Master's Thesis

Comparison of different 256-bit Elliptic Curves for Low-Resource Devices

Wolfgang Wieser
0831773

Graz, 2014

Institute for Applied Information Processing and Communications
Graz University of Technology

Assessor: Karl-Christian Posch

Advisor: Michael Hutter

Abstract

Elliptic Curve Cryptography has gained increasing interest in the last decade, especially in the context of low-resource devices such as embedded systems, smartcards, and RFID devices. Its advantage can be found in the efficient implementation and the possibility of using short keys.

Many elliptic-curve types have been published in the past decades. They mainly differ in the underlying finite-field, the form, and recommended curve parameters such as the prime modulus, the prime order, curve coefficients, or the base point. The question of which curve type is the most suitable one for highly restricted environments is in fact an open research question. While there exist many publications that present single implementations of various elliptic curves, a comparison of those solutions is not only missing but also largely unfeasible. This is because each implementation was made using different design tools, techniques, security levels, and process technologies. A fair comparison of state of the art elliptic curve types in terms of efficiency is therefore highly demanded not only by industry but also by different standardization bodies and related organizations.

In this work, we propose a generic hardware architecture that allows efficient implementations of various elliptic curves. We explore the design space of low-area implementations of three commonly used curves, i.e., NIST P256, Brainpool P256r1, and Curve25519. All of these curves share a security level of 128 bits and are thus comparable. Furthermore, we used the same tools and techniques to allow the fair evaluation of area, power, and speed. For each curve, we applied optimization techniques to reduce the area and power footprint and propose a novel method on efficient multi-precision multiplication for this type of hardware architecture.

We synthesized all implementations using a 0.13-μm low leakage CMOS process technology. The results show that all three curves can be implemented with less than 7 000 GEs (without considering RAM). Curve25519 showed to be the most performant curve. It needs 877 339 cycles for a random point scalar multiplication while NIST P256 needs 1 718 398 cycles and Brainpool P256r1 needs 3 043 325 cycles. Our smallest design needs 5 192 GEs. The power consumption for all designs is between 78 and 284 μW.

Keywords: elliptic curve cryptography, low-resource device, Curve25519, NIST P256, Brainpool P256r1, CMOS, evaluation

Zusammenfassung

Elliptische-Kurven-Kryptographie hat in den letzten zehn Jahren zunehmendes Interesse erfahren, besonders in Verbindung mit Geräten mit beschränkten Ressourcen, wie eingebettete Systeme, Chipkarten und RFID Geräten. Sie hat den Vorteil, dass sie effizient implementiert werden kann und die Verwendung kurze kryptografischer Schlüssel ermöglicht.

In den letzten Jahrzehnten wurden viele Arten elliptischer Kurven veröffentlicht. Diese unterscheiden sich hauptsächlich im verwendeten endlichen Körper, der Kurvengleichung und den empfohlenen Kurven-Parametern, wie etwa dem Modulo, der Ordnung, den Koeffizienten oder dem Generatorpunkt. Es ist eine offene Frage welche Art von elliptischen Kurven sich am besten für stark eingeschränkte Geräte eignet. Es wurden zwar in vielen Publikationen einzelne Implementierungen von verschiedenen Kurven vorgestellt, aber es fehlt ein Vergleich dieser Vorschläge. Dieser ist auch nur schwer möglich. Das liegt daran, dass jede Implementierung mit unterschiedlichen Tools, Techniken, Sicherheitslevels und Prozess-Technologien erstellt wurde. Ein gerechter Vergleich der Effizienz von aktuellen Arten von elliptischen Kurven wird daher von der Industrie, verschiedenen Standardisierungsgremien und ähnlichen Organisationen stark nachgefragt.

In dieser Arbeit schlagen wir eine generische Hardware-Architektur vor, die ein effizientes Implementieren von elliptischen Kurven erlaubt. Dabei loten wir die Möglichkeiten eines Designs mit einer kleinen Fläche für drei elliptische Kurven aus, nämlich NIST P256, Brainpool P256r1 und Curve25519. Diese drei Kurven bieten alle ein Sicherheitslevel von 128 Bits und sind daher direkt miteinander vergleichbar. Außerdem haben wir für alle drei Kurven dieselben Tools und Techniken verwendet um einen fairen Vergleich der Fläche, des Stromverbrauchs und der Geschwindigkeit zu ermöglichen. Für jede Kurve haben wir Optimierungen angewandt, um die Fläche und den Stromverbrauch zu reduzieren. Zusätzlich schlagen wir eine neue Methode für Langzahlmultiplikation für diesen Typ von Hardware vor.

Alle Varianten wurden mit einer 0.13-μm CMOS Technologie synthetisiert. Dabei konnten alle drei Kurven mit einer Fläche kleiner als 7 000 GEs (ohne RAM) implementiert werden. Das Design für Curve25519 ist schneller und kleiner als das der anderen Kurven. Curve25519 benötigt 877 339 Takte für eine Skalarmultiplikation mit einem zufälligen Punkt, während die schnellsten Varianten von NIST P256 (1 718 398 Takte) und Brainpool P256r1 (3 043 325 Takte) eine deutlich längere Laufzeit haben. Unser kleinstes Design braucht 5 192 GE. Der Stromverbrauch für alle Varianten liegt zwischen 78 und 284 μW.

Schlüsselwörter: Elliptische-Kurven-Kryptographie, Geräte mit beschränkten Ressourcen, Curve25519, NIST P256, Brainpool P256r1, CMOS, Vergleich

Acknowledgements

At first, I would like to thank Michael Hutter to guide me through many telematics masters lectures, exercises, projects and finally my masters thesis. I would like to thank him for many ideas, hints, and showing me different exciting aspects of cryptography and designing hardware. Many thanks also to Karl Christian Posch, who taught me the basics of designing hardware in his lecture "Rechnerorganisation". Working three years as teaching assistant for this lecture, allowed me to acquire more in-depth knowledge and gave me some interesting experiences. Equally, I would like to thank my family for their support. I also would like to thank my fellow students for making the studying pleasant even in stressful times and supporting me in group works and learning for exams. Last but not least, I want to thank my girlfriend Fabia for supporting and pushing me to finish my second thesis and her big understanding in all the stressful times.

Contents

1 Introduction

Cryptography has experienced a rapid progress in the last hundred years. Nowadays, in developed countries almost everyone uses it – often without noticing it. Cryptography is applied in car keys, television, wireless networks, mobile phones, debit cards, online banking, passports, and many other applications. Thereby, cryptography can guarantee attributes like confidentiality, data integrity, authenticity, or non-repudiation. Since processing units keep getting faster, the length of cryptographic keys must extend to guarantee a constant level of security. As a result, calculation time, needed memory, and needed bandwidth increase. Thus, elliptic curve cryptography becomes more popular, because it can provide the same level of security as former algorithms using a significant shorter key length than for example RSA. This is because up to now it takes much longer to break elliptic curve cryptography than breaking RSA with the same key length. Smaller parameters lead to significant advantages in power consumption and requirements in computing power and memory. This makes cryptography with adequate level of security feasible for low-resource devices such as embedded systems, smart cards, and RFID tags. We are using these devices constantly and in near future in the "Internet of things" RFID tags will be attached to everything. They will interact with each other and may exchange sensitive data. Therefore, they must provide reasonable cryptographic functions to guarantee a secure communication. As a result, it is very important to work on small and fast implementations having low power consumption. In this work, we describe a highly efficient architecture for elliptic curve cryptography. To find the most efficient elliptic curve parameters and algorithms, we implement three different curves in several variants and compare them with each other.

Operations on elliptic curves use digits with several hundred bits. If the used hardware cannot work on such large numbers, they must be divided into a number of words and handled one by one. This so called "multiple precision arithmetic" divides an operation into several partial operations. As a result, it is slower than if the value can be handled at once. To keep the number of partial operations small and thus ensure a sufficient level of efficiency, it is important that the words are as large as possible. Common CPUs are able to work on 64 or even 128 bits. In the last years some CPUs and GPUs were released which are even able to work on 256 or even 512-bit words, but they are not very common yet. To support operations on large words much area is necessary. Thus, arithmetical units for low-resource devices such as chip cards typically have a much smaller word size. They provide typically only operations on 8, 16, or 32 bits. In this work, we present an efficient 32-bit architecture for 256-bit elliptic curves, which can be used on strongly limited systems too.

Cryptographic protocols based on elliptic curves consist of several layers. On the very top, there is the service. It defines the purpose of the used cryptography. As mentioned before, it can be used to ensure confidentiality, data integrity, authenticity, or non-repudiation. The second layer is the protocol, which defines how many messages must be exchanged between the involved entities. The number of the messages also depends on the next layer, the scheme. Schemes provide a set of cryptographic operations which are typically combined within a protocol. Common schemes are key-agreement schemes,

identification schemes, encryption schemes, or signature schemes. The next layer defines how the point-multiplication is performed. Like for a common multiplication of two numbers, there are several algorithms for calculation. Point-multiplication uses point doubling and point-addition operations. They are defined in the fifth layer and are based on the geometrical structure of the used curve. Therefore, the operations differ for different types of curves and point representations. Finally, the last layer implements the finite field arithmetic. There exist several finite fields, but in this work, we just focus on elliptic curves based on a prime field. To find the most efficient algorithms for point multiplication, we implement several variants for each curve. Therefore we use state-of-the-art formulas for the curves NIST P256 and Brainpool P256r1 and propose new formulas for Curve25519. We also describe a new algorithm for efficient multiple precision multiplication, the so called zigzag product-scanning multiplication method.

The reminder of this work is organized as follows. In Chapter 2, we illustrate how modular multiple precision arithmetic works and explain finite fields. Chapter 3 gives a summary about elliptic curves, common algorithms for point-multiplication, and widely used protocols based on elliptic curves. Additionally, we give an overview of feasible attacks and possible countermeasures. Then, in Chapter 4, we describe the used curves with their parameters. Afterwards, in Chapter 5, we give details of our hardware architecture. In Chapter 6, we discuss the results of all of our variants for all implemented curves and evaluate the costs of several components. Finally, in Chapter 7 we give a conclusion of the work.

2 Arithmetical Background

In this chapter, we discuss the mathematical background, which is needed to use elliptic curves for cryptographic purposes. Finite field arithmetic is used in many cryptographic areas. For instance, the two very common encryption standards AES and RSA use finite field arithmetic. It also forms the base for elliptic curve cryptography. All operations on elliptic curves are done using arithmetic on the underlying field. Furthermore, finite field arithmetic uses modular arithmetic which often uses multiple precision arithmetic to handle big numbers. In this chapter, we explain multiple precision arithmetic, modular arithmetic, finite field arithmetic and inversion in finite fields.

2.1 Multiple Precision Arithmetic

The size of a number a CPU can handle at once is limited. Modern CPUs in personal computers typically support operations on values with a length of 64 bits. For CPUs in embedded systems, smartcards, or RFID tags, word sizes of even only 8, 16, or 32 bits are quite common. To provide a sufficient level of security, numbers in the cryptographic context are often are many times larger. Therefore, the values must be divided into several words, with a word size w. Several programming languages like Java, C#, or Python have built-in support to relieve the programmer from this task. Hardware designers either must create a CPU with an appropriate word size or they must use multiple precision arithmetic. In the following subsections we describe algorithms for multiple precision addition, subtraction, and multiplication.

2.1.1 Addition and Subtraction

Given two n-bit numbers a, b, and the word size w, the addition with multiple precision arithmetic works as follows. The two numbers are divided into $q = \lceil n/w \rceil$ words, so that $a = \{a_{q-1}, ..., a_1, a_0\}, b = \{b_{q-1}, ..., b_1, b_0\}$. Starting with index 0, all corresponding pairs of words $\{a_i, b_i\}$ must be added. If these partial sums are larger than the word size, the additional bits are used as carry for the addition of the next pair. A detailed description of the algorithm can be found in Algorithm 1. The length of the total result can be $n + 1$. The result of the algorithm is given in $\{carry, c\}$.

Signed values typically are represented using two's complement. In this representation the most significant bit (MSB) indicates the sign of the value. If it is 1, the number is negative. A number with n bits can represent integers in the range from -2^{n-1} to $2^{n-1} - 1$. To negate a number, all bits must be inverted and the value 1 must be added. The concept is demonstrated in the following example.

Example 1. *Calculating* $14 - 12$ *at word size* $w = 5$:

- $-12 = -0b01100 = 0b10011 + 1 = 0b10100$
- $14 - 12 = 14 + (-12) = 0b01110 + 0b10100 = 0b00010 = 2$ (5-bit result)

If numbers are represented in two's complement, multiple precision subtraction works the same way as for addition.

2.1.2 Multiplication

There are many algorithms for multiple precision multiplication. They differ in the number of necessary load and store operations, depending on how the operands are processed. In this thesis, we only consider algorithms with quadratic complexity. For those algorithms, each word of a must be multiplied with each word of b. As a consequence, for the most methods the same operands must be loaded multiple times or must be kept in registers. Thus, either the number of memory operations is high or many registers are necessary.

Operand-scanning multiplication. This method is also known as schoolbook or row-wise multiplication method. Thereby, one word of a is loaded and multiplied with all words of b. This is done for all words of a. The partial products are added to the intermediate result with an offset $(i + j) \cdot w/2$ defined by the indices i, j of the operands. A detailed description can be found in Algorithm 2. Figure 1a illustrates the sequence of partial products. The advantage of this method is, that each word of a is used only once.

Algorithm 1: Multiple precision addition

Input: $a = \{a_{q-1}, ..., a_1, a_0\}, b = \{b_{q-1}, ..., b_1, b_0\}, w, q$
Output: $carry, c = \{c_{q-1}, ..., c_1, c_0\} = a + b$

```
1 carry = 0
2 for i = 0 to q − 1 do
3 |   t = a_i + b_i + carry
4 |   carry = t/2^w
5 |   c_i = t − carry · 2^w
6 return carry, c
```

Algorithm 2: Operand-scanning multiplication

Input: $a = \{a_{q-1}, ..., a_1, a_0\}, b = \{b_{q-1}, ..., b_1, b_0\}, w, q$
Output: $c = a \cdot b$

```
1 c = 0
2 for i = 0 to q − 1 do
3 |   op_A = a_i
4 |   for j = 0 to q − 1 do
5 |   |   op_B = b_j
6 |   |   c = c + (op_A · op_B) · 2^((i+j)·w/2)
7 return c
```

Product-scanning multiplication. This method is also known as column-wise multiplication method. There, the partial products are calculated sorted after their offset – column wise. Thereby, the result can be calculated word wise. Thus, it is possible to hold the intermediate result in an accumulating register so that they do not have to be stored or loaded during the calculations. Additionally, the handling of carry values is simpler than for the operand-scanning method. In return, the algorithm is more complicated and for each partial product the index of both operands changes. A detailed description can be found in Algorithm 3. Figure 1b illustrates the sequence of partial products.

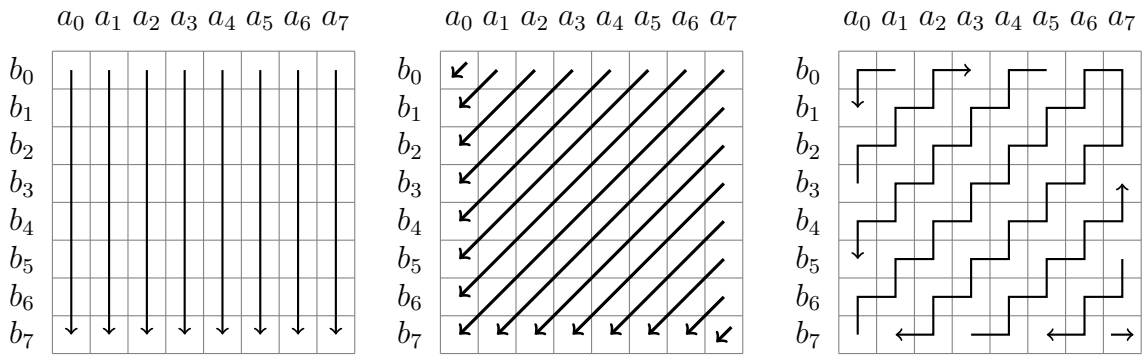
Zigzag product-scanning multiplication. In our work, we propose a new variant of the product-scanning multiplication. Thereby, two columns are calculated simultaneously in a zigzag pattern as shown in Figure 1c. This has the advantage that for each partial product only the index of one operand changes and the result is calculated almost word-wise. Thus, less memory operations are necessary. Therefore, this algorithm allows a multiplication in two cycles even if only one operand can be loaded per cycle. In the first cycle the next operand is loaded and in the second one the result is written to the RAM. However, the algorithm is more complex than the other ones and the accumulator must be bigger. Algorithm 4 gives a detailed description. Instead of calculating x, y it is easier to precalculate them for certain a number of words q.

Wenger and Werner presented in [WW11] another zigzag product-scanning multiplication method. However, they calculate only one column at once. Thus, in their method both operands change for most of the 32-bit multiplications. Therefore, in our architecture their method has no advantage compared to the classical product-scanning multiplication

Algorithm 3: Product-scanning multiplication

Input: $a = \{a_{q-1}, ..., a_1, a_0\}, b = \{b_{q-1}, ..., b_1, b_0\}, w, q$
Output: $c = a \cdot b$

```
1  c = 0
2  for e = 0 to q − 1 do
3      for d = 0 to e − 1 do
4          op_A = a_{e−d−1}
5          op_B = b_d
6          c = c + (op_A · op_B) · 2^{(e)·w/2}
7  for e = 0 to q − 2 do
8      for d = 0 to w − 2 − e do
9          op_A = a_{w−1−d}
10         op_B = b_{e+1+d}
11         c = c + (op_A · op_B) · 2^{(e+2)·w/2}
12 return c
```



(a) Operand-scanning (b) Product-scanning (c) Zigzag variant

Figure 1: Comparison of three multiplications methods

algorithm.

More variants and details can be found in [HW11]. All presented methods have to calculate n^2 partial products. Thus, they have a running time of $\mathcal{O}(n^2)$. Beside the presented methods, there exist more multiple precision multiplication algorithms. However, some of them need less multiplications and thus are faster but they are considered to be too complex for an implementation on a low-resource device. Two examples are the Karatsuba algorithm [KO63] with a running time of $\mathcal{O}(n^{\log_2(3)})$ and Fürer's algorithm [Für09] using fast Fourier transformation to achieve a running time of $\mathcal{O}(n \log(n) \log(\log(n)))$.

2.2 Modular Arithmetic

In modular arithmetic, numbers wrap around when they reach a certain value, which is called the modulus. If two numbers are multiplied, the result can be twice as long. Thus, in many cryptographic algorithms the result is reduced to keep the length constantly at the size of the modulus m. If a number $b = a + k \cdot m$, a is congruent to b. Thereby k can be any negative or positive number. This is written as $a \equiv b \pmod{m}$ and b can be seen as reminder by the integer division of b/m. In modular arithmetic, a and b are equivalent, thus all arithmetical operations on them have the same reduced result as shown in following example.

Example 2. *Modular arithmetic with* $16 = 16 + 2 \cdot 23 \equiv 62 \pmod{23}$:

- Adding 9: $16 + 9 = 25 = 2 + 1 \cdot 23 \equiv 2 \pmod{23}$,
  $62 + 9 = 71 = 2 + 3 \cdot 23 \equiv 2 \pmod{23}$
- Multiply with 19: $16 \cdot 19 = 304 = 5 + 13 \cdot 23 \equiv 5 \pmod{23}$,
  $62 \cdot 19 = 1178 = 5 + 51 \cdot 23 \equiv 5 \pmod{23}$

Algorithm 4: Combined multiplication method

Input: $a = \{a_{q-1}, ..., a_1, a_0\}, b = \{b_{q-1}, ..., b_1, b_0\}, w, q$
Output: $c = a \cdot b$

```
1  x = −1; y = 0; s = −1; m = −1; c = 0; op_A = a_1; op_B = 0
2  for e = 0 to q − 1 do
3      if e is even then
4          x = x + 2
5          y = y + s
6      else
7          y = y + 2
8          x = x + s
9      m = m − 4 · s
10     for d = 0 to m do
11         if e is even then
12             if d is even then
13                 y = y − s
14                 op_B = b_y
15             else
16                 x = x + s
17                 op_A = a_x
18         else
19             if d is even then
20                 x = x − s
21                 op_A = a_x
22             else
23                 y = y + s
24                 op_B = b_y
25         if s < 0 then
26             c = c + (op_A · op_B) · 2^((2·c+(d+1 (mod 2)))·w/2)
27         else
28             c = c + (op_A · op_B) · 2^((2·c+(d (mod 2)))·w/2)
29     if c + 1 = n/2 then
30         t = x
31         x = y
32         y = t
33         x = x − 1
34         m = m + 2
35 return c
```

The result of adding two n-bit numbers can have $(n+1)$ bits. In this case it is sufficient to subtract the modulus when the $(n+1)^{th}$ bit is set, as used in Algorithm 1. However, the result of a multiplication can have $2n$ bits. In worst case it would take 2^n steps to reduce the result by subtracting the modulus. Another possibility is to use division, but division is typically a very expensive operation. Thus other methods must be used. Some of them are presented in the following subsections.

2.2.1 Barrett Reduction

This algorithm was published by Paul Barrett in 1986 [Bar87]. The basic principle behind this algorithm is shown in following equation:

$$a \pmod m = a - \left\lfloor \frac{a}{m} \right\rfloor \cdot m \tag{1}$$

If the modulus m is constant, some values can be precalculated to speed up the calculation. The equation above can be extended as follows [DV11]:

$$a \pmod m = a - \left\lfloor \frac{\frac{a}{b^{n-1}} \cdot \frac{b^{2\cdot k}}{m}}{b^{n+1}} \right\rfloor \cdot m = a - \left\lfloor \frac{\frac{a}{b^{n-1}} \cdot \mu}{b^{n+1}} \right\rfloor \cdot m, \tag{2}$$

where b typically is chosen as a multiple of the word size. Thereby, $\mu = \frac{b^{2\cdot k}}{m}$ can be precalculated, since it depends only on the modulus. Division by powers of two are simple shifting operations. Thus, if b is a multiple of two, divisions by b^{n-1} and b^{n+1} can be done very fast. n is the number of bits of the modulus. Because of the shifting operations, only two partial multi-precision multiplications are needed. The complete description can be found in Algorithm 5. If the values are calculated with an appropriate precision, Line 3 to 6 are not necessary. Then, the running time of the algorithm is independent from the input.

Algorithm 5: Barrett reduction

Input: $m,\ b \geq 3, k = \lfloor log_2(m) \rfloor + 1,\ 0 \leq a < b^{2\cdot k},\ \mu = \lfloor b^{2\cdot k}/m \rfloor$
Output: $a \pmod m$

1 $q = \left\lfloor \lfloor z/b^{k-1} \rfloor \cdot \mu / b^{k+1} \right\rfloor$
2 $r = (a \pmod{b^{k+1}}) - (q \cdot m \pmod{b^{k+1}})$
3 **if** $r < 0$ **then**
4 $\quad r = r + b^{k+q}$
5 **while** $r \geq m$ **do**
6 $\quad r = r - m$
7 **return** r

2.2.2 Montgomery Multiplication

The algorithm was introduced in 1985 by Peter Montgomery [Mon85]. In contrast to the Barrett reduction, this algorithm includes the multiplication. Therefore, the input data is transformed into the Montgomery form, which makes modulo operations easier. Then, one or more multiplications with an additional Montgomery reduction are performed. Finally, the result is converted back to normal form. Let $R > m$ and $gcd(R, m) = 1$, so that the modulus of R is easy to calculate. Thus, R typically is a multiple of the word size w. To transform a number a to its Montgomery form $\bar{a}$, it is multiplied with R. Then the result of the multiplication can be calculated as follows:

$$a \cdot R \pmod{m} \cdot b \cdot R \pmod{m} = \bar{a} \cdot \bar{b} = \bar{c} \cdot R = c \cdot RR \tag{3}$$

As one can see in this equation, the result of the multiplication must be divided by R to get the Montgomery form. To convert it back to normal form it must be divided a second time. This division can be replaced by the Montgomery reduction operation. The complete algorithm can be found in Algorithm 6. Despite of simplified reduction, because of the additional conversations, this method is generally less efficient than a naive multiplication and reduction. This method is only advantageous if a sequence of multiplications is performed, as instance for modular exponentiation.

2.2.3 Reduction for Pseudo-Mersenne Primes

Mersenne primes are all prime numbers which can be written as $2^n - 1$, for instance $127 = 2^7 - 1$. However, Pseudo-Mersenne primes are all prime numbers which can be written as $2^n - q$, whereby $0 < |q| < 2^{\lfloor n/2 \rfloor}$ [Sol11].

Assuming the prime number is $p = 2^{255} - 19$, then $2^{256} \equiv 38 \pmod{p}$, $2^{257} = 2 \cdot 2^{256} \equiv 2 \cdot 38 \pmod{p}$ and so on. Thus, for this prime number, the reduction of a 512-bit number c can be done as follows:

$$c = \{c_{256-511}, c_{0-255}\} \pmod{p} \equiv c_{0-255} + c_{256-511} \cdot 38 \tag{4}$$

The result can still be bigger than p. Thus, this step must be repeated twice and finally p must be subtracted for a complete reduction. This prime is used in Curve25519.

Algorithm 6: Montgomery multiplication

Input: $a, b, m, R, \beta = -N^{-1} \pmod{R}$
Output: $c = a \cdot b \pmod{m}$

1 $\bar{a} = a \cdot R \pmod{N}$
2 $\bar{b} = b \cdot R \pmod{N}$
3 $p = \bar{a} \cdot \bar{b}$
4 $\bar{c} = (p + (p \cdot \beta \pmod{R}) \cdot m)/R$
5 $c = (\bar{c} + (\bar{c} \cdot \beta \pmod{R}) \cdot m)/R$
6 **return** c

It works similar also for more complex prime numbers, as long as their coefficients are powers of 2. For instance, NIST P-256 uses the modulus $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. Let c be a 512-bit number, which can be written as $c = \{c_{15}, ..., c_{01}, c_{00}\}$, whereby c_i are 32-bit numbers. Then reduction of c is defined by [GFD09]

$$c \equiv c' \pmod{p} = s_0 + 2 \cdot s_1 + 2 \cdot s_2 + s_3 + s_4 - s_5 - s_6 - s_7 - s_8, \tag{5}$$

where s_i is a 256-bit number and is given by one of following equations.

$$\begin{aligned}
s_0 &= \{c_{07}, \ c_{06}, \ c_{05}, \ c_{04}, \ c_{03}, \ c_{02}, \ c_{01}, \ c_{00}\} \\
s_1 &= \{c_{15}, \ c_{14}, \ c_{13}, \ c_{12}, \ c_{11}, \ 0, \ 0, \ 0\} \\
s_2 &= \{\ 0, \ c_{15}, \ c_{14}, \ c_{13}, \ c_{12}, \ 0, \ 0, \ 0\} \\
s_3 &= \{c_{15}, \ c_{14}, \ 0, \ 0, \ 0, \ c_{10}, \ c_{09}, \ c_{08}\} \\
s_4 &= \{c_{08}, \ c_{13}, \ c_{15}, \ c_{14}, \ c_{13}, \ c_{11}, \ c_{10}, \ c_{09}\} \\
s_5 &= \{c_{10}, \ c_{08}, \ 0, \ 0, \ 0, \ c_{13}, \ c_{12}, \ c_{11}\} \\
s_6 &= \{c_{11}, \ c_{09}, \ 0, \ 0, \ c_{15}, \ c_{14}, \ c_{13}, \ c_{12}\} \\
s_7 &= \{c_{12}, \ 0, \ c_{10}, \ c_{09}, \ c_{08}, \ c_{15}, \ c_{14}, \ c_{13}\} \\
s_8 &= \{c_{13}, \ 0, \ c_{11}, \ c_{10}, \ c_{09}, \ 0, \ c_{15}, \ c_{14}\}
\end{aligned} \tag{6}$$

2.3 Introduction to Finite Fields

A finite field is also called Galois field and it is a set that contains a finite number of elements and defines operations addition (denoted by $+$) and multiplication (denoted by $\cdot$) on these elements. Additionally, it has the following arithmetic properties [HVM04]:

- The result of an operation must also be an element of the group.
- $(\mathbb{F}, +)$ is an abelian group with (additive) identity denoted by 0.
- $(\mathbb{F} \setminus \{0\}, \cdot)$ is an abelian group with (multiplicative) identity denoted by 1.
- The distributive law holds: $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in \mathbb{F}$.

The notation of a finite field is $\mathbb{F}_{p^m}$ or $GF(p^m)$. Thereby p is a prime number called the characteristic of the field and m is a positive integer. For every combination of p and m, there exists a finite field containing p^m elements. Moreover, two finite fields with the same number of elements are isomorphic. If $p = 2$, it is called binary field ($\mathbb{F}_{2^m}$), and if $m = 1$, it is called a prime field ($\mathbb{F}_p$).

2.3.1 Prime Fields

A prime field $\mathbb{F}_p$ consists of p numbers $\{0, 1, 2, ...p - 1\}$. Thereby, additions and multiplications are performed modulo p. This means that if a result r is not within $[0, p - 1]$, it is reduced so that $r \equiv r' = r - k \cdot p, \ 0 \leq r' \leq p - 1$, where k is an integer.

Example 3. *Arithmetic operations on prime field* $\mathbb{F}_{11}$:

- Addition: $7 + 10 \equiv 6 \pmod{11}$, since $7 + 10 = 17,\ 17 - 1 \cdot 11 = 6$
- Subtraction: $7 - 10 \equiv 8 \pmod{11}$, since $7 - 10 = -3,\ -3 + 1 \cdot 11 = 8$
- Multiplication: $7 \cdot 10 \equiv 4 \pmod{11}$, since $7 \cdot 10 = 70,\ 70 - 6 \cdot 11 = 4$
- Inversion: $7^{-1} \equiv 8 \pmod{11}$, since $7 \cdot 8 = 56,\ 56 - 5 \cdot 11 = 1$

2.3.2 Binary Fields

A binary field $\mathbb{F}_{2^m}$ can be constructed from a polynomial base. Then it can be written as $\mathbb{F}_{2^m} = \{a_{m1} \cdot z^{m1} + a_{m2} \cdot z^{m2} + ... + a_2 \cdot z^2 + a_1 \cdot z^1 + a_0 \cdot z^0\}$. Thereby $a_i \in \{0, 1\}$ and $f(z)$ is an irreducible binary polynomial of degree m. An irreducible polynomial cannot be factored as a product of two binary polynomials each of degree less than m. On arithmetic operations each coefficient is calculated separately and modulo 2. Thus, addition and subtraction lead to the same result and correspond to a boolean XOR operation [HVM04].

To reduce results of multiplication, which are larger than $f(x)$, long division is used.

Example 4. *All elements of* $\mathbb{F}_{2^4}$:

$\{a_3, a_2, a_1, a_0\}$	$f(z)$	$\{a_3, a_2, a_1, a_0\}$	$f(z)$
0000	0	1000	z^3
0001	1	1001	$z^3 + 1$
0010	z	1010	$z^3 + z$
0011	$z + 1$	1011	$z^3 + z + 1$
0100	z^2	1100	$z^3 + z^2$
0101	$z^2 + 1$	1101	$z^3 + z^2 + 1$
0110	$z^2 + z$	1110	$z^3 + z^2 + z$
0111	$z^2 + z + 1$	1111	$z^3 + z^2 + z + 1$

Example 5. *Arithmetic operation on* $\mathbb{F}_{2^4}$ *with* $f(z) = z^4 + z^2 + 1$:

- Addition: $(z^3 + z^2) + (z^3 + z^1 + z^0) = (z^2 + z^1 + z^0)$, since $1100_2 \oplus 1011_2 = 0111_2$
- Subtraction: $(z^3 + z^2) - (z^3 + z^1 + z^0) = (z^2 + z^1 + z^0)$, since $1100_2 \oplus 1011_2 = 0111_2$
- Multiplication : $(z^3 + z^2) \cdot (z^3 + z^1 + z^0) = (z^6 + z^4 + \cancel{z^3} + z^5 + \cancel{z^3} + z^2) = (z^3 + z^1)$,
  since the remainder of $1110100_2 \pmod{10101_2}$ is 1010_2

$$\begin{array}{rl} & 0100000 \\ \oplus & \underline{10101} \\ & 001010 \end{array} \qquad \begin{array}{rl} & 1110100 \\ \oplus & \underline{10101} \\ & 0100000 \end{array}$$

- Inversion : $(z^3 + z^2)^{-1} = (z^1 + z^0)$, since $(z^3 + z^2) \cdot (z^1 + z^0) = (z^0)$

2.4 Field Inversion in Prime Fields

The inverse x of a in $\mathbb{F}_|$ is the unique element so that $a \cdot x \equiv 1 \pmod{p}$. In this section we present three algorithms which can be used to find the inverse element. Beside the presented algorithms there exist, for instance, several adaptations of the Euclidean algorithms [Knu05, MVOV96], algorithms using the Chinese reminder theorem [MVOV96], the almost inverse algorithm [SOOS95] or methods based on Itoh and Tsujiis inversion [IT88].

2.4.1 Extended Euclidean Algorithm

The Extended Euclidean Algorithm can be used to calculate the inverse of numbers in prime fields. Let a and b be two integers, where $a \neq 0, b \neq 0, b \geq a$ and $b = q \cdot a + r$. Then the greatest common divisor of a and b is the largest integer $d = gcd(a, b)$ that divides a and b. Moreover, a and b can be written as $a = a' \cdot d$ and $b = b' \cdot d$. Furthermore, $gcd(a, b) = gcd(b - c \cdot a, a)$, since $b - c \cdot a = b' \cdot d - c \cdot a' \cdot d = d \cdot (b' - c \cdot a')$. Combining the facts that $b = q \cdot a + r$ and $gcd(a, b) = gcd(b - c \cdot a, a)$, it can be concluded that $gcd(a, b) = gcd(r, a)$, where $(r, a) < (a, b)$. In the Euclidean Algorithm this step is repeated until one of the factors is zero. Then the other factor must be d, since the result of $gcd(a, b) = gcd(r, a) = ... = gcd(0, d)$ still must be d. The algorithm determines, because the values are strictly decreasing. In the following, u and v represent the intermediate values of $gcd(a, b) = gcd(u, v) = d$ during the calculation.

The Euclidean Algorithm can be extended that it maintains x_1, y_1, x_2, y_2 so that $a \cdot x_1 + b \cdot y_1 = u$ and $a \cdot x_2 + b \cdot y_2 = v$ in each iteration. Remember that the algorithm determines if $u = 0$ and $v = gcd(a, b)$. If b is a prime and $0 < a \leq b$, $gcd(a, b) = v = 1$. Thus, in the last iteration $1 = a \cdot x_2 + b \cdot y_2 \equiv a \cdot x_2 \pmod{b}$, since $b \cdot y_2 \equiv 0 \pmod{b}$. As a result of $a \cdot x_2 \equiv 1 \pmod{b}$, $x_2 = a^{-1} \pmod{b}$. Thereby, y_1 and y_2 are not relevant for the result. These observations lead to the Extended Euclidean Algorithm, as described in Algorithm 7 [HVM04]. In the algorithm, we check $u \neq 1$, because we swap the values of u and v at the end of the iteration to prepare them for the next iteration. An example to this algorithm can be found in Example 6.

2.4.2 Binary GCD Algorithm

The disadvantage of Algorithm 7 is that it needs a division in Line 6. In the binary GCD algorithm this problem is solved by replacing the division by a right shift (division by 2) and subtraction. The complete description can be found in Algorithm 8. Looking at the algorithm, it is clear that at the start and the end of each iteration at least one of u and v are even. Thus, at least one of u and v are divided by two. Therefore, the total number of iterations is at most $2n$, where n is the maximum length of a and b [HVM04]. An example to this algorithm can be found in Example 7 in the appendix.

Algorithm 7: Extended Euclidean algorithm

Input: Prime $p, a \in [1, p-1]$
Output: $a^{-1} \pmod{p}$

```
1  u = a
2  v = p
3  x_1 = 1
4  x_2 = 0
5  while u ≠ 1 do
6      q = ⌊v/u⌋
7      r = v − q · u
8      x = x_2 − q · x_1
9      v = u
10     u = r
11     x_2 = x_1
12     x_1 = x
13 return x_1 (mod p)
```

2.4.3 Fermat's Little Theorem

Fermat's little theorem states that if p is a prime $a^p \equiv a \pmod{p}$ for any integer a. It can be concluded that if $a < p, a^{p-1} \equiv 1 \pmod{p}$ and further $a^{p-2} = ap - 1/a \equiv a^{-1}$ [Fer]. Algorithm 9 is a very generic variant using Fermat's little theorem. It can be optimized for some special primes, especially for Mersenne or Pseudo-Mersenne primes. Optimized variants for the implemented curves are discussed in Section 4. An example for an optimized variant of the algorithm can be found in Example 8. For a random prime number, the algorithm needs about $3/2 \cdot n$ multiplications and reductions because on average half of the bits are one.

Algorithm 8: Binary GCD Algorithm

Input: Prime $p, a \in [1, p-1]$
Output: $a^{-1} \pmod{p}$

```
1  u = a
2  v = p
3  x_1 = 1
4  x_2 = 0
5  while v ≠ 1 do
6      while u is even do
7          u = u ≫ 1
8          if x_1 is odd then
9              x_1 = x_1 + p //without reduction
10         x_1 = x_1 ≫ 1
11     while v is even do
12         v = v ≫ 1
13         if x_2 is odd then
14             x_2 = x_2 + p //without reduction
15         x_2 = x_2 ≫ 1
16     if u ≥ v then
17         u = u − v; x_1 = x_1 − x_2
18     else
19         v = v − u; x_2 = x_2 − x_1
20 return x_2 (mod p)
```

Algorithm 9: Inversion using Fermat's little theorem

Input: Prime $p, n = \lceil log_2(p) \rceil, a \in [1, p-1]$
Output: $a^{-1} \pmod{p}$

```
1  q = p − 2
2  r = a
3  for i = n − 2 downto 0 do
4      r = r · r (mod p)
5      if q_i = 1 then
6          r = r · a (mod p)
7  return r
```

3 Elliptic Curve Cryptography

Elliptic curves have been studied by mathematicians for several hundred years [Hew05]. Though, their usage for cryptography was first proposed in 1985 by Neal Koblitz [Kob87] and Victor Miller [Mil86] independently. The advantage of elliptic curve cryptography is the short key length. For n bits of security, the key in ECC must have a length of at least $2n$ while for integer-factorization cryptography (RSA) the key length must be much longer. Comparing the runtime of software implementations of 256-bit ECC / 3072-bit RSA security level, ECC is 20 to 60 times faster [Cry04]. Thus, in the last decade elliptic curve cryptography has become more and more important. Many different elliptic curves have been defined and several algorithms for point-multiplication have been developed to improve the speed and the security of the calculations.

In Section 3.1, we will explain the elliptic curve discrete logarithm problem, which is essential for the security of cryptography based on elliptic curves. Then, in Section 3.2 we give an overview over three common elliptic curves: the Weierstrass Curve, the Montgomery Curve, and the twisted Edward's Curve. Afterwards, in Section 3.3 we explain often used algorithms for point-multiplication. Common cryptographic protocols using elliptic curves for key-agreement, encryption, and the creation of digital signature are presented in Section 3.4. Finally, in Section 3.5, we describe possible attacks and countermeasures.

3.1 The Elliptic Curve Discrete Logarithm Problem

The security of elliptic curve cryptography depends on the hardness of solving the elliptic curve discrete logarithm problem (ECDLP). This problem is defined as: given an elliptic curve E defined over a finite field $\mathbb{F}_q$, two points $P, Q \in E(\mathbb{F}_q)$ find $k \in [0, n-1] \in \mathbb{N}$ such that $Q = k \cdot P$. Then k is called the discrete logarithm of Q to the base P [HVM04]. The most naive way to search k is to compute $P, 2P, 3P, ...$ until the result is Q. The running time of this algorithm is $n/2$ on average and n in the worst case. Advanced attacks have a running time of $\mathcal{O}(\sqrt{p})$, where p is the largest prime divisor of n. Therefore, n should be selected sufficiently large, so that this algorithm requires an infeasible amount of computation. However, there is no proof that no efficient algorithm for solving the ECDLP can be found. Nevertheless, the problem has been extensively studied in the last 30 years, but so far no subexponential-time algorithm was found[HVM04].

Table 1 compares the key lengths of integer-factorization cryptography (IFC) with elliptic curve cryptography (ECC) for several levels of security. One can see that the ratio between both key lengths decreases with an increasing level of security.

3.2 Elliptic Curves

Elliptic curves are algebraic curves of third order on which a geometrical defined operations can be performed. In the last thirty years several, different curves have been proposed for cryptographic usage. Some of them have special properties, which allow to do faster calculations. Elliptic curves find also application in an integer factorization algorithm of

Table 1: Comparing key lengths of IFC and ECC [BBB+12]

Bits of security	IFC	ECC	ECC/IFC
80	1024	160	15,63 %
112	2048	224	10,94 %
128	3072	256	8,33 %
192	7680	384	5,00 %
256	15360	>512	>3,33 %

Lenstra ([LJ87]) and a primality proving algorithm by Goldwasser and Kilian ([GK86]). In this section we describe the Weierstrass Curve, the Montgomery Curve, and the twisted Edward Curve. More curves than described in following sections can be found in [BL14].

3.2.1 Weierstrass Curve

An elliptic curve E over a field K is a set of points $(x, y) \in K \times K$ satisfying the Weierstrass equation:

$$y^2 + a_1 \cdot x \cdot y + a_3 \cdot y = x^3 + a_2 \cdot x^2 + a_4 \cdot x + a_6, \tag{7}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and the discriminant Δ must not be zero to ensure that the curve is smooth. If the characteristic of the field K is not equal to 2 or 3, E can be transformed to

$$y^2 = x^3 + a \cdot x + b \tag{8}$$

This variant is called the short Weierstrass form. The discriminant of this curve is $\Delta = 16 \cdot (4 \cdot a^3 + 27 \cdot b^2)$. Thus, for $\Delta \neq 0$, $4 \cdot a^3 \neq 27 \cdot b^2$ must be fulfilled.

All points satisfying Formula 8 form a group. For being a group, they also have to meet four properties. First, the result of an operation on one or two points on the curve must also be an element of the group. Additionally the associative property must hold for all points $((P + Q) + R = P + (Q + R))$. Furthermore, there must be a neutral element $\mathcal{O}$, so that $P + \mathcal{O} = P$. Finally, for every element P there must exist a negative element $-P$, so that $P + -P = \mathcal{O}$.

The group operation is called point addition. The geometrical addition of two points $P + Q = R$ is done by drawing a line through P and Q. This line intersects with the curve in a third point, which must be reflected in the x-axis. The result is unambiguous, because a line can have at most three intersections with an elliptic curve. Formally, the result of $P + Q = R$, where $P = (x_1, y_1), Q = (x_2, y_2)$, and $R = (x_3, y_3)$, can be calculated as follows:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2, \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{9}$$

This formula cannot be used to calculate $P + P = 2P = R$, called point doubling. Instead of the line through two points, for point doubling, the tangent to the curve at that point

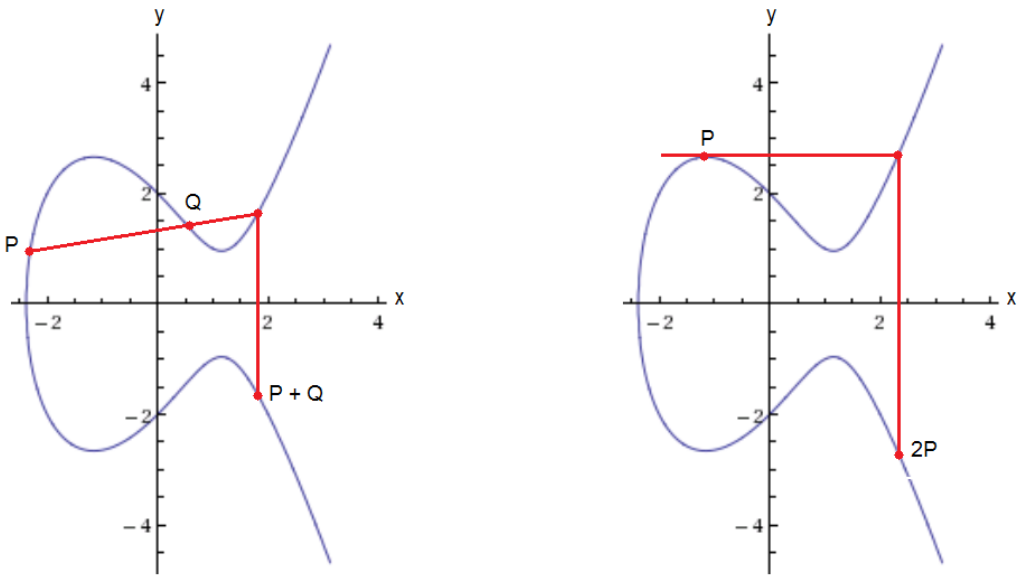
is used. Therefore, the result can be calculated with following formula:

$$x_3 = \left(\frac{3 \cdot x_1^2 + a}{2 \cdot y_1}\right)^2 - 2 \cdot x_1, \quad y_3 = \left(\frac{3 \cdot x_1^2 + a}{2 \cdot y_1}\right)(x_1 - x_3) - y_1 \tag{10}$$

An illustration of the principle behind these two operations can be found in Figure 2. To complete the properties of the group, the neutral element $\mathcal{O}$ must be defined. This element can be calculated by adding a point to its negative equivalent. However, the line connecting these two points is vertical. Looking at Formula 8, it is clear that there is no third intersection with the curve for a vertical line. Therefore, the neutral point is defined to be infinitely far away and thus is mathematically above every point. As a result, every vertical line hits the neutral point.

As one can see in Formulas 9 and 10, these calculations need divisions and thus a field inversion ($\frac{x}{y} = x \cdot y^{-1}$). An inversion on a finite field consists of several multiplications and thus is much more expensive than a single multiplication. Therefore often alternative formulas are used in which the inversion is traded by some additional multiplications. This can be done using projective instead of affine point coordinates. In the projective formulas x and y are substituted by X/Z^c and Y/Z^d, where c and d are constants and depend on the used projective coordinate systems. Thus, the divisor of the resulting formulas can be stored in Z and division is only necessary for conversion back to affine coordinates. The projective form of the Weierstrass equation is

$$Y^2 \cdot Z + a_1 \cdot X \cdot Y \cdot Z + a_3 \cdot Y \cdot Z^2 = X^3 + a_2 \cdot X^2 \cdot Z + a_4 \cdot X \cdot Z^2 + a_6 \cdot Z^3 \tag{11}$$



(a) Point addition $P + Q$

(b) Point doubling $2 \cdot P$

Figure 2: Arithmetic operations on elliptic curves

If the characteristic of the field K is not equal to 2 or 3, E can be transformed to

$$Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3. \tag{12}$$

Comparing this formula with the affine Weierstrass equation in Formula 8, one can see that they are equal for $Z = 1$. Thus, the projective point $(X : Y : 1)$ corresponds with the affine point (x, y). Furthermore, every projective point can be converted to its affine equivalent by $(X : Y : Z) = (X/Z^c : Y/Z^d : 1) = (x, y)$. This conversion must be done only if affine coordinates are needed, typically at the end of the point-multiplication. The values of c and d depend on the used coordinates systems. For standard projective coordinates, $c = 1$ and $d = 1$. Using this representation, the neutral point is at $(0 : 1 : 0)$. Beside standard projective coordinates there exist other representations. For instance, Jacobian projective coordinates, where $c = 2, d = 3$, and $\mathcal{O} = (1 : 1 : 0)$. Different representations lead to different formulas for point addition and doubling. Depending on the implementation, one representation can be more efficient than others [HVM04].

A plot of an exemplary Weierstrass curve can be found in Figure 3a. The parameters $a = -4, b = 4$ for the shown example are chosen in a way, that all values can be represented by a single curve. It exist also parameters for which the values lie on two distinct curves, for instance $a = -4, b = 3$. Then the curve looks similar to the example for the Montgomery curve in Figure 3b.

3.2.2 Montgomery Curve

Peter L. Montgomery introduced a new elliptic curve in 1987 [Mon87]. The Montgomery curve is defined by

$$b \cdot y^2 = x^3 + a \cdot x^2 + x, \tag{13}$$

where $b \cdot (a^2 - 4) \neq 0$ must be fulfilled. This equation can be transformed to the short Weierstrass form by dividing it by b^3 and substituting x by x/b and y by y/b. Thus, every Montgomery curve can be mapped to a Weierstrass curve; they are birationally equivalent.

The formulas for point addition are given by

$$x_3 = \frac{b \cdot (y_2 - y_1)^2}{(x_2 - x_1)^2} - a - x_1 - x_2, \quad y_3 = \frac{(2 \cdot x_1 + x_2 + a) \cdot (y_2 - y_1)}{(x_2 - x_1)} - \frac{b \cdot (y_2 - y_1)^3}{(x_2 - x_1)^3} - y_1 \tag{14}$$

The result of point doubling can be calculated as follows:

$$\begin{aligned} x_3 &= \frac{b \cdot (3 \cdot x_1^2 + 2 \cdot a \cdot x_1 + 1)^2}{(2 \cdot b \cdot y_1)^2} - a - 2 \cdot x_1, \\ y_3 &= \frac{(3 \cdot x_1 + a) \cdot (3 \cdot x_1^2 + 2 \cdot a \cdot x_1 + 1)}{2 \cdot b \cdot y_1} - \frac{b \cdot (3 \cdot x_1^2 + 2 \cdot a \cdot x_1 + 1)^3}{(2 \cdot b \cdot y_1)^3} - y_1 \end{aligned} \tag{15}$$

A point P on this curve can be represented by projective Montgomery coordinates $P = (X : Z)$, without using Y. Thus, there is no distinction between the affine points (x, y) and $(x, -y)$, because they are both given by $(X : Z)$. Thus, an implementation using

Montgomery coordinates has to store only two instead of three projective coordinates. Additionally, state of the art formulas using projective coordinates for Montgomery curves require less arithmetic operations for point addition and point doubling than formulas for Weierstrass curves [BL14]. Curve25519 uses this type of curve and is discussed in Section 4.0.5. A plot of an exemplary Montgomery curve can be found in Figure 3b.

3.2.3 Twisted Edwards Curve

Harold Edwards proposed a new elliptic curve in 2007 [Edw07]. It has the following equation:

$$x^2 + y^2 = c^2 \cdot (1 + x^2 \cdot y^2). \tag{16}$$

Daniel J. Bernstein and Tanja Lange founded its usage in cryptography and pointed out several advantages compared to the Weierstrass form. They generalized the formula above to

$$x^2 + y^2 = c^2 \cdot (1 + d \cdot x^2 \cdot y^2), \tag{17}$$

where $c \cdot d \cdot (1 - d \cdot c^4) \neq 0$. This form is called twisted Edwards curve and is birationally equivalent to an elliptic curve in Montgomery form.

The result of a point addition can be calculated by

$$x_3 = \frac{x_1 \cdot y_2 + y_1 \cdot x_2}{c \cdot (1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2)}, \quad y_3 = \frac{y_1 \cdot y_2 - x_1 \cdot x_2}{c \cdot (1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2)}. \tag{18}$$

The formulas for point doubling are given as

$$x_3 = \frac{2 \cdot x_1 \cdot y_1 \cdot c}{x_1^2 + y_1^2}, \quad y_3 = \frac{(y_1^2 - x_1^2) \cdot c}{2 \cdot c^2 - (x_1^2 + y_1^2)}. \tag{19}$$

More details and an extensive comparison with other curves can be found in [BL07]. A plot of an exemplary Edward's curve can be found in Figure 3c.

3.3 Point-Multiplication Algorithms

Cryptographic algorithms based on elliptic curves typically multiply a point P on the curve with an integer k. This operation is called point-multiplication or scalar multiplication. To calculate $k \cdot P$, point addition and point doubling operations are used. Since k typically has more than hundred binary digits (n), the point-multiplication is a computationally very intensive operation. The number of necessary point operations either is constant for secure implementations or depends on the number of ones in k. Thus, there exist algorithms which converts k in an alternative, more convenient representation, for instance the non-adjacent form (NAF). However, in this section, we present only algorithms we used in our work. Some more can be found in [HVM04].

3.3.1 Binary Method

The most straight forward way to implement point-multiplication is using the binary method. One of the most common variants is double-and-add. It is shown in Algorithm 10. Starting with the most significant bit (MSB) of k the intermediate result Q is doubled in each round. Additionally it adds P to the intermediate result if the bit k_i is one. Since the loop starts at the MSB, this variant is called left-to-right double-and-add. A right-to-left

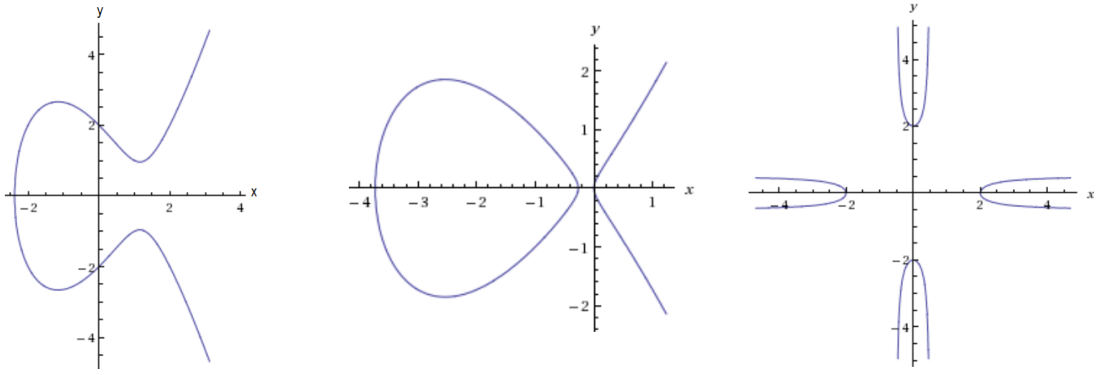
Algorithm 10: Left-to-right binary method for point-multiplication

Input: $k = (k_{n-1}, ..., k_1, k_0)_2, P$
Output: kP

```
1 Q = O
2 for i = n − 1 downto 0 do
3 |   Q = 2Q
4 |   if k_i = 1 then
5 |   |   Q = Q + P
6 return Q
```

variant of this algorithm exists too. However, doing it right-to-left requires two variables. Therefore, the left-to-right variant is more common, because it needs only one variable.

Since on average the number of ones in k is $n/2$, the expected runtime for this algorithm is $n \cdot (A/2 + D)$, where A denotes the point addition and D the point doubling operation. Thereby, P keeps constant during the whole computation. Thus P can be stored in affine coordinates, while Q is in projective coordinates. Some formulas can take advantage of



(a) Weierstrass curve ($a = -4, b = 4$)

(b) Montgomery curve ($a = 4, b = 2$)

(c) Edward's curve ($c = 2$)

Figure 3: Comparison of different elliptic curves

these mixed coordinates and are faster than pure projective coordinates [HVM04]. To demonstrate this algorithm, Example 9 uses it for a integer multiplication.

3.3.2 Montgomery's Method

This algorithm was first proposed by Peter L. Montomery [Mon87] and is also known as the Montgomery (powering) ladder. Meanwhile there exist various variants of the algorithm. A basic variant is shown in Algorithm 11. The advantage of this algorithm is that each

Algorithm 11: Montgomery ladder

Input: $k = (k_{n-1}, ..., k_1, k_0)_2, P$
Output: kP

```
1  R_0 = O
2  R_1 = P
3  for i = n − 1 downto 0 do
4      if k_i = 1 then
5          R_0 = R_0 + R_1
6          R_1 = 2 · R_1
7      else
8          R_1 = R_0 + R_1
9          R_0 = 2 · R_0
10 return R_0
```

iteration needs one addition and one doubling independent of the value of k. This makes Montgomery's method more resistant against implementation attacks (for more details see Section 3.5). However, the expected runtime is $n \cdot (A + D)$. Thus it is on average $A \cdot n/2$ slower than the binary method. Another advantage is that the point addition and doubling can be calculated independently. Thus, these Lines 5 and 6 or rather 8 and 9 can be calculated in parallel to speed up the whole multiplication [JY03]. To demonstrate this algorithm, Example 10 uses it for a integer multiplication.

3.3.3 Fixed-Base Comb Method

Sometimes P has a predefined value, for instance for the elliptic curve digital signature algorithm (see section 3.4.2). Therefore, some multiples of P can be precomputed and stored in a lookup table. As a result, the point-multiplication itself can be accelerated. In the former methods, P is added to Q, when $k_i = 1$. Here, several bits of k are handled at once. Therefore, k is sliced into blocks of length w. In the phase of precomputation, all values of $f \cdot P$, for $0 \leq f \leq 2^w - 1$ are stored in a lookup table. During the actual computation, the slices of k are used as index for the lookup table. With this variant, a speedup of w can be achieved. The runtime is determined by $m/w \cdot (A + D)$. Additional

speedup can be achieved, if the values in the lookup table are stored in affine form. Then, they have a projective representation of $(X : Y : 1)$. Considering, that on Z-coordinate is 1, the formulas for point addition can be optimized. Thus, addition using so called mixed coordinates, is typically faster than pure projective coordinates [HVM04]. A description of the fixed-base comb method can be found in Algorithm 12. To demonstrate this algorithm, Example 11 uses it for a integer multiplication.

Algorithm 12: Fixed-base comb method

Input: Window width w, $d = \lceil n/w \rceil, k = (k_{n-1}, ..., k_1, k_0)_2, P$
Output: kP

```
1  T = 2P
2  LUT[0] = O
3  LUT[1] = P
4  LUT[2] = T
5  for f = 3 to 2^w − 1 do
6  |   T = T + P
7  |   LUT[i] = T
8  Q = O
9  for i = d − 1 downto 0 do
10 |   t = [k_{i·w+w−1}, ..., k_{i·w}]
11 |   for j = 0 to w − 1 do
12 |   |   Q = 2 · Q
13 |   Q = Q + LUT[t]
14 return Q
```

3.4 Cryptographic Protocols

Cryptographic protocols are based on hard mathematical problems. This section describes protocols based on the elliptic curve discrete logarithm problem. It can be combined with the DiffieHellman problem (DHP) to implement a key agreement protocol (ECDH). The Elliptic Curve Digital Signature Algorithm (ECDSA), which is variant of the Digital Signature Algorithm (DSA), can be used to generate digital signatures. With the Elliptic Curve Integrated Encryption Scheme (ECIES), a variant of the ElGamal public-key encryption scheme, data can be encrypted. In the following we describe these three algorithms.

3.4.1 Elliptic Curve Diffie-Hellman

In cryptography it can be distinguished between symmetric- and asymmetric-key algorithms. In symmetric-key algorithms the same key for both encryption and decryption is used. Thus, all parties must have access to the same key. Therefore, the key must be

transmitted over a secure channel or it is generated with a key agreement protocol. For such protocols asymmetric-key algorithms are used. They are called asymmetric-key algorithms, because they use two different keys, a private (secret) and a public one. The public key is used for encryption or to verify a digital signature, while the private key is used for decryption or to create a digital signature. Therefore, these two keys must be the inverse of the other somehow. However, it must be very hard to derive the private key from the public key. Thus, asymmetric-key algorithms are based on hard mathematical problems like integer factorization or the discrete logarithm. In contrast, symmetric-key algorithms can use easier algorithms, because the used key is private. Thus, symmetric-key algorithms are typically faster than asymmetric-key algorithms.

Common key agreement protocols are based on the Diffie-Hellman problem, first proposed by Whitfield Diffie and Martin Hellman [DH76]. The Elliptic Curve Diffie-Hellman protocol is based on this problem. Therefore, each participant must have an elliptic curve public-private key pair (d, Q), where $Q = d \cdot G$ and G is a generator. If Alice (d_A, Q_A) wants to establish a shared key with Bob (d_B, Q_B), then both compute $(x_d, y_d) = d \cdot Q$ doing a point-multiplication. Then the shared key is x_d. This works, because $k_A \cdot Q_B = d_A \cdot d_B \cdot G = d_B \cdot d_A \cdot G = d_B \cdot Q_A$ [BJS07].

3.4.2 Elliptic Curve Digital Signature Algorithm

To proof the authenticity and integrity of a digital message or document it can be signed digitally. Therefore, the Elliptic Curve Digital Signature Algorithm can be used. It is the most widely standardized elliptic curve-based signature scheme, first standardized in [Ser05]. To create a digital signature of message m the following steps are necessary:

- Calculate $h = \text{HASH}(m)$, where HASH is a cryptographic hash function
- Let z be the $log_2(n)$ leftmost bits of h
- Select a random integer k from $[1, n-1]$, where n is the order of G
- Calculate $(x_1, y_1) = k \cdot G$
- Calculate $r = x_1 \pmod n$, restart if $r = 0$
- Calculate $s = k^{-1}(z + r \cdot d) \pmod n$, where d is the private key, restart if $s = 0$
- The signature consists of the tuple (r, s)

The following steps are necessary to verify a digital signature:

- Verify r and s are integers in [1, n-1]
- Calculate $h = \text{HASH}(m)$
- Let z be the $ld(n)$ leftmost bits of h
- Calculate $w = s^{-1} \pmod n$
- Calculate $u_1 = z \cdot w \pmod n$, $u_2 = r \cdot w \pmod n$
- Calculate $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_A$
- If $r \equiv x_1 \pmod n$, the signature is valid

3.4.3 Elliptic Curve Integrated Encryption Scheme

ECIES is an encryption standard ([Ser01]) based on a proposal of Bellare and Rogaway [ABR99], which is a variant of the ElGamal public-key encryption scheme [ElG85]. The ElGamal encryption scheme has similarities to the Diffie-Hellman key agreement protocol. ECIES is a hybrid encryption scheme which uses a public-key and a symmetric-key crypto-system as well. The first key is responsible to encapsulate the second key, which is used to encrypt the data. Thus, the user must provide an elliptic curve public-private key pair (d, Q), where $Q = d \cdot G$ and G is a generator. This scheme also provides semantic security, so that it is not possible to reveal information about the plain text from the content and length of the cipher text. Therefore, ECIES uses a key derivation function (KDF) and a message authentication code (MAC). A key derivation function can derive a key with an arbitrary length from an input. A message authentication code can be used to authenticate a message and allows the detection of message changes. For encryption, the public key Q of the receiver is used:

- Select a random integer k from $[1, n-1]$, where n is the order of G
- Calculate $(x_1, y_1) = k \cdot G, (x_2, y_2) = h \cdot k \cdot Q$, where $h = \frac{|E(\mathbb{F}_q)|}{n}$
- Calculate the keys $k_1 || k_2 = \mathrm{KDF}(x_2)$
- Calculate $C = E_{k_1}(m)$, $T = \mathrm{MAC}_{k_2}(C)$
- Send (R, C, T)

To decrypt the cipher (R, C, T), the private key d of the receiver must be used, whereby nobody else can decrypt it:

- Calculate $(x_2, y_2) = h \cdot d \cdot R$
- Calculate the keys $k_1 || k_2 = \mathrm{KDF}(x_2)$
- Check if $T = \mathrm{MAC}_{k_2}(C)$
- Calculate $m = D_{k_1}(C)$

3.5 Attacks and Countermeasures

Modern cryptographic algorithms are designed in a way that they are secure, even if the adversary has a complete description of the algorithm, and knows all public keys. Furthermore, the security must be given if she can read all messages and even if she can send her own messages. The only thing she does not know is the secret key. Thus, nowadays an attacker has two possibilities to break the security of such systems. In the first type of attacks, some additional informations about the system are collected. Therefore she can use side-channel attacks, fault attacks and probing attacks, which are presented in the following subsections.

Second, she can use several algorithms to calculate the discrete logarithm of $Q = k \cdot P$. The only possibility to prevent such attacks is to use a key with an adequate length. Some examples are the Pohlig-Hellman algorithm [PH78], Pollard's rho algorithm for logarithms [Pol78] and Shanks baby step-giant step algorithm [Sha71]. These algorithms

have a runtime of $\mathcal{O}(\sqrt{p})$. Therefore, the key in ECC must be twice as long as the needed bit security. The current record for the longest broken discrete logarithm is held by Wenger and Wolfger. They broke a 113-bit elliptic curve in extrapolated 24 days [WW14]. In future, it may be possible to break much larger keys by using quantum computers. To break a 256-bit elliptic curve about 1800 qubits are necessary to break them in $6 \cdot 10^9$ 1-qubit additions [PZ03]. The actual biggest commercially available quantum computer D-Wave Two has a 512-qubit CPU [DW14].

3.5.1 Side-Channel Attacks

Side-channel attacks are passive attacks in which the calculation time or physical properties such as the power consumption or the electro-magnetic emanation of a device are exploited. These attacks consist of two phases. First, side-channel information is measured. Then, in the second phase one calculates expected values for a guessed key. These values are compared with the measured values. If the measured values match the calculated ones, the guessed key is correct. This technique can only be used if the measured values depend at least partially on the used secret key [HVM04].

Timing analysis. This method uses the fact that the execution time often depends on the processed values. It was first published by Paul C. Kocher [Koc96]. With regard to elliptic curve cryptography, an attacker can measure the running time of a point-multiplication. If the implementation uses the binary method, the runtime depends among other things on the number of 1s in the key. Additionally, the duration of a reduction after an integer multiplication may depend on the length of the result and thus on the input values. Repeated executions with different input but the same key can be used to perform statistical correlation analysis of timing information to recover the secret key.

To protect against such attacks, the running time of the implementation must be completely independent of the values of all inputs and keys. Thus, for instance, it is recommended to use rather Montgomery's method than the binary method for point-multiplication. Nonetheless, it is often hard to make an implementation completely time-invariant, especially if it is a software implementation. This is because the time it takes to load data from memory depends on if it was recently used and still in the cache or not. The small time difference between loading data from cache or not can also be used to reveal the secret. A detailed description for a cache-timing attack against AES can be found in [Ber05].

Power analysis. In power analysis the attacker measures the power consumption of a cryptographic hardware device and was first published in 1998 [KJJ99]. Therefore, the attacker must have physical access to the device. The actual power consumption depends on the actually executed instruction and data. This is because for different instructions different parts of the hardware are active. Additionally, the power consumption of a transistor is higher if it changes its value than if it stays at the same value. Thus, conclusions on the values can be drawn from the power consumption.

In a simple power analysis, information is deduced directly by examining the power trace from a single execution. With regard to elliptic curve cryptography, an attacker is able to distinguish between a point doubling and point addition operation. If the binary method for point-multiplication is used, the key could be directly read from the power trace, if a typical power profile of both operations is known.

Differential power analysis, exploits the fact that the power consumption is related to the current values. Since these variations are typically much smaller than those used for simple power analysis, statistical methods must be used. Therefore, the power trace of several executions must be collected. Then, the attacker makes iteratively guesses for each bit of the key and partitions the power traces into two groups according to the predicted value of the bit. If the guess is correct, the difference in the power consumption between two groups is larger than for false guesses.

Similar to timing attacks, a countermeasure against this attack is to make the execution of an algorithm independent from the data. Another countermeasure is to randomize the intermediate values [Cor99], the register usage, or the instruction execution [MMS01]. For instance, it is possible to calculate $k \cdot (P \cdot R)$ and divide it at the end by R to get the result of the point-multiplication $k \cdot P$. Another possibility is to use a workflow which reduces the information leakage through side channels as described in [TV03, TV05].

Electro-magnetic analysis. Electro-magnetic emanations are caused by the flow of current through a CMOS device and can be collected by placing a sensor close to the device. Measuring electro-magnetic emanations is more sensitive than measuring the power consumption. Therefore, the attacker must have very close access to the devices and the target component must be isolated the other components. The signals of different gates have different characteristics and thus can be separated and analysed individually. This is the advantage compared to power analysis, where only the combined power of all active units can be measured. For measurement typically a coiled copper wire of outer diameters varying between 150 and 500 microns is placed directly above the target component. Once, data is recorded, similar methods as for power analysis can be applied to gather secret information [GMO01].

One possible countermeasure is to make the program execution independent from the processed data. Furthermore, additional layers of aluminium or copper can be added to dam the electro-magnetic emanations, because they are not good ferromagnetic metals. Sensitive parts of the chip can even surrounded with a Faraday cage. There exist also several technologies with very low power consumption and thus a reduced electro-magnetic emanation. More details can be found in [QS01].

3.5.2 Fault Attacks and Probing Attacks

In the former presented methods the attacker is passive and uses information the device provides anyway. The attacks in the techniques described in this section are active. This means that the adversary must interfere the conditions of execution, the execution itself, or open the hardware to get information she would not get otherwise.

Fault attacks. In contrast to side-channel analysis, in fault attacks the behavior of the device is influenced. Thereby, a fault is injected intentionally to gain erroneous computation results. Based on the result, it is possible to reveal information about the secret key. Such errors may be introduced by an adversary who has physical access to the device [BDL97].

One example is the safe-error attack, which can be applied to implementations achieving a constant running time with dummy operations. Typically, the result of these dummy operations has no influence on the real result. Thus, if an error is induced while a particular operation is executed and the result remains the same, the attacker knows that that operation was a dummy operation. The binary method for point-multiplication can be secured against side-channel attacks by doing a dummy point addition when the actual bit of the key is zero. This variant is called double-and-add-always. If the adversary can execute this algorithm several times with the same secret key and interfere a single point addition operation, she can learn the value of a single bit of the key. Doing this for all bits of the secret key, she can deduce the complete key.

A possible countermeasure is to perform crucial parts redundant on multiple places of the chip maybe in different ways. It is harder to manipulate several parts of the chip at once. If the results of the different calculations are different the calculations must have been manipulated. Then the execution can be stopped. Additionally error-detection codes can be used. More details can be found in [MSY06].

Probing attacks. While fault attacks try to manipulate the device, probing attacks aim to spy on intermediate values directly on the chip. Therefore, the attacker places a metal needle on a wire of interest and measures the value carried along that wire during the computations. Thus, the chip must be opened to place a probe at the wire. The advantage of this method is that it allows access directly to the values inside the chip. In return it is quite expensive.

It is very hard to protect the chip against this kind of attack. One possibility is to use light-sensitive photodiodes and seismometers to detect if the chip is opened [ABCS06]. Another possibility is to distribute the computations into several areas of the chip, so that the whole secret is never at one place. Thus, more probes must be performed, which increases the complexity and costs. A further possibility is to use randomized circuits as proposed in [ISW03].

4 Efficient Implementation of ECC

An elliptic curve is defined by a set of domain parameters. It starts with the underlying field, which is a prime field for all of our implemented curves. However, all of them use a different prime p. Furthermore, they use different curve parameters (a, b) or even different equations. They also use different base points $G = (G_x, G_y)$ as generator.

There is a vast number of ways to calculate point doubling and addition for each curve. The right choice depends on several properties. For each set of formulas following characteristics can be defined:

- M ... number of needed multiplications
- S ... number of needed squaring operations
- A ... number of needed additions
- M_x ... number of needed multiplications with the constant x
- R ... number of needed registers

To get a rating of the formulas, these values can be weighted and accumulated. Thereby, the weights depend on the architecture. For instance, it is possible to do a squaring faster than a multiplication. Thus, a common weighting is $1S = 0.8M$. However, to keep our implementation as small as possible, we did not implement an extra squaring operation. Thus, squaring operations are done using multiplication and so $1S = 1M$. Additions are typically much faster than multiplications. Thereby, multiplications with small constants can be implemented as a series of additions. For a small implementation the number of registers is also a crucial factor.

In the following sections, we give more details of the three implemented curves and the used formulas.

4.0.3 Brainpool P256r1

In 2005, the working group ECC-Brainpool has published elliptic curves with 160 up to 512 bits. They chose all parameters pseudo-randomly to prevent attacks against a certain class of parameters and patent violations. For 256 bits they chose following parameters [LM10]:

- y^2 $= (x^3 + a \cdot x + b) \pmod p$ (short Weierstrass curve)
- a $=$ `0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9`
- b $=$ `0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6`
- p $=$ `0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377`
- G_x $=$ `0x8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE3262`
- G_y $=$ `0x547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997`

In our implementation we use the Montgomery's method for point-multiplication and point representation with a common Z-coordinate, where $x = X/Z, y = Y/Z$ [HJS11]. The formulas have costs of $11M + 4S + 1M_a + 1M_{4b} + 14A; (7 + X_0, Y_0)R$, where M_{4b} is a multiplication with $4 \cdot b$. For some protocols X_0 and Y_0 are constant and can be stored in a ROM instead of register. Therefore, the costs for them are stated explicitly. The explicit

formulas to calculate $X1' = X1 + X2,\ X2' = 2 \cdot X2$ can be found in Algorithm 13. In the evaluation this algorithms is referred as CoZ1. To recover the y-coordinate Algorithm 16 must be used after the scalar multiplication. Their costs are $8M + 2S + 1M_a + 1M_{4b} + 8A; (7 + X_0, Y_0)R$.

We also implemented faster variants of these formulas, which need fewer multiplications but more registers. The variant CoZ2 has costs of $9M+5S+1M_a+1M_{4b}+14A; (8+X_0, Y_0)R$. For these formulas the same algorithm for recovery can be used as for CoZ1. A full description can be found in Algorithm 14.

The fastest variant of these formulas reuses results from the former iteration and is referred as CoZ3. Therefore, additional registers are necessary to store these values. For the first iteration, these values must be initialized as follows: $Ta = a \cdot Z1,\ Tb = 4b \cdot Z2$, and $TD = X_0 \cdot Z$. In return, they need fewer multiplications and have costs of $10M + 5S + 13A; 10R$. The according formulas for point doubling and addition can be found in Algorithm 15. A list of the formulas to recover the y-coordinate is given in Algorithm 17.

4.0.4 NIST P-256

In 2000, the National Institute of Standards and Technology (NIST) has published elliptic curves over prime fields with 192 up to 521 bits. For 256 bits they chose following parameters [GFD09]:

- $y^2 = (x^3 + a \cdot x + b) \pmod p$ (short Weierstrass curve)
- $a =$ `-0x3`
- $b =$ `0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B`
- $p = 2^256 - 2^224 + 2^192 + 2^96 - 1$
  $=$ `0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff`
- $G_x =$ `0xDE2444BEBC8D36E682EDD27E0F271508617519B3221A8FA0B77CAB3989DA97C9`
- $G_y =$ `0xC093AE7FF36E5380FC01A5AAD1E66659702DE80F53CEC576B6350B243042A256`

Because this curve is the same type as the curve of Brainpool, we use the same formulas for point doubling and addition. However, the given choice of a and p allows two speed ups compared to random values. Since p is a Pseudo-Mersenne prime, fast reduction is possible. The second advantage is that M_a can be done by two additions. As a result the same formulas as for the Brainpool curve can be implemented faster.

Additionally, we also implemented a variant using the fixed-base comb method. Therefore we cannot use the same formulas as for Montgomery's method. The used formulas can be found in Algorithm 18 and 19. In these formulas the coordinates are represented as Jacobian coordinates, where $x = X/Z^2,\ y = Y/Z^3$. The values in the look-up table are stored as affine coordinates, with $Z = 1$. With this fact the formulas for the addition are faster than general ones.

4.0.5 Curve25519

In 2006 Daniel J. Berstein proposed Curve25519 based on an elliptic curve with the Montgomery-Form. It uses [Ber06]:

- $b \cdot y^2 = (x^3 + a \cdot x + x) \pmod{p}$ (Montgomery curve)
- $a = \texttt{0x76d06}$
- $b = \texttt{0x1}$
- $p = 2^{255} - 19$
  $= \texttt{0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed}$
- $G_x = \texttt{0x9}$

Compared to the two former curves, the prime of this curve has only 255 bits. This curve only uses the x-coordinate and the parameters allow fast arithmetic.

In our implementation we optimized the X-coordinate double-and-add formulas, where $x = X/Z$ given in [Ber06]. They have costs of $5M + 4S + M_{a24} + 8A; (6 + X_0)R$, where $a24 = (a + 2)/4$. The explicit formulas can be found in Algorithm 20.

Since the most efficient formulas for Weierstrass curves use a common Z-coordinate, we tried to search such a variant for Curve25519. A naive approach to convert the XZ-coordinates into CoZ representation is to multiply both Z-coordinates and each X-coordinates with the other Z-coordinate:

$$\begin{aligned} X1 &= X_1 \cdot Z_2 \\ X2 &= X_2 \cdot Z_1 \\ Z &= Z_1 \cdot Z_2 \end{aligned} \tag{20}$$

Thus, the naive approach requires three additional multiplications. In return storing the coordinates requires only three registers. However, our best result needs $8M + 5S + 1M_a + 7A; (6 + X_0)R$ (see Algorithm 21). Maybe this can be explained by the fact that in the original formulas only few coefficients are multiplied by both Z_1 and Z_2, which can be simplified by $Z_1 = Z_2$.

Algorithm 13: CoZ-coordinate double-and-add formulas for short Weierstrass curves, variant 1 with costs of $11M + 4S + 1M_a + 1M_{4b} + 14A; (7 + X_0, Y_0)R$: $(X1, Z1)' = (X1, Z1) + (X2, Z2), (X2, Z2)' = 2 \cdot (X2, Z2), Z = Z1 = Z2$

1 $R1 = X1 * X2;$	12 $X1 = R4 * Z;$	23 $X2 = X2 + X2;$
2 $R3 = Z * Z;$	13 $R3 = R3 + R3;$	24 $R2 = R2 - R1;$
3 $R4 = Z * R3;$	14 $R3 = R3 + R1;$	25 $R1 = R4 + R4;$
4 $R2 = R3 * a;$	15 $Z = X2 * R4;$	26 $R4 = X2 + R4;$
5 $R1 = R1 + R2;$	16 $R4 = R1 * X2;$	27 $X2 = R2 * R2;$
6 $X1 = X1 + X2;$	17 $R1 = X2 * X2;$	28 $R1 = X2 - R1;$
7 $R3 = X1 * R1;$	18 $R2 = R1 + R2;$	29 $X2 = R1 * Z;$
8 $X1 = X1 - X2;$	19 $R1 = R1 + R1;$	30 $Z = X1 * R4;$
9 $X1 = X1 - X2;$	20 $X2 = X0 * X1;$	31 $X1 = R3 * R4;$
10 $R1 = b4 * R4;$	21 $R3 = R3 - X2;$	
11 $R4 = X1 * X1;$	22 $X2 = R1 * R2;$	

Algorithm 14: CoZ-coordinate double-and-add formulas for short Weierstrass curves, variant 2 with costs of $9M + 5S + 1M_a + 1M_{4b} + 14A; (8 + X_0, Y_0)R$: $(X1, Z1)' = (X1, Z1) + (X2, Z2), (X2, Z2)' = 2 \cdot (X2, Z2), Z = Z1 = Z2$

1 $R2 = Z * Z;$	11 $R5 = R5 + R5;$	21 $R1 = R1 - R3;$
2 $R3 = R2 * a;$	12 $R5 = R5 + R2;$	22 $X1 = X1 + X2;$
3 $R1 = Z * R2;$	13 $R1 = R1 + R3;$	23 $X2 = X1 * R1;$
4 $R2 = B4 * R1;$	14 $R3 = X1 * X1;$	24 $X2 = X2 + R2;$
5 $R1 = X2 * X2;$	15 $R1 = R1 + R3;$	25 $R2 = Z * R3;$
6 $R5 = R1 - R3;$	16 $X1 = X1 - X2;$	26 $Z = X0 * R2;$
7 $R4 = R5 * R5;$	17 $X2 = X2 + X2;$	27 $X2 = X2 - Z;$
8 $R1 = R1 + R3;$	18 $R3 = X2 * R2;$	28 $X1 = R5 * X2;$
9 $R5 = X2 * R1;$	19 $R4 = R4 - R3;$	29 $X2 = R3 * R4;$
10 $R5 = R5 + R5;$	20 $R3 = X1 * X1;$	30 $Z = R2 * R5;$

Algorithm 15: CoZ-coordinate double-and-add formulas for short Weierstrass curves, variant 3 with costs of $10M + 5S + 13A; 10R$: $(X1, Z1)' = (X1, Z1) + (X2, Z2), (X2, Z2)' = 2 \cdot (X2, Z2), Z = Z1 = Z2$

1 $R2 = X1 - X2;$	11 $R3 = R5 * R2;$	21 $X2 = R1 * R4;$
2 $R1 = R2 * R2;$	12 $R3 = R3 + Tb;$	22 $R2 = R1 * R3;$
3 $R2 = X2 * X2;$	13 $R5 = X1 + X2;$	23 $R3 = R2 * Tb;$
4 $R3 = R2 - Ta;$	14 $R2 = R2 + Ta;$	24 $R4 = R2 * R2;$
5 $R4 = R3 * R3;$	15 $R2 = R2 - R1;$	25 $R1 = TD * R2;$
6 $R5 = X2 + X2;$	16 $X2 = X1 * X1;$	26 $R2 = Ta * R4;$
7 $R3 = R5 * Tb;$	17 $R2 = R2 + X2;$	27 $Tb = R3 * R4;$
8 $R4 = R4 - R3;$	18 $X2 = R5 * R2;$	28 $X1 = X1 - R1;$
9 $R5 = R5 + R5;$	19 $X2 = X2 + Tb;$	29 $TD = R1;$
10 $R2 = R2 + Ta;$	20 $X1 = R3 * X2;$	30 $Ta = R2;$

Algorithm 16: (X, Y,Z)-recovery from CoZ-coordinate for short Weierstrass curve for variant 1,2 with costs of $8M + 2S + 1M_a + 1M_{4b} + 8A; (7 + X_0, Y_0)R$: $(X, Y, Z)' = recover(X1, X2, Z)$

1 $R1 = X0 * Z;$	8 $R3 = X2 * a;$	15 $R2 = R1 * X1;$
2 $R2 = X1 - R1;$	9 $R2 = R2 + R3;$	16 $X1 = R2 * X2;$
3 $R3 = R2 * R2;$	10 $R3 = R2 * R1;$	17 $R2 = X2 * Z;$
4 $R4 = R3 * X2;$	11 $R3 = R3 - R4;$	18 $Z = R2 * R1;$
5 $R2 = R1 * X1;$	12 $R3 = R3 + R3;$	19 $R4 = B4 * R2;$
6 $R1 = X1 + R1;$	13 $R1 = Y0 + Y0;$	20 $X2 = R4 + R3;$
7 $X2 = Z * Z;$	14 $R1 = R1 + R1;$	

Algorithm 17: (X, Y,Z)-recovery from CoZ-coordinate for short Weierstrass curve for variant 1,2 with costs of $10M + 3S + 8A; (9 + X_0, Y_0)R$:
$(X, Y, Z)' = recover(X1, X2, Z, TD, Ta, Tb)$

1 $R1 = TD * X1;$	8 $R4 = R4 - R3;$	15 $R3 = R3 + R3;$
2 $R2 = R1 + Ta;$	9 $R4 = R4 + R4;$	16 $X = R3 * R1;$
3 $R3 = X1 + TD;$	10 $R4 = R4 + Tb;$	17 $R1 = R2 * TD;$
4 $R4 = R2 * R3;$	11 $R2 = TD * TD;$	18 $Z = R3 * R1;$
5 $R3 = X1 - TD;$	12 $R3 = X1 * R2;$	19 $R2 = X0 * X0;$
6 $R2 = R3 * R3;$	13 $R1 = X0 * R3;$	20 $R3 = R2 * X0;$
7 $R3 = R2 * X2;$	14 $R3 = yd + yd;$	21 $Y = R3 * R4;$

Algorithm 18: Point doubling for short Weierstrass curve where $a = -3$ in Jacobian coordinates with costs of $3M + 5S + 26A; 7R$:
$(X1, Y1, Z1)' = 2 \cdot (X1, Y1, Z1)$

1 $R1 = Z1 * Z1;$	10 $R3 = X1 * R2;$	19 $R3 = R1 + R1;$
2 $R2 = Y1 * Y1;$	11 $Y1 = R1 + R1;$	20 $Y1 = R3 - X1;$
3 $R3 = Y1 + Z1;$	12 $R4 = R1 + Y1;$	21 $R1 = R2 * R2;$
4 $R4 = R3 * R3;$	13 $R1 = R4 * R4;$	22 $R3 = R1 + R1;$
5 $R3 = R4 - R2;$	14 $Y1 = R3 + R3;$	23 $R1 = R3 + R3;$
6 $Z1 = R3 - R1;$	15 $X1 = Y1 + Y1;$	24 $R3 = R1 + R1;$
7 $R3 = X1 - R1;$	16 $Y1 = X1 + X1;$	25 $R2 = R4 * Y1;$
8 $R4 = X1 + R1;$	17 $X1 = R1 - Y1;$	26 $Y1 = R2 - R3;$
9 $R1 = R3 * R4;$	18 $R1 = R3 + R3;$	

Algorithm 19: Point addition for short Weierstrass curve with $a = -3, Z2 = 1$ in Jacobian coordinate with costs of $7M + 4S + 14A; 7R$:
$(X1, Y1, Z1)' = (X1, Y1, Z1) + (X2, Y2, 1)$

1 $R1 = Z1 * Z1;$	10 $R2 = R3 * R3;$	19 $Y1 = R3 + R3;$
2 $R4 = Z1 * R1;$	11 $R1 = Z1 - R1;$	20 $R3 = R4 * R4;$
3 $R2 = X2 * R1;$	12 $Z1 = R1 - R2;$	21 $R3 = R3 - X1;$
4 $R3 = Y2 * R4;$	13 $R1 = R2 + R2;$	22 $X1 = R3 - R2;$
5 $R4 = R3 - Y1;$	14 $R2 = R1 + R1;$	23 $R2 = R1 - X1;$
6 $R4 = R4 + R4;$	15 $R1 = X1 * R2;$	24 $R1 = R4 * R2;$
7 $R3 = R2 - X1;$	16 $X1 = R3 * R2;$	25 $Y1 = R1 - Y1;$
8 $R2 = Z1 + R3;$	17 $R2 = R1 + R1;$	
9 $Z1 = R2 * R2;$	18 $R3 = Y1 * X1;$	

Algorithm 20: X-coordinate double-and-add formulas for Curve25519 where $a24 = (a+2)/4$ with costs of $5M + 4S + 1M_{a24} + 8A; (6 + X_0)R$: $(X1, Z1)' = 2 \cdot (X1, Z1), (X2, Z2)' = (X1, Z1) + (X2 + Z2)$

1 $R1 = X2 + Z2;$	7 $Z2 = Z2 * Z2;$	13 $X1 = Z2 * X1;$
2 $X2 = X2 - Z2;$	8 $X1 = X1 * X1;$	14 $Z2 = R1 - X2;$
3 $Z2 = X1 + Z1;$	9 $R2 = Z2 - X1;$	15 $Z2 = Z2 * Z2;$
4 $X1 = X1 - Z1;$	10 $Z1 = R2 * a24;$	16 $Z2 = Z2 * X0;$
5 $R1 = R1 * X1;$	11 $Z1 = Z1 + X1;$	17 $X2 = R1 + X2;$
6 $X2 = X2 * Z2;$	12 $Z1 = R2 * Z1;$	18 $X2 = X2 * X2;$

Algorithm 21: CoZ double-and-add formulas for Curve25519 with costs of $8M + 5S + 1M_a + 7A; (6 + X_0)R$: $X1, Z)' = 2 \cdot (X1, Z), (X2, Z2)' = (X1, Z1) + (X2 + Z2), Z = Z1 = Z2$

1 $R1 = Z * Z;$	8 $R3 = R2 - X1;$	15 $R1 = R1 + R2;$
2 $R2 = X1 * Z;$	9 $R2 = R3 * R3;$	16 $R2 = X1 + X1;$
3 $R3 = X1 * X2;$	10 $R3 = X0 * R2;$	17 $X1 = R2 + R2;$
4 $X1 = X2 * Z;$	11 $R2 = Z - R1;$	18 $R2 = X1 * R1;$
5 $Z = X2 * X2;$	12 $R1 = Z + R1;$	19 $X1 = X2 * R2;$
6 $R3 = R3 - R1;$	13 $Z = R2 * R2;$	20 $X2 = Z * R3;$
7 $X2 = R3 * R3;$	14 $R2 = X1 * a;$	21 $Z = R3 * R2;$

5 Efficient 32-Bit Elliptic Curve Processor

Our life becomes more and more digital. Devices like computers and smart phones and in a near future smart glasses and smart watches are used extensively for communication and are integrated in our daily routine. Thus, they "know" many things about us, even things we do not want to share with the world. These devices know private data like our passwords and have access to our pictures, mails, and bank accounting. Furthermore, the producers of such devices or the used software try to gather as many information about us to adapt their products to our habits and requirements, so that we feel more comfortable with their products. This data must be protected. As a result, all smart devices must provide some cryptographic primitives. This also applies for smart cards like debit cards and our Austrian e-cards. Cryptography is not only used to ensure data privacy, but also to guarantee data integrity, authenticity, or non-repudiation. For instance, after signing a digital contract, it should be impossible to alter the content. Furthermore, all contracting partners must be identifiable and must not have any possibility to deny his signature. However, smart phones and smart cards are devices with a very limited amount of resources. Their computational power and their power supply as well are limited. Thus, it is important to implement the cryptographic primitives as efficiently as possible.

In this chapter we present our architecture of an efficient elliptic curve processor. The given architecture can be also used to implement other cryptographic primitives like hash functions. Thus, it is possible to implement a complete cryptographic system using this architecture.

5.1 Related Work

In the last years, several implementations for CPUs, GPUs, microcontroller, and sensor nodes were published. Since they have other restrictions and conditions, we will not compare our results with them. In our work, we will only compare with other hardware implementations. Most of the published implementations are for 192 bits. Since, the needed area and running time increase non-linearly with increasing key size, we will furthermore compare only with hardware implementations of 256-bit elliptic curves. Nonetheless, other hardware implementations use similar concepts and thus were used as guides for our implementation [OP01, KP06, SBM$^+$06, HFP10, WFF10, WH11, HFW11, WH12].

There are many implementations for field-programmable gate arrays (FPGAs) supporting 256-bit elliptic curves. The two most notable implementations are from Verbauwhede et al. [SDMPV06] and Güneysu and Paar [GP08]. However, FPGA implementations are quite hard to compare with ASIC implementations. Table 2 gives an overview of ASIC implementations supporting 256-bit elliptic curves. The table lists the used CMOS technologies, supported curve types, and the used word sizes. Moreover, it shows the area and needed cycles, calculation time and power consumption for the point-multiplication of the implementations. Most of these architectures of these implementations are designed for high-speed scalar multiplication and most of them also support binary-field curves. Some of them are compatible with the IEEE 1363 standard (see [iee00]). All listed implemen-

tations only support curves using the short Weierstrass equation, thus they cannot run Curve25519.

Table 2: Hardware implementations of 256-bit elliptic curves

	Technology	Curves	Word size [bit]	Area [GE]	Cycles	Time [ms]	Power [mW]
Satoh [ST03]	0.13-μm	any	8	19 935	9 385 000	28,0	n.a
			16	25 051	2 711 000	11,6	n.a
			32	43 521	880 000	5,4	n.a
			64	106 659	340 000	2,7	n.a
Wolkerstorfer [Wol04]	0.35-μm	any $\in GF(p^{256})$	256	14 763	1 175 451	17,2	n.a
Chen [CBC07]	0.13-μm	any $\in GF(p^{256})$	n.a	122 000	562 000	1,0	n.a
Lai [LH09]	0.13-μm	IEEE 1363	32	197 028	252 067	1,2	n.a
Muthukumar [MJ10]	0.13-μm	IEEE 1363	256	184 000	54 568	0,4	68.4

Wolkerstorfer ([Wol04]) implemented an elliptic-curve processor for low area and low power consumption. His design supports ECDSA signatures, but he did not implement a hash function and a random number generator. It uses a dual field arithmetic unit and supports curves from 192 to 256 bits. Arithmetic operations are performed in the Montgomery domain doing bit-serial multiplication with interleaved modular reduction. For point-multiplication projective Jacobian coordinates are used with Montgomery algorithm for modular multiplication.

Lai and Huang ([LH09]) use the addition-subtraction (see [WHGH$^+$08]) method for the point scalar multiplication and Jacobian's projective coordinates. They store all intermediate values in a register file, which consists of seven 256-bit buffers. The arithmetic unit consists of four parallel 32-bit word-based multipliers and four 64-bit word-based adders. The controller decomposes the equations into a sequence of atomic operations of single additions and multiplications. Thereby, the Montgomery algorithm for modular multiplication is used. All curve parameters (a, b, p, G) are input parameters to the arithmetic unit. To reduce the length of the critical path, the output of the multipliers produce a redundant carry-save representation. The adders are implemented by two carry-save adders and one carry-propagation adder (see Section 5.5.1).

In contrast to the other implementations, Chen, Bai, and Chen [CBC07] use a systolic network for the arithmetic operations. They also use the Montgomery algorithm for modular multiplication. A scheduling algorithm for modular reduction selects the operations depending on the values. The systolic network also implements modular division. Thus, they use affine coordinates instead of projective coordinates.

Muthukumar and Jeevananthan ([MJ10]) use Jacobian's projective coordinates in their implementation. Multiplication is done with Montgomery modular multiplication algorithm. The arithmetic unit consists of four parallel and serial 32-bit adders and a 32-bit multiplier. The 32-bit multiplier internally uses four 16-bit multiplier. All intermediate values are stored in the register file, which consists of seven 256-bit buffers. They use a power management scheduler, which controls the power consumption and adjusts the

clock frequency when the power consumption reaches a threshold value. This mechanism controls also the number of active addition units in the arithmetic unit.

The implementation of Satoh and Takano ([ST03]) supports elliptic curves from 160 up to 256 bits in binary and prime fields. Additionally, they can set their word of 8 up to 64 bits. Their results show that the computation time grows more than quadratically with the prime size. Additionally, the computation time and needed area strongly depend on the word size. For the finite-field multiplication Montgomery multiplication and for squaring Montgomery squaring is used. The point-multiplication is done by using a binary method. For higher speed, the controller can skip the operations for null bits in the key. The arithmetic unit consists of a Wallace tree (see [Wal64]) and a carry propagation adder for multiplication. The memory is divided into two parts to provide both operands of the multiplication at the same time.

A comparison of different architectures can be found in [TCW+05].

5.2 Architecture Overview

Our goal was to design a small architecture which is still fast and flexible enough to use different curves. It should be even possible to implement other cryptographic primitives on the same architecture. To meet these contrary goals, we optimized our architecture to perform multiple precision multiplications as fast as possible with as few resources as possible. We decided to use a word size of 32 bits, because the running time for 16 bits would be four times as long. This is reasonable, because most of the running time is required for these multiplications. In this section, we describe how we make a tradeoff between speed and area to create a high efficient implementation which also can be used for limited devices like RFID chips. Additionally, we describe our machine code programs for the different curves.

Figure 4 shows a high-level block diagram of our architecture. Our design consists of three parts: The memory, a controller, and the arithmetic unit. The memory contains a random access memory (RAM), a read only memory with needed constants (ROM) and map, which maps the address at the input to the addresses of the RAM and ROM. In the controller multiple ROMs and counter are used. Furthermore, it contains a call stack and an optional multiplication controller. The arithmetic logic unit (ALU) consists of an input buffer to store one or more operands, a rotation logic, several adders, and an accumulator.

5.3 Memory

Our memory consists of a RAM, a ROM, and some logic to map the addresses and swap pairs of registers. A detailed overlook of the memory can be found in Figure 5.

RAM. In the most arithmetic operations two operands are necessary. Therefore, a memory with two outputs providing both operands at the same time would be comfortable. However, a dual-ported RAM needs two address and data busses. Thus it needs more area than a single-ported RAM. To save area, we only use a single-ported RAM in our

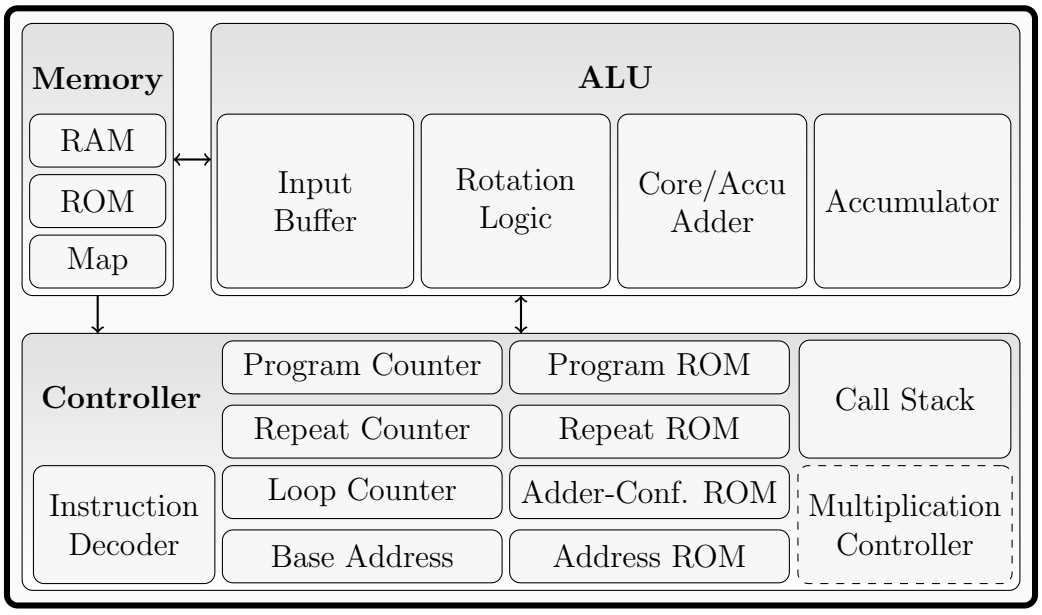


Figure 4: High-level block diagram of our architecture

architecture. To use still two operands at the same time, the ALU needs an input buffer. Additionally, the RAM provides the data from requested address at the next clock cycle. This makes it necessary to fetch data from RAM one cycle in advance. The RAM consists of R 256-bit registers, which consists of eight separately addressable 32-bit words.

ROM and address map. Since registers are very expensive in terms of area, we tried to keep their number as low as possible. Thus, all constants like a, b, and p are stored in a ROM. Nevertheless, it must be possible to access these constants in the same way as values from the registers to use them in the point operations. Therefore we map the ROM values in the address space directly behind the registers from the RAM. This makes it necessary to calculate the address in the ROM from the *address*. Additionally, a multiplexer is required to provide either the values from the RAM or the ROM on the output of the memory. In contrast to a RAM, a ROM provides the data from requested address at the same clock cycle. To make it behaving like the RAM, the ROM address must be delayed and the multiplexer on output must switch with a delay of one cycle. Using p during a reduction, it is clear that this value comes from ROM. To speed up the reduction, we added the possibility to read data from ROM without the artificial delay described above. This option can be selected by the *rom_mode* flag. The ROM consists of C 256-bit registers, which consists of eight separately readable 32-bit words.

Swapping registers. Looking at the Montgomery ladder for point-multiplication (see Algorithm 11), it is necessary to swap the content of two registers depending on a bit of the key. Exchanging the values from two registers needs many operations and thus

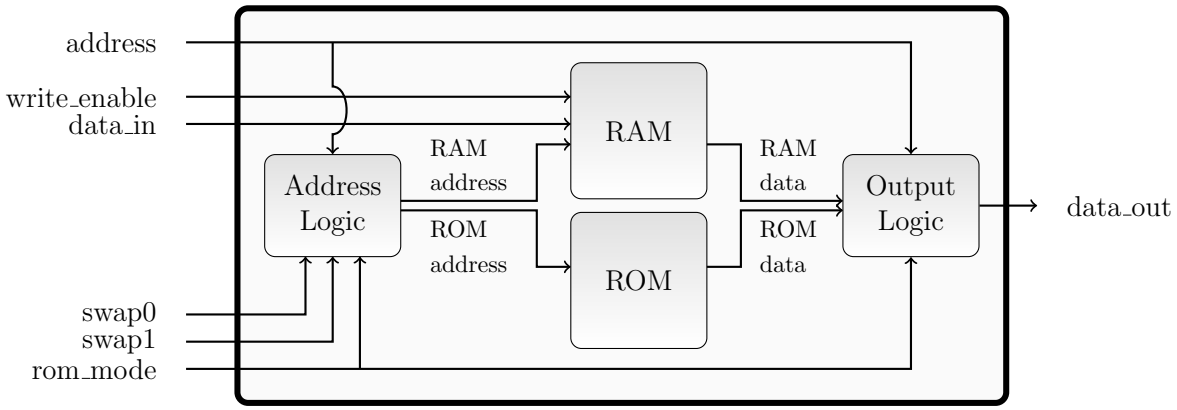


Figure 5: Detailed block diagram of the memory

increases the running time noticeably. Thus, we added a swapping functionality to the address map. This allows to exchange the address mapping of one or two register pairs in one cycle. From outside, the memory looks as if the registers had swapped their values. For a constant power consumption it possible to perform a dummy ($swap0$) instead a real swapping operation ($swap1$).

Support for comb method. The Comb Method for point-multiplication uses a look-up table with precomputed values. Thereby, bits from the key define the index in the table. Thus, it must be possible to create an address dynamically depending on a certain value. Therefore, the controller provides the needed index and the address logic uses them to generate the address for the ROM.

5.4 Controller

Our controller mainly consists of a ROM containing machine code, an instruction decoder to generate the control signals, a call stack, and some counters to represent the program flow. Figure 6 gives an overview over the structure of the controller. Compared to the implementations presented in related work, we do not use scheduling algorithms, but the sequence of instructions is predetermined.

Program counter and program ROM. Every elliptic curve algorithm is different. Especially the formulas for point operations and the algorithms for modular arithmetic differ. Thus, the architecture must be flexible and must allow various algorithms. To achieve this, the algorithms are not fixed state machines but we use a machine code (see Section 5.6). This machine code is stored in a program ROM and the program counter indicates the index of the current instruction. Since the program ROM contains several hundred instructions and thus is quite complex, it takes a noticeable time that the output reacts on changes of the input. In addition it takes time for the signals to go through

the instruction decoder to the ALU to choose the right values and more time to do the operations on these values. To reduce this signal running time, the instruction is buffered and executed in the next cycle. As a result, the program counter points to one instruction ahead.

Call logic and call stack. Typically, this counter increases by one after executing an instruction. However, for point-multiplication some operations are repeated in a loop n times. For this purpose, it also must be possible to set the program counter on a predefined value instead of incrementing it. This mechanism can be used for calling subroutines as well. After executing a subroutine, the program must be continued at the instruction after the one which is called the subroutine. Therefore, this address is stored in the call stack. In our implementation the call stack consists of a ring buffer to allow calls to the top of a loop, which will not return without wasting slots. The instructions must be buffered before they are executed and we use no additional hardware to predict calls. Thus, it needs one extra cycle to buffer the first instruction of the subroutine after each call. Furthermore, also after returning from a subroutine the first instruction must be buffered. Therefore, each call and return cause a delay of one instruction. Table 3 shows an exemplary time line with a call and return instruction.

Table 3: Example for sequence of executed instruction with delay after CALL and RET

Cycle	1	2	3	4	5	6	7	8	9
PC	0	1	2	3	17	17	18	3	4
ROM[PC]	`ADD`	`SUB`	`CALL 17`	`ROL32`	`MUL`	`RET`	`NOP`	`ROL32`	`SUB`
Instruction		`ADD`	`SUB`	`CALL 17`		`MUL`	`RET`		`ROL32`

Loop counter and skip logic. The loop for the point-multiplication must be iterated from $LC = n - 1$ to 0. To leave the loop counter, the value of LC is compared with 0. The instruction for comparison is followed by the instruction to call the top of the loop again. Thus, this instruction must be skipped if $LC = 0$. In this case the program counter must be increased by two.

Base address and address logic. To reduce the length of the machine code instruction, it is only possible to address four 256-bit entries in the memory at a time. These four entries are defined by the base-address in the base-address ROM. To use more registers, it is possible to change the index of this ROM. This mechanism has the advantage that subroutines can be called with different base addresses to work on different registers. This is comparable with parameters in function calls. The entries in the base-address table are freely configurable to allow all possible necessary combinations of register usage. Additionally, in the Montgomery ladder it is necessary to swap values from registers depending on k_{LC}. Thus, it must be possible to load the corresponding word from the memory and the higher bits of the LC are used in the address.

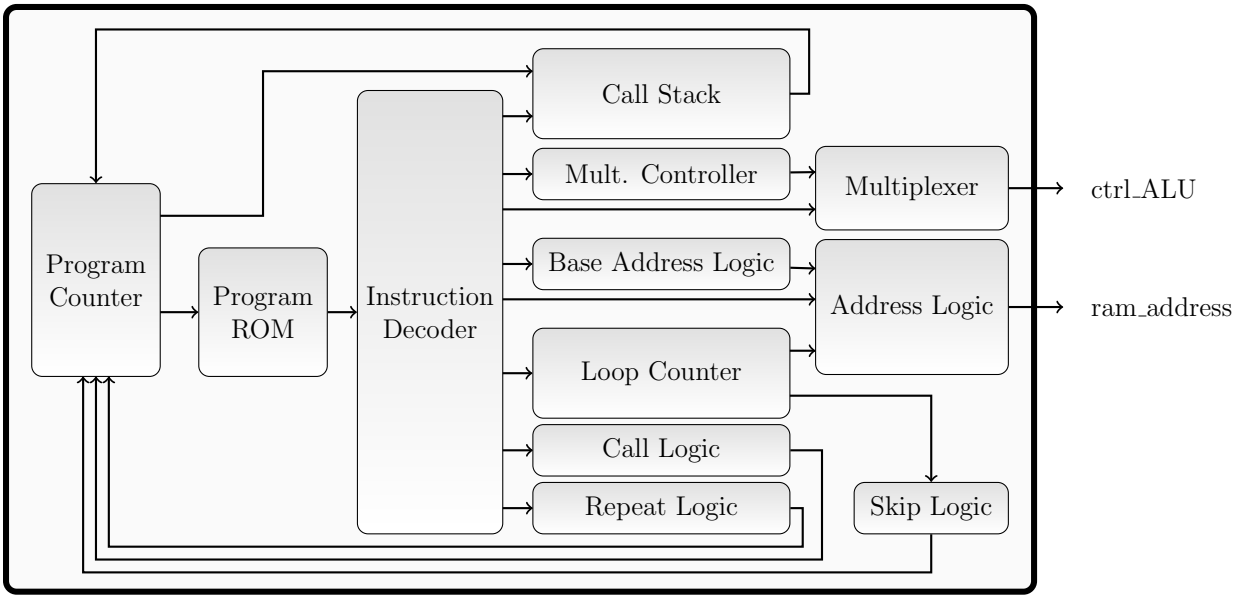


Figure 6: Detailed block diagram of the controller

Multiplication controller. As mentioned before, it is only possible to load one value from memory per cycle. Additionally, the result is first available at the next cycle. This produces an additional overhead to operations like multiplying two values. Since it is very important to make multiplications as fast as possible an optional multiplication controller can be used. This executes a 256-bit multiplication in an optimized way and preloads the operands for next 32-bit multiplication. In total, the architecture support three different data flow variants during a 256-bit multiplication. For the first variant, no multiplication controller is used. Therefore, the loading of the operands, the 32-bit multiplications and the writing of the results must be written in machine code (see Figure 7a). For the other variants the multiplication controller does the complete 256-bit multiplication using the product-scanning algorithm. If a 32-bit multiplication needs at least three cycles, the controller can use these three cycles to store the previous result and preload the next operands. For high-speed implementations, the 32-bit multiplication can be done in two cycles (compare Figure 7b). Then, it is no longer possible to load two operands for each 32-bit multiplication. Thus, in this case the zigzag product-scanning multiplication algorithm is used (see Figure 7c). A comparison of this three variants shows that multiplications with the additional controller needs no intermediate loading phase and thus is much faster. Using the multiplication controller instead of doing the multiplication with machine code reduces the code length. However, the additional control logic increases the area of the controller. Additionally, a multiplexer is necessary to select either the controlling signals from the multiplication controller or from the main controller. The costs for the multiplication controller will be discussed more detailed in Section 6.7.

Cycle	1	2	3	4	5	6	7	8	9
Phase	Load		Multiply			Store	Load		Multiply
Address	B1	A1				C1	B2	A2	
Data		B1	A1					B2	A2
BufferB			B1						B2
inA		B1	A1					B2	A2
inB			B1						B2
result						C1			
write C						1			

(a) Multiplication without multiplication controller within 3 cycles

Cycle	1	2	3	4	5	6	7	8	9
Phase	Load		Multiply			Multiply			Multiply
Address	B1	A1		B2	A2	C1	B3	A3	C2
Data		B1	A1		B2	A2		B3	A3
BufferA				A1			A2		
BufferB			B1			B2			B3
inA			A1			A2			A3
inB			B1			B2			B3
result						C1			C2
write C						1			1

(b) Multiplication with multiplication controller within 3 cycles

Cycle	1	2	3	4	5	6	7	8	9
Phase	Load		Multiply		Multiply		Multiply		Multiply
Address	B1	A1		B2	C1	A3	C2	B4	C3
Data		B1	A1		B2		A3		B4
BufferA				A1				A3	
BufferB			B1			B2			
inA			A1				A3		
inB			B1		B2				B4
result					C1		C2		C3
write C					1		1		1

(c) Multiplication with multiplication controller within 2 cycles

Figure 7: Dataflow during 256-bit multiplications

Repeat logic. An additional mechanism to reduce the code length is the repeat logic. This logic allows to repeat instructions for a constant number of iterations. The number of iterations can be different for different instructions and is stored in the repeat ROM. If the repeated instruction is a call of a subroutine, several instructions can be executed in a loop. This is useful to implement modular inversion with Fermat's little theorem. If this algorithm is optimized for a certain prime number, it contains sequences of multiplications using the same parameters. Those sequences can be implemented with fewer instructions using the repeat logic.

5.5 Arithmetic Logic Unit

We implemented an arithmetic logic unit (ALU), which can be modified by a set of constants. This makes our architecture very flexible and allows to adopt it for different needs. An overview can be found in Figure 8. In this section, we describe all components of the ALU and explain two different types of adder.

Input buffer. As mentioned earlier, the input buffer is used to provide both operands at the same time. This at least includes a register to buffer *data_in* for in_B, since the memory can not provide in_A and in_B at the same time. For the high-speed variant, this unit contains four buffers to store both input values for the current and the next operation.

Core adder. This unit consists of several adders, depending on the constant which defines the number of cycles for a 32-bit multiplication (NUM_CYCLES). The adders are connected in series and shifted by one bit each. This structure allows to use them as a digit-serial multiplier. More details about their structure can be found in Section 5.5.1. Thus, it can be used to multiply inB with a ($\lceil 32/NUM\_CYCLES \rceil$)-bit number. This number either is one of $\{in\_A_{NUM\_CYCLES-1}, ..., in\_A_1, in\_A_0\}$ or set by the instruction. The former possibility is used in the 32-bit multiplications, the latter one for other arithmetic and logic operations. Using the same unit for multiplication and other arithmetic operations reduces the area.

Accumulator and rotation logic. To use the ALU for multiplication, the partial sums from the core adder must be summed up. Therefore the value of the accumulator must be rotated to the right for ($\lceil 32/NUM\_CYCLES \rceil$) bits, added to the output of the core adder and stored into the accumulator.

The rotation logic can be used to rotate the value of the accumulator for any other value. For a 256-bit multiplication the accumulator must be able to store the result of 8

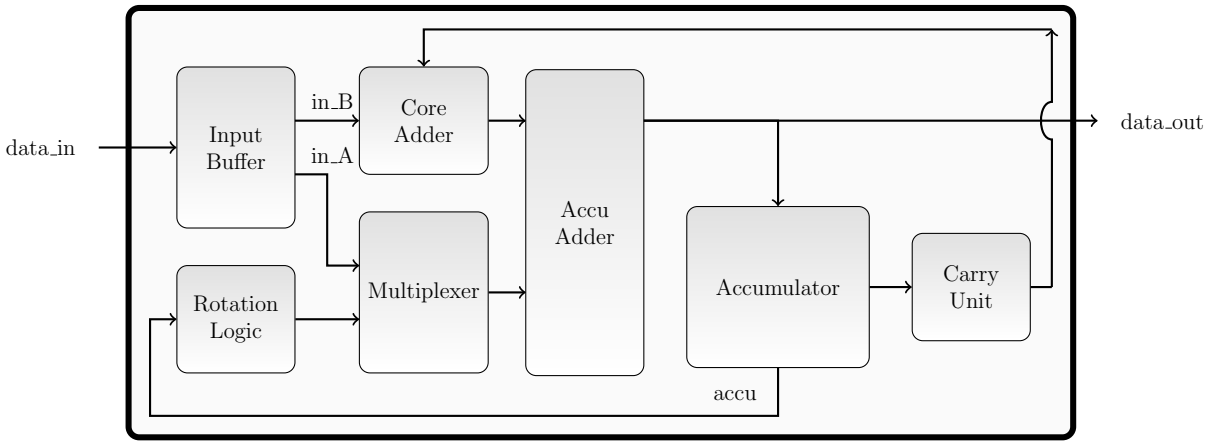


Figure 8: Detailed block diagram of the ALU

32-bit multiplications. Thus, the size of the accumulator typically is $2 \cdot 32 + 3$ bits. When $NUM\_CYCLES = 2$, it is not possible to load two operands for each 32-bit multiplication. Thus, in this case the proposed zigzag product-scanning multiplication algorithm is used. Therefore, it is necessary to increase the size of the accumulator to $3 \cdot n + 3$ bits. When doing a 256-bit addition or subtraction, for each 32-bit operation, the carry bit of the former one must added. This can be achieved by shifting the result 32 bits before adding the next one. Since the accumulator is circular and allows only rotation, the low part must be set to zero to achieve the same behavior. In the following, the high 32-bit part of the accumulator is denoted by $ACCU32$. This part is used to store the result of a 32-bit addition, subtraction, or logic operation. If the additional 3 bits for carry should be used, it is denoted by $ACCU35$.

Accu-adder and multiplexer. This adder is used to add the result of the core adder to the accumulator. During a multiplication, the value of the accumulator can be stored rotated. Thus, this adder must be circular. This means that the highest carry bit is used as carry in at the least significant bit to guarantee that the addition is performed correctly. To allow the addition or subtraction of in_A and in_B it must be possible to use in_A instead of *accu* in the accu-adder. Thus, a multiplexer is necessary at its second input.

Dynamic logic. It may be necessary to perform logic operations with in_A and in_B. Thus, the part of the accu-adder, which has calculated the result for $ACCU32$, also supports logic operations. Therefore, we created a dynamic logic circuit which extends the functionality of a fulladder by OR, XOR, and AND. Table 4 shows the output values of these functions for all possible input combinations in a and b. A fulladder can be used to generate the sum and the carry bit of two input bits and an input carry. Section 5.5.1 shows how multi-bit values can be added using fulladders. Figure 9 compares a classic fulladder with our dynamic logic. The little circles in the figure symbolize an inversion of the signal. The additional circuit mainly uses NAND-gates which can be built in hardware especially small. Setting the control signals right, it can be chosen between OR, XOR, AND, or the functionality of a fulladder. For AND, in_B must be inverted.

Table 4: Truth tables of some logic functions and a fulladder

b	a	OR	XOR	AND	$NAND$
0	0	0	0	0	1
0	1	1	1	0	1
1	0	1	1	0	1
1	1	1	0	1	0

c_{in}	b	a	sum	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

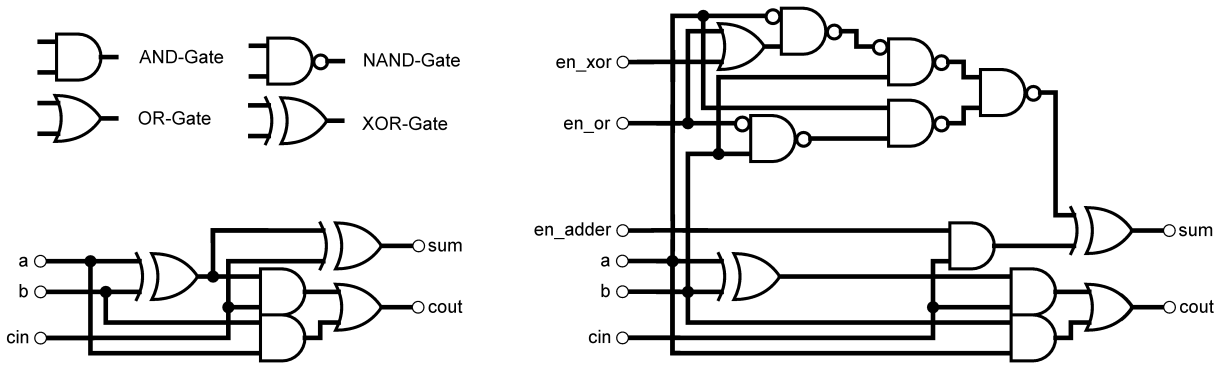


Figure 9: Schematic of a fulladder and a dynamic logic cell

Carry unit. After a 256-bit addition, it depends on the carry bit if the prime number must be subtracted or not. However, for a secure implementation, in both cases the same operations must be done. Thus, in one case the prime number is subtracted, in the other case zero is subtracted using the same instructions. In order to achieve this, the carry bit is stored in a secure buffer, which will be described below. If $carry = 1$, the first adder in the core adder is active and the prime number is subtracted. Otherwise, the first adder is inactive and nothing is subtracted. The procedure for reduction after a 256-bit subtraction is similar.

In the NIST-reduction, it is even possible to have two carry bits. Additionally, they can be positive or negative. Thus, in this case an extended secure buffer with three bits is used. The overflow buffer can be used in the same way as the carry buffer. Besides, it is possible to decrement the overflow value depending if it was negative or positive. This can be achieved with the instruction `UPDOF`. If the value stored in the overflow buffer is zero, again the first adder in the adder core is inactive during adding the prime number. Otherwise, the prime number is added or subtracted, depending if the value in the overflow buffer is negative or positive.

LC-Bit. In point-multiplication, the operations depend on the key. Thus, it is necessary to extract one or more bits from the key depending on the loop counter. Therefore the controller generates the address for the according word from the register holding the key. This 32-bit word is loaded into the buffer for in_B. Then, a multiplexer chooses the bit or bits respectively depending on the loop counter. These values are used in the controller to decide if registers must be swapped or not.

Secure buffer. The secure buffer is used to store values which depend on the key and thus may not leak the device. It has a buffer B_1 to store the actual value and a dummy buffer B_2 which changes its value, the value in the real buffer does not change. In this way in any case one value changes to ensure a constant power consumption. However,

the synthesizer removes all elements which have no influence on other parts of the design. Thus, the output of the B_2 must be connected with the output of B_1 to avoid that it is removed. To not influence to output of B_1, the output of the B_2 is kept constant. Therefore we use two registers, whereby one of them is one and the other is zero. Thus the result of connecting them with an XOR-gate is constantly one. For reasons of symmetry B_1 also must consists of two registers. Table 5 shows all possible states and transitions of the secure buffer. Figure 10 shows the schematic. This buffer is also used to store the carry and overflow bits.

Table 5: Truth tables of the secure buffer

set	B_2	B_1	B_2'	B_1'	out'
0	10	01	**01**	01	0
0	01	01	**10**	01	0
0	10	10	**01**	10	0
0	01	10	**10**	01	0
1	10	01	10	**10**	1
1	01	01	01	**10**	1
1	10	10	10	**01**	1
1	01	10	01	**01**	1

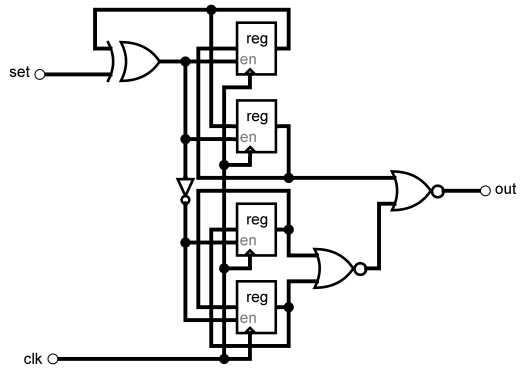


Figure 10: Schematic of the secure buffer

5.5.1 Core Adder

In our work, we use a 32-bit digit serial multiplication algorithm. As we showed in [WH14], is the most efficient one in terms of area and power this type of multiplier In this algorithm, operand b is multiplied with each single bit of operand a. Multiplication with a single bit is a very simple operation, because the result either can be b or 0. The partial results are shifted and accumulated. Figure 11 shows the principle behind this algorithm. In a digit-serial multiplier, NA such 1-bit multiplications are done in parallel, whereby NA stands for

"number of additions". In our architecture NA can be between 2 and 16. For NA parallel 1-bit multiplications $NA-1$ n-bit adder are necessary to calculate the partial sum and one $2n$-bit adder to add it to the intermediate result in the accumulator. In our work, we used two different structures, which are presented in the following two sections. To keep them understandable, we pretend that all adders are active. In our implementation some adders have an additional AND-gate which defines if the adder is active or not.

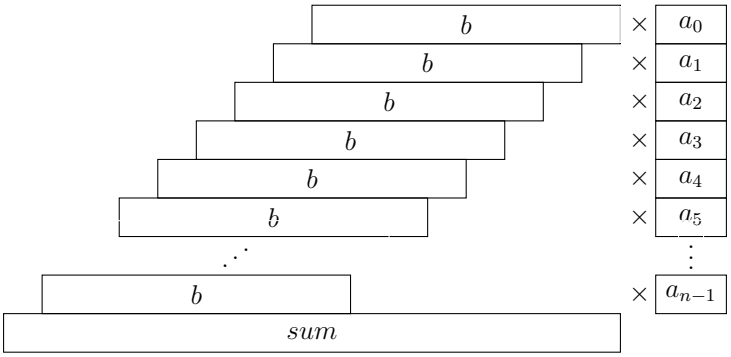


Figure 11: Principle of digit serial multiplier

Carry-Ripple Adder Chain The easiest way to build an n-bit adder from n full adder is to connect them in series as shown in Figure 12. This type of adder is called carry-ripple adder. It can calculate the sum of two n-bit number a, b, and an input carry. The result including the output carry has $n+1$ bits. To calculate the last bit, the carry bits must be propagated through all full adders. Since, they need some time to react on changes of the input, they have some delay d. Thus, the total calculation has a total delay of $n \cdot d$.

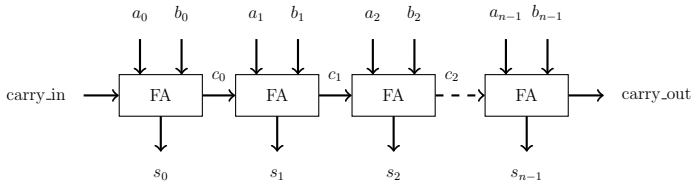


Figure 12: n-bit carry-ripple adder

Figure 13 shows how we used three such adders in series to build a core adder with $NA = 4$. All adders are connected in series, hence the carry must propagate through all adders. As a result, the ripple-carry adder is quite slow with a total delay of $(NA-1) \cdot n \cdot d$. In return the layout of such an adder is simple, and thus has a short design time.

Carry-Save Adder Tree To improve the running time for such a big adder, carry-save adder (CSA) can be used. In contrast to the carry-ripple adder, the carry does not propagates through all adder instances. This can be achieved by representing the output

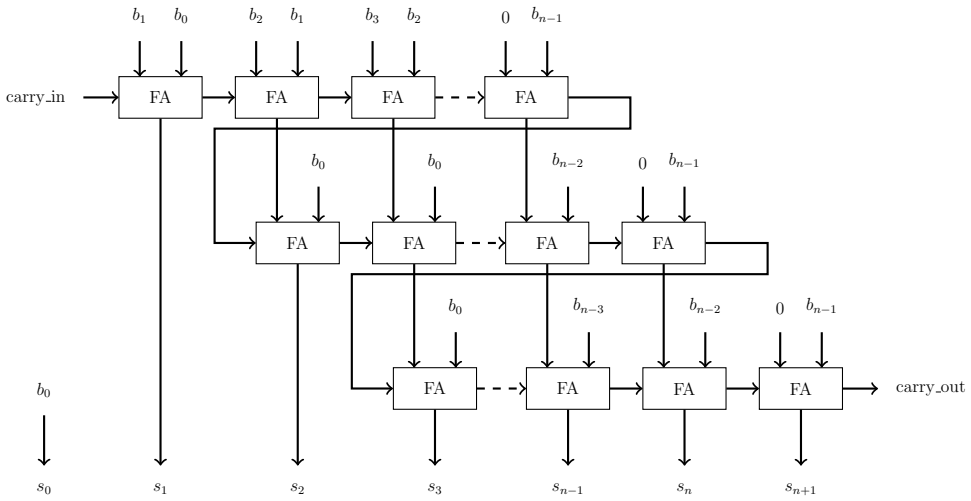


Figure 13: n-bit ripple adder

of each adder as a 2-bit number. Therefore the fulladders are disjoint and consume three input bits and a 2-bit output. This representation is used until the final result is needed. Figure 14 shows an n-bit carry-save adder [WE93]. The delay of this adder is d.

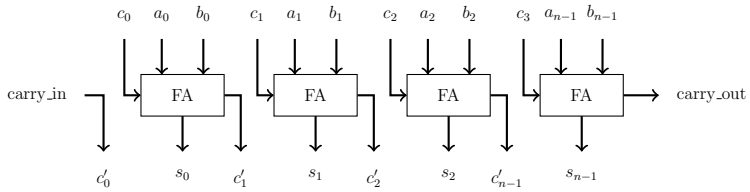


Figure 14: n-bit carry-safe adder

For the core-adder, we use $NA-1$ CSA and connect them in a tree structure to reduce the longest path. Figure 13 shows how we built a core adder with $NA = 4$. Only the final adder uses the carry-ripple mode to convert the 2-bit representation from the CSA to normal binary representation. The advantage of this structure is the short delay of about $2 \cdot log_2(NA) + n$. In return, due to the irregular structure it is harder to design.

5.6 Machine Code

All instructions have nine bits, which results in $2^9 = 512$ possibilities. Nevertheless, we have not implemented 512 different instructions. We provide two types of instructions: instructions without address and instructions with an address. Thereby the length of the address can be up to six bits.

If instructions are especially often used before operations which require values from memory, it is useful to load these values during these instructions. Thereby, two bits of

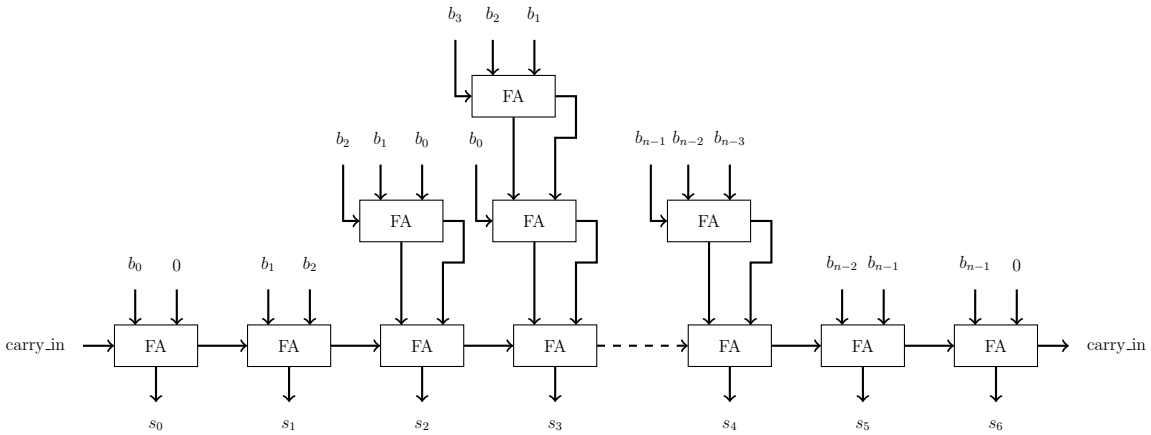


Figure 15: n-bit ripple adder

the machine code specify the address from the base address entry. Additionally, three bits define which 32-bit word from the 256-bit value should be loaded from the memory. Thus, in this case five bits are reserved for the address. This reduces the number of possible instructions, but improves speed because values can be loaded for the next instruction. In other instructions, the address can also define the index of one of the ROMs in the controller. For instance, the address at the instruction `REPEAT` defines the index of the repeat constant in the repeat ROM.

Tables 6 and 7 list all implemented instructions. Their first column shows the instruction name and a possible address parameter. To improve the readability, we write for instance simply RC instead "the value of the repeat ROM at the index RC". A description of the instruction can be found in the last column. Table 6 has an additional column with the maximal address size.

In the following we make remarks to the notation and some special instructions. The structure of the adder chain allows to add or subtract multiples of inB to inA or the accumulator. Thereby, the active adder can be selected by the parameter AS. To determine the LC-bit, it is necessary to load the word containing it from the memory, which can be done with the instruction `FLC`. For multi-precision subtraction it is important that all high bits share the value of the sign. The high bits are stored in the low part of the accumulator. To set them to the correct value after rotation, the instruction SETSGN can be used. To use MUL256, the multiplication controller must be added to the design. The result of the 256-bit multiplication has 512 bits. The low half of this results is stored in the third register defined by the base address ($BA[2]$), the high half is stored in the fourth register ($BA[3]$). As operand registers one ($BA[0]$) and two ($BA[1]$) are used.

5.7 Machine Programs of Point-Multiplication

We implemented the point-multiplication in various ways, but all machine programs consist of four parts: initialization, a loop for the point multiplication, modular inversion in the

Table 6: List of instructions with address

Mnemonic		Max. address	Description
NOP	A	32	fetches word with address A from memory
FLC	A	32	fetches A[LC] from memory
ROL32	A	32	rotates ACCU 32 bits to left, fetches address A from memory
BUFALU	A	32	loads A (must be in ROM) directly into input buffer
BUF	A	32	save current output of memory into input buffer, fetches A
RST	A	32	resets the complete ACCU, fetches A
RSTL	A	32	resets the low part of the ACCU, fetches A
SETSGN	A	32	sets the low part of the ACCU to current sign, fetches A
WR	A	32	writes low part of the ACCU into the memory at address A
WRROR	A	32	writes ACCU32 to A, rotates ACCU $(32 - NA)$ bits to the right
ADD	AS	8	calculates inA + inB·AS
ADDC	AS	8	calculates inA + inB·AS + carry
SUB	AS	8	calculates inA – inB·AS
SUBC	AS	8	calculates inA – inB·AS + carry
ADDAE	AS	8	calculates ACCU32 + inB·AS
SUBAE	AS	8	calculates ACCU32 – inB·AS
SUBH	AS	8	calculates ACCU35 – inB·AS
CALL	CT	16	sets program counter to CT
SETBA	BA	64	sets base address to BA
CMPI	SC	4	skips next instruction when LC-bit = SC, increments LC
CMPD	SC	4	skips next instruction when LC-bit = SC, decrements LC
REPEAT	RC	16	repeats following instruction RC times

finite-field, and subroutines for modular arithmetic. Several machine code examples can be found in the Appendix B.

For all programs, we varied the number of cycles for the multiplication from two to sixteen. Additionally, we implemented all of them with and without multiplication controller. This controller needs additional area, but the 256-bit multiplication needs fewer cycles and must not be written in machine code. This saves 209 instructions and thus area for a smaller program ROM. Using a multiplication controller, the machine code on average has a total length of 647 instructions. Thereby, the length of the machine code strongly depends on the reduction algorithm. For Brainpool P256r1, more than the half of the machine code is used for the code to perform the reductions after multiplication, addition, and subtraction. Another quarter of the code defines the arithmetic operations itself. The rest is the code for the initialization and the loop for the point-multiplication. If the multiplication controller is not used, it needs only a small area to control the 32-bit multiplication. Not every implementation needs the whole instruction set. To safe area, the parts for the unused instructions are removed from the design by the synthesizer.

Table 7: List of instructions without address

Mnemonic	Description
ROL8	rotates ACCU *NA* bits to left
ROR8	rotates ACCU *NA* bits to right
ROL16	rotates ACCU $(32 - 2 \cdot NA)$ bits to left
ROL24	rotates ACCU $(32 - NA)$ bits to left
ROR24	rotates ACCU $(32 - NA)$ bits to right
SUBACC	calculates inB – ACCU32
LDCOMB	load the needed comb-constant from the ROM
RED	calculates inA±inB depending on overflow
REDC	calculates inA±inB + carry depending on overflow
AND	calculates inA AND not inB
OR	calculates inA OR inB
XOR	calculates inA XOR inB
MUL	calculates ACCU + inA·inB
MULNEG	calculates ACCU – inA·inB
MUL256	256-bit multiplication ({BA[3]BA[2]} = BA[0]·BA[1])
RET	sets program counter to top value of call stack + 1
INCBA	increments base address
DECBA	decrements base address
SWAP0	swap state becomes 0, register are no longer swapped
SWAP1	swap state becomes 1, register are swapped
SWAPLC	swap state is XORed with LC-bit
SKIPB	skips next instruction when LC-bit = 1
INCLC	increments LC
DECLC	decrements LC
HLT	stops execution
SAVEC	saves carry
SAVECI	saves inverted carry
SAVECX	saves old carry XORed with actual carry
SAVEOF	saves overflow
UPDOF	updates overflow

Initialization. The first part of the programs initializes the memory with values like $0, 1, P, or 2P$, depending on the curve and the used algorithm. For Curve25519 it is also necessary to perform masking operations on the key. In total the initialization for Curve25519 needs 77 instructions and cycles. For NIST P256 and Brainpool P256r1 it is necessary to calculate $2P$ doing a point addition. Therefore, the initialization for these curves includes several calls of subroutines and need much longer, but still less than one percent of the total running time.

Loop for point-multiplication. We designed our implementation in a way to protect it against basic cryptographic attacks. Therefore, it is most important to choose the right

Table 8: Costs of implemented formulas

Curve	Algorithm	Costs
Brainpool, NIST	CoZ1 - diff. dbl & add [HJS11]	$11M + 4S + 1M_a + 1M_{4b} + 14A; (7 + X_0, Y_0)R$
	CoZ2 - diff. dbl & add [HJS11]	$9M + 5S + 1M_a + 1M_{4b} + 14A; (8 + X_0, Y_0)R$
	CoZ3 - diff. dbl & add [HJS11]	$10M + 5S + 13A; 10R$
Curve25519	XZ - diff. dbl & add (Sec.4.0.5)	$5M + 4S + 1M_{a24} + 8A; (6 + X_0)R$
Brainpool, NIST	CoZ1, CoZ2 - Y-recovery [HJS11]	$8M + 2S + 1M_a + 1M_{4b} + 8A; (7 + X_0, Y_0)R$
	CoZ3 - Y-recovery [HJS11]	$10M + 3S + 8A; (9 + X_0, Y_0)R$
NIST	COMB - double [BL14]	$3M + 5S + 26A; 7R$
	COMB - add [BL14]	$7M + 4S + 14A; 7R$

algorithms, the algorithms must need the same time for any input. In our programs, we use the Montgomery ladder. To see how much the point-multiplication could be sped up we also implemented a modified comb method. We had to modify the comb method, because for the used formulas no neutral element $\mathcal{O}$ exists. Thus, we use a dummy variable Q_0 when we should add $\mathcal{O}$ when $t = 0$. In our modified comb method is shown in Algorithm 22. The protection is not perfect, but his algorithm is only used to see how fast the point-multiplication could be.

Both, the Montgomery ladder and the modified comb method, need a loop to iterate over all bits of the key. In this loop we implement the formulas for the point addition and doubling operations. Therefore, it calls the subroutines for modular arithmetic several times with different base addresses. This loop causes about 90 percent of the total running time. A comparison of the costs of the implemented formulas can be found in Table 8. We need one register to store the high half of the 256-bit multiplications, additionally to the listed number of registers.

Modular inversion in the finite-field. After calculating the result in projective coordinates, it must be converted back to affine coordinates. Therefore, the inverse of the Z-coordinate must be calculated. For the calculation, we use Fermat's little theorem. This can be implemented by iterating over all bits of $p - 2$, where p is the prime of the curve and causes almost ten percent of the running time.

Subroutines for modular arithmetic The previous parts of the programs need modular addition, subtraction and multiplication. These functions are written in subroutines. So, they can be called with different base addresses to operate on different entries in the memory. The instructions for these functions are responsible for a major part of the program size. Thereby, the length depends heavily on the used reduction algorithms. The reductions for Curve25519 and NIST P256 fast reduction methods can be applied, Brainpool P256r1 requires a Barrett reduction after the multiplication. A detailed comparison of the costs for reduction can be found in Section 6.7.

It is important to mention, that all reduction algorithms are performed in constant time. To reduce the result of an addition or subtraction the prime number must be add

Algorithm 22: Modified Fixed-base comb method

Input: Window width w, $d = \lceil t/w \rceil, k = (k_{t-1}, ..., k_1, k_0)_2, P$
Output: kP

```
1  T = 2P
2  LUT[0] = P
3  LUT[1] = P
4  LUT[2] = T
5  for f = 3 to 2^w − 1 do
6  |   T = T + P
7  |   LUT[i] = T
8  Q_0 = P
9  Q_1 = LUT[k_{d·w−1}, ..., k_{(d−1)·w}]
10 for i = d − 2 downto 0 do
11 |   t = [k_{i·w+w−1}, ..., k_{i·w}]
12 |   for j = 0 to w − 1 do
13 |   |   Q_1 = 2 · Q_1
14 |   if t = 0 then
15 |   |   Q_0 = Q_0 + LUT[t]
16 |   else
17 |   |   Q_1 = Q_1 + LUT[t]
18 return Q_1
```

or subtract only if the result is negative or higher than the prime number. Otherwise we do the same operations, but without subtracting or adding the prime number. This can be achieved by storing the carry bit and disabling all adders depending on this bit.

6 Implementation Results

In this chapter, we compare different implementations of elliptic curve cryptography with a 128-bit security. For a fair comparison, we created a generic platform and used the same tools and workflow for all variants.

The first section describes the setup of our evaluation. In the Sections 6.2 to 6.4, we present the results in time, area and power of the three curves separately. Then, in Section 6.5, we compare these results with each other. In Section 6.6, compare our results with the implementations presented in the related work (see Section 5.1). Finally, in Section 6.7, we evaluate the costs of several components of the implementations.

6.1 Evaluation Setup

Table 9 explains the used abbreviations in this section. In our work, we implemented the finite field inversion using Fermat's little Theorem. As mentioned in Section 2.4.3, the algorithm can be optimized for Pseudo-Mersenne primes as used by NIST P256 and Curve25519. To evaluate the impact of the optimization on performance we implemented the inversion in three variants. First variant is the unoptimized variant as shown in Algorithm 9. For the second variant (FI1), we list the necessary multiplications for the optimized variant in the machine code without a loop. Since, for Curve25519 this contains up to 99 identical multiplications in a row, in the third variant (FI2), we do some loops to reduce the length of the machine code. The list does not show the value of all parts of the design. Thus, the sum for the ALU, controller, and memory are not only the sum of the listed values but the actual total area. For the evaluation we stored a fixed base point in the ROM and a variable key in the RAM.

Many systems already have a RAM and thus no additional costs would arise. If a system does not have a RAM, it could be implemented with standard cells or RAM macros. Thereby, the total area does not include the area for the RAM. Thus, we have not added the area for the RAM to the total area of the design. However, a standard-cell-based RAM is considered for the evaluation of the total power consumption.

The hardware results were generated for a frequency of 1 MHz using the following tools:

- Faraday UMC 0.13um 1.2V/3.3V 1P8M LL Logic Process ($5.12\mu m^2$/GE) [Far03]
- Cadence® Encounter RTL Compiler v08.10-s238_1
- Cadence® First Encounter 08.10-s273_1 (64 bit)

6.2 NIST P-256

For the NIST curve, we implemented the Montgomery ladder and a fixed-base comb method. The implementations using the comb method need a larger ROM to store the precalculated values. Additionally, more registers are necessary to store the X-, Y-, Z-coordinates of the point coordinates of the dummy value. Moreover, four temporary registers and one register to store the high half of the result of the multiplication are

Table 9: Abbreviations

- Alg. ... used algorithm
- MC ... number of cycles for 32-bit multiplication
- FM ... 1 if MUL256 is used
- FI ... 1 or 2 if fast inversion is used
- CS ... comb size
- Cycles ... number of total cycles for point-multiplication
- RAM ... area for RAM [GE]
- ROM ... area for ROM [GE]
- MCtrl ... area for memory controller [GE]
- Map ... area for address map [GE]
- Mem ... total area for memory without RAM [GE]
- Buffer ... area for input buffer
- Adder ... area for core and accu adder [GE]
- Rot ... area for rotation logic [GE]
- Accu ... area for accumulator [GE]
- ALU ... total area for ALU [GE]
- PROM ... area for program ROM [GE]
- Mult ... area for multiplication controller [GE]
- Ctrl ... total area for controller [GE]
- Total ... total area for the ECC-core without RAM [GE]
- P-Mem ... power consumption of memory [nW]
- P-ALU ... power consumption of ALU [nW]
- P-Ctrl ... power consumption of the controller [nW]
- P-Total ... total power consumption of the ECC-core [nW]
- At ... area-time product (area without RAM) [GE / 10^9]
- AtP ... area-time-power product (area without RAM) [GE nW / 10^{12}]
- Ins ... number of required instructions in machine code

necessary. COMB2 handles two bits of the key per iteration, COMB4 even four bits per iteration. Thus, COMB2 stores four and COMB4 sixteen precalculated values as X- and Y-coordinates each. In return fewer point additions are necessary, because only one is done per iteration. COMB2 needs only half of point additions, COMB4 even only a quarter.

Table 10 shows all values of the implementations using the Montgomery ladder, Table 11 contains the values of the implementations using the comb method. The smallest implementation uses CoZ1 and requires 16 cycles for a multiplication without a multiplication controller. The fastest design with Montgomery's method uses CoZ3, but only requires two cycles for multiplication and uses a multiplication controller. It is more than three times faster but needs 80% more area than the smallest implementation. The power consumption is 125% higher. The fastest implementation using the comb method is even 18% faster than the fastest variant using the Montgomery ladder. Additionally, it needs

17% more area, but the power consumption is reduced by 38%. The implementation which uses CoZ1 and a multiplication controller calculating 32-bit multiplications in four cycles has the lowest power consumption has . Looking at the At and AtP metrics, implementations which need three cycles for a multiplication are the best. Thus, for NIST curves it seems to be most efficient to do multiplication 3, 4, or 8 cycles using ten adder in the core adder. Thereby, the design with $MC = 3$ and CoZ1 has the best AtP product.

6.3 Brainpool P256r1

For the Brainpool curve we only used the Montgomery ladder for point-multiplication since the speedup with the comb method is similar as for the NIST curves. Table 12 shows all results. The smallest implementation uses CoZ3, needs 16 cycles, a 32-bit multiplication and does not use a multiplication controller. The fastest design is 354% faster but needs only 66% more area. This design only needs two cycles for a 32-bit multiplication, uses a multiplication controller and CoZ3. Looking at the At and AtP metrics, implementations which need two cycles for a multiplication are the best. The design which uses CoZ3 and a multiplication controller doing 32-bit multiplications in eight cycles has the smallest power consumption.

6.4 Curve25519

For Curve25519 we implemented only one formula for point-multiplication but used three different methods for integer inversion. The first is used identically for the other curves, the second one uses an optimized algorithm without loops, and the third variant uses loops in the machine code for the optimized variant. The fastest variant with fast inversion is seven percent faster than the same variant without fast inversion. This variant only needs five percent more area for the longer machine code but needs even five percent less power due to the shorter runtime. The variants using the optimized algorithm with loop counter have a shorter machine code but need some additional cycles caused by the loop. Additionally we implemented some of the variants with carry-save adder instead of carry-ripple adder. These implementations need at maximum only 80 GEs more area. Also the difference in the power consumption is below two percent.

For the reduction, in Curve25519, the high bits must be multiplied with 36 as described in Section 2.2.3. This only can be done in one step if at least six adders are used. Thus, for the small design more steps are necessary and the runtime increases disproportionately. The implementation with the lowest power consumption does not use fast inversion and multiplication controller and needs two cycles for a 32-bit multiplication. The smallest implementation is very similar but needs sixteen cycles for a 32-bit multiplication. The fastest implementation is completely contrary. It uses the fast inversion without loops and a multiplication controller which needs two cycles per multiplication. This design also has the best rating at the AtP metric. The design with the best value at the At metric uses the fast inversion with loops and needs three cycles for a multiplication.

Table 10: All results for NIST P256 using the Montgomery ladder

Alg.	MC	FM	**Cycles**	RAM	ROM	MCtrl	Map	**Mem**	Buffer	Adder	Rot	Accu	**ALU**
	2		**2867899**	18286	304	164	45	**512**	226	3888	546	472	**5601**
	3		**3171003**	18286	304	164	45	**512**	226	2775	487	472	**4383**
CoZ1	4		**3474107**	18286	304	164	45	**512**	226	2115	384	472	**3619**
	8		**4686523**	18286	304	164	45	**512**	226	1235	384	472	**2739**
	16		**7111355**	18286	304	164	45	**512**	226	792	384	472	**2296**
	2		**2228539**	18277	304	164	45	**513**	1191	4064	806	698	**7343**
	3		**2498491**	18277	304	164	45	**513**	1191	2775	487	472	**5348**
CoZ1	4	1	**2801595**	18277	304	164	45	**513**	1191	2115	384	472	**4584**
	8		**4014011**	18277	304	164	45	**513**	1191	1235	384	472	**3704**
	16		**6438843**	18277	304	164	45	**513**	1191	792	384	472	**3261**
	2		**2729689**	20200	304	162	45	**511**	226	3888	546	472	**5601**
	3		**3016473**	20200	304	162	45	**511**	226	2775	487	472	**4383**
CoZ2	4		**3303257**	20200	304	162	45	**511**	226	2115	384	472	**3619**
	8		**4450393**	20200	304	162	45	**511**	226	1235	384	472	**2739**
	16		**6744665**	20200	304	162	45	**511**	226	792	384	472	**3039**
	2		**2129235**	20200	304	162	42	**508**	821	4064	806	698	**6973**
	3		**2384652**	20200	304	162	42	**508**	821	2775	487	472	**4978**
CoZ2	4	1	**2671436**	20200	304	162	42	**508**	1119	2115	384	472	**4512**
	8		**3818572**	20200	304	162	42	**508**	1119	1235	384	472	**3632**
	16		**6112844**	20200	304	162	42	**508**	1119	792	253	472	**3189**
	2		**2595499**	24439	304	157	45	**506**	226	3888	546	472	**5601**
	3		**2866347**	24439	304	157	45	**506**	226	2775	487	472	**4383**
CoZ3	4		**3137195**	24439	304	157	45	**506**	226	2115	384	472	**3619**
	8		**4220587**	24439	304	157	45	**506**	226	1235	384	472	**2739**
	16		**6387371**	24439	304	157	45	**506**	226	792	384	472	**2296**
	2		**2024179**	24197	304	157	45	**506**	1191	4064	806	698	**7344**
	3		**2265403**	24197	304	157	45	**506**	1191	2775	487	472	**5348**
CoZ3	4	1	**2536251**	24197	304	157	45	**506**	1191	2115	384	472	**4584**
	8		**3619643**	24197	304	157	45	**506**	1191	1235	384	472	**3704**
	16		**5786427**	24197	304	157	45	**506**	1191	792	384	472	**3261**

Alg.	MC	FM	PROM	Mult	**Ctrl**	**Total**	P-Mem	P-ALU	P-Ctrl	**P-Total**	At	AtP
	2		1476	309	**3022**	**9136**	50237	4917	47184	**102338**	26,2	2681
	3		1476	348	**3066**	**7962**	48157	4976	84886	**138019**	25,2	3478
CoZ1	4		1476	342	**3060**	**7191**	48157	4900	84584	**137641**	25,0	3441
	8		1476	354	**3068**	**6320**	58671	4157	71049	**133877**	29,6	3963
	16		1476	362	**3074**	**5883**	37539	4200	79208	**120947**	41,8	5056
	2		1003	354	**2621**	**10477**	49614	91976	44740	**186330**	23,3	4341
	3		1003	379	**2640**	**8501**	31981	22271	25445	**79697**	21,2	1690
CoZ1	4	1	1003	366	**2628**	**7725**	31981	21385	25460	**78826**	21,6	1703
	8		1003	389	**2655**	**6872**	49114	33623	32865	**115602**	27,6	3191
	16		1003	394	**2660**	**6434**	49998	44574	38079	**132651**	41,4	5492
	2		1413	309	**2986**	**9097**	62414	3968	88782	**155164**	24,8	3848
	3		1413	348	**3026**	**7919**	50676	4794	65350	**120820**	23,9	2888
CoZ2	4		1413	342	**3019**	**7149**	50668	4717	65131	**120516**	23,6	2844
	8		1413	354	**3035**	**6284**	59973	4817	97290	**162080**	28,0	4538
	16		1413	362	**3039**	**6588**	41564	6723	66012	**114299**	44,4	5075
	2		922	361	**2537**	**10018**	63506	16578	59474	**139558**	21,3	2973
	3		922	363	**2534**	**8020**	36295	55535	29474	**121304**	19,1	2317
CoZ2	4	1	922	366	**2533**	**7553**	39395	14852	49107	**103354**	20,2	2088
	8		922	392	**2569**	**6709**	62411	16585	65772	**144768**	25,6	3706
	16		922	395	**2568**	**6264**	56807	19786	52427	**129020**	38,3	4941
	2		1631	309	**3214**	**9321**	72861	43487	99320	**215668**	24,2	5219
	3		1631	348	**3254**	**8143**	67228	21577	123722	**212527**	23,3	4952
CoZ3	4		1631	342	**3247**	**7372**	67227	9883	122598	**199708**	23,1	4613
	8		1631	354	**3265**	**6510**	52474	10553	53622	**116649**	27,5	3208
	16		1631	362	**3273**	**6075**	79339	4580	86098	**170017**	38,8	6597
	2		1054	354	**2706**	**10556**	66678	146985	59121	**272784**	21,4	5838
	3		1054	379	**2729**	**8583**	52629	27696	65904	**146229**	19,4	2837
CoZ3	4	1	1054	366	**2717**	**7806**	52628	27094	65984	**145706**	19,8	2885
	8		1054	389	**2738**	**6948**	67384	44837	57790	**170011**	25,1	4267
	16		1054	394	**2744**	**6511**	42232	23713	43129	**109074**	37,7	4112

Table 11: All results for NIST P256 using comb method

CS	MC	FM	**Cycles**	RAM	ROM	MCtrl	Map	**Mem**	Buffer	Adder	Rot	Accu	**ALU**
	2		**2606336**	24439	628	171	42	**845**	226	3888	546	472	**5596**
	3		**2850496**	24439	628	171	42	**845**	226	2775	487	472	**4378**
2	4		**3094656**	24439	628	171	42	**845**	226	2115	384	472	**3613**
	8		**4071296**	24439	628	171	42	**845**	226	1235	384	472	**2734**
	16		**6024576**	24439	628	171	42	**845**	226	792	384	472	**2291**
	2		**2095126**	24439	629	171	42	**845**	821	4064	806	698	**6969**
	3		**2312581**	24439	629	171	42	**845**	821	2775	487	472	**4973**
2	4	1	**2556741**	24439	629	171	42	**845**	1119	2115	384	472	**4507**
	8		**3533381**	24439	628	171	42	**845**	1119	1235	384	472	**3627**
	16		**5486661**	24439	628	171	42	**845**	1119	792	384	472	**3184**
	2		**2133128**	24439	2257	203	42	**2510**	226	3888	546	472	**5647**
	3		**2331208**	24439	2256	203	42	**2510**	226	2775	487	472	**4429**
4	4		**2529288**	24439	2256	203	42	**2510**	226	2115	384	472	**3665**
	8		**3321608**	24439	2257	202	42	**2510**	226	1235	384	472	**2785**
	16		**4906248**	24439	2259	202	42	**2512**	226	792	384	226	**2342**
	2		**1718398**	24439	2257	204	42	**2512**	821	4064	806	698	**7020**
	3		**1894813**	24439	2256	203	42	**2510**	821	2775	487	472	**5024**
4	4	1	**2092893**	24439	2257	203	42	**2511**	1119	2115	384	472	**4558**
	8		**2885213**	24439	2256	202	42	**2509**	1119	1235	384	472	**3678**
	16		**4469853**	24439	2259	202	42	**2512**	1119	792	384	472	**3235**

CS	MC	FM	PROM	Mult	**Ctrl**	**Total**	P-Mem	P-ALU	P-Ctrl	**P-Total**	At	AtP
	2		1585	326	**3222**	**9663**	87376	4547	33673	**125596**	25,2	3165
	3		1585	351	**3247**	**8470**	73456	4988	43180	**121624**	24,1	2931
2	4		1585	341	**3237**	**7696**	73446	4912	42974	**121332**	23,8	2888
	8		1585	356	**3247**	**6826**	97190	3693	68987	**169870**	27,8	4722
	16		1585	371	**3267**	**6403**	99123	4747	56575	**160445**	38,6	6193
	2		1096	361	**2775**	**10589**	104199	23030	66887	**194116**	22,2	4309
	3		1096	341	**2752**	**8570**	89509	93029	83708	**266246**	19,8	5272
2	4	1	1096	351	**2763**	**8115**	68466	17742	53591	**139799**	20,7	2894
	8		1096	362	**2773**	**7245**	72178	18203	14026	**104407**	25,6	2673
	16		1096	375	**2785**	**6814**	92493	29880	80162	**202535**	37,4	7575
	2		1481	328	**3179**	**11336**	99159	31414	63644	**194217**	24,2	4700
	3		1481	351	**3202**	**10141**	122581	5457	83220	**211258**	23,6	4986
4	4		1481	341	**3193**	**9368**	122626	5381	83000	**211007**	23,7	5001
	8		1481	356	**3207**	**8502**	116543	5769	67656	**189968**	28,2	5357
	16		1481	371	**3222**	**8076**	137056	48552	98731	**284339**	39,6	11260
	2		1060	362	**2808**	**12340**	92829	24756	51553	**169138**	21,2	3586
	3		1060	363	**2809**	**10343**	137842	26952	64276	**229070**	19,6	4490
4	4	1	1060	366	**2810**	**9879**	94681	21425	41442	**157548**	20,7	3261
	8		1060	392	**2836**	**9023**	115664	21187	71207	**208058**	26,0	5410
	16		1060	395	**2838**	**8586**	140268	36140	65950	**242358**	38,4	9307

Table 12: All results for Brainpool P256r1

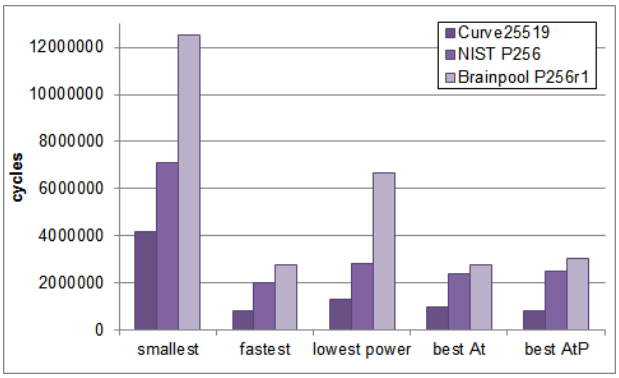
Alg.	MC	FM	**Cycles**	RAM	ROM	MCtrl	Map	**Mem**	Buffer	Adder	Rot	Accu	**ALU**
CoZ1	2		**3677949**	20200	561	154	45	**760**	226	3883	546	472	**5471**
	3		**4412029**	20200	561	154	45	**760**	226	2793	487	472	**4277**
	4		**5146109**	20200	561	154	45	**760**	226	2149	384	472	**3529**
	8		**8082429**	20200	561	154	45	**760**	226	1288	384	472	**2669**
	16		**13955069**	20200	561	154	45	**760**	226	856	384	472	**2236**
CoZ1	2	1	**3043325**	20200	561	154	45	**760**	821	4059	806	698	**6845**
	3		**3744253**	20200	561	154	45	**760**	821	2793	487	472	**4872**
	4		**4478333**	20200	561	154	45	**760**	1119	2149	384	472	**4423**
	8		**7414653**	20200	561	154	45	**760**	1119	1288	384	472	**3562**
	16		**13287293**	20200	561	154	45	**760**	1119	856	384	472	**3129**
CoZ2	2		**3499959**	22171	561	159	45	**765**	226	3883	546	472	**5471**
	3		**4194514**	22171	561	159	45	**765**	226	2793	487	472	**4277**
	4		**4889069**	22171	561	159	45	**765**	226	2149	384	472	**3634**
	8		**7667289**	22171	561	159	45	**765**	226	1288	384	472	**2669**
	16		**13223729**	22171	561	159	45	**765**	226	856	384	472	**2236**
CoZ2	2	1	**2899505**	22171	561	159	45	**765**	821	4059	806	698	**6845**
	3		**3562693**	22171	561	159	45	**765**	821	2793	487	472	**4872**
	4		**4257248**	22171	561	159	45	**765**	1119	2149	384	472	**4423**
	8		**7035468**	22171	561	159	45	**765**	1119	1288	384	472	**3562**
	16		**11905480**	22171	561	159	45	**765**	1119	856	384	472	**3129**
CoZ3	2		**3318752**	26223	561	157	45	**763**	226	3883	546	472	**5471**
	3		**3974712**	26223	561	157	45	**763**	226	2793	487	472	**4277**
	4		**4630672**	26223	561	157	45	**763**	226	2149	384	472	**3529**
	8		**7254512**	26223	561	157	45	**763**	226	1288	384	472	**2669**
	16		**12502192**	26223	561	157	45	**763**	226	856	384	472	**2236**
CoZ3	2	1	**2751664**	26231	561	157	45	**762**	821	4059	806	698	**6844**
	3		**3378000**	26231	561	157	45	**762**	821	2793	487	472	**4872**
	4		**4033960**	26231	561	157	45	**762**	1119	2149	384	472	**4423**
	8		**6657800**	26231	561	157	45	**762**	1119	1288	384	472	**3562**
	16		**11905480**	26231	561	157	45	**762**	1119	856	384	472	**3129**

Alg.	MC	FM	PROM	Mult	**Ctrl**	**Total**	P-Mem	P-ALU	P-Ctrl	**P-Total**	At	AtP
CoZ1	2		2072	309	**3581**	**9812**	95182	65552	77191	**237925**	36,1	8589
	3		2072	348	**3620**	**8656**	75960	47100	91200	**214260**	38,2	8185
	4		2072	342	**3613**	**7902**	75809	44794	90913	**211516**	40,7	8609
	8		2072	354	**3626**	**7054**	59399	44645	48350	**152394**	57,0	8686
	16		2072	362	**3633**	**6629**	94045	88101	79567	**261713**	92,5	24208
CoZ1	2	1	1733	361	**3316**	**10920**	56539	14342	37673	**108554**	33,2	3604
	3		1733	341	**3294**	**8926**	64846	63231	82147	**210224**	33,4	7021
	4		1733	351	**3308**	**8490**	85098	33377	97950	**216425**	38,0	8224
	8		1733	362	**3318**	**7639**	76654	24621	74636	**175911**	56,6	9957
	16		1733	376	**3333**	**7222**	68897	30854	87434	**187185**	96,0	17970
CoZ2	2		2092	328	**3628**	**9863**	38473	40562	48632	**127667**	34,5	4405
	3		2092	352	**3645**	**8686**	55570	90993	73009	**219572**	36,4	7992
	4		2092	341	**3634**	**8033**	42889	44581	72672	**160142**	39,3	6294
	8		2092	356	**3652**	**7085**	85132	65591	53537	**204260**	54,3	11091
	16		2092	371	**3667**	**6668**	49184	66106	88775	**204065**	88,2	17999
CoZ2	2	1	1777	362	**3366**	**10975**	52081	16800	79221	**148102**	31,8	4710
	3		1777	341	**3340**	**8977**	59532	64443	116529	**240504**	32,0	7696
	4		1777	351	**3366**	**8553**	60986	19199	96398	**176583**	36,4	6428
	8		1777	362	**3366**	**7692**	49428	11749	65214	**126391**	54,1	6838
	16		1777	376	**3381**	**7275**	47331	16416	88351	**152098**	86,6	13172
CoZ3	2		2009	328	**3584**	**9817**	96418	41064	54037	**191519**	32,6	6244
	3		2009	352	**3606**	**8645**	75552	21256	31904	**128712**	34,4	4428
	4		2009	341	**3595**	**7887**	75493	21063	31720	**128276**	36,5	4682
	8		2009	356	**3606**	**7037**	48795	65748	62510	**177053**	51,1	9047
	16		2009	371	**3624**	**6623**	74662	66366	93913	**234941**	82,8	19453
CoZ3	2	1	1739	361	**3377**	**10983**	56334	18709	95758	**170801**	30,2	5158
	3		1739	363	**3381**	**9015**	72631	109866	69458	**251955**	30,5	7685
	4		1739	366	**3376**	**8561**	71395	16986	81916	**170297**	34,5	5875
	8		1739	392	**3403**	**7727**	46692	12961	46936	**106589**	51,4	5479
	16		1739	395	**3407**	**7299**	47331	16416	88351	**152098**	86,9	13217

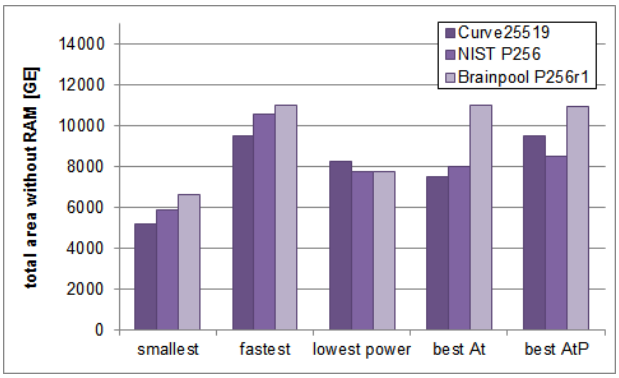
Table 13: All results for Curve25519

MC	FM	FI	**Cycles**	RAM	ROM	MCtrl	Map	**Mem**	Buffer	Adder	Rot	Accu	**ALU**
2			**1286977**	16355	98	128	46	**272**	226	3745	546	472	**5288**
3			**1482625**	16355	97	128	46	**271**	226	2674	487	472	**4150**
4			**1678273**	16355	97	128	46	**271**	226	2037	384	472	**3410**
8			**2552575**	16355	98	128	46	**272**	226	1188	431	472	**2608**
16			**4178899**	16355	98	128	46	**272**	226	760	487	472	**2236**
2			**877339**	16355	98	128	46	**272**	821	3924	806	698	**6665**
3			**1051588**	16355	97	128	46	**271**	821	2674	487	472	**4746**
4	1		**1247236**	16355	97	128	46	**271**	1119	2037	384	472	**4303**
8			**2121538**	16355	98	128	46	**272**	1119	1188	431	472	**3501**
16			**3747862**	16355	98	128	46	**272**	1119	759	487	472	**3129**
2			**1194511**	16355	95	135	46	**276**	226	3745	546	472	**5288**
3			**1374735**	16355	95	135	46	**276**	226	2674	487	472	**4150**
4		1	**1554959**	16355	95	135	46	**276**	226	2037	384	472	**3410**
8			**2360335**	16355	95	135	46	**276**	226	1188	431	472	**2608**
16			**3858447**	16355	95	135	46	**276**	226	760	487	250	**2236**
2			**817167**	16225	95	135	46	**275**	821	3924	806	698	**6666**
3			**977679**	16225	95	135	46	**275**	821	2674	487	472	**4746**
4	1	1	**1157903**	16225	95	135	46	**275**	1119	2037	384	472	**4303**
8			**1963279**	16225	95	135	46	**275**	1119	1188	431	472	**3502**
16			**3461391**	16225	95	135	46	**275**	1119	759	487	472	**3129**
2			**1195102**	16355	95	135	46	**276**	226	3745	546	472	**5288**
3			**1375326**	16355	95	135	46	**276**	226	2674	487	472	**4150**
4		2	**1555550**	16355	95	135	46	**276**	226	2037	384	472	**3410**
8			**2360926**	16355	95	135	46	**276**	226	1188	431	472	**2608**
16			**3859038**	16355	95	135	46	**276**	226	760	487	472	**2236**
2			**817758**	16225	95	135	46	**275**	821	3924	806	698	**6666**
3			**978270**	16225	95	135	46	**275**	821	2674	487	472	**4746**
4	1	2	**1158494**	16225	95	135	46	**275**	1119	2037	384	472	**4303**
8			**1963870**	16225	95	135	46	**275**	1119	1188	431	472	**3502**
16			**3461982**	16225	95	135	46	**275**	1119	759	487	472	**3129**

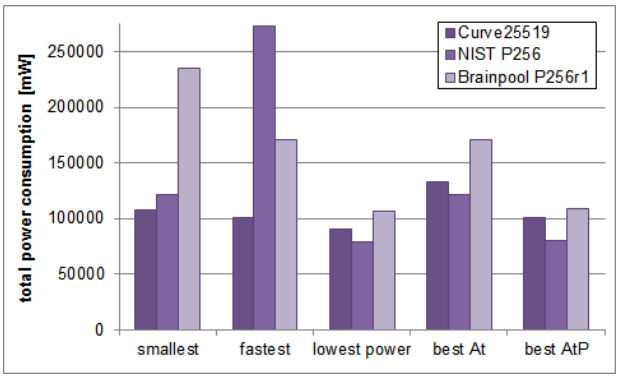
MC	FM	FI	PROM	Mult	**Ctrl**	**Total**	P-Mem	P-ALU	P-Ctrl	**P-Total**	At	AtP
2			1437	309	**2694**	**8252**	38990	3104	67332	109426	10,6	1160
3			1437	348	**2734**	**7154**	38990	3030	66824	108844	10,8	1176
4			1437	343	**2728**	**6408**	60930	2905	81475	145310	14,5	2107
8			1492	356	**2812**	**5692**	42254	3435	61531	107220	21,7	2327
16			1355	371	**2685**	**5192**	50973	14551	42033	107557	7,9	850
2			788	341	**2093**	**9029**	56186	29184	78983	164353	7,5	1233
3			788	341	**2090**	**7106**	60988	17813	77506	156307	8,3	1297
4	1		788	351	**2096**	**6670**	40145	13610	63708	117463	13,0	1527
8			1067	364	**2374**	**6146**	60482	14047	56960	131489	21,2	2788
16			942	377	**2259**	**5659**	42610	3057	80203	125870	10,3	1296
2			1707	327	**3050**	**8613**	62903	2820	75260	140983	10,3	1452
3			1707	353	**3076**	**7501**	62903	2746	74837	140486	10,5	1475
4		1	1707	341	**3064**	**6749**	51572	2543	54508	108623	13,6	1477
8			1519	356	**2895**	**5778**	54667	3061	61471	119199	21,0	2503
16			1556	371	**2939**	**5451**	30951	13934	56454	101339	7,7	780
2			1171	357	**2537**	**9478**	48938	17195	79642	145775	7,4	1079
3			1171	362	**2538**	**7558**	33018	15906	69458	118382	8,2	971
4	1	1	1171	366	**2544**	**7121**	44224	14800	45506	104530	12,8	1338
8			1333	393	**2726**	**6503**	42803	13855	40002	96660	21,3	2059
16			1344	395	**2746**	**6150**	63489	3060	76600	143149	10,1	1446
2			1511	327	**2876**	**8439**	61283	68737	88955	218975	10,1	2212
3			1511	353	**2903**	**7328**	61283	65376	88465	215124	10,2	2194
4		2	1511	341	**2891**	**6576**	42613	66469	72512	181594	13,4	2433
8			1379	356	**2778**	**5661**	52205	47946	44458	144609	20,8	3008
16			1454	371	**2867**	**5378**	49166	18040	84587	151793	7,7	1169
2			1089	357	**2454**	**9396**	43214	15571	74457	133242	7,3	973
3			1089	362	**2461**	**7482**	32364	13707	65225	111296	8,2	913
4	1	2	1089	366	**2471**	**7049**	44156	14800	47689	106645	12,7	1354
8			1232	393	**2680**	**6457**	42809	13855	35813	92477	21,0	1942
16			1228	395	**2670**	**6074**	39463	2881	48956	91300	10,7	977



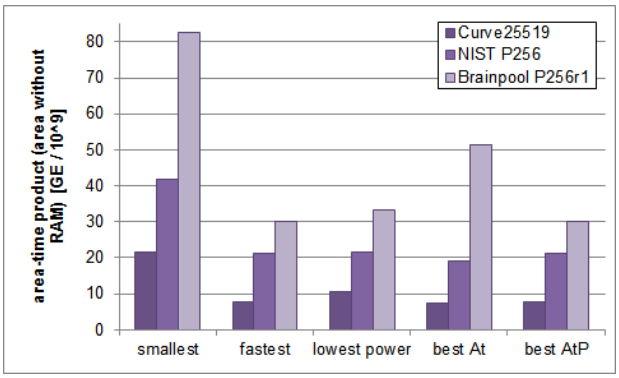
(a) Comparison of needed cycles



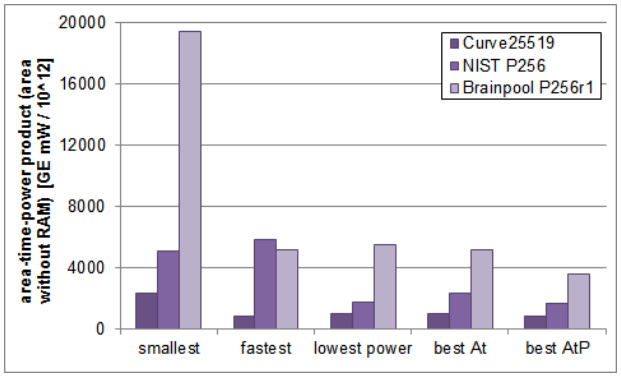
(b) Comparison of area



(c) Comparison of power consumption



(d) Comparison of area-time product



(e) comparison of area-time-power product

Figure 16: Comparison of implemented curves for different metrics

6.5 Comparison of Curves

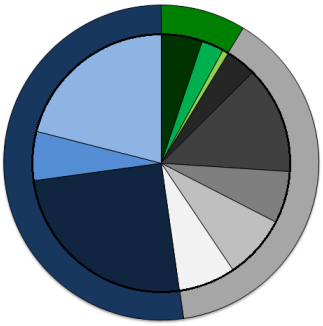
For this comparison, the comb method are not considered. Figure 16 compares the values of the best implementations of all curves at area, cycles, power, At and AtP. Figure 16a shows the needed cycles for a point-multiplication of the best implementations of all curves in these five categories. In each case, Curve25519 needs the smallest number of cycles for the point-multiplication. The reason for this is that the formulas for Curve25519 need fewer multiplications and the reduction can be done faster. Figure 16b compares the needed area of the best implementations. This shows that in the category lowest power consumption and best AtP the best designs of the other curves are partially smaller than the one of Curve25519. However, the areas of all curves are quite similar, because they mainly differ in the length of the machine code and the size of the ROM. In Figure 16c one can see the power consumption of the top implementations. Thereby, NIST P256 has a lower power consumption in the categories power, At and AtP. In area and time the implementations of Curve2559 have the lowest power consumption. The comparison of the At metric is also dominated by Curve25519 as shown in Figure 16d. Also comparing the values the AtP metric Curve25519 has the best values in all categories. Summarizing it can be said that Curve25519 dominates the most metrics, only the power consumption of some NIST implementations is lower. Partially, Brainpool has much higher values compared to the other curves partially.

Figures 17 to 19 compare the proportioning of the components in the smallest and fastest implementation of all curves. The legend for the used colors is shown in Figure 20. Table 14 lists the corresponding values. In the fastest variants, the core- and accu-adder are responsible for more than a third of the area. In contrast, the PROM needs the most area in the smallest implementations. The share of the area for the memory controller is twice as large in the fast implementations as in the small ones.

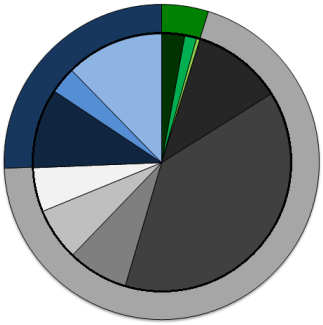
Table 14: Shares of area of smallest and fastest variants

Alg	Metric	Parameter	ROM	MCtrl	Map	**Mem**	PROM	Mult	**Ctrl**
Curve25519	smallest	MC=16, FI=0, FI=0	1,89%	2,47%	0,89%	**5,24%**	26,10%	7,15%	**51,71%**
NIST P256	smallest	CoZ1, MC=16, FM=0	5,17%	2,79%	0,76%	**8,70%**	25,09%	6,15%	**52,25%**
Brainpool P256r1	smallest	CoZ3, MC=16, FM=0	8,47%	2,37%	0,68%	**11,52%**	30,33%	5,60%	**54,72%**
Curve25519	fastest	MC=02, FM=1, FI=1	1,00%	1,42%	0,49%	**2,90%**	12,35%	3,77%	**26,77%**
NIST P256	fastest	CoZ3, MC=02, FM=1	2,88%	1,49%	0,43%	**4,79%**	9,98%	3,35%	**25,63%**
Brainpool P256r1	fastest	CoZ3, MC=02, FM=0	5,11%	1,43%	0,41%	**6,94%**	15,83%	3,29%	**30,75%**

Alg	Metric	Parameter	Buffer	Adder	Rot	Accu	**ALU**
Curve25519	smallest	MC=16, FI=0, FI=0	4,35%	14,64%	9,38%	9,09%	**43,07%**
NIST P256	smallest	CoZ1, MC=16, FM=0	3,84%	13,46%	6,53%	8,02%	**39,03%**
Brainpool P256r1	smallest	CoZ3, MC=16, FM=0	3,41%	12,92%	5,80%	7,13%	**33,76%**
Curve25519	fastest	MC=02, FM=1, FI=1	8,66%	41,40%	8,50%	7,36%	**70,33%**
NIST P256	fastest	CoZ3, MC=02, FM=1	11,28%	38,50%	7,64%	6,61%	**69,57%**
Brainpool P256r1	fastest	CoZ3, MC=02, FM=0	7,48%	36,96%	7,34%	6,36%	**62,31%**

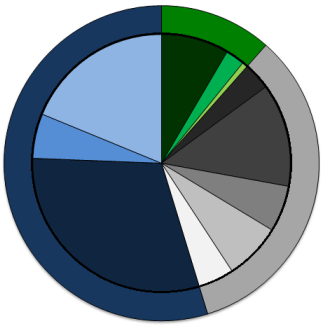


(a) Smallest variant ($CoZ1, MC = 16, FM = 0$)

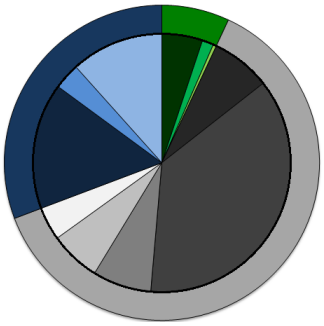


(b) Fastest variant ($CoZ3, MC = 02, FM = 1$)

Figure 17: Shares of area of smallest and fastest variants for NIST P256

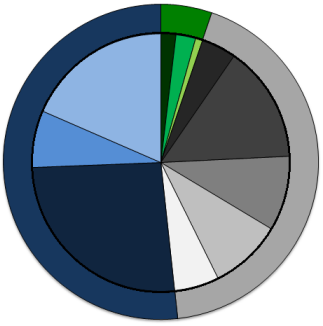


(a) Smallest variant ($CoZ1, MC = 16, FM = 0$)

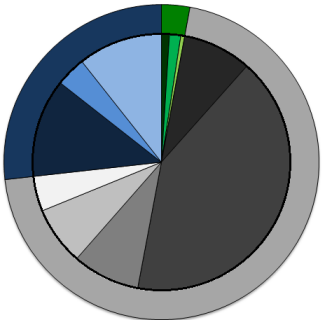


(b) Fastest variant ($CoZ3, MC = 02, FM = 1$)

Figure 18: Shares of area of smallest and fastest variants for Brainpool P256r1



(a) Smallest variant ($MC = 16, FM = 0$)



(b) Fastest variant ($MC = 02, FM = 1$, FI=1)

Figure 19: Shares of area of smallest and fastest variants for Curve25519

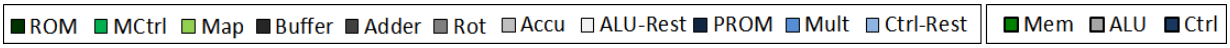


Figure 20: Legend

6.6 Comparison with Related Work

In Table 15, we compare the results of the related work with our designs. Since, the power consumption is not given for their implementations, we can only compare the area and running time. Area for the RAM is not included. All designs except the one from Wolkerstorfer use a 0.13-μm technology and implement a 256-bit elliptic curve over a prime field and thus are fairly comparable. A comparison of the results of Satoh shows that a word size of 64 bits has the best At-product. However, it is only slightly better than the 32-bit implementation, which needs much less area. Thus, a 32-bit implementation seems to be a quite good compromise between speed and area.

The area of all of our implementations is significantly smaller than the smallest design from the related work. However, the running time is longer in many cases, because our design is primarily designed for small area and low power and not high-speed. Totally, our Curve25519 implementations have the best At-product of all compared designs. Our designs for the NIST curve have a bigger At-product than the implementation of Muthukumar. This is due to much higher running time which is caused by our generic design without dedicated reduction structure in the ALU.

Table 15: Comparison of 256-Bit implementations

	Curves	Word size [bit]	Area [GE]	Cycles	At [GE/10^9]
Satoh [ST03]	any	8	19 935	9 385 000	187,09
		16	25 051	2 711 000	67,91
		32	43 521	880 000	38,30
		64	106 659	340 000	36,26
Wolkerstorfer [Wol04]	any$\in GF(p^{256})$	256	14 763	1 175 451	17,35
Chen [CBC07]	any$\in GF(p^{256})$	n.a	122 000	562 000	68,56
Lai [LH09]	IEEE 1363	32	197 028	252 067	49,66
Muthukumar [MJ10]	IEEE 1363	256	184 000	54 568	10,04
Smallest Curve25519 design	Curve25519	32	5 192	4 178 899	21,70
Fastest Curve25519 design	Curve25519	32	9 478	817 167	7,75
Smallest At Curve25519 design	Curve25519	32	7 482	978 270	7,32
Smallest NIST P256 design	NIST P256	32	5 883	7 111 355	41,84
Fastest NIST P256 design (w/o comb)	NIST P256	32	10 556	2 024 179	21,37
Fastest NIST P256 design (comb)	NIST P256	32	12 340	1 718 398	21,21
Smallest At NIST P256 design	NIST P256	32	8 020	2 384 652	19,12

6.7 Evaluation of Costs

In Table 16 we evaluate the costs of the elements with a major share on area and calculation time. Thereby the costs for the needed cycles are related to the whole point-multiplication. The area is only an estimation, because the growth of the area is not linear. The table shows that the same effort in additional hardware has different impact on different curves. For

instance, adding the multiplication controller saves 630 000 cycles for NIST (up to 3,5%) and Brainpool (up to 4,8%) curves, but it saves only 400 000 cycles at Curve25519 (up to 2,1%). Reducing MC has more influence on Brainpool than on the other curves, because the reduction after a 256-bit multiplication also uses 32-bit multiplications. In contrast to the other curves, if MC becomes greater than four at Curve25519, the size of the PROM even increases. This is because at the reduction of Curve25519 the high part of the result must be multiplied with 38. If the number of adders in the core-adder decreases below six, the multiplication with 38 is no longer possible within one step. Then, the multiplication must be done in more than one step which requires additional instructions. Sometimes the size of the PROM shrinks if the number of instructions is increased. For instance, the reduction after multiplication for Curve25519 with $MC \leq 4$ needs 93 instructions, while the reduction with $MC = 8$ requires 123 instructions. However, removing the 93 instructions in the first case reduces the area by 314GE, while removing the 130 instructions in the second case reduces the area only by 130GE. This seems to be the case, if the number of instructions after increasing is slightly above a power of two. Then the synthesizer can optimize the size of the ROM better. The table shows also a main reason why the curves of NIST and Brainpool are much slower than Curve25519: Their running time is strongly influenced by the reduction after the 256-bit multiplication. The running time for pure multiplication is comparable and the running time of other arithmetic operations is negligible.

Table 16: Comparison of costs in time and area

Description	Ins	Area [GE]	Cycles
Addition/Subtraction with reduction - Curve25519 (FI)	131	354	267 240
Addition/Subtraction with reduction - NIST (CoZ3)	165	836	550 440
Addition/Subtraction with reduction - Brainpool (CoZ3)	199	-169	663 864
Pure multiplication - Curve25519 (FI)	210	22	$410\,990 + MC \cdot 180\,160$
Pure multiplication - NIST (CoZ3)	210	-919	$618\,748 + MC \cdot 271\,232$
Pure multiplication - Brainpool (CoZ3)	210	-175	$618\,748 + MC \cdot 271\,232$
Multiplication-reduction - Curve25519 (FI, $2 \leq MC \leq 4$)	93	314	261 795
Multiplication-reduction - Curve25519 (FI, $MC = 8$)	123	130	346 245
Multiplication-reduction - Curve25519 (FI, $MC = 16$)	143	156	402 545
Multiplication-reduction - NIST (CoZ3)	263	577	1 114 594
Multiplication-reduction - Brainpool (CoZ3)	326	1 042	$995\,930 + MC \cdot 385\,658$
Multiplication controller for FM - Curve25519 ($MC = 2$)	-209	1 341	$-409\,638$
Multiplication controller for FM - Curve25519 ($MC > 2$)	-209	539	$-431\,037$
Multiplication controller for FM - NIST ($MC = 2$)	-209	1 341	$-639\,360$
Multiplication controller for FM - NIST ($MC > 2$)	-209	539	$-672\,512$
Multiplication controller for FM - Brainpool ($MC = 2$)	-209	1 341	$-634\,624$
Multiplication controller for FM - Brainpool ($MC > 2$)	-209	539	$-667\,776$
Using fast inversion (FI) - Curve25519 ($MC = 2$)	482	1845	$-92\,466$
Using fast inversion (FI2) - Curve25519 ($MC = 2$)	277	954	$-91\,875$
Fast inversion with `REPEAT` - Curve25519 ($MC = 2$)	270	1603	$-91\,946$
ALU with MC=8 instead of MC=16 - Curve25519 (FI)	-20	372	$-1\,498\,112$
ALU with MC=4 instead of MC=16 - Curve25519 (FI)	-50	1 174	$-2\,303\,488$
ALU with MC=3 instead of MC=16 - Curve25519 (FI)	-50	1 914	$-2\,483\,712$
ALU with MC=2 instead of MC=16 - Curve25519 (FI)	-50	3 052	$-2\,644\,224$
ALU with MC=8 instead of MC=16 - NIST (CoZ3)	0	372	$-2\,166\,784$
ALU with MC=4 instead of MC=16 - NIST (CoZ3)	0	1 174	$-3\,250\,176$
ALU with MC=3 instead of MC=16 - NIST (CoZ3)	0	1 914	$-3\,521\,024$
ALU with MC=2 instead of MC=16 - NIST (CoZ3)	0	3 052	$-3\,791\,872$
ALU with MC=8 instead of MC=16 - Brainpool (CoZ3)	0	414	$-5\,247\,680$
ALU with MC=4 instead of MC=16 - Brainpool (CoZ3)	0	1 264	$-7\,871\,520$
ALU with MC=3 instead of MC=16 - Brainpool (CoZ3)	0	2 022	$-8\,527\,480$
ALU with MC=2 instead of MC=16 - Brainpool (CoZ3)	0	3 194	$-9\,183\,440$
Additional register in RAM	-	1 971	-
Additional register in ROM	-	136	-

7 Conclusions

In this work, we presented a dynamic platform to compare three different frequently used 256-bit elliptic curves. The architecture is very flexible and allows very small designs. This is due to several facts. First, we try to reuse the components in the arithmetic unit as much as possible. For instance, we have no separated addition and multiplication unit. Furthermore, we use a 9-bit machine code to reduce the size of the program ROM. Additionally, we use only a single-ported RAM which is smaller than a dual-ported RAM for a further reduction of the area requirements.

We implemented three different elliptic curves: NIST P256, Brainpool P256r1, and Curve25519. Then, we compared their calculation time, area, power consumption, their area-time product, and their area-time-power product. For NIST P256 and Brainpool P256r1 we used a common projective Z-coordinate representation to reduce the memory requirements. In order to evaluate different configurations, three different formulas (CoZ1, CoZ2, and CoZ3) were implemented. These formulas differ in the number of required operations and registers. For Curve25519, we presented new explicit formulas in XZ-coordinate representation. These formulas need only six multiplications and fewer registers than previous published formulas.

In our architecture, the number of needed clock cycles for a 32-bit multiplication can be varied from 2 to 16. For high-speed implementations, an additional multiplication controller can be used. It can perform a 256-bit multiplication in an optimized way. Typically, it uses the classic product-scanning multiplication algorithm. To perform a 32-bit multiplication in only two cycles, in the high-speed variant, it uses our proposed zigzag product-scanning multiplication algorithm. This reduces the running time from 5% (Brainpool P256r1 with $MC = 16$) up to 46% (Curve25519 with $MC = 2$).

For the point-multiplication we used the Montgomery ladder and for NIST P256 additionally the fixed-base comb method. The fastest implementation using the comb method is 18% faster than the fastest variant using the Montgomery ladder. Additionally it needs 17% more area, but the power consumption is reduced by 38%. The variants with the comb method have slightly better values at the At and AtP metrics than for the others.

For Curve25519, we additionally implemented two optimized variants for modular inversion in the finite field. The machine code for the first one, defines each multiplication separately. Since the code contains up to 99 identical multiplications in a row, the second variant uses loops for a shorter machine code. The fastest variant with fast inversion is seven percent faster than the same variant without fast inversion. Additionally it needs five percent less power due to the shorter runtime. In return this variant needs five percent more area, caused by the longer machine code.

The fastest implementation of Curve25519 needs 817167 cycles for the point-multiplication. The fastest implementations for NIST P256 and Brainpool P256r1 are much slower. To get designs with a small area, it is possible to reduce the number of used adders from sixteen to two. In return, the number of needed cycles for the multiplication increases. All three curves can be implemented in fewer than 7000 GE without RAM. Additionally, one can choose between carry-ripple and carry-safe adder for a shorter critical path. The

area and power consumption of both variants are almost the same. Looking at the best implementations at the At and AtP metrics, they all use a multiplication controller and need two or three cycles for the 32-bit multiplication. Thus, the additional area for fast multiplication seems to be compensated by the faster calculation.

The generic instruction set allows the implementation of any elliptic curve and even other algorithms. Thus the functionality of the proposed design can be extended to implement almost any cryptographic protocol. For protection against basic side-channel attacks, all algorithms have a constant runtime.

As future work our design can be tested against side-channel attacks to find and fix vulnerable points to improve the security. Moreover, additional functions can be implemented to provide a complete cryptographic system.

References

[ABCS06] Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.

[ABR99] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHAES: An encryption scheme based on the Diffie-Hellman problem. *Available at citeseer. ist. psu. edu/abdalla99dhaes. html*, 1999.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in cryptologyCRYPTO86*, pages 311–323. Springer, 1987.

[BBB+12] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. NIST Special Publication 800-57: Recommendation for Key Management - Part 1, 2012.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in CryptologyEUROCRYPT97*, pages 37–51. Springer, 1997.

[Ber05] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[BJS07] Elaine Barker, Don Johnson, and Miles Smid. NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, 2007.

[BL07] Daniel J Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Advances in cryptology–ASIACRYPT 2007*, pages 29–50. Springer, 2007.

[BL14] Daniel J Bernstein and Tanja Lange. Hyperelliptic - explicit-formulas database, 2014. `http://hyperelliptic.org/EFD/index.html`.

[CBC07] Gang Chen, Guoqiang Bai, and Hongyi Chen. A high-performance elliptic curve cryptographic processor for general curves over GF(p) based on a systolic arithmetic unit. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(5):412–416, 2007.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems*, pages 292–302. Springer, 1999.

[Cry04] Elliptic Curve Cryptography. The advantages of elliptic curve cryptography for wireless security. *IEEE Wireless Communications*, page 63, 2004.

[DH76] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[DV11] Vincent Dupaquis and Alexandre Venelli. Redundant Modular Reduction Algorithms. In *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 102–114. 2011.

[DW14] D-Wave. Quantum computing, 2014.

[Edw07] Harold Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[Far03] Faraday. $0.13\mu m$ low leakage high density standard cells - fsc0l_d core cell, December 2003.

[Fer] Pierre de Fermat. letter to Frénicle de Bessy (1640-10-18). `https://web.archive.org/web/20061222105104/http://www.cs.utexas.edu/users/wzhao/fermat2.pdf`.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[GFD09] Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. FIPS PUB 186-3 federal Information processing Standards Publication Digital Signature Standard (DSS), 2009.

[GK86] Shafi Goldwasser and Joe Kilian. Almost all primes can be quickly certified. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 316–329. ACM, 1986.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded SystemsCHES 2001*, pages 251–261. Springer, 2001.

[GP08] Tim Güneysu and Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 62–78. Springer, 2008.

[Hew05] Paul Hewitt. A brief history of elliptic curves, 2005.

[HFP10] Michael Hutter, Martin Feldhofer, and Thomas Plos. An ECDSA processor for RFID authentication. In *Radio Frequency Identification: Security and Privacy Issues*, pages 189–202. Springer, 2010.

[HFW11] Michael Hutter, Martin Feldhofer, and Johannes Wolkerstorfer. A cryptographic processor for low-resource devices: canning ECDSA and AES like sardines. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 144–159. Springer, 2011.

[HJS11] Michael Hutter, Marc Joye, and Yannick Sierra. Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation. In *Progress in Cryptology–AFRICACRYPT 2011*, pages 170–187. Springer, 2011.

[HVM04] Darrel Hankerson, Scott Vanstone, and Alfred J Menezes. *Guide to elliptic curve cryptography*. Springer, 2004.

[HW11] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 459–474. Springer, 2011.

[iee00] IEEE Standard Specifications for Public-Key Cryptography. *IEEE Std 1363-2000*, 2000.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003*, pages 463–481. Springer, 2003.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[JY03] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 291–302. Springer, 2003.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in CryptologyCRYPTO99*, pages 388–397. Springer, 1999.

[Knu05] Donald Ervin Knuth. *The art of computer programming*. Pearson Education, 2005.

[KO63] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in CryptologyCRYPTO96*, pages 104–113. Springer, 1996.

[KP06] Sandeep Kumar and Christof Paar. Are standards compliant elliptic curve cryptosystems feasible on RFID. In *Workshop on RFID Security*, pages 12–14, 2006.

[LH09] Jyu-Yuan Lai and Chih-Tsun Huang. A highly efficient cipher processor for dual-field elliptic curve cryptography. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 56(5):394–398, 2009.

[LJ87] Hendrik W Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[LM10] M. Lochter and J. Merkle. RFC 5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation. Technical report, March 2010.

[Mil86] Victor S Miller. Use of elliptic curves in cryptography. In *Advances in CryptologyCRYPTO85 Proceedings*, pages 417–426. Springer, 1986.

[MJ10] B Muthukumar and S Jeevananthan. High speed hardware implementation of an elliptic curve cryptography (ECC) co-processor. In *Trendz in Information Sciences & Computing (TISC), 2010*, pages 176–180. IEEE, 2010.

[MMS01] David May, Henk L Muller, and Nigel P Smart. Random register renaming to foil DPA. In *Cryptographic Hardware and Embedded SystemsCHES 2001*, pages 28–38. Springer, 2001.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mon87] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[MSY06] Tal G Malkin, François-Xavier Standaert, and Moti Yung. A comparative cost/security analysis of fault attack countermeasures. In *Fault Diagnosis and Tolerance in Cryptography*, pages 159–172. Springer, 2006.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[OP01] Gerardo Orlando and Christof Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. In *Cryptographic Hardware and Embedded SystemsCHES 2001*, pages 348–363. Springer, 2001.

[PH78] Stephen C Pohlig and Martin E Hellman. An improved algorithm for computing logarithms over and its cryptographic significance (corresp.). *Information Theory, IEEE Transactions on*, 24(1):106–110, 1978.

[Pol78] John M Pollard. Monte carlo methods for index computation (). *Mathematics of computation*, 32(143):918–924, 1978.

[PZ03] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv preprint quant-ph/0301141*, 2003.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer, 2001.

[SBM+06] Kazuo Sakiyama, Lejla Batina, Nele Mentens, Bart Preneel, and Ingrid Verbauwhede. Small-footprint ALU for public-key processors for pervasive security. In *Workshop on RFID Security*, volume 12, 2006.

[SDMPV06] Kazuo Sakiyama, Elke De Mulder, Bart Preneel, and Ingrid Verbauwhede. A parallel processing hardware architecture for elliptic curve cryptosystems. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 3, pages III–III. IEEE, 2006.

[Ser01] American National Standards Institute Standards Committee Financial Services. *Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Curve Cryptography: ANSI American National Standard for Financial Services, X9.63-2001.* American national standard / ANSI. Accredited Standards Committee X9, Incorporated, 2001.

[Ser05] American National Standards Institute Standards Committee Financial Services. *Public Key Cryptography for the Financial Services Industry - the Elliptic Curve Digital Signature Algorithm (ECDSA): ANSI American National Standard for Financial Services, ANS X9.62-2005.* American national standard / ANSI. Accredited Standards Committee X9, Incorporated, 2005.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.

[Sol11] Jerome A Solinas. Pseudo-mersenne prime. In *Encyclopedia of Cryptography and Security*, pages 992–992. Springer, 2011.

[SOOS95] Richard Schroeppel, Hilarie Orman, Sean OMalley, and Oliver Spatscheck. *Fast key exchange with elliptic curve systems*. Springer, 1995.

[ST03] Akashi Satoh and Kohji Takano. A scalable dual-field elliptic curve cryptographic processor. *Computers, IEEE Transactions on*, 52(4):449–460, 2003.

[TCW+05] Timothy J Todman, George A Constantinides, Steven JE Wilton, Oscar Mencer, Wayne Luk, and Peter YK Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005.

[TV03] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against DPA at the logic level. In *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 125–136. Springer, 2003.

[TV05] Kris Tiri and Ingrid Verbauwhede. A VLSI design flow for secure side-channel attack resistant ICs. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*, pages 58–63. IEEE Computer Society, 2005.

[Wal64] Christopher S Wallace. A suggestion for a fast multiplier. *Electronic Computers, IEEE Transactions on*, (1):14–17, 1964.

[WE93] Neil HE Weste and Kamran Eshraghian. *Principles of CMOS VLSI design*, volume 2. Addison-Wesley Reading, MA, 1993.

[WFF10] Erich Wenger, Martin Feldhofer, and Norbert Felber. A 16-Bit Microprocessor Chip for Cryptographic Operations on Low-Resource Devices. In J. Sturm C. Zhang M. Köberle M. Ley, E. Ofner, editor, *Austrochip 2010*, pages 55 – 60. Fachhochschule Kärnten, 2010.

[WH11] Erich Wenger and Michael Hutter. A hardware processor supporting elliptic curve cryptography for less than 9 kGEs. In *Smart Card Research and Advanced Applications*, pages 182–198. Springer, 2011.

[WH12] Erich Wenger and Michael Hutter. Exploring the design space of prime field vs. binary field ECC-hardware implementations. In *Information Security Technology for Applications*, pages 256–271. Springer, 2012.

[WH14] Wolfgang Wieser and Michael Hutter. Efficient Multiplication on Low-Resource Devices. In IEEE Computer Society, editor, *Euromicro Conference on Digital System Design Architectures, Methods and Tools – DSD 2014, 2014, Proceedings*. IEEE Computer Society, 2014.

[WHGH+08] William Whyte, Nick Howgrave-Graham, Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, and Philip S. Hirschhorn. IEEE P1363 Draft 9: Standard Specifications for Public Key Cryptography - Annex A, Number-Theoretic Background. *IACR Cryptology ePrint Archive*, 2008.

[Wol04] Johannes Wolkerstorfer. *Hardware Aspects of Elliptic Curve Cryptography.* PhD thesis, 2004.

[WW11] Erich Wenger and Mario Werner. Evaluating 16-bit Processors for Elliptic Curve Cryptography. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 166 – 181. Springer, 2011.

[WW14] Erich Wenger and Paul Wolfger. Solving the Discrete Logarithm of a 113-bit Koblitz Curve with an FPGA Cluster. 2014.

A Examples to Algorithms

In this chapter, we show examples of previously described algorithms.

Field Inversion

Example 6. *: Inverting $a = 212$ in $\mathbb{F}_{\text{|}}$, where $p = 2^8 - 2^4 - 1 = 239$ using the extended euclidean algorithm.*

#	u	v	x_1	x_2		q	u'	v'	x_1'	x_2'
1	212	239	1	0	$\rightarrow$	1	27	212	-1	1
2	27	212	-1	1	$\rightarrow$	7	23	27	8	-1
3	23	27	8	-1	$\rightarrow$	1	4	23	-9	8
4	4	23	-9	8	$\rightarrow$	5	3	4	53	-9
5	3	4	53	-9	$\rightarrow$	1	**1**	3	**-62**	53

$\rightarrow a^{-1} = x_1' \pmod{p} = -62 \pmod{239} \equiv 177$

Example 7. *: Inverting $a = 212$ in $\mathbb{F}_{\text{|}}$, where $p = 2^8 - 2^4 - 1 = 239$ using the binary gcd algorithm.*

#	Calculation steps	
1.1	$u = u \gg 1 = 106,$	$x_1 = (x_1 + p) \gg 2 = 120$
1.2	$u = u \gg 1 = 53,$	$x_1 = x_1 \gg 2 = 60$
1.3	$v = v - u = 186,$	$x_2 = x_2 - x_1 = -60$
2.1	$v = v \gg 1 = 93,$	$x_2 = x_2 \gg 2 = -30$
2.2	$v = v - u = 40,$	$x_2 = x_2 - x_1 = -90$
3.1	$v = v \gg 1 = 20,$	$x_2 = x_2 \gg 2 = -45$
3.2	$v = v \gg 1 = 10,$	$x_2 = (x_2 + p) \gg 2 = 97$
3.3	$v = v \gg 1 = 5,$	$x_2 = (x_2 + p) \gg 2 = 168$
3.4	$u = u - v = 48,$	$x_1 = x_1 - x_2 = -108$
4.1	$u = u \gg 1 = 24,$	$x_1 = x_1 \gg 2 = -54$
4.2	$u = u \gg 1 = 12,$	$x_1 = x_1 \gg 2 = -27$
4.3	$u = u \gg 1 = 6,$	$x_1 = (x_1 + p) \gg 2 = 106$
4.4	$u = u \gg 1 = 3,$	$x_1 = x_1 \gg 2 = 53$
4.5	$v = v - u = 2,$	$x_2 = x_2 - x_1 = 115$
5.1	$v = v \gg 1 = 1,$	$x_2 = (x_2 + p) \gg 2 = 177$
5.2	$u = u - v = 2,$	$x_1 = x_1 - x_2 = -124$

$\rightarrow a^{-1} = x_2' \pmod{p} = 177 \pmod{239} \equiv 177$

Example 8. *: Inverting $a = 212$ in $\mathbb{F}_{\text{!}}$, where $p = 2^8 - 2^4 - 1 = 239$ using Fermat's little theorem. The optimized variant can be used only for given prime and reuses intermediate results. Thus, the optimized variant needs one multiplication less. Though, it needs two additional variables to store the intermediate results.*

#	Calculation steps		
	Algorithm 9	Optimized	
1	$r = r \cdot r = 12$	$r = a \cdot a = 12$	a^{2^1}
2	$r = r \cdot a = 154$	$r = r \cdot a = 154$	$a^{2^2-2^0}$
3	$r = r \cdot r = 55$	$s = r \cdot r = 55$	$a^{2^3-2^1}$
4	$r = r \cdot a = 188$	$t = s \cdot a = 188$	$a^{2^3-2^0}$
5	$r = r \cdot r = 211$	$r = t \cdot t = 211$	$a^{2^4-2^1}$
6	$r = r \cdot r = 67$	$r = r \cdot r = 67$	$a^{2^5-2^2}$
7	$r = r \cdot a = 103$	$r = r \cdot r = 187$	$a^{2^6-2^3}$
8	$r = r \cdot r = 93$	$r = r \cdot r = 75$	$a^{2^7-2^4}$
9	$r = r \cdot a = 118$	$r = r \cdot r = 128$	$a^{2^8-2^5}$
10	$r = r \cdot r = 62$	$r = r \cdot t = 164$	$a^{2^8-2^5+2^3-2^0}$
11	$r = r \cdot r = 20$	$r = r \cdot s = \mathbf{177}$	$a^{2^8-2^5+2^3-2^0+2^3-2^1} = p - 2$
12	$r = r \cdot a = \mathbf{177}$		

Point-Multiplication

Example 9. *: Using binary method to multiply $k = 0x10011101$ by $P = 11010010$.*

i	k_i	Calculation steps
7	1	$Q = 0000000000000000_2 \cdot 2 + 11010010_2 = 0000000011010010_2$
6	0	$Q = 0000000011010010_2 \cdot 2 = 0000000110100100_2$
5	0	$Q = 0000000110100100_2 \cdot 2 = 0000001101001000_2$
4	1	$Q = 0000001101001000_2 \cdot 2 + 11010010_2 = 0000011101100010_2$
3	1	$Q = 0000011101100010_2 \cdot 2 + 11010010_2 = 0000111110010110_2$
2	1	$Q = 0000111110010110_2 \cdot 2 + 11010010_2 = 0001111111111110_2$
1	0	$Q = 0001111111111110_2 \cdot 2 = 0011111111111100_2$
0	1	$Q = 0011111111111100_2 \cdot 2 + 11010010_2 = 1000000011001010_2$

Example 10. *: Using Montgomery's method to multiply* $k = 0x10011101$ *by* $P = 11010010$.

i	k_i	Calculation steps	
7	1	$R_0 = R_0 + R_1 = 0000000011010010_2;$	$R_1 = 2 \cdot R_1 = 0000000110100100_2$
6	0	$R_1 = R_0 + R_1 = 0000001001110110_2;$	$R_0 = 2 \cdot R_0 = 0000000110100100_2$
5	0	$R_1 = R_0 + R_1 = 0000010000011010_2;$	$R_0 = 2 \cdot R_0 = 0000001101001000_2$
4	1	$R_0 = R_0 + R_1 = 0000011101100010_2;$	$R_1 = 2 \cdot R_1 = 0000100000110100_2$
3	1	$R_0 = R_0 + R_1 = 0000111110010110_2;$	$R_1 = 2 \cdot R_1 = 0001000001101000_2$
2	1	$R_0 = R_0 + R_1 = 0001111111111110_2;$	$R_1 = 2 \cdot R_1 = 0010000011010000_2$
1	0	$R_1 = R_0 + R_1 = 0100000011001110_2;$	$R_0 = 2 \cdot R_0 = 0011111111111100_2$
0	1	$R_0 = R_0 + R_1 = 1000000011001010_2;$	$R_1 = 2 \cdot R_1 = 1000000110011100_2$

Example 11. *: Using fixed-base comb method to multiply* $k = 0x10011101$ *by* $P = 11010010$.

i	t	Calculation steps
		$LUT[0] = 0000000000_2$
		$LUT[1] = 0011010010_2$
		$LUT[2] = 0110100100_2$
		$LUT[3] = 1001110110_2$
3	10_2	$Q = 0000000000000000_2 \cdot 2 + 0110100100_2 = 0000000110100100_2$
2	01_2	$Q = 0000000110100100_2 \cdot 2 + 0011010010_2 = 0000011101100010_2$
1	11_2	$Q = 0000011101100010_2 \cdot 2 + 1001110110_2 = 0001111111111110_2$
0	01_2	$Q = 0001111111111110_2 \cdot 2 + 0011010010_2 = 1000000011001010_2$

B Machine Code Examples

In this chapter, we give some examples for the machine code. The real machine code only contains the index of the used parameter. To keep the following examples understandable, we define the parameter directly in the code. Thus, the following examples are written in some kind of pseudo code. The actually used entry of the base-address-ROM is denoted by a USE statement. Then, for instance the third word of the first register given in this statement can be addressed by V0[2].

Copy a 32-bit value. To copy a 32-bit value to another register, the value first must be loaded into the buffer. In the next step it is added to the accumulator so that it finally can be written into another memory location.

```
USE R0, R1, R2, R3
RST    V0[0]
BUF
ADDAE 1
WRROR V1[0]
```

To add `R0[0]` to a value stored in the accumulator, the instruction in the first line must be replaced by a `NOP` to avoid the deletion of the value of the accumulator.

Adding two 32-bit values. The following code stores the sum of the least 32 bits of R0 and R1 into R2. Instead of parameter 1 at `ADD` another value q can be used to calculate `R2[0] = q` $\cdot$ `R0[0] + R1[0]`. This functionality for instance can be used in the reduction for Curve25519.

```
USE R0, R1, R2, R3
RST    V0[0]
BUF    V1[0]
ADD    1
WRROR V2[0]
```

To subtract, multiply, or do logical operations the instruction `ADD` must be replaced by `SUB`, `MUL`, `AND`, `OR`, or `XOR` respectively.

Using carry bit from the previous 32-bit addition. To add values larger than 32 bit, it is necessary to do several 32-bit additions sequentially. Thereby, the carry bit of the previous addition must be used. The following code can be used to calculate `R2[1] = R0[1] + R1[1] + c`, whereby `c` is the carry bit from the former example.

```
RSTL    V0[1]
BUF     V1[1]
ADDC    1
WRROR   V2[1]
```

May add a value depending on the carry bit. For the reduction it is necessary to add or subtract the prime (`P`) only if the carry bit is set after the last addition. Therefore the carry bit can be stored and used to enable or disable the adder. This is indicated by using 0 at `ADD` or `SUB`.

```
USE R0, R1, R2, P
SAVEC
RST     P[0]
BUF     V2[0]
SUBC    0
WRROR   V2[0]
```

Doing a loop from 255 to 0. For the point-multiplication it is necessary to perform a loop from 255 to 0. This can be done by reducing the value loop counter and jump back to the top of the loop until the value of the loop counter is zero.

```
  DECLC
Loop:
  //instructions in loop
  CMPD 0
  CALL Loop
  //instructions after loop
```

Calculate the sum of two multiplications. For 256-bit multiplications using the product-scanning method, the results of several multiplications must be accumulated. Therefore the value in the accumulator must be rotated between the single multiplications. The following example shows how to calculate `R2[1] = R0[0] · R1[1] + R0[1] · R1[0]`.

```
USE R0, R1, R2, R3
RSTL    V0[0]
BUF     V1[1]
MUL
ROL32   V0[1]
BUF     V1[0]
MUL
WR      V2[1]
```

256-bit product-scanning multiplication. Based on the code in the previous example the code for a full 256-bit multiplication {R3,R2} = R0 · R1 looks as follows. It has the same behavior as MUL256.

```
USE R0, R1, R2, R3
RST     V0[0]
BUF     V1[0]
MUL
WR      V2[0]

RSTL    V0[0]
BUF     V1[1]
MUL
ROL32   V0[1]
BUF     V1[0]
MUL
WR      V2[1]

RSTL    V0[0]
BUF     V1[2]
MUL
ROL32   V0[1]
BUF     V1[1]
MUL
ROL32   V0[2]
BUF     V1[0]
MUL
WR      V2[2]

RSTL    V0[0]
BUF     V1[3]
MUL
ROL32   V0[1]
BUF     V1[2]
MUL
ROL32   V0[2]
BUF     V1[1]
MUL
ROL32   V0[3]
BUF     V1[0]
MUL
WR      V2[3]

RSTL    V0[0]
BUF     V1[4]
MUL
ROL32   V0[1]
BUF     V1[3]
MUL
ROL32   V0[2]
BUF     V1[2]
MUL
ROL32   V0[3]
BUF     V1[1]
MUL
ROL32   V0[4]
BUF     V1[0]
MUL
WR      V2[4]

RSTL    V0[0]
BUF     V1[5]
MUL
ROL32   V0[1]
BUF     V1[4]
MUL
ROL32   V0[2]
BUF     V1[3]
MUL
ROL32   V0[3]
BUF     V1[2]
MUL
ROL32   V0[4]
BUF     V1[1]
MUL
ROL32   V0[5]
BUF     V1[0]
MUL
WR      V2[5]
```

```
RSTL    V0[0]
BUF     V1[6]
MUL
ROL32   V0[1]
BUF     V1[5]
MUL
ROL32   V0[2]
BUF     V1[4]
MUL
ROL32   V0[3]
BUF     V1[3]
MUL
ROL32   V0[4]
BUF     V1[2]
MUL
ROL32   V0[5]
BUF     V1[1]
MUL
ROL32   V0[6]
BUF     V1[0]
MUL
WR      V2[6]

RSTL    V0[0]
BUF     V1[7]
MUL
ROL32   V0[1]
BUF     V1[6]
MUL
ROL32   V0[2]
BUF     V1[5]
MUL
ROL32   V0[3]
BUF     V1[4]
MUL
ROL32   V0[4]
BUF     V1[3]
MUL
ROL32   V0[5]
BUF     V1[2]
MUL
ROL32   V0[6]
BUF     V1[1]
MUL
ROL32   V0[7]
BUF     V1[0]
MUL
WR      V2[7]

RSTL    V0[1]
BUF     V1[7]
MUL
ROL32   V0[2]
BUF     V1[6]
MUL
ROL32   V0[3]
BUF     V1[5]
MUL
ROL32   V0[4]
BUF     V1[4]
MUL
ROL32   V0[5]
BUF     V1[3]
MUL
ROL32   V0[6]
BUF     V1[2]
MUL
ROL32   V0[7]
BUF     V1[1]
MUL
WR      V3[0]

RSTL    V0[2]
BUF     V1[7]
MUL
ROL32   V0[3]
BUF     V1[6]
MUL
ROL32   V0[4]
BUF     V1[5]
MUL
ROL32   V0[5]
BUF     V1[4]
MUL
ROL32   V0[6]
BUF     V1[3]
```

```
MUL
ROL32   V0[7]
BUF     V1[2]
MUL
WR      V3[1]

RSTL    V0[3]
BUF     V1[7]
MUL
ROL32   V0[4]
BUF     V1[6]
MUL
ROL32   V0[5]
BUF     V1[5]
MUL
ROL32   V0[6]
BUF     V1[4]
MUL
ROL32   V0[7]
BUF     V1[3]
MUL
WR      V3[2]

RSTL    V0[4]
BUF     V1[7]
MUL
ROL32   V0[5]
BUF     V1[6]
MUL
ROL32   V0[6]
BUF     V1[5]
MUL
ROL32   V0[7]
BUF     V1[4]
MUL
WR      V3[3]

RSTL    V0[5]
BUF     V1[7]
MUL
ROL32   V0[6]
BUF     V1[6]
MUL
ROL32   V0[7]
BUF     V1[5]
MUL
WR      V3[4]

RSTL    V0[6]
BUF     V1[7]
MUL
ROL32   V0[7]
BUF     V1[6]
MUL
WR      V3[5]

RSTL    V0[7]
BUF     V1[7]
MUL
WR      V3[6]

ROR24
ROR8
WR      V3[7]
```

Curve25519 reduction after multiplication. After the multiplication the result can have up to 512 bits. Thus it must be reduced. Thereby, the reduction for Curve25519 is the most simplex one, because therefore only the upper half must be multiplied with 38 and added to the lower half. This must be repeated a second time to reduce the result of this addition, too. Finally, a single possible carry bit must be added to the first 32-bit word of the result. These three steps to calculate `R0 = reduce({R3,R2})` can be implemented as follows.

```
USE R0, R1, R2, R3
RST    V3[0]
BUF    V2[0]
ADD    38
WRROR  V3[0]

RSTL   V3[1]
ROR8
BUF    V2[1]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[1]

RSTL   V3[2]
ROR8
BUF    V2[2]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[2]

RSTL   V3[3]
ROR8
BUF    V2[3]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[3]

RSTL   V3[4]
ROR8
BUF    V2[4]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[4]

RSTL   V3[5]
ROR8
BUF    V2[5]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[5]

RSTL   V3[6]
ROR8
BUF    V2[6]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[6]

RSTL   V3[7]
ROR8
BUF    V2[7]
ADDAE  38
BUF
ADDAE  1
WRROR  V3[7]

————————— REDUCE2 —————————

ROR8
WRROR  V1[0]

RST    V0[0]
BUF    V3[0]
ADD    38
WRROR  V0[0]

RSTL   V3[1]
BUF
ADDAE  1
WRROR  V0[1]

RSTL   V3[2]
BUF
ADDAE  1
WRROR  V0[2]

RSTL   V3[3]
BUF
ADDAE  1
WRROR  V0[3]
```

```
RSTL   V3[4]
BUF
ADDAE  1
WRROR  V0[4]

RSTL   V3[5]
BUF
ADDAE  1
WRROR  V0[5]

RSTL   V3[6]
BUF
ADDAE  1
WRROR  V0[6]

RSTL   V3[7]
BUF
ADDAE  1
WRROR  V0[7]

————— REDUCE3 ————

ROR8
WRROR  V1[0]

RST    V0[0]
BUF    V0[0]
ADD    38
WRROR  V0[0]
```

NIST P256 reduction after multiplication. The reduction for NIST P256 is more complex than the reduction for Curve25519. In a first step several words must be added as defined in Section 2.2.3. The result of this addition can be positive and negative and can have up to two carry bits. These bits are stored in the overflow-buffer in a signed representation. Depending on them either the prime must be added or subtracted until the overflow is zero. Therefore REDUCE1 must be repeated three times. The complete reduction to calculate `R0 = reduce({R3,R2})` can be implemented as follows.

```
USE    R0, P, R2, R3
RST    V2[0]
BUF    V3[0]
ADD    1
NOP    V3[1]
BUF    V3[3]
ADDAE  1
BUF    V3[4]
SUBH   1
BUF    V3[5]
SUBH   1
BUF    V3[6]
SUBH   1
BUF
SUBH   1
WRROR  V2[0]

SETSGN V2[1]
ROR8
BUF    V3[1]
ADDAE  1
BUF    V3[2]
ADDAE  1
BUF    V3[4]
ADDAE  1
BUF    V3[5]
SUBH   1
BUF    V3[6]
SUBH   1
BUF    V3[7]
SUBH   1
BUF
SUBH   1
WRROR  V2[1]

SETSGN V2[2]
ROR8
BUF    V3[2]
ADDAE  1
BUF    V3[3]
```

```
ADDAE  1
BUF    V3[5]
ADDAE  1
BUF    V3[6]
SUBH   1
BUF    V3[7]
SUBH   1
BUF
SUBH   1
WRROR  V2[2]

SETSGN V2[3]
ROR8
BUF    V3[3]
ADDAE  1
BUF    V3[4]
ADDAE  2
BUF    V3[5]
ADDAE  2
BUF    V3[7]
ADDAE  1
BUF    V3[0]
SUBH   1
BUF    V3[1]
SUBH   1
BUF
SUBH   1
WRROR  V2[3]

SETSGN V2[4]
ROR8
BUF    V3[4]
ADDAE  1
BUF    V3[5]
ADDAE  2
BUF    V3[6]
ADDAE  2
BUF    V3[1]
ADDAE  1
BUF    V3[2]
SUBH   1
BUF
SUBH   1
WRROR  V2[4]
SETSGN V2[5]
ROR8
BUF    V3[5]
ADDAE  1
BUF    V3[6]
ADDAE  2
BUF    V3[7]
ADDAE  2
BUF    V3[2]
ADDAE  1
BUF    V3[3]
SUBH   1
BUF
SUBH   1
WRROR  V2[5]

SETSGN V2[6]
ROR8
BUF    V3[6]
ADDAE  1
BUF    V3[7]
ADDAE  2
BUF    V3[6]
ADDAE  2
BUF    V3[5]
ADDAE  1
BUF    V3[0]
ADDAE  1
BUF    V3[1]
SUBH   1
BUF
SUBH   1
WRROR  V2[6]

SETSGN V2[7]
ROR8
BUF    V3[7]
ADDAE  1
BUF    V3[7]
ADDAE  2
BUF    V3[0]
ADDAE  1
```

```
BUF     V3[2]
ADDAE   1
BUF     V3[3]
SUBH    1
BUF     V3[4]
SUBH    1
BUF     V3[5]
SUBH    1
BUF
SUBH    1
WRROR   V2[7]

SAVEOF
INCBA

————— REDUCE1 ————

RST     V1[0]
BUF     V3[0]
RED
WRROR   V0[0]

RSTL    V1[1]
BUF     V3[1]
REDC
WRROR   V0[1]

RSTL    V1[2]
BUF     V3[2]
REDC
WRROR   V0[2]

RSTL    V1[3]
BUF     V3[3]
REDC
WRROR   V0[3]

RSTL    V1[4]
BUF     V3[4]
REDC
WRROR   V0[4]

RSTL    V1[5]
BUF     V3[5]
REDC
WRROR   V0[5]

RSTL    V1[6]
BUF     V3[6]
REDC
WRROR   V0[6]

RSTL    V1[7]
BUF     V3[7]
REDC
WRROR   V0[7]

UPDOF
```

Brainpool P256r1 reduction after multiplication. After the multiplication at Brainpool P256r1 we do a Barrett reduction. First, we have to calculate a partial 256-bit product of the result of the multiplication {R3,R2} and a constant M. However, this constant has more than 256 bits. Thus, we must store it into two registers in the ROM. To reduce the number of switches between these two registers, MH contains also some values of ML.
ML = 0xA1C55B7EBB73ABA8322A7BF29B4F54A0FF6A2FA9B62AE6301180DD0C6B117C94,
MH = 0x00000001818C1131A1C55B7EBB73ABA8322A7BF29B4F54A0FF6A2FA9B62AE630.
In a second step we multiply this product with P and subtract it from the R2. For this code we need three entries in the address-base-ROM, thus the USE statement is longer. To switch between these entries the instructions INCBA and DECBA are used. The complete reduction to calculate R0 = reduce({R3,R2}) can be implemented as follows.

```
USE R1, ML, R2, R3;
    R1, MH, R2, R3;
    P,  R1, R2, R0
BUFALU V1[0]
RST    V3[7]
MUL
BUFALU V1[1]
ROL32  V3[6]
MUL
BUFALU V1[2]
ROL32  V3[5]
MUL
BUFALU V1[3]
ROL32  V3[4]
MUL
BUFALU V1[4]
ROL32  V3[3]
MUL
BUFALU V1[5]
ROL32  V3[2]
MUL
BUFALU V1[6]
ROL32  V3[1]
MUL
BUFALU V1[7]
ROL32  V3[0]
MUL
INCBA
BUFALU V1[6]
ROL32  V2[7]
MUL
BUFALU V1[7]
ROL32  V2[6]
MUL
DECBA
BUFALU V1[1]
RSTL   V3[7]
MUL
BUFALU V1[2]
ROL32  V3[6]
MUL
BUFALU V1[3]
ROL32  V3[5]
MUL
BUFALU V1[4]
ROL32  V3[4]
MUL
BUFALU V1[5]
ROL32  V3[3]
MUL
BUFALU V1[6]
ROL32  V3[2]
MUL
BUFALU V1[7]
ROL32  V3[1]
MUL
INCBA
BUFALU V1[6]
ROL32  V3[0]
MUL
BUFALU V1[7]
ROL32  V2[7]
MUL
BUFALU V1[0]
RSTL   V3[7]
MUL
BUFALU V1[1]
ROL32  V3[6]
MUL
BUFALU V1[2]
ROL32  V3[5]
MUL
BUFALU V1[3]
ROL32  V3[4]
MUL
BUFALU V1[4]
ROL32  V3[3]
MUL
BUFALU V1[5]
ROL32  V3[2]
MUL
BUFALU V1[6]
ROL32  V3[1]
MUL
BUFALU V1[7]
ROL32  V3[0]
```

```
MUL
WR      V0[0]

BUFALU  V1[1]
RSTL    V3[7]
MUL
BUFALU  V1[2]
ROL32   V3[6]
MUL
BUFALU  V1[3]
ROL32   V3[5]
MUL
BUFALU  V1[4]
ROL32   V3[4]
MUL
BUFALU  V1[5]
ROL32   V3[3]
MUL
BUFALU  V1[6]
ROL32   V3[2]
MUL
BUFALU  V1[7]
ROL32   V3[1]
MUL
WR      V0[1]

BUFALU  V1[2]
RSTL    V3[7]
MUL
BUFALU  V1[3]
ROL32   V3[6]
MUL
BUFALU  V1[4]
ROL32   V3[5]
MUL
BUFALU  V1[5]
ROL32   V3[4]
MUL
BUFALU  V1[6]
ROL32   V3[3]
MUL
BUFALU  V1[7]
ROL32   V3[2]
MUL
WR      V0[2]

BUFALU  V1[3]
RSTL    V3[7]
MUL
BUFALU  V1[4]
ROL32   V3[6]
MUL
BUFALU  V1[5]
ROL32   V3[5]
MUL
BUFALU  V1[6]
ROL32   V3[4]
MUL
BUFALU  V1[7]
ROL32   V3[3]
MUL
WR      V0[3]

BUFALU  V1[4]
RSTL    V3[7]
MUL
BUFALU  V1[5]
ROL32   V3[6]
MUL
BUFALU  V1[6]
ROL32   V3[5]
MUL
BUFALU  V1[7]
ROL32   V3[4]
MUL
WR      V0[4]

BUFALU  V1[5]
RSTL    V3[7]
MUL
BUFALU  V1[6]
ROL32   V3[6]
MUL
BUFALU  V1[7]
```

```
ROL32   V3[5]
MUL
WR      V0[5]

BUFALU  V1[6]
RSTL    V3[7]
MUL
BUFALU  V1[7]
ROL32   V3[6]
MUL
WR      V0[6]

BUFALU  V1[7]
RSTL    V3[7]
MUL
WR      V0[7]

INCBA

BUFALU  V0[0]
RST     V1[0]
MULNEG
NOP     V2[0]
BUF     V2[0]
ROL24
ADDAE   1
WRROR   V3[0]

BUFALU  V0[1]
SETSGN  V1[0]
MULNEG
BUFALU  V0[0]
ROL32   V1[1]
MULNEG
NOP     V2[1]
BUF     V2[1]
ROL24
ADDAE   1
WRROR   V3[1]

BUFALU  V0[2]
SETSGN  V1[0]
MULNEG
BUFALU  V0[1]
ROL32   V1[1]
MULNEG
BUFALU  V0[0]
ROL32   V1[2]
MULNEG
NOP     V2[2]
BUF     V2[2]
ROL24
ADDAE   1
WRROR   V3[2]

BUFALU  V0[3]
SETSGN  V1[0]
MULNEG
BUFALU  V0[2]
ROL32   V1[1]
MULNEG
BUFALU  V0[1]
ROL32   V1[2]
MULNEG
BUFALU  V0[0]
ROL32   V1[3]
MULNEG
NOP     V2[3]
BUF     V2[3]
ROL24
ADDAE   1
WRROR   V3[3]

BUFALU  V0[4]
SETSGN  V1[0]
MULNEG
BUFALU  V0[3]
ROL32   V1[1]
MULNEG
BUFALU  V0[2]
ROL32   V1[2]
MULNEG
BUFALU  V0[1]
ROL32   V1[3]
MULNEG
BUFALU  V0[0]
```

```
ROL32   V1[4]
MULNEG
NOP     V2[4]
BUF     V2[4]
ROL24
ADDAE   1
WRROR   V3[4]

BUFALU  V0[5]
SETSGN  V1[0]
MULNEG
BUFALU  V0[4]
ROL32   V1[1]
MULNEG
BUFALU  V0[3]
ROL32   V1[2]
MULNEG
BUFALU  V0[2]
ROL32   V1[3]
MULNEG
BUFALU  V0[1]
ROL32   V1[4]
MULNEG
BUFALU  V0[0]
ROL32   V1[5]
MULNEG
NOP     V2[5]
BUF     V2[5]
ROL24
ADDAE   1
WRROR   V3[5]

BUFALU  V0[6]
SETSGN  V1[0]
MULNEG
BUFALU  V0[5]
ROL32   V1[1]
MULNEG
BUFALU  V0[4]
ROL32   V1[2]
MULNEG
BUFALU  V0[3]
ROL32   V1[3]
MULNEG
BUFALU  V0[2]
ROL32   V1[4]
MULNEG
BUFALU  V0[1]
ROL32   V1[5]
MULNEG
BUFALU  V0[0]
ROL32   V1[6]
MULNEG
NOP     V2[6]
BUF     V2[6]
ROL24
ADDAE   1
WRROR   V3[6]

BUFALU  V0[7]
SETSGN  V1[0]
MULNEG
BUFALU  V0[6]
ROL32   V1[1]
MULNEG
BUFALU  V0[5]
ROL32   V1[2]
MULNEG
BUFALU  V0[4]
ROL32   V1[3]
MULNEG
BUFALU  V0[3]
ROL32   V1[4]
MULNEG
BUFALU  V0[2]
ROL32   V1[5]
MULNEG
BUFALU  V0[1]
ROL32   V1[6]
MULNEG
BUFALU  V0[0]
ROL32   V1[7]
MULNEG
NOP     V2[7]
BUF     V2[7]
ROL24
```

```
ADDAE   1                       WRROR   V3[7]
```

Curve25519 reduction after addition and subtraction. The reduction after additions and subtractions are shorter, because only a single bit must be reduced. Following code shows the reduction for Curve25519. Therefore the carry bit must be multiplied with 38. Following code shows the reduction `R0 = reduce({R1})` after a 256-bit addition.

```
USE     R0, R1, C, P            ADDC    1
SAVEC                           WRROR   V0[3]
RST     V2[0]  //0x26           RSTL    V1[4]
BUF     V1[0]                   ADDC    1
ADD     0                       WRROR   V0[4]
WRROR   V0[0]                   RSTL    V1[5]
RSTL    V2[1]                   ADDC    1
BUF     V1[1]                   WRROR   V0[5]
ADDC    1                       RSTL    V1[6]
WRROR   V0[1]                   ADDC    1
RSTL    V1[2]                   WRROR   V0[6]
ADDC    1                       RSTL    V1[7]
WRROR   V0[2]                   ADDC    1
RSTL    V1[3]                   WRROR   V0[7]
```

In the reduction after subtraction the carry bit must be reduced a second time.

```
USE     R0, R1, C, P            SUBIAC
SAVECI                          WRROR   V0[4]
RST     V2[1]  //0V25           RSTL    V1[5]
BUF     V1[0]                   BUF
SUBC    0                       SUBIAC
WRROR   V0[0]                   WRROR   V0[5]
RSTL    V1[1]                   RSTL    V1[6]
BUF                             BUF
SUBIAC                          SUBIAC
WRROR   V0[1]                   WRROR   V0[6]
RSTL    V1[2]                   RSTL    V1[7]
BUF                             BUF
SUBIAC                          SUBIAC
WRROR   V0[2]                   WRROR   V0[7]
RSTL    V1[3]
BUF                             SAVECX
SUBIAC                          RST     V2[2]
WRROR   V0[3]                   BUF     V0[0]
RSTL    V1[4]                   SUBC    0
BUF                             WRROR   V0[0]
```

NIST and Brainpool reduction after addition and subtraction. The reduction after addition works identically for the curves from NIST and Brainpool. Therefore the prime must be subtracted when the carry bit is set. For Brainpool this reduction must be repeated a second time, because the prime is much smaller.

```
USE     R0, R1, R2, P
SAVEC
BUFALU  V3[0]
RSTL    V1[0]
SUBC    0
WRROR   V0[0]
BUFALU  V3[1]
RSTL    V1[1]
SUBC    0
WRROR   V0[1]
BUFALU  V3[2]
RSTL    V1[2]
SUBC    0
WRROR   V0[2]
BUFALU  V3[3]
RSTL    V1[3]
SUBC    0
WRROR   V0[3]
BUFALU  V3[4]
RSTL    V1[4]
SUBC    0
WRROR   V0[4]
BUFALU  V3[5]
RSTL    V1[5]
SUBC    0
WRROR   V0[5]
BUFALU  V3[6]
RSTL    V1[6]
SUBC    0
WRROR   V0[6]
BUFALU  V3[7]
RSTL    V1[7]
SUBC    0
WRROR   V0[7]
```

The code for the reduction after a subtraction looks as follows. For the subtraction of both curves this code must be repeated a second time. Thereby the first instruction must be replaced by a `SAVECX`, since a second reduction is only necessary if the first one created a carry bit.

```
USE     R0, R1, R2, P
SAVECI
BUFALU  V3[0]
RSTL    V1[0]
ADD     0
WRROR   V0[0]
BUFALU  V3[1]
RSTL    V1[1]
ADDC    0
WRROR   V0[1]
BUFALU  V3[2]
RSTL    V1[2]
ADDC    0
WRROR   V0[2]
BUFALU  V3[3]
RSTL    V1[3]
ADDC    0
WRROR   V0[3]
BUFALU  V3[4]
RSTL    V1[4]
ADDC    0
WRROR   V0[4]
BUFALU  V3[5]
RSTL    V1[5]
ADDC    0
WRROR   V0[5]
BUFALU  V3[6]
RSTL    V1[6]
ADDC    0
WRROR   V0[6]
BUFALU  V3[7]
RSTL    V1[7]
ADDC    0
WRROR   V0[7]
```