptic curve cryptosystem.

## 3.1. Technical Analysis

### 3.1.1. FLINT Library

FLINT/C (Functions for Large Integers in Number Theory and Cryptography)[9] is a library for calculating with large numbers. It is the core element of software realization for this thesis. It offers numerous modules for arithmetics with numbers, polynomials, power series and matrices over multi-precision integers/rationals, real and complex numbers as well as finite fields. Nonetheless, due to limitations coming with microcontroller's architecture most of the modules offered by FLINT are unavailable. It heavily depends on the MPIR[1]/GMP[2] and MPFR[3] libraries. All the package modules in the library are optimized for x86 and x86-64 CPUs. Thus, building the complete library from the source code for ARM architecture was not an option. It might be theoretically possible by setting up a custom cross-toolchain[4] explicitly for Cortex-M3 (see 3.1.2), but this approach was proved to be extremely complex hence out of scope for this work.

With points outlined above the usage of FLINT/C was trimmed to memory management and bitwise logical operations (it includes left/right shift operations, bitwise relations, direct access to individual bits as well as comparison operations).

### 3.1.2. Microcontroller and tools

Using a microcontroller as a target device comes with a trade-off between ease of software implementation of ECC logic and execution of simple power analysis (i.e. direct access to peripherals of a board). As it was mentioned in Section 3.1.1 a microcontroller is not capable of running FLINT library with all features that it offers, if one wants to avoid extra work. Thus, having LPC1769 provided, this section will present introductory information regarding the chip architecture, peripheral complements of the board for power analysis and development tools.

[1]Highly optimized library for bignum arithmetic based on GMP library. [10].

[2]A library for arbitrary precision arithmetic used for cryptography applications and research, Internet security, computational algebra research, etc. [7]

[3]C library for multiple-precision floating-point computations with correct rounding [6]

[4]Cross-toolchain (cross-compiler) is a compiler which is capable of converting instructions into machine code for a device other than that on which it is running. For more detailed information reader can refer to [3]

#### LPC1769

LPC1769 is an ARM Cortex-M3 based microcontroller by NXP for embedded applications, which can operate at CPU frequencies up to 120 MHz. The ARM Cortex-M3 uses Harvard architecture with three dedicated buses for instruction, data and peripherals. Current consumption measurement on CPU can be directly carried out by taking probes on dedicated pins (denoted as J7 in Figure 3.1). This is described in Chapter 4, Section 4.1, which goes through the necessary setup and equipment for this explicit task. The detailed information of all the features available on the board reader can find in User Manual for LPC176x/5x series [14]. An important facet of LPC board is the JTAG (LPC-Link) debug interface, which makes the development convenient, since it is supported by numerous development environments, to name a few uVision from Keil/ARM, Embedded Workbench from IAR and LPCExpresso IDE.

#### LPCXpresso IDE and Git

The realization code base is large enough[5] and would be inadequate to maintain it with a simple text editor lacking version control. To deal with this difficulty LPCXpresso IDE coupled with Git was used. LPCXpresso IDE offers a fully featured environment for development, it has great compatibility with LPC-Link as well, which is of much importance. It is available for Linux, however it has to be started via a script to ensure that all the user interface components are visible. Git is a distributed version control system used to track the changes to source files. In order to familiarize themselves with the concept of version controlling in software development, readers can refer to specific literature such as "Pro Git" by S. Chacon and B. Straub [2].

### 3.1.3. Verification of implementation

Evincing the validity of computations is essential. For this cause free open-source mathematics software system SageMath was used. It is based on numerous packages, some of them were already mentioned in section related to long integer representation (see Section 3.1.1). It should be mentioned that one of these underlying packages is FLINT. The software was installed and used under Linux environment, it supports several user interfaces including command line sage prompt with Python (programming language) based syntax. It offers an enormous amount of features, most importantly, implementation of elliptic curves and arithmetics over various fields. Detailed test cases of binary field arithmetics and elliptic curves operations are given in Chapter 3, Section 3.3. For information concerning SageMath one can consult official documentation of the project [17].

[5]Ranging between 1900-2000 lines of code, excluding external sources

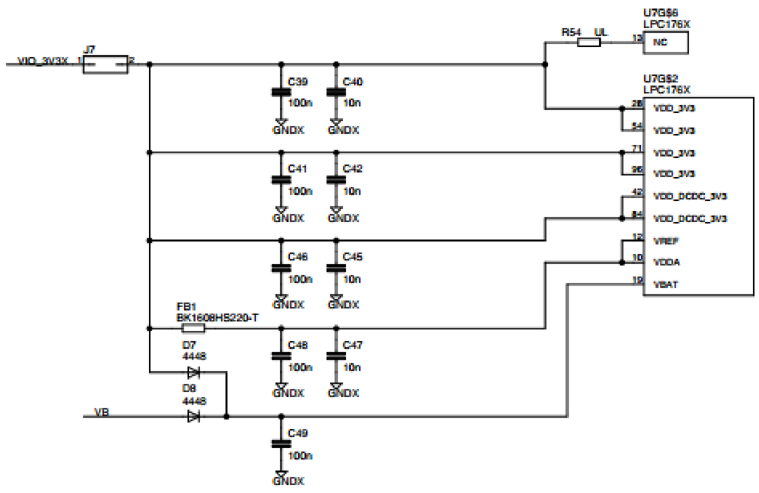


Figure 3.1.: LPC176x target side ("LCPXpresso LPC1769 rev B", Sheet 5 [5])

## 3.2. Implementation of Binary field Arithmetics and Elliptic Curve Operations

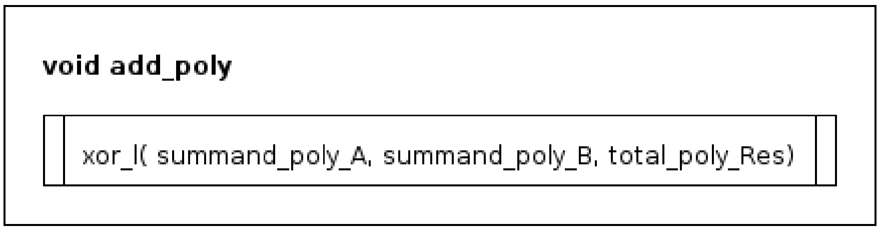
Mathematical definition of binary field operations is given in Chapter 2 Section 2.1.2 where it is mentioned, that the technical specification varies from provided formulation in sense of actual implementation. Therefore, the subsection 3.2.1 of this section extends on this matter, whereas subsection 3.2.2 addresses the design of elliptic curve operations.

### 3.2.1. Binary Field Arithmetics Realization

#### Addition

It is the corner stone of the entire field arithmetics, despite being the simplest operation to implement in means of logic complexity. The desired functionality is to add $a(x), b(x) \in GF(2^m)$, which are represented in form of binary polynomials. Since FLINT/C is used for large integer representation, it is possible to make use of data structure CLINT [6], which will hold the polynomials. Furthermore, addition in the field can be executed by binary exclusive $OR$ (also know as $XOR$). Once more, a ready-made function from the library is used to $XOR$ polynomials and store the result of calculation in third polynomial. Thus, following figure (see Figure 3.2) encapsulates the description.

[6]For the detailed information refer to "Cryptography in C and C++", Chapter 2 [18]

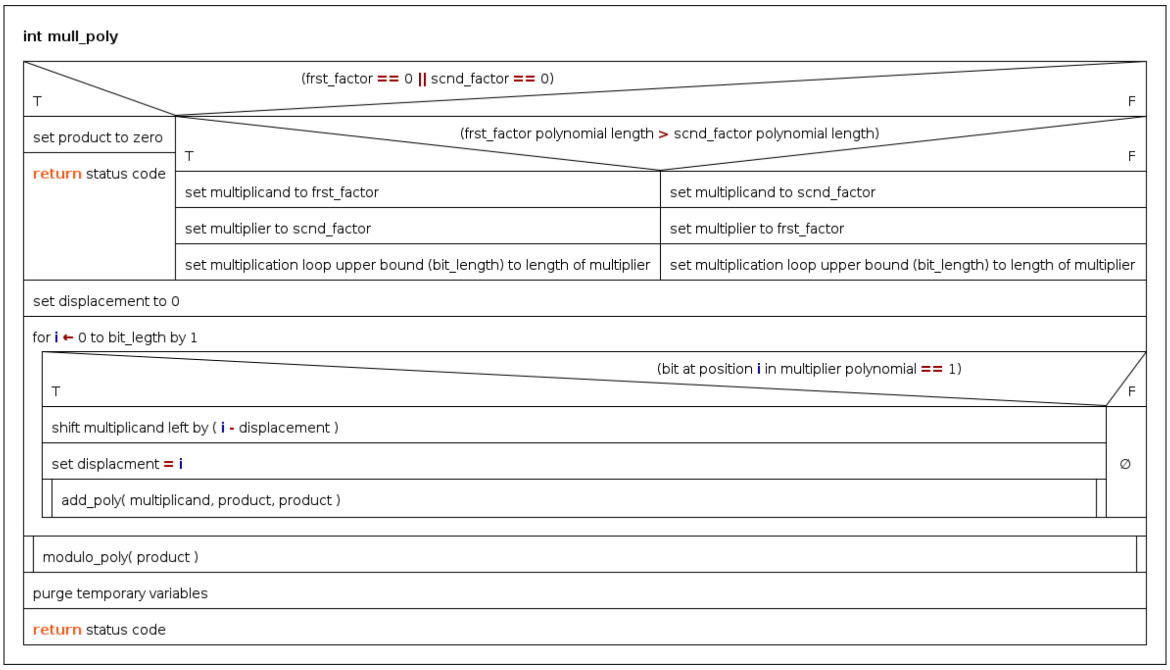


```
void add_poly
    xor_l( summand_poly_A, summand_poly_B, total_poly_Res)
```

Figure 3.2.: Addition

### Multiplication and Squaring

Multiplication of two elements in $GF(2^m)$ such as $a(x), b(x)$ can be carried out by multiplying $a(x)$ by each term of $b(x)$ separately, where each calculation iteration is equivalent to combination of a shift operation followed by summation. The addition is performed as stated in Figure 3.2. The logic behind of the multiplication function is depicted in the Nassi-Shneidermann diagram below (see Figure 3.3) and it can be seen, that there is an instruction performing shifting. To execute it a function provided by FLINT/C library is used.



```
int mull_poly
    (frst_factor == 0 || scnd_factor == 0)
    T:
        set product to zero
        return status code
    F:
        (frst_factor polynomial length > scnd_factor polynomial length)
        T:
            set multiplicand to frst_factor
            set multiplier to scnd_factor
            set multiplication loop upper bound (bit_length) to length of multiplier
        F:
            set multiplicand to scnd_factor
            set multiplier to frst_factor
            set multiplication loop upper bound (bit_length) to length of multiplier
    set displacement to 0
    for i ← 0 to bit_legth by 1
        (bit at position i in multiplier polynomial == 1)
        T:
            shift multiplicand left by ( i - displacement )
            set displacment = i
            add_poly( multiplicand, product, product )
        F:
            ∅
    modulo_poly( product )
    purge temporary variables
    return status code
```

Figure 3.3.: Multiplication

Multiplication function operates with three parameters, which are CLINT variables each holding multiplicand, multiplier and product respectively. The function has a return value, to facilitate debugging and general control flow. The explicit implementation of squaring operation was omitted. Multiplication is used instead, with both parameters holding the same value. It definitely affects the calculation performance, but at the same time it was a conscious decision on trying to keep the overall workload manageable within a certain time slot. It is obvious that the function produces results which have greater polynomial length as any of its inputs. Therefore, there is a direct call of reduction function ***modulo_poly***.[7]

## Inversion and Division

Having two elements in $GF(2^m)$ such as $a(x), b(x)$, then division $a(x)/b(x)$ is calculated as the multiplication in the form of $a(x) \cdot b^{-1}(x)$, where $b^{-1}(x)$ is the inverse of $b(x)$. It is obvious that inversion and division are tightly related operations. Thus, in order to implement division over binary field some logic that administers the calculation of inverse should be conceptualized. To accomplish this two additional functions are implemented: one which executes long polynomial division and the other which implements the extended Euclidean algorithm to calculate the inverse. It is worthwhile to mention that the realization of long division with remainder was the most laborious subtask. For this reason, in spite of the provided NSD[8](see Figure 3.4) depicting this function, the careful explanation of the algorithm would be appropriate.

For the long division, similar to the multiplication, shifting coupled with addition is used. But in contrast to the described functions, in order to develop long division, some better understanding of FLINT/C library functionalities was required. Therefore, before explaining the intricate parts of the actual implementation, it is reasonable to expand on how binary polynomials are handled in memory. As it was mentioned earlier, they are managed by CLINT data type from FLINT/C. A vague way to visualize how the library handles this task is to imagine that polynomials are stored as binary strings, or as a sequence of 0's and 1's. Each character or term of this string can be accessed via a library function and be inspected for its value, whether it is a 1 or a 0. Additionally, there is a dedicated function which returns the length of such a string. With this piece of information provided, the examination of division function should become less confusing.

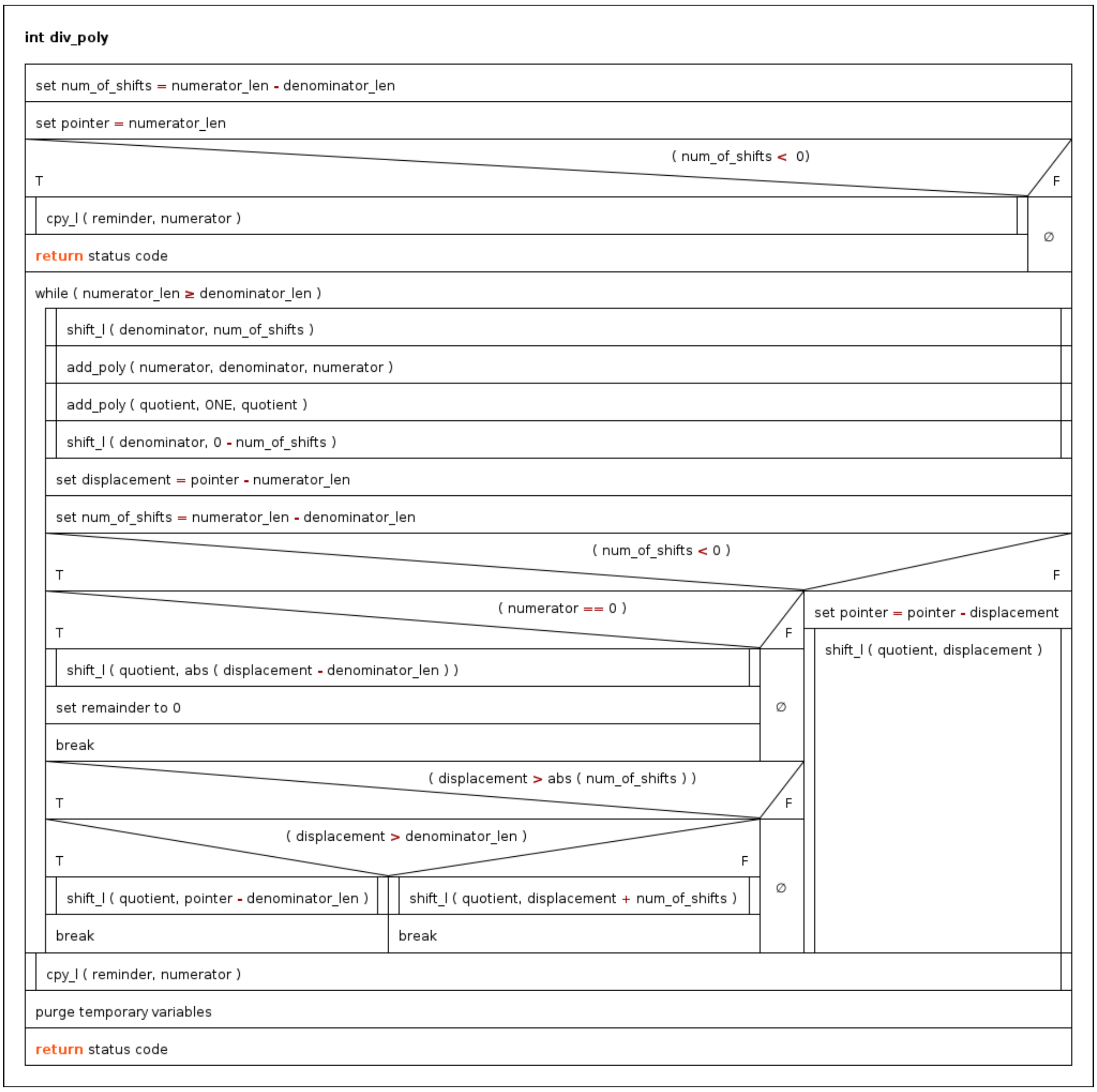
[7]Implementation information of this function is given later in this section
[8]Nassi-Shneidermann diagram

The core idea behind of the implementation is to left shift the denominator by appending trailing zeros to the end of it and then subtract it from the nominator, this way reducing the degree (i.e. the length) of the divisor polynomial. This procedure should be executed repeatedly until the length of the nominator is less than the denominator's. Since addition and subtraction are the same in binary field, it is accomplished by the addition function described in subsection 3.2.1. The amount of shifting is regulated by the difference between lengths of the nominator and denominator polynomials. Based on its value the divisor is shifted by such amount that its highest term aligns with the highest term of the polynomial representing the nominator. After each successful iteration the quotient is updated by shifting its value to the left and appending 1 to its end. As a last commentary to the realization of long division with reminder: there are several nested ***if*** statements in the NSD diagram, which serve the purpose of correctly shifting the denominator and quotient polynomials. After this brief explanation, it should become easier to understand the algorithm depicted in Figure 3.4.

Further examination of the functions which perform division over binary field, it is appropriate to concentrate on the function which calculates the inverse. Opposite to the long division, the fundamental algorithm and its implementation for this function are well known. It calculates the multiplicative inverse based on the extended Euclidian algorithm. Therefore, Figure 3.5 simply illustrates the realization of the algorithm, which utilizes some of the FLINT/C library functions, the long division with reminder, the multiplication and addition functions described in details above. However, with the aim of avoiding possible ambiguity, it suitable to comment on a function from the library called ***fswap_l***. It is used instead of a simple copy function (i.e. ***cpy_l***), since it keeps the code better structured and directly interchanges the contents of two CLINT variables, taking into consideration the necessary memory management tasks. To summarize, the binary field division in this thesis is performed in two steps: first, the inverse is calculated; next, the multiplication is carried out.

#### Reduction

Reduction or modulo is the last operation of interest and its implementation is not challenging. It relies on the long division function. At its core it simply invokes the mentioned function and passes to it a polynomial to be reduced and the irreducible polynomial of underlying field. The description of this functionality can be seen in Figure 3.6.



```
int div_poly
set num_of_shifts = numerator_len - denominator_len
set pointer = numerator_len
( num_of_shifts < 0 )
  T:
    cpy_l ( reminder, numerator )
    return status code
  F:
    ∅
while ( numerator_len ≥ denominator_len )
    shift_l ( denominator, num_of_shifts )
    add_poly ( numerator, denominator, numerator )
    add_poly ( quotient, ONE, quotient )
    shift_l ( denominator, 0 - num_of_shifts )
    set displacement = pointer - numerator_len
    set num_of_shifts = numerator_len - denominator_len
    ( num_of_shifts < 0 )
      T:
        ( numerator == 0 )
          T:
            shift_l ( quotient, abs ( displacement - denominator_len ) )
            set remainder to 0
            break
          F:
            ∅
        ( displacement > abs ( num_of_shifts ) )
          T:
            ( displacement > denominator_len )
              T:
                shift_l ( quotient, pointer - denominator_len )
                break
              F:
                shift_l ( quotient, displacement + num_of_shifts )
                break
          F:
            ∅
      F:
        set pointer = pointer - displacement
        shift_l ( quotient, displacement )
cpy_l ( reminder, numerator )
purge temporary variables
return status code
```

Figure 3.4.: Division

### 3.2.2. Implementation of Elliptic Curves: Point Addition and Doubling

The existence of functions embodying binary field arithmetics remarkably contributes to the goal of achieving functional elliptic curve cryptosystem. To accomplish the objective, functions outlining the group law described in Section 2.3 need to be implemented. This subsection emphasizes the actual realization of point addition and doubling operations. Before going through the details of how the computation is carried out, couple of remarks need to be made on the point representation technique used for this work. There are several

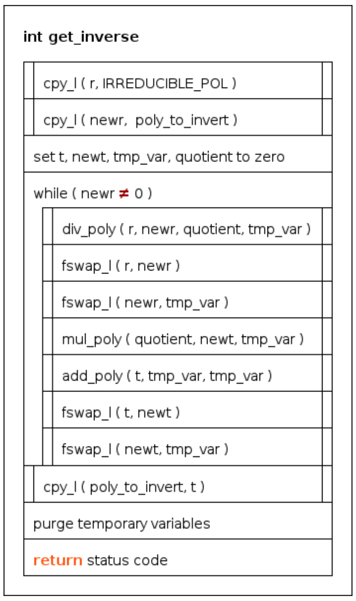

int get_inverse
cpy_l ( r, IRREDUCIBLE_POL )
cpy_l ( newr, poly_to_invert )
set t, newt, tmp_var, quotient to zero
while ( newr ≠ 0 )
div_poly ( r, newr, quotient, tmp_var )
fswap_l ( r, newr )
fswap_l ( newr, tmp_var )
mul_poly ( quotient, newt, tmp_var )
add_poly ( t, tmp_var, tmp_var )
fswap_l ( t, newt )
fswap_l ( newt, tmp_var )
cpy_l ( poly_to_invert, t )
purge temporary variables
return status code

Figure 3.5.: Inverse

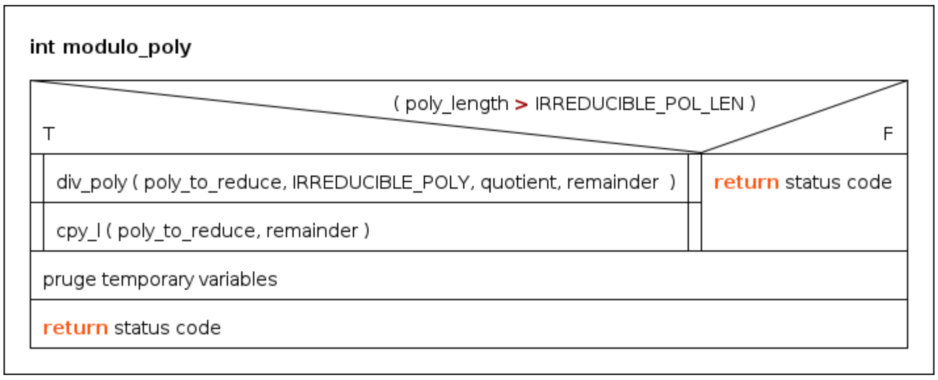

int modulo_poly
( poly_length > IRREDUCIBLE_POL_LEN )
T
F
div_poly ( poly_to_reduce, IRREDUCIBLE_POLY, quotient, remainder )
cpy_l ( poly_to_reduce, remainder )
return status code
pruge temporary variables
return status code

Figure 3.6.: Reduction

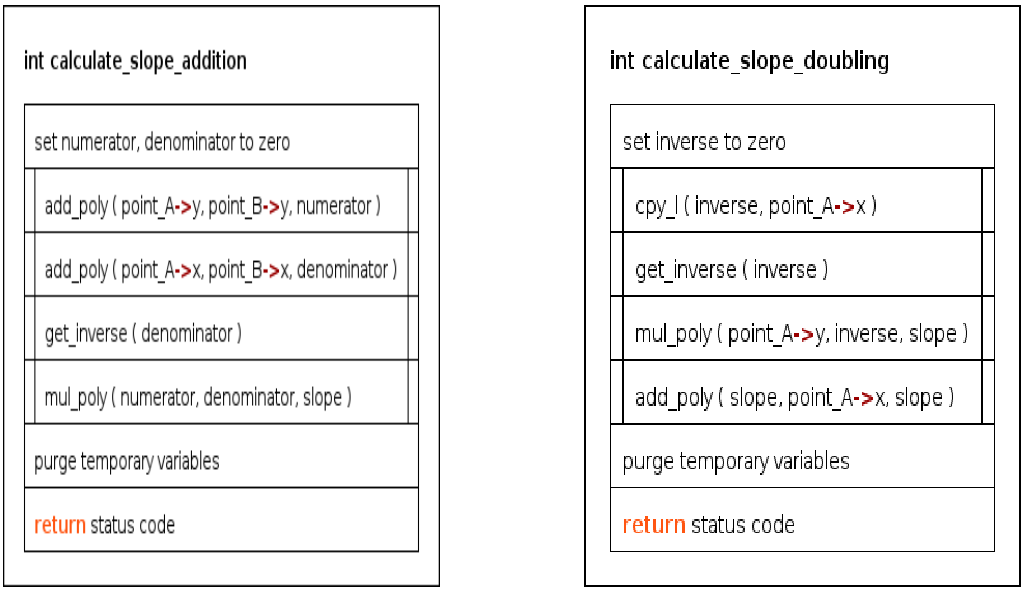
possibilities to represent an elliptic curve point (refer to Chapter 3 of "Elliptic Curves in Cryptography" [1] for detailed explanation). Each of them comes with a trade off between the ease of calculation and the computational efficiency. For this particular implementation of elliptic curve operations affine coordinates are used to describe points on a curve. This decision was made relying on the operational simplicity of the affine representation compared to the conventional projective coordinates.

### Point Addition

The realization of point addition conducts the calculations defined by the algebraic formulae in Equation (2.13) in Chapter 2, Section 2.3. The function fulfills the regulations of group law, by examining the points to be added. It is done in the following way by checking:

- whether either of the points is the point at infinity
- whether one of the points is negative of the other

The complete implementation of the function is illustrated in Figure 3.8. After careful examination of which, one can notice, that in addition to the checks defined above and the functions performing binary field arithmetics, there is a function call for a method named ***calculate_slope_addition***. The sole purpose of this auxiliary function is to calculate $\lambda$ from the formula (refer to Equation (2.13)). Its realization is described in Figure 3.7a. The another important aspect is that the squaring of $\lambda$ (it is called **slope** in Nassi-Shneidermann Diagram) is performed by multiplying the value by itself. The reasoning for this approach was given in section describing the multiplication function (see 3.2.1). The consequences of this implementation will be evaluated in Section 4, which covers the power analysis.

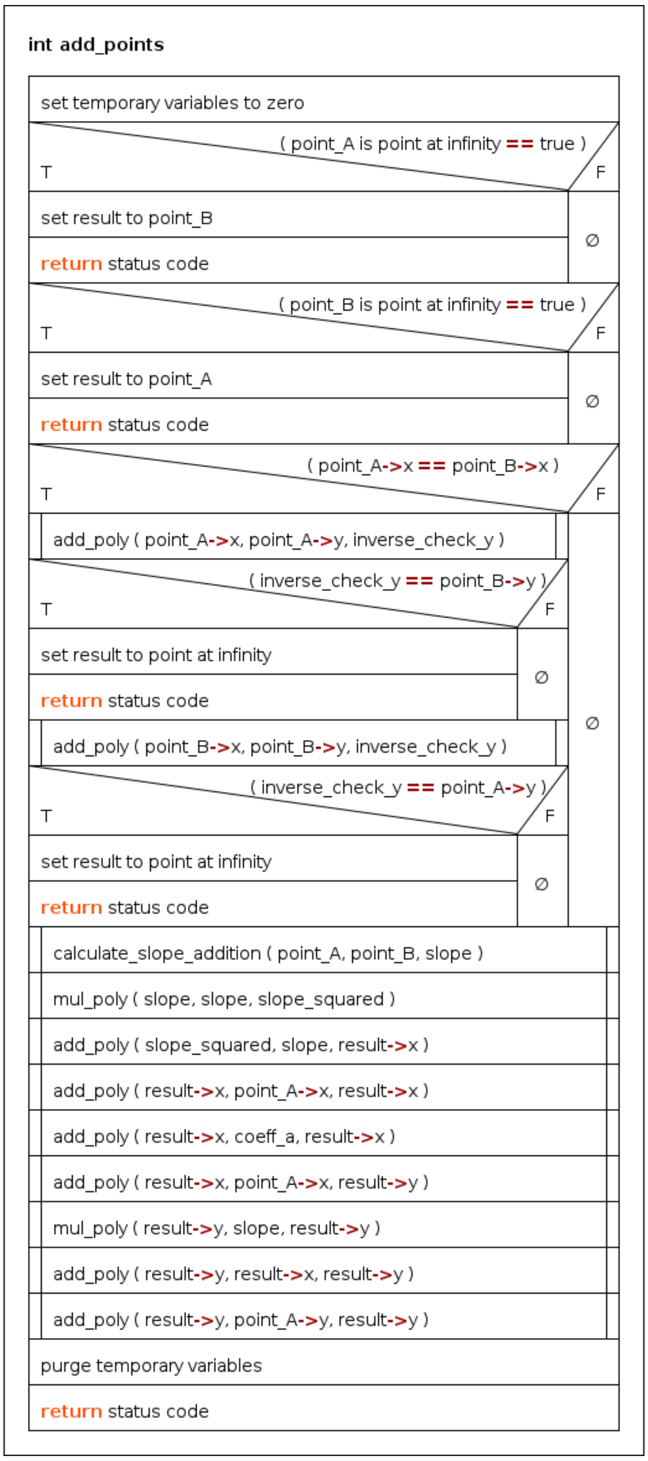


```
int calculate_slope_addition
    set numerator, denominator to zero
    add_poly ( point_A->y, point_B->y, numerator )
    add_poly ( point_A->x, point_B->x, denominator )
    get_inverse ( denominator )
    mul_poly ( numerator, denominator, slope )
    purge temporary variables
    return status code
```

```
int calculate_slope_doubling
    set inverse to zero
    cpy_l ( inverse, point_A->x )
    get_inverse ( inverse )
    mul_poly ( point_A->y, inverse, slope )
    add_poly ( slope, point_A->x, slope )
    purge temporary variables
    return status code
```

(a) Compute $\lambda$ for Point Addition (b) Compute $\lambda$ for Point Doubling

Figure 3.7.: Auxiliary functions for point addition and doubling

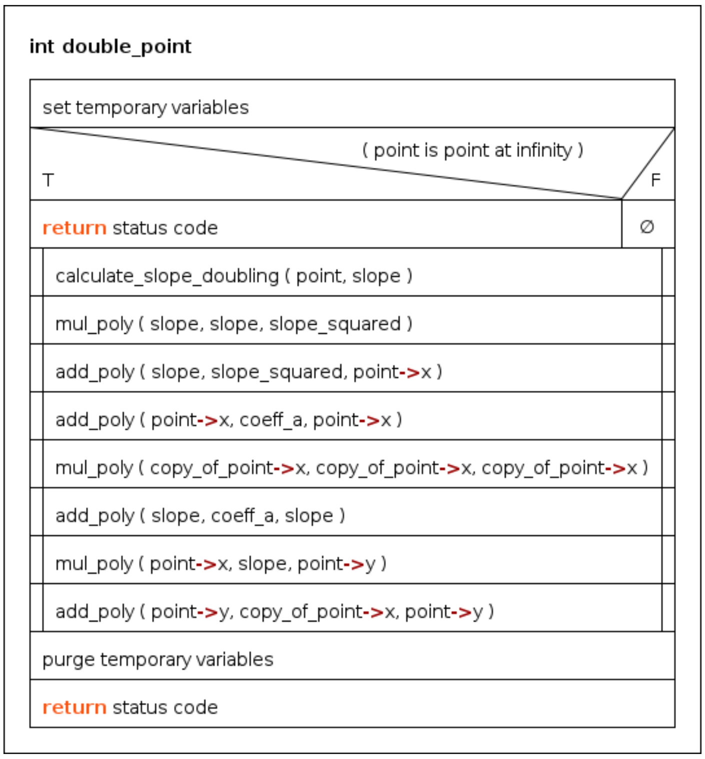


```
int add_points
    set temporary variables to zero
    ( point_A is point at infinity == true )
        T:
            set result to point_B
            return status code
        F:
            ∅
    ( point_B is point at infinity == true )
        T:
            set result to point_A
            return status code
        F:
            ∅
    ( point_A->x == point_B->x )
        T:
            add_poly ( point_A->x, point_A->y, inverse_check_y )
            ( inverse_check_y == point_B->y )
                T:
                    set result to point at infinity
                    return status code
                F:
                    ∅
            add_poly ( point_B->x, point_B->y, inverse_check_y )
            ( inverse_check_y == point_A->y )
                T:
                    set result to point at infinity
                    return status code
                F:
                    ∅
        F:
            ∅
    calculate_slope_addition ( point_A, point_B, slope )
    mul_poly ( slope, slope, slope_squared )
    add_poly ( slope_squared, slope, result->x )
    add_poly ( result->x, point_A->x, result->x )
    add_poly ( result->x, coeff_a, result->x )
    add_poly ( result->x, point_A->x, result->y )
    mul_poly ( result->y, slope, result->y )
    add_poly ( result->y, result->x, result->y )
    add_poly ( result->y, point_A->y, result->y )
    purge temporary variables
    return status code
```

Figure 3.8.: Point Addition

### Point Doubling

Analogously to the point addition, the doubling function straightforwardly implements the formulas given by Equation (2.14), which explicitly define how calculations should be carried out when adding point to itself. The realization considers a case of how a doubling of the point at infinity should be handled. This is implementation specific. To elaborate more, the point at infinity is defined as a point with coordinates $(0, 0)$. Additionally, at the beginning of calculations, in order to exclude the possibility of error propagations by falsely set variables, they all are preset to zero. Therefore, to countermeasure the occurrence of division by zero error this check is incorporated into the logic. The comprehensive description of the explained functionality is shown in Figure 3.9. Moreover, the point doubling relies on a supplementary method called ***calculate_slope_doubling***. Similarly to its operative equivalent in point addition, it calculates $\lambda$ from the Equation (2.14). To view the implementation details of this function reader can refer to Figure 3.7b.



```
int double_point
set temporary variables
( point is point at infinity )
T:
    return status code
F:
    ∅
calculate_slope_doubling ( point, slope )
mul_poly ( slope, slope, slope_squared )
add_poly ( slope, slope_squared, point->x )
add_poly ( point->x, coeff_a, point->x )
mul_poly ( copy_of_point->x, copy_of_point->x, copy_of_point->x )
add_poly ( slope, coeff_a, slope )
mul_poly ( point->x, slope, point->y )
add_poly ( point->y, copy_of_point->x, point->y )
purge temporary variables
return status code
```

Figure 3.9.: Point Doubling

## 3.3. Testing Implementation Validity

Confirmation of the fact that each component of the software performs as expected is a vital part of any implementation. All the functions described in previous sections are the subject of unit testing. Developer constructing individual modules is responsible to ensure, that the piece of software produces reasonable output. Concept of the result or output being credible can be further defined. Thus, for a well known input or inputs the unit under test must deliver a result, which is consistent no matter how many times the same input was provided to the unit. Furthermore, trustworthiness of the result must be verified against independent software or tool capable of reconstructing the implemented functionality or must be evaluated by any other means. External open-source software known as SageMath[9] (refer to Section 3.1.3 for more details) was used for this particular aim in this work.

### 3.3.1. Test Cases for Binary Field Arithmetics Realization

A separate test suite was created for each function described in Section 3.2.1. For some functions lookup tables containing inputs and corresponding outputs were generated to enable semi-automated testing, whereas others were tested manually. Such an approach with mixed testing techniques speeds up the overall realization without sacrificing quality assurance. Indeed, for some functions implementing binary field arithmetics, extensive tests are unfitting. As an example addition function can be viewed. Due to its simplicity and complete reliance on FLINT/C library function, a standalone test suite with lookup tables and additional methods to conduct verification of each test case would be irrational. Therefore, a simple manual comparison of one or two results of calculation against SageMath simulation is sufficient.

In case of complex function realizations, like long division with remainder or multiplication, semi-automated testing is a better choice. Those functions have several intricate parts and to test them manually is too laborious. Thus, it is desirable to set up lookup tables and use them for affirmation of calculation correctness. A lookup table is a matrix (or it is better to think of it as a two-dimensional array) with values taken from SageMath, which is used to simulate the behavior of a specific function (e.g. multiplication, long division with remainder, etc.). Each row of this array consists of a tuple representing input(s)[10] and output(s)[11]. To expound this approach, test suite of multiplication function is examined. It was mentioned in Section 3.2.1, that function expects as input two values, which hold the factors

[9] Also known as Sage or SAGE (System for Algebra and Geometry Experimentation)

[10] Depending on the function signature

[11] Again, it depends on function signature, i.e. log division with remainder produces two results: quotient and remainder

to be multiplied, yet produces a single result containing the product. Thus, using SageMath, multiplication over finite field of characteristic 2 was performed numerous times (for details refer Section "Finite Fields" of SageMath Manual [17]), then values used in calculations were directly passed to a two-dimensional array, with each row storing values of individual computation. Figure 3.10a illustrates how this array looks like by providing an excerpt from actual source code. In pursuance of avoiding manual invocation of multiplication function for each test triplet of values (e.g. a row from array), additional function executing specified test cases is developed. It simply iterates through the array, picks up the set of values, feeds it to the function under test and, as a last step, verifies the result of computation against the value from SageMath simulation. NSD depicted in Figure 3.10b expresses the described logic in details. Very same approach is used to verify function performing long division[12].

Functions calculating inverse and performing reduction are tested manually. As an argument to be adduced in support of this decision, it can be pointed out that main blocks in constitution of those functions are multiplication and(or) division functions. Therefore, successful throughout testing of multiplication and long division with remainder drastically reduces chances of miscalculations for them. To verify validity of computations by inverse and reduction functions, similarly to addition, a few test cases can be conducted and checked against SageMath simulations. Source code for test suites and lookup tables with additional commentaries regarding execution can be found on the CD.

### 3.3.2. Test Cases for Point Addition and Doubling Functions

By carefully examining the actual implementations of point addition (see Figure 3.8) and doubling (see Figure 3.9) operations one can notice, that both of them exploit the functions described in Section 3.2.1. To be specific, they actually consist of numerous invocations of the functions performing binary field arithmetics coupled with the logic which handles regulations of the group law. Furthermore, an inspection of the auxiliary methods (see Figure 3.7) indicates, that those are composed of the field arithmetic functions as well. After this preamble, it should not be surprising that the confirmation of calculations' accuracy of the field arithmetics functions, using the strategy described in Section 3.3.1, significantly simplifies the testing of the point addition and doubling functions. Once again, SageMath was used to warrant that the designed functions produce error-free results. It offers the necessary functionality to conduct calculations with elliptic curves over various finite fields.[13] The test suites of the point addition and doubling were developed in an uncomplicated and straightforward manner.

[12]For multiplication and division 64 distinct test cases per function were generated

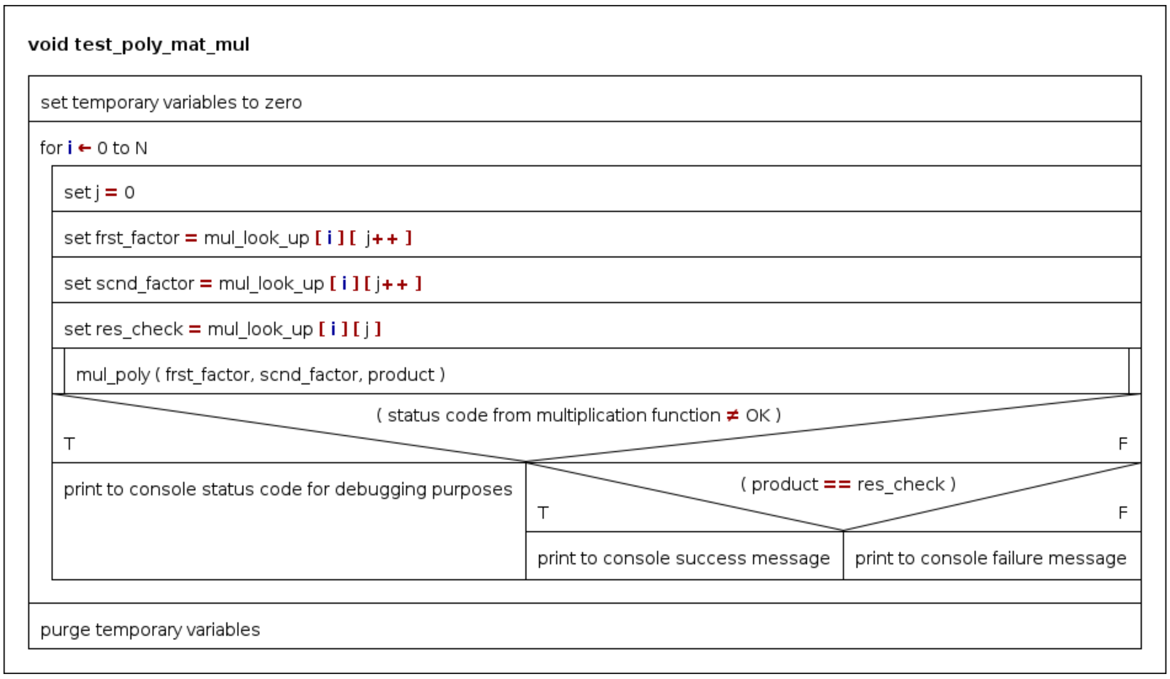
[13]For the detailed description of the features offered by SageMath module, reader can consult Section "Elliptic curves over finite fields" of the SageMath documentation [17]

At first a finite field of characteristics 2 should be defined, then an elliptic curve needs to be specified with the previously created field as its underlying field. Then, some random points from the curve can be selected using the methods offered by SageMath. Next, point manipulations, like adding two distinct points or doubling a single point can be executed. After that, the generated values can be extracted in form of binary strings and be used in the implementation for the testing purposes. Finally, by executing the self-designed functions with the test data from SageMath and comparing the results of calculations from self-implemented functions against the results of SageMath simulations for being identical, it can be asserted that the functions operate as expected. This procedure was done several times with various values of $m$ (i.e. the order of $GF(2^m)$)[14] and different curves. In order to facilitate a faster prosecution of the described routine, a simple set of SageMath prompt compatible instructions was created. Reader can find the file with explicit instructions and log file containing the test results on the CD.

[14]For the values ranging from $2^{16}$ up to $2^{1024}$

```
char * mul_look_up[N][3]={
{"1110010000010011", "1101000010010010", "10001010011000111110000010010110"},
{"1111010110010101", "1110011111100010", "10110111000110101111101010010010"},
{"1000110000010111", "0010000010000011", "00100011100010101111111110111001"},
{"0010001101111101", "1110100111010100", "00111000010110100101110111001000"},
{"0010011000111001", "1010000111100011", "00101111110001110100110101010110"},
{"0100000111101101", "0001100011111001", "00001100010111101011000000110101"},
{"1101011100111010", "0100011101001100", "01101111101101100001101101110000"},
{"0100000100011001", "1000110000000001", "01000111000000001101101000110010"},
{"1101000001110101", "1111010100100000", "10010001010010110011111101000000"},
{"0011111001011000", "1111011100000011", "00101000010111101001010111010000"},
{"1110110111100101", "0010100001011010", "00110101110000010011011111100100"},
{"0101100110011001", "1111100001111001", "01101101101011011100001101000010"},
```

(a) Snippet from Lookup Table for Multiplication Test Suite


void test_poly_mat_mul
set temporary variables to zero
for i ← 0 to N
set j = 0
set frst_factor = mul_look_up [ i ] [ j++ ]
set scnd_factor = mul_look_up [ i ] [ j++ ]
set res_check = mul_look_up [ i ] [ j ]
mul_poly ( frst_factor, scnd_factor, product )
( status code from multiplication function ≠ OK )
T
F
print to console status code for debugging purposes
( product == res_check )
T
F
print to console success message
print to console failure message
purge temporary variables

(b) Test Suite of Multiplication Function

Figure 3.10.: Verification of Multiplication Function

# 4. Side Channel Analysis

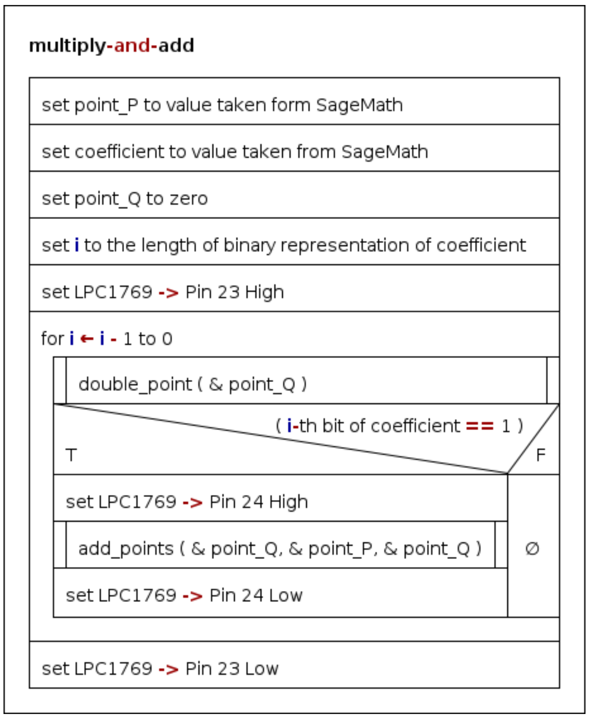
The side channel analysis operates with "side channel information", such as timing details (e.g. time that operations take), radiation of various sorts and power consumption statistics, which can be retrieved from encryption devices. The most common subtypes of this sort of analysis are: simple and differential power analysis, timing and fault attacks. Simple Power Analysis (SPA) is predominantly based on looking at the visual representation of the power consumption while performing an encryption operation and direct interpretation of the collected measurements. Since the amount of consumed power varies for divergent operations performed by a microcontroller, SPA can reveal the differences in power profiles and identify these operations. In this work SPA is used to distinguish point doubling and addition operations of ECC implementations and attempt to yield information about the key material. For the visual representation of the collected measurements an oscilloscope of DPO4054 series was used. It features sample rates up to $2.5$ GS/s and $10$ M points record length. The LPC1769 board facilitates the measurements by providing the dedicated pins which are easily accessible for probing. According to the documentation [5] J7 (see Figure 3.1) is shorted on the board, but if the connection between two pins is opened, it can be used for the current consumption measurement on the CPU.

It is stated in Chapter 2, Section 2.3.2, that a scalar multiplication is the core of elliptic curve cryptosystems, which is defined by an equation in form of: $Q = [k]P = P + P + \cdots + P\ (k\ times)$. Multiply-and-add method is one of the widely known algorithms to perform this calculation (also known as binary method). It relies on the binary expansion of $k$ and has a simple logic behind it. The binary representation of $k$ is inspected from its most significant bit down to the least significant one. In case of a bit being set two operations are performed: point doubling followed by point addition, otherwise only a doubling is executed. Due to the fact that elliptic curve operations produce different power traces, inspection of the power profile may reveal the value of $k$. That being so leads to the conclusion, that the requisite for conducting a successful side channel analysis is the ability to distinguish the power traces produced by the curve operations.

The explanation of the physical configuration, the use of additional electrical components and the necessary measurement arrangements for the oscilloscope to improve the readability of data can be found in Section 4.1. Additionally, it contains the information about how some extra triggering enables doubling and addition operations discovery from the overall power profile. The description of the domain parameters and the measurements' evaluation of the curves complying with NIST[1]/SECG[2] standards are given in Section 4.2.

## 4.1. Simple Power Analysis Realization

The implementation of the binary method to perform a scalar multiplication is simple. However, it is necessary to show the actual realization to ease the explanation of the following paragraphs. As it can be seen from Figure 4.1, the presented block carries out the logic of the binary method, with only one difference: some additional instructions are incorporated into algorithm's execution to take the advantage of triggering.



```
multiply-and-add
set point_P to value taken form SageMath
set coefficient to value taken from SageMath
set point_Q to zero
set i to the length of binary representation of coefficient
set LPC1769 -> Pin 23 High
for i ← i - 1 to 0
    double_point ( & point_Q )
    ( i-th bit of coefficient == 1 )
    T
        set LPC1769 -> Pin 24 High
        add_points ( & point_Q, & point_P, & point_Q )
        set LPC1769 -> Pin 24 Low
    F
        ∅
set LPC1769 -> Pin 23 Low
```

Figure 4.1.: Multiply-and-add

[1]National Institute of Standards and Technology is an agency of the United States Department of Commerce

[2]Standards for Efficient Cryptography Group (SECG), an industry consortium, that facilitate the adoption of efficient cryptography

Before proceeding to details of the triggers' role in facilitating discrimination of curve operations, the physical arrangement for conducting the measurement should be described. It was mentioned earlier, that LPC1769 board simplifies the task accomplishment by providing dedicated pins (J7 on Figure 4.2a), which can be directly accessed for the current consumption measurement on the CPU. However, according to the documentation [5] they are short soldered in between and for measurement purposes this connection must be broken off. In order to indicate the voltage representation of the current flow between two pins a $100\Omega$ resistor is used (it can be seen in Figure 4.2b with the probes attached to it). Thus, by attaching the oscilloscope probes to this resistor[3] the power consumption on the CPU can be observed.

After this preamble, the explanation of triggers' role in aiding the power analysis would be suitable. They simplify the detection of the overall computational block's power profile and the recognition of the independent operations. Thus, it was decided to use two peripheral pins[4] for this purpose: one of the them is supposed to mark the portion of the power profile signal where the execution of scalar multiplication is taking place, whereas another pin is configured to aid the curve operations distinction. The physical set up for this approach is illustrated in Figure 4.2b. The further elaboration can be formulated as follows: pin 23 of the LPC1769 board is set high before the execution of the multiply-and-add block calculations and low after its accomplished. The manipulation of pin 24 is coordinated so that it outputs high voltage around the addition operation, whereas low voltage output persists during the rest of computational steps. The described operational flow can be seen in the diagram illustrated in Figure 4.1. To observe these manipulations, the mentioned pins and J7 are connected to the oscilloscope of DPO4054 series, with the additional configurations applied with purpose of improving the quality of readings. To summarize:

- The channel displaying the overall power consumption, is set to have band-limit of 20 MHz and AC coupling applied. It is scaled to $1mV$ peak-to-peak
- The channels displaying the triggers are set to have DC coupling and the signal resolution for both is set to 5 V
- The record length of the oscilloscope is adjusted to be $10k$ points resulting into sampling rate of 500 S/s
- High resolution mode is turned on.

In addition to these settings, a special probe with attenuation factor of 1 is connected to J7, whereas for the trigger signals ordinary probes with attenuation factor of 10 are used. The measurement result, collected with this setup, is depicted in Figure 4.3a. A strong distortion in the signal (labeled as $P\_P$, channel 2) represents the power consumption on the CPU

[3]For the sake of clarity further mentions of J7 mean this connection and not factual pins themselves
[4]for information about LPC1769 board peripherals refer to [14]

and completely overlaps with the window, where the signal labeled as $CB\_T$ is high, which appears for the voltage output at pin 23. The examination of Figure 4.3a shows, that trace of the multiply-and-add implementation is distinguishable from the overall power profile even without the additional triggering. This ability to identify the relevant portion of the measurements empowers the further investigation and shifts the focus on the differentiation of the power traces left by the distinct curve operations.

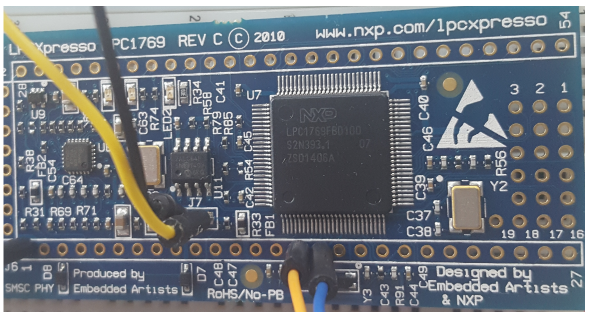
Therefore,the closer evaluation of the parts where the signal of channel $3$ labeled as $OPB\_T$ is high should be main point of the interest. Channel $3$ monitors the voltage levels at pin $24$. The controlled output of this pin signifies the exact moments when an addition operation is carried out. This hack serves a goal of finding the repetitive patterns in the power profile, which can be confirmed later to represent the power trace of the aforementioned operation (see Figure 4.3a). The attentive examination of $P\_P$ signal indeed reveals the recurrent appearances of the clearly recognizable traces within an observation window which is defined by the voltage level alterations of $OPB\_T$ signal. Furthermore, they persist outside of the observation window as well. This behavior is documented in Figures 4.3b, 4.3c and 4.3d.

A confirmation of the fact that observations hold in general and are not confined to any specific case is done by conducting measurements several times keeping the same physical arrangements. The demonstration of how this conclusion was achieved is critical. Moreover, an explanation of the work-flow needs to be given, the fundamentals of which are valid for all the measurement cases performed. Focusing on a particular case for the sake of clarity, the following actions are taken to conduct a measurement. At first, a SageMath simulation of an elliptic curve with its underlying field is generated. The field order is explicitly set to $128$ and the coefficient value is chosen so, that its binary representation is $(1000100010001)_2$. The purpose of a coefficient manipulation is obvious, because $1$'s and $0$'s dictate the order and quantity of the point addition and multiplication functions to be executed. Thus, the observation of $P\_P$ signal must yield traces of thirteen doubling and four addition operations. Since explicit triggering is not required to detect the signal portion of interest, the instructions used for pin 23 manipulations are removed. The results of the measurements with this slightly adjusted implementation of the multiply-and-add (see Figure 4.1) are shown in Figures 4.4a and 4.4b. The first notice is that the absence of the trigger signal from pin 23 does not affect the shape of $P\_P$ signal. The second, instead of four pulses in the channel labeled as $OPB\_T$ only three can be seen. Despite the mismatches in the expected and the actual appearances of the power profile and trigger signals, it should be mentioned, that the binary representation of the coefficient still can be spotted to an acceptable extend.

As it was noted in the description of the multiply-and-add algorithm, in case of a bit being set a doubling operation will be followed by an addition, otherwise only a point doubling takes place. It was also mentioned that the control logic of pin $24$ delimits the traces generated by execution of point addition operation. Therefore, repetitive samples in overall profile occurring midst two pulses in signal $OPB\_T$ must be the result of the doubling function execution(s), whereas the pattern within a single pulse is the outcome of the point addition calculation. This can be best presented graphically and Figure 4.4b is therefor: the distinct operations and the corresponding bit value of the coefficient are manually separated and marked. Now, the reason of the discrepancy in appearances of the power profile and trigger signals can be given. It is implementation specific. Due to the fact, that there are several function calls before the code block that implements the multiply-and-add algorithm, they inevitably affect the code execution[5] and the overall power consumption in general. This causes the deprivation of certain measurement readings, in particular the power traces of the initial curve operations and the trigger signal alterations. Pursuing the goal of the distortions minimization and the exclusion of the possible effects of triggering logic in measurements, the instructions responsible for pin $24$ manipulations were removed from the implementation of the multiply-and-add algorithm. However, even taking these changes into account, the quality of the measurements has not significantly improved. To illustrate this, the coefficient is set to $(101011101101)_2$ and the measurements are done over again. It can be seen from Figure 4.4c, that the curve operations are still traceable and the coefficient value can be partially extracted; nonetheless, the problem persists and it is impossible to retrieve the information regarding the power traces of the initial curve operations.

This particular realization of curve the operations (detailed description in Section 3.2.2) is not performance efficient and causes a large difference in their computational time. Furthermore, it yields a greater power trace for a doubling operation compared to an addition, which is unusual and normally it is the other way around. Such kind of abnormality needs to be analyzed and the possible causes of the observable behavior should be specified. Section 4.2 focuses on this matter by evaluating the execution mean time per a curve operation as well as the overall calculation time for the curves of the different sizes.

[5] Even if the compiler optimization is configured to be off


(a) J7 on LPC1769 board

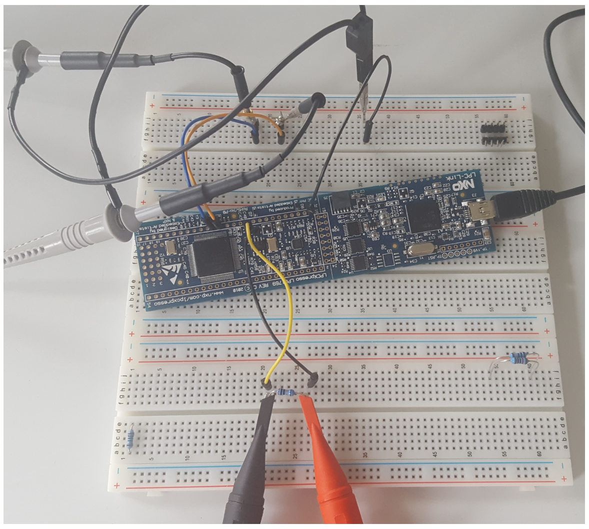
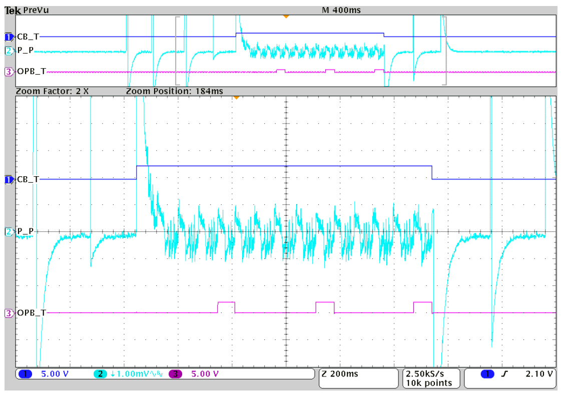

(b) Peripheral pins for triggering

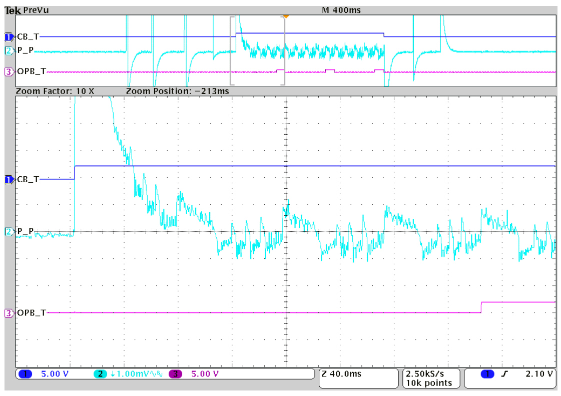
Figure 4.2.: Complete measurement setup

## 4.2. Analysis of Curves Complying with Commercial Standards

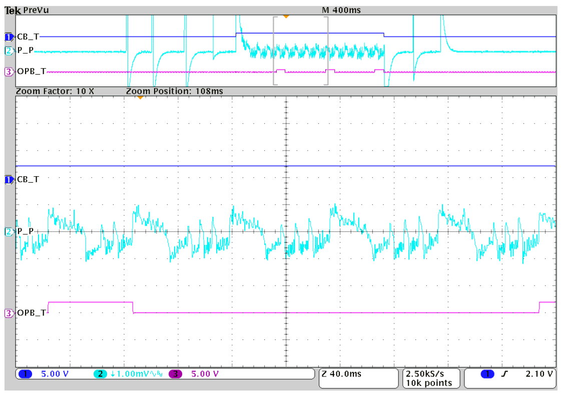
In continuation of Section 4.1, which describes the operation discovery from the repetitive patterns in the power profile, this section reports on the further analysis, which examines the performance characteristics of independent operations. For this purpose several curves with the different domain parameters were chosen. The brief overview of them is given in Table 4.1a. It contains information about the underlying field sizes, the approximate bit lengths of an RSA or DSA moduli at analogous strength specifications. The reduction polynomials of the binary fields are listed in Table 4.1b. It is important to mention, that the measurement routine was not changed. However, the additional triggering logic is removed from the software implementation and the physical arrangement, in the pursue of minimizing the unnecessary distortions in the appearance of the power consumption signal. The procedure



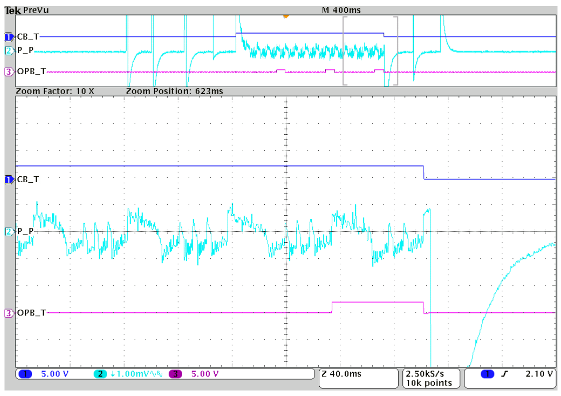
(a) Measurement Sample



(b) Enlargement of signal portion 1



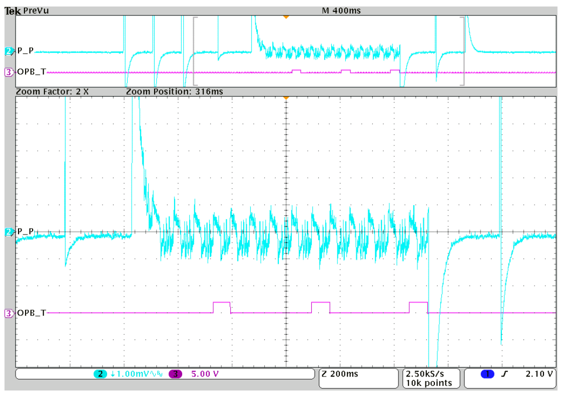
(c) Enlargement of signal portion 2



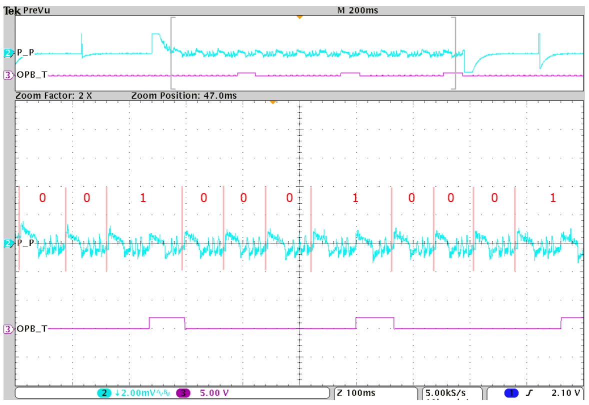
(d) Enlargement of signal portion 3

Figure 4.3.: Usage of triggers

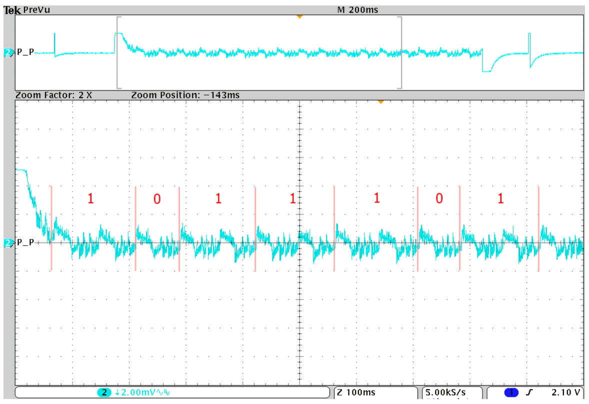
of the coefficient value extraction is preserved, it is done by studying the observable patterns and manually separating the operations traces. Due to the implementation specific behavior mentioned in Section 4.1, a direct evaluation of the execution mean time per operation is performed. In order to ease the import of a curve and its field parameters, SageMath is used. Exploiting the approach described in Chapter 3, Section 3.3.2, an elliptic curve and its field are specified, then the values are exported in form of binary strings and are passed to the program execution. The instructions facilitating this work-flow can be found on the CD. Since all of the curves are Koblitz curves (see Chapter 2, Section 2.2.1), there is no need to explicitly import the coefficients $a$ and $b$. They are set utilizing FLINT/C library functions. Moreover, the value of the $k$ coefficient, which specifies the scalar multiplication is maintained unchanged for all six cases. Such a deliberate coefficient manipulation contributes to a better graphical designation of the evolving computational complexity as the sizes of the fields grow. Thus, the binary representation of the $k$ coefficient is $(100010110111)_2$ yielding twelve executions of doubling and seven point addition operations. To estimate the mean time of an operation execution, several measurement samples of a solitary operation's completion time are taken over the number of the samples. Table 4.3 illustrates the results



(a) Resulting power profile



(b) Separation of individual operations



(c) Persistence of information loss problem

Figure 4.4.: Observation of signal with specific coefficient

achieved by this approach. The domain parameters of the curves used for the analysis are taken from "SEC 2: Recommended Elliptic Curve Domain Parameters" [4]. The curves to be examined are defined by a simplified Weierstrass equation in form of:

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

where the coefficients $a$ and $b$, the underlying field and its reduction polynomial change their value per curve. Therefore, to prevent unnecessary repetition of Weierstrass equations for each curve, just the values of the coefficients, the reduction polynomial and the compressed form of the curve base point are listed and can be found on the CD.

| Parameters | Size | RSA/DSA | Koblitz or random |
|---|---|---|---|
| sect163k1 | 163 | 1024 | k |
| sect233k1 | 233 | 2240 | k |
| sect239k1 | 239 | 2304 | k |
| sect283k1 | 283 | 3456 | k |
| sect409k1 | 409 | 7680 | k |
| sect571k1 | 571 | 15360 | k |

(a) Curves used for observations

| Field | Reduction Polynomial(s) |
|---|---|
| $\mathbf{F}_{2^{163}}$ | $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ |
| $\mathbf{F}_{2^{233}}$ | $f(x) = x^{233} + x^{74} + 1$ |
| $\mathbf{F}_{2^{239}}$ | $f(x) = x^{239} + x^{36} + 1$ or $x^{239} + x^{158} + 1$ |
| $\mathbf{F}_{2^{283}}$ | $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$ |
| $\mathbf{F}_{2^{409}}$ | $f(x) = x^{409} + x^{87} + 1$ |
| $\mathbf{F}_{2^{571}}$ | $f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$ |

(b) Reduction Polynomials of $GF(2^m)$

Table 4.1.: "SEC 2: Recommended Elliptic Curve Domain Parameters" Chapter 3, pages 14-15 [4])

### 4.2.1. Observations

Figure 4.5 clearly shows that the binary representation of the $k$ coefficient can be revealed from the power profile signal for all of the curves of interest. However, on the grounds of the issue described in Section 4.1 there are still certain bits of the coefficient, which can not be identified. A closer examination of the figures indeed reflects this behavior. Even though the $k$ coefficient is set to be $(100010110111)_2$ the power traces of two doubling and one addition operations are not perceptible, hence the visible part of the coefficient is $(0010110111)_2$. The second important finding is that for any given curve a doubling operation takes significantly more calculation time than an addition. In order to clarify the main cause of this phenomenon the actual implementation of the curve operations should be closely studied. Table 4.2a contains the information about how many times certain functions are called on the average during the execution of a curve operation. Table 4.2b shows those values for the helper functions, which were additionally implemented to calculate $\lambda$ from Equations (2.13) and (2.14).

At a first glance one can conclude that the statement about the abnormal execution time is false, since a point addition has more than twice amount of the calls for the *add_poly* and various library functions. However, the measurements prove the opposite. Even though a greater number of the internal function calls affects the power trace of an addition operation, the majority of the invoked functions perform two simple tasks: value copying and purging of the temporary variables at the end of the calculations. Furthermore, those are the library functions, which are already optimized. Regarding the *add_poly*: by reviewing the Nassi-Shneidermann diagram illustrated in Figure 3.2 it becomes obvious, that the only overhead of this function is expressed by the wrapper nature of its implementation. Being explicit, it simply calls a library function to perform $XOR$ of two CLINT variables. Therefore, it should not be surprising, that the most of the computational time is consumed by the rest of self-implemented functions for binary field arithmetics. Those are the multiplication, division and calculation of the inverse functions (refer to Chapter 3, Section 3.2.1). Due to the fact that those functions realize field arithmetics without utilizing any advanced methods, they are extremely inefficient. Moreover, the squaring operation is not implemented at all, instead the multiplication function is used. Thus, the difference in the execution time of the curve operations is predominantly defined by the invocation frequency of the mentioned self-implemented functions. When comparing among the total number of the multiplication function calls per curve operation (including the helper functions) in Table 4.2, it can be seen, that a point doubling has four, whereas a point addition requires three calls. Furthermore, a half of those calls actually imitates the squaring for a doubling operation. On the other hand, only one out of three calls utilizes multiplication to perform the squaring in a point addition operation. Going back to the multiplication function realization in Chapter 3, Section 3.2.1 it can be seen that in order to keep the computation result within the the finite field the modulo operation (i.e. reduction function) is executed. Now, by taking into consideration the fact that the squaring of a value is more likely to produce a result which needs to be reduced, it can be said that for a doubling operation the reduction is performed on average more often than for a point addition. This distinction has to have a significant impact, whence the reduction is performed by exploiting the long polynomial division function, which itself is loosely implemented.

The timing analysis of the functions, which compose the elliptic curve operations, could have shed more light on the assumptions and the conclusions made in the paragraph above. However, before conducting it, one should take into the account an important aspect. Such kind of an examination is labor-intensive. A careful and elaborated analysis of five functions, which execution time depends on the several factors, implies a humongous workload. Therefore, it is a conscious decision to keep the evaluation within certain boundaries and review at provided depth only these functions, which realize elliptic curve arithmetics, whence meeting the requirements of this work. Table 4.3 summarizes a simple timing analysis by displaying the total execution time and the mean execution time[6] of the point doubling, addition operations for the curves listed in Table 4.1a. As expected the overall execution time rises according to the growth of the underlying field size. The curve operations' execution mean time[7] lengthens as well, preserving the tendency of a doubling requiring more time than an addition operation.

| Function Name | $\approx n$ calls in **add_points** | $\approx n$ calls in **double_point** |
|---|---|---|
| **FLINT/C** *functions* | 22 | 8 |
| *add_poly* | 9 | 4 |
| *mul_poly* | 2 | 3 |
| *Respective auxiliary function* | 1 | 1 |

(a) Analysis of curve operations constituents

| Function Name | $\approx n$ calls in **calculate_slope_addition** | $\approx n$ calls in **calculate_slope_doubling** |
|---|---|---|
| **FLINT/C** *functions* | 4 | 2 |
| *add_poly* | 2 | 1 |
| *mul_poly* | 1 | 1 |
| *get_inverse* | 1 | 1 |

(b) Analysis of auxiliary functions constituents

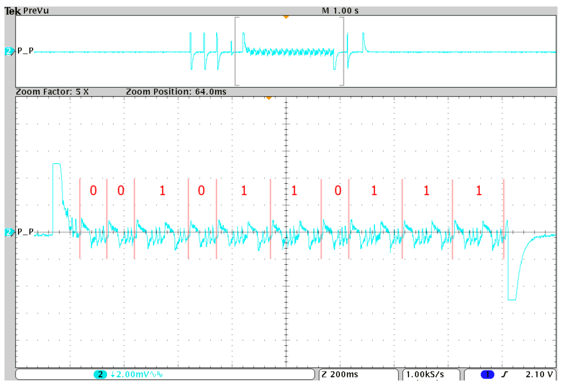
Table 4.2.: Examination of implemented curve operations

| Curve | Total Computational Time [s] | Execution MT per Doubling [ms] | Execution MT per Addition [ms] |
|---|---|---|---|
| sect163k1 | 1,67 | 104,56 | 88,96 |
| sect233k1 | 2,78 | 173,68 | 144,96 |
| sect239k1 | 3,07 | 193,28 | 158,00 |
| sect283k1 | 3,55 | 216,72 | 189,68 |
| sect409k1 | 6,13 | 380,40 | 321,92 |
| sect571k1 | 10,21 | 633,00 | 535,20 |

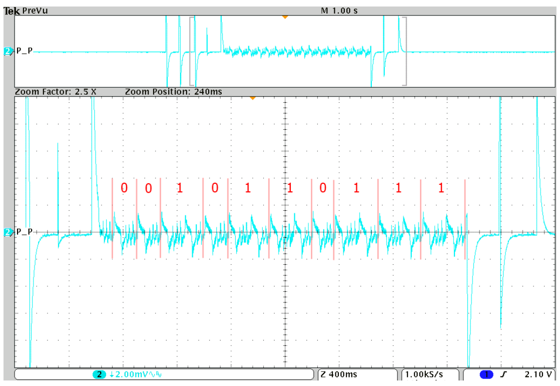
Table 4.3.: Computational Time Characteristics

[6] Mean execution time is designated as MT in Table 4.3

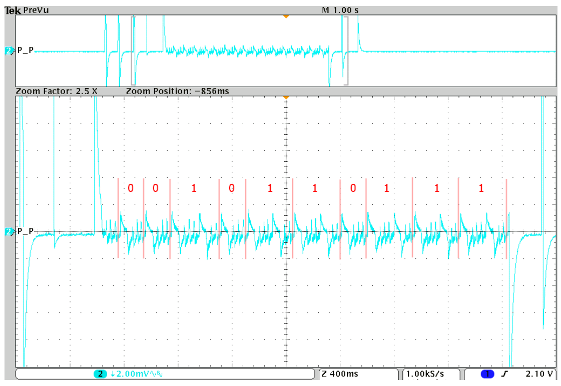
[7] Execution mean time was calculated as the arithmetic mean (average) time of the operations' execution



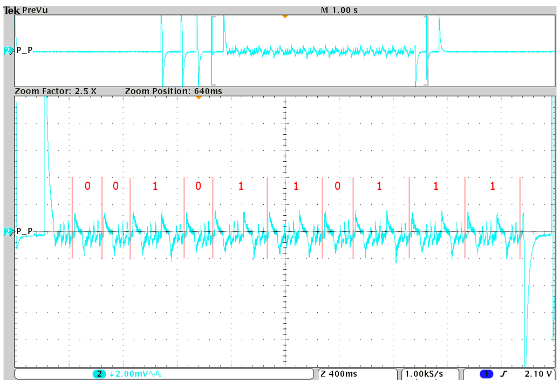
(a) **sect163k1**



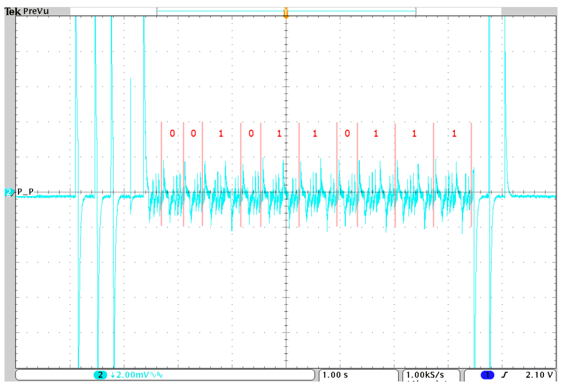
(b) **sect233k1**



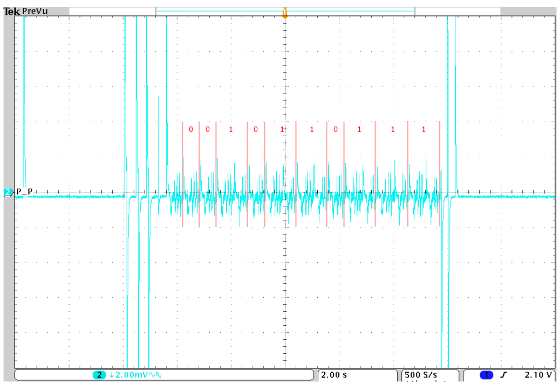
(c) **sect239k1**



(d) **sect283k1**

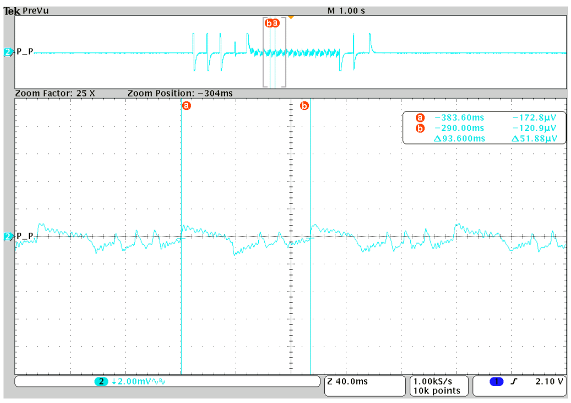


(e) **sect409k1**

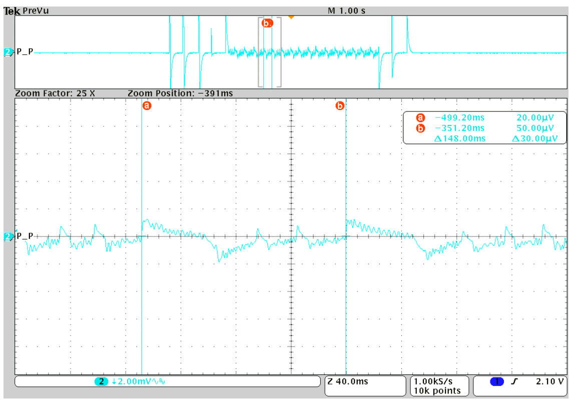


(f) **sect571k1**

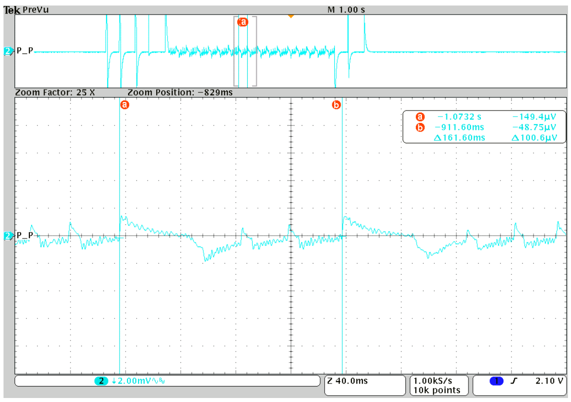
Figure 4.5.: Coefficient extraction of NIST/SECG curves



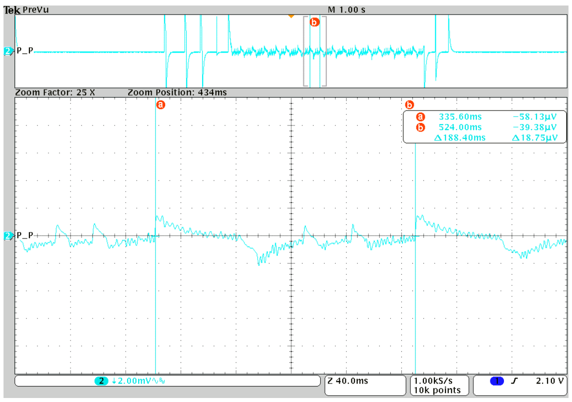
(a) **sect163k1**



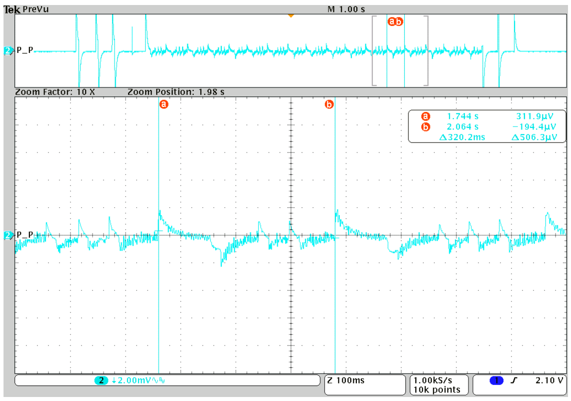
(b) **sect233k1**



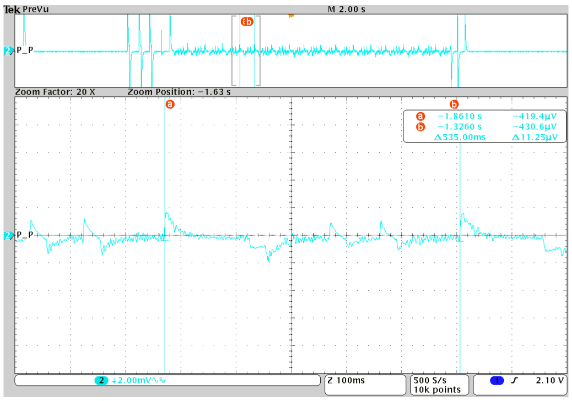
(c) **sect239k1**



(d) **sect283k1**

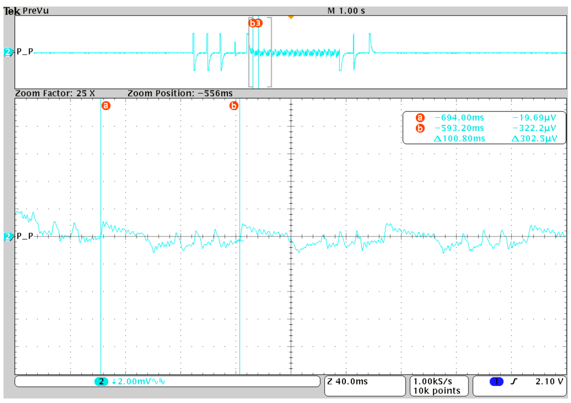


(e) **sect409k1**

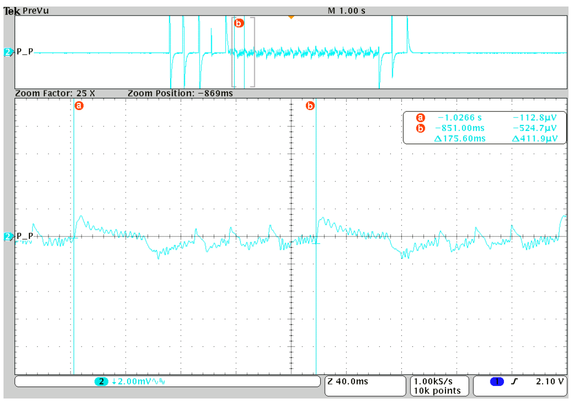


(f) **sect571k1**

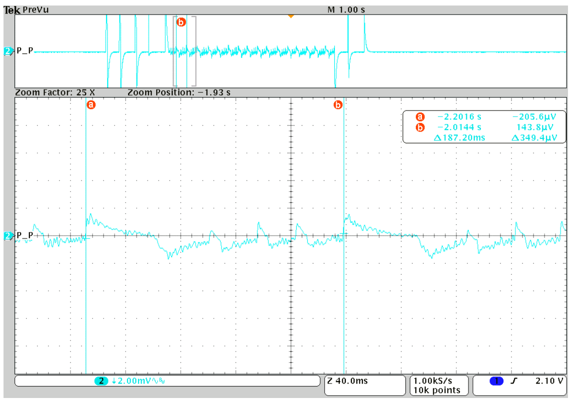
Figure 4.6.: Comparison of addition operation execution time



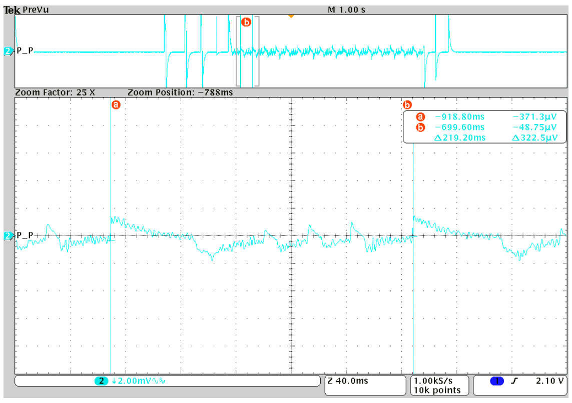
(a) **sect163k1**



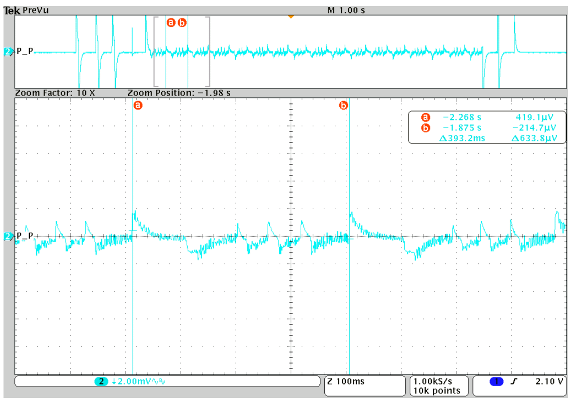
(b) **sect233k1**



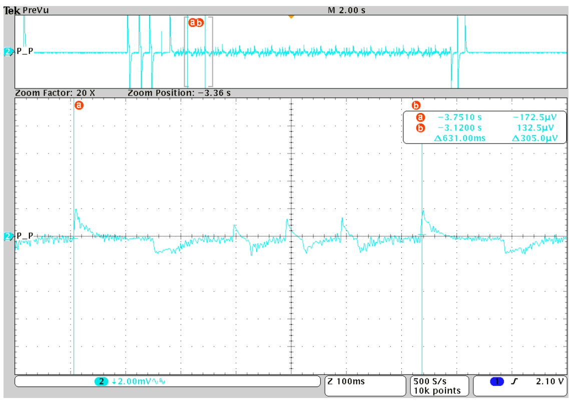
(c) **sect239k1**



(d) **sect283k1**



(e) **sect409k1**



(f) **sect571k1**

Figure 4.7.: Comparison of doubling operation execution time

# 5. Conclusion

There are several issues present, which can be resolved effectively and it would be appropriate to review them. First, the corruption in the power profile that leads to the inability of a complete coefficient identification from the observable traces. This drawback could be eradicated by inserting an empty loop prior to the multiply-and-add block execution, which in turn would possibly stabilize the signal to be examined. However, this would have contradicted the goal of keeping analysis as close as possible to the real side channel attack. Next, the absence of an actual implementation for a field squaring operation. This limitation could be suppressed either by leveraging the full potential of FLINT/C library or by explicitly implementing the operation itself. Unfortunately, none of the listed options can be easily achieved and the brief descriptions of related problems are given in Chapter 3.

Further suggestions for the quality improvement of the performed analysis can be made in addition to the description of the existing problems and their potential solutions. Even though a custom cross-toolchain for target device was ruled out as a laborious task, it can be viewed as the most optimal method. This way the complete functionality of the FLINT/C library could have been utilized, which in turn would have led to the simplification of binary field arithmetics realization. As a first possible outcome, the simple power analysis could have been made the only priority in contrast to having two cross-related and equally important tasks: the field arithmetics implementation with verification and the power analysis itself. The other possibility could have been the further investigation in the efficient field and elliptic curve arithmetics implementation based on the FLINT/C library.

Nonetheless, this work shows a possible implementation of binary field arithmetics and studies its effects on an elliptic curve system realization. It was demonstrated how the implementation of the field and curve arithmetics can be achieved just relying on the bare FLINT/C library functionality. The credibility of the calculations produced by the functions carrying out the field operations was asserted against an external tool. SageMath was used to simulate binary field arithmetics and export computation data, which was further employed by test suites. Exploiting very same approach the accuracy of the curve operation calculations was demonstrated. The execution of the side channel analysis was carefully examined and described, straightforward evaluation of the measurement results was covered.

# References

[1] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[2] S. Chacon and B. Straub. *Pro Git*. Apress, second edition, 2014.

[3] D. Coupvent-Desgraviers. Small reminder for linux user very keen of embedded system. http://pmc.polytechnique.fr/pagesperso/dc/arm-en.html, 2012 (accessed June 8, 2017).

[4] Daniel R. L. Brown. Standards for Efficient Cryptography 2 (SEC 2). Technical report, Certicom Research, January 2010.

[5] Embedded Artists AB. Lpcxpresso lpc1769 rev b. https://www.embeddedartists.com/sites/default/files/docs/schematics/LPCXpressoLPC1769revB.pdf, 2011 (accessed June 10, 2017).

[6] Free Software Foundation. The gnu mpfr library. http://www.mpfr.org/, 2016 (accessed June 8, 2017).

[7] Free Software Foundation. The gnu multiple precision arithmetic library. https://gmplib.org/, 2016 (accessed June 8, 2017).

[8] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., 2004.

[9] W. Hart. Flint: Fast library for number theory. http://www.flintlib.org/index.html, 2016 (accessed June 1, 2017).

[10] W. Hart. Mpir: Multiple precision integers and rationals. http://mpir.org/, 2017 (accessed June 8, 2017).

[11] R. Lidl and H. Niederreiter. *Introduction to finite fields and their application*. Cambridge University Press, 1986.

[12] M. S. Anoop. Elliptic curve cryptography an implementation guide. http://www.infosecwriters.com/Papers/Anoopms_ECC.pdf, 2005 (accessed March 2, 2017).

[13] A. J. Menezes, T. Okamoto, and S. Vanstone. *Reducing elliptic curve logarithms to logarithms in a finite field*. IEEE, 2002.

[14] NXP Semiconductors. *LPC176x/5x User manual*. NXP, "rev.4.1 - 19 december 2016" edition, December 2016.

[15] Oxford University Press. Communication definition from open oxford dictionary of english language. https://en.oxforddictionaries.com/definition/communication, 2017 (accessed March 2, 2017).

[16] A. Soni and N. Saxena. Elliptic curve cryptography: An efficitent approach for encryption and decryption of data sequence. *International Journal of Science and Research (IJSR), India Online ISSN: 2319-7064*, 2(5):203–208, May 2013.

[17] The Sage Development Team. Sage reference v.7.6. http://doc.sagemath.org/html/en/reference/index.html, 2005-2017 (accessed June 10, 2017).

[18] M. Welschenbach. *Cryptography in C and C++*. United States of America 9 8 7 6 5 4 3 2 1, 2005.

# Appendices

# A. CD Contents

**flint.h** Header file of FLINT/C library

- Autor: Michael Welschenbach, 1998-2001 by Springer-Verlag Berlin, Heidelberg

**flint.c** Source file of FLINT/C library

- Autor: Michael Welschenbach, 1998-2001 by Springer-Verlag Berlin, Heidelberg

**ecc_mat.h** Header file of Elliptic Curve Operations

**ecc_mat.c** Source code of ECC operations implementation

**poly_mat.h** Header file of Binary Field Arithmetics Operations

**poly_mat.c** Source code of Binary Field Arithmetics implementation

**ECC_Thesis.c** Source file with main function

**test_suite.h** Header file of Test Suites

**test_suite.c** Source code of Test Suites

**lookup_tables.c** Source file with lookup-tables for Test Suites

**setting_up_curves.py** Collection of instructions to facilitate SageMath of ECC curves

**sage_generate.py** Collection of instructions for ECC and field arithmetics testing

**CurveSpecifications.pdf** Values of coefficients, reduction polynomials and compressed base points of curves used in analysis

**MeasurementsExecutionExplanation.txt** Journal of conducted measurements

**SAGE_Samples** Folder of helper scripts used for test data generation

**TestLogs** Folder of Test Suite Logs

**Mes_Photos** Folder with photo documentation of all the measurements

# Declaration

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor report has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

|   |   |
|---|---|
| Hamburg, August 11, 2017 |  |
| City, Date | sign |