KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Embedded Security for Car Telematics and Infotainment

Anthony Coyette

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: elektrotechniek, optie Geïntegreerde elektronica

Promotor:
Prof. Dr. Ir. I. Verbauwhede

Assessoren:
Prof. Dr. Ir. B. Preneel
Prof. Dr. Ir. F.-X. Standaert

Begeleiders:
Dr. Ir. B. Gierlichs
Ir. J. Balasch

Academiejaar 2011 – 2012

Preface

I would like to thank my assistants Benedikt and Josep for their regular support and advice. Without them I would still be thinking the world is secure.

Another team who deserves my thanks is my family. I think about my Dad and his "explanations", my mother and her boundless care. I should maybe also give a word about my brother and my sister, but I won't.

Finally, I would like to thank Caroline who, by many ways, brought me here. I love you.

Anthony Coyette

Contents

Abstract

Elliptic Curve Cryptography appeared in 1985 and since became an increasingly important crypto-system in public-key cryptography. Numerous articles have come to improve the field. One of these milestones is Edwards Curves which appeared in 2007 and proposed advantageous properties against SPA attacks.

This thesis proposes the design, implementation and comparison of several small co-processors in hardware acceleration work flow. We start from scratch the software implementation of a scalar multiplication of Edwards Curves defined over $\mathbb{F}_{P_{192}}$ in projective coordinates. In a first phase, 8-bit, 16-bit, 32-bit multipliers interfaced on the parallel port of a 8051 are successively developed. Then, the system is improved again by the implementation of a memory mapped 192-bit broadcast multiplier including a modular reduction. These five hardware configurations combined to an assembly optimization finally furnish six versions which are compared on four axes : time, energy consumption, area and security.

On the one hand the side-channel attack still exposes a SPA-weakness in the implementation of all the version. On the other hand, measures show the expected results that the 192-bit multiplier provides a faster and lower power system at the expense of multiplying the silicon area by three.

Key words : Edwards Curves, Prime Field, Co-Design, Security.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

AL	Artihmetic Layer
DPA	Differential Power Analysis
ECC	Elliptic Curve Cryptography
EL	Elliptic Layer
FSM	Finite State Machine
SCA	Side-Channel Attack
SPA	Simple Power Analysis

Symbols

E	Elliptic Curve
$\mathbb{F}_{P_{192}}$	Prime field with NIST-P192 as prime number
K	Field

Chapter 1

Introduction

1.1 Motivation

In the automotive world, manufacturers provide their consumers with the newest technology: GPS, DVD player,etc. With that technology emergence inside the car, the classic car-radio theft became a more lucrative activity. Such that manufacturers have now to defend their costumers against these thefts. In that area, cryptography has a role to play. Instead of a mechanical protection, a crypto-system scheme can verify the matching between an entertainment unit and the car when this one tries to connect itself on the CAN bus. This would prevent the use of a device out of an authorized car and make the theft device useless.

Cryptography proposes more than one solution to this task. A public-key authentication system based on Elliptic Curve Cryptography is one of the possibilities. The idea is to force the device to follow a protocol to prove its identity on its power on. During the design of such a crypto-system, the manufacturer has to face a lot of choices. First, concerning the mathematical security, several types of elliptic curves exist and are defined over different finite fields with different key sizes. Then, during the implementation, the manufacturer has to face more options. While software implementations are cheap and flexible but slow, specific ASICs are fast, but fixed and expensive. The co-design proposes an hardware/software trade-off and appears like a good opportunity but increases the design parameters again.

Since so many options are offered, we think that a comparative study would help to enlighten those choices. Of course, the field is too vast to be entirely covered. We propose then to partially cover the co-design trade-off in a fixed context.

1.2 Contribution

In this work, we started from scratch the software implementation of a scalar multiplication on Edwards curves in projective coordinates. We made that implementation run on a 8051 processor synthesized on a Virtex 2 FPGA. On the basis of that pure software implementation, we did a analysis in order to locate the bottleneck. This analysis made us implement several hardware multipliers of different sizes. We

interfaced them all with the 8051 to end up with 5 different co-designs. We analyzed then the performances of the six versions according to four parameters: time, energy consumption, area and security.

1.3 Structure

This thesis is structured around eight chapters. In order to familiarize the reader with the field of Elliptic Curve Cryptography and their implementation, Chapter 2 overviews the theoretical concepts, the specificities of their implementation and the security issues. Chapter 3 lays the basis of our implementation, the main choices that define the design of our work. Then, the software implementation is described in Chapter 4. Chapter 5 explains the co-design work flow applied to the software version in order to improve the performances. On the basis of the system implemented, Chapter 6 covers the security analysis that we made. And finally, Chapter 7 handles the comparison of the 6 versions from the four points of view previously stated: time, area, consumption and security. In Chapter 8, we conclude and give perspectives for a future work.

Chapter 2

Elliptic Curve Cryptography

Elliptic curves are mathematical objects that have been studied in mathematics for a long time, some roots of this study takes place BC with Diophantus[30]. In this work, a more specific look will be taken on elliptic curves when they are used as the basis of a crypto-system. The idea of using these mathematical constructions in cryptography appeared around 1985 and is attributed to Miller and Kolbitz with their respective papers [17] [22].

In this chapter, a basic introduction presents the elliptic curves and the operations that are defined on them. Afterwards, prime fields are introduced to explain how elliptic curves are used in cryptography. Then, algorithms are given to illustrate different coordinate systems, and types of curves. Next, we outline the main advantages of Elliptic Curve Cryptography. Finally, the last section will cover the attacks that threaten implementations.

2.1 Elliptic Curves

As stated in Hankerson's book [10], an Elliptic Curve E over a field K is defined by an equation

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

where $a_1, a_2, a_3, a_4, a_6 \in K$. [1] This equation – also called the Weierstrass form – represents a general description of elliptic curves. More restrictive assumptions on the coefficients (for example, $a_1 = 1$) lead to special curves which exhibits some properties leading to the optimization of their computation. In the following pages, a few types of curves will be studied depending on the context and the goal. To begin, the simplified Weierstrass form in affine coordinates on $\mathbb{R}$ will be used for their readability. Afterwards, the same curves but in projective coordinates will be handled and finally Edwards curves and their properties will be presented. Simultaneously, finite fields will be introduced to replace $\mathbb{R}$.

[1] A condition exists on the value of the a_i coefficients. But this detail is not covered here.

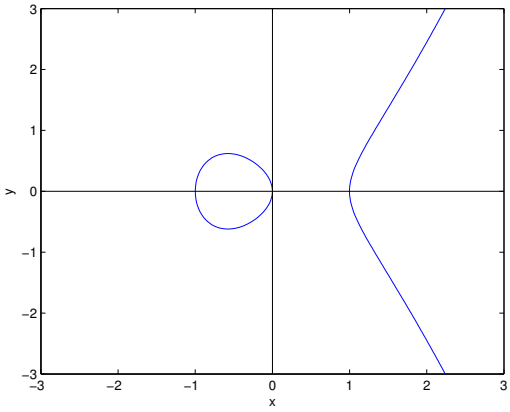

Figure 2.1: $y^2 = x^3 - x$.

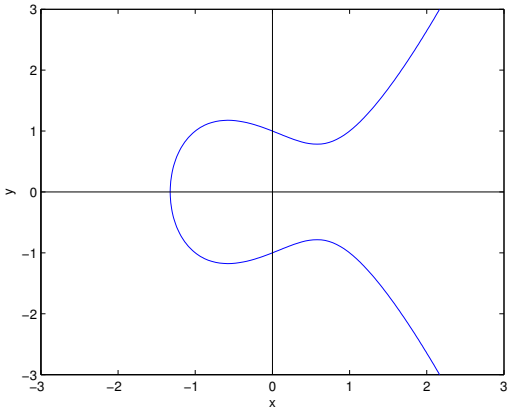

Figure 2.2: $y^2 = x^3 - x + 1$.

Besides these three types of curves and the two fields cited, in order to present the basic ideas and tools around elliptic curves, the $\mathbb{R}$ field will be used for its intuitive representation. Later on, other fields will be presented to the reader but for the moment,the reader should keep in mind that cryptography does not employ the $\mathbb{R}$ field.

Figures 2.1 and 2.2 represent graphically two elliptic curves. They are the two usual pictures shown as an introduction to give to the reader an intuition what the EC look like.

2.1.1 Operations on Elliptic Curves

Without entering too much into the details, the curves presented in the previous section allow the definition of a few mathematical operations on their elements. Stricto

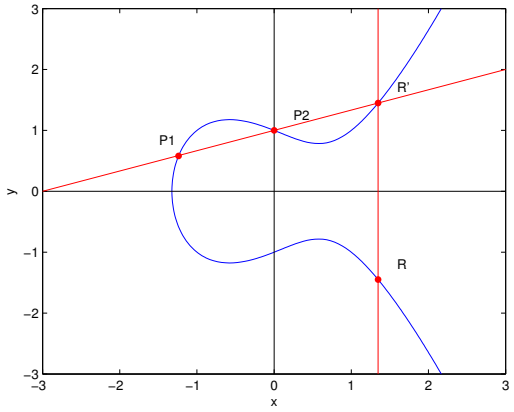

FIGURE 2.3: Point Addition.

sensu, there exists only one operation, the Group Law: the addition of two points or Point Addition. The name of the group law comes from the mathematical formulation. One says that the set of points on an elliptic curve plus the point at infinity associated with the Group Law form an abelian group, which is a mathematical construction.

- Addition $+ : E \times E \to E$

From now on, mathematical details will be left apart and the focus will be put on the computational aspects. For example, in a larger sense, besides the addition two other operations are constantly used and cited even if they are not defined in the Group. But as it will be seen in the next section, these operations are based on the Point Addition.

- Doubling $D : E \to E$
- Scalar multiplication $* : K \times E \to E$

Addition. The addition of two points on the curve is introduced here in a graphical way, to help the reader get an intuition about the operation. In a second phase, the mathematical expressions associated to these graphical constructions will be given.

As presented in Figure 2.3, the addition of two different points P_1 and P_2 on the curve, with the additional condition that $P_1 \neq -P2$, consists first in the computation of the intersection between the curve and the straight line defined by the two points. And the final result is the opposite of the intermediate point - its image by an orthogonal symmetry of axis X. Mathematically, let us define :

$P_1 = (x_1, y_1)$
$P_2 = (x_2, y_2)$

$$R = P_1 + P_2 = (x_3, y_3) = \left(\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2, \left(\frac{y_2 - y_1}{x_2 - x_1} \right)(x_1 - x_3) - y_1 \right).$$

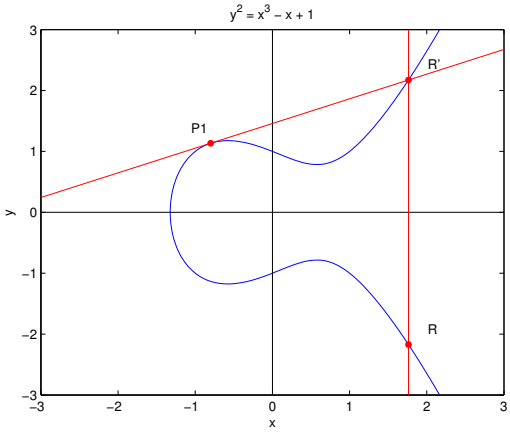

Figure 2.4: Point Doubling.

These expressions can be computed with a minimum number of operations according to the procedure given in Appendix A [2]: 3M+6S+1I.

An important point should be noted from now on. In this work, the same algorithm is applied to do a multiplication or a squaring and so no distinction is made in the amount of computation as it is usually done. Effectively, the redundancy of information in the case of the squaring allows to increase the performance of this operation in comparison with the multiplication. Nevertheless, according to the purpose of this work, the implementation of the multiplication intends to vary a lot. Hence, it was decided to make no distinction in order to not double the work.

Doubling. Doubling a point is simply the addition of twice the same point. But a quick look at the formulae stated in the previous section shows that the same equations can not be used. Effectively, with twice the same coordinates, the previous expressions would make a zero appear at the denominator - which is a not defined. The basic idea of the straight line defined by two points must be rethought in the extreme case of two points getting closer and closer. The same idea appears in the definition of the derivative of a curve and that is what is used here. As presented in Figure 2.4, doubling a point on the curve consists first in the computation of the tangent line to the curve at this point. Then, an intermediate point is computed as the intersection between the tangent line and the curve. The final result is the opposite of the intermediate point - its image by a orthogonal symmetry of axis X.

The mathematical translation of the process gives as result :

$$P = (x_1, y_1)$$

$$R = -R' = 2P = (x_3, y_3) = \left(\left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1, \left(\frac{3x_1^2 + a}{2y_1} \right)(x_1 - x_3) - y \right).$$

[2] M : Multiplication, S : Subtraction, I : Inversion.

From the computational point of view, the algorithm corresponding to these expressions – in Appendix A – needs [3]: 3M + 3m + 1A + 3S + 1I.

Scalar Multiplication. The scalar multiplication, as stated before, consists in the multiplication between one point on the curve and an element from the field. Basically, to multiply by k means that the point P is added k times with itself. In other words, for $k \in K$ and $P \in E$:

$$k * P = P + P + ... + P\,.$$

The multiplication can be constructed on the basis of the operations seen earlier. For example, as stated just above, the multiplication could be seen as one doubling (P+P) and k-1 Additions (2P+P,3P+P,...). Of course, it would make the computation really inefficient considering the fact that big numbers are manipulated[4]. A classic way to tackle this problem is to apply the square-and-multiply algorithm. Basically this approach starts from the rewriting of the scalar multiplication in the form :

$$k * P = P + 2 * (P + (P + 2 * (P + (...))))\,.$$

This reformulation leads directly to Algorithm 1 which computes the wanted result. The kernel of the idea relies on the binary representation of the number k

$$k = (k_{n-1}, k_{n-2}, ..., k_1, k_0)\,.$$

But even though this algorithm is mathematically well designed, we usually do not use it in Elliptic Curve Cryptography because its structure is weak against the several side-channel attacks that we cover later on [29]. Instead, algorithms like the Montgomery Ladder – illustrated in Algorithm 2 – are used. At the expense of efficiency, they offer a better security.

2.1.2 Finite Field

As previously said, in cryptography one does not use $\mathbb{R}$ as the arithmetic field upon which the elliptic curves are built. Instead, fields with a finite number of elements – also called finite fields – are used. According to [10], three main classes exist: binary fields, prime fields and extension fields. In this work, we make use of the prime fields such that it is the only one covered.

Prime fields. Mathematically speaking: Let p be a prime number in $\mathbb{N}$, a prime field $\mathbb{F}_p$ is defined as :

$$\mathbb{F}_p = \{x \in \mathbb{Z} : x < p\}\,.$$

[3] m: simplified multiplication. A multiplication by 2 is a shift with a modular verification. And a multiplication by 3 allows short-cuts.

[4] In the following, the prime fields will be presented. This work uses numbers up to the order of 2^{192}.

Algorithm 1 Left-to-Right Scalar Multiplication.

Input: $P \in E$ and $k \in K$
Output: $R = k * P$

```
j=max_i{k_i = 1}
j ← j − 1
R ← P
for i from j to 0 do
    R ← 2R
    if k_j = 1 then
        R ← R + P
    end if
end for
```

Algorithm 2 Montgomery Ladder.

Input: $P \in E$ and $k \in K$
Output: $R_0 = k * P$

```
j=max_i{k_i = 1}
j ← j − 1
R_0 ← 0
R_1 ← P
for i from j to 0 do
    if k_j = 0 then
        R_0 ← R_0 + R_1
        R_1 ← 2R_0
    else
        R_0 ← R_0 + R_1
        R_1 = 2R_1
    end if
end for
```

In other words, the field $\mathbb{N}$ contains all the naturals from 0 to p-1. For what concerns the operations on this field, four binary operations are accepted in a large sense. Indeed, the +,-,*,/ can be sum up to +,* if the subtraction is defined as the addition of the opposite, and the division is defined by the multiplication by the inverse. Otherwise, they are basically the same as the operations defined on $\mathbb{Z}$ but the result undergoes a modulo to keep the range [0;p-1]. As an example, some operations on $\mathbb{F}_7$ are presented below. Operations on $\mathbb{F}_7 = \{0,1,2,3,4,5,6\}$:

Addition : $(4 + 5)_{\mathbb{F}_7} = 4 + 5 \bmod 7 = 2$

Subtraction : $(2 - 4)_{\mathbb{F}_7} = 2 - 4 \bmod 7 = 5$

Multiplication : $(4 * 5)_{\mathbb{F}_7} = 4 * 5 \bmod 7 = 6$

Division : $(4/2)_{\mathbb{F}_7} = 4 * 4 \bmod 7 = 2$

To be complete and being able to divide, an algorithm to find the inverse of an element must be implemented. [10] proposes several ones: the extended Euclidian algorithm (not efficient), the binary inversion and the Montgomery inversion. The binary inversion is designated in order to get a reasonable efficiency without entering the Montgomery's complexity with its change of representation and special multiplication.

Algorithm 3 Binary Inversion in $\mathbb{F}_p$.

Input: $a \in \mathbb{F}_p$
Output: a^{-1} mod p
  $u \leftarrow a$, $v \leftarrow p$
  $x_1 \leftarrow 1$, $x_2 \leftarrow 0$
  while $u \neq 0$ and $v \neq 0$ **do**
    while u is even **do**
      $u \leftarrow \frac{u}{2}$
      if x_1 is even **then**
        $x_1 \leftarrow \frac{x_1}{2}$
      else
        $x_1 \leftarrow \frac{x_1+p}{2}$
      end if
    end while
    while v is even **do**
      $v \leftarrow \frac{v}{2}$
      if x_2 is even **then**
        $x_2 \leftarrow \frac{x_2}{2}$
      else
        $x_2 \leftarrow \frac{x_2+p}{2}$
      end if
    end while
    if $u \geq v$ **then**
      $u \leftarrow u - v$
      $x_1 \leftarrow x_1 - x_2$
    else
      $v \leftarrow v - u$
      $x_2 \leftarrow x_2 - x_1$
    end if
    if u=1 **then**
      return x_1 mod p
    else
      return x_2 mod p
    end if
  end while

2.2 Projective coordinates

Taking a look at the two previous sections, a big problem appears and threatens the performance of any program using the up to now explained materials. The basic formulae for point addition and doubling make use of a division. This operation in a prime field supposes to first find the inverse of the divisor and then multiply it by the numerator. The first step uses Algorithm 3 stated before which is demanding and makes an implementation slow. A manner of avoiding this inverse-finding exploits the rewriting of the expressions in a different coordinate system.

The X-Y system used heretofore is called an Affine Coordinates System. The proposed change is to go from this system to another equivalent representation. Several representations exist but the key point is that these new spaces work with representative points. A point in affine coordinates has several representative points in projective coordinates. It should be noted now that this detail gives an advantage against the DPA because it allows to compute an operation with different points but obtain an equivalent result if the projective points are representative of the same affine points. Here are some examples of such projective spaces :

Projective coordinates : $(x, y) \sim (X, Y, Z)$ if $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$

Jacobian coordinates : $(x, y) \sim (X, Y, Z)$ if $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$

Chudnovsky coordinates : $(x, y) \sim (X, Y, Z, Z^2, Z^3)$ if $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$.

In the rest of this section, the simplified Weierstrass curve will be kept and the used projective coordinates are the Jacobian ones. In a first phase, the equations to convert the affine coordinates to projective coordinates are described and a short study of the effects on the point addition and doubling equation is given. It should be noted that there are a lot of existing projective coordinates systems. They have their own properties that lead to special optimizations in combination with defined curves.

2.3 Edwards Curves

As stated in the general introduction to elliptic curves, a lot of different types of curves exist. Edwards curves are the set of curves described by the equation :

$$x^2 + y^2 = c^2(1 - dx^2y^2)$$

where c,d $\in \mathbb{F}_p$. Making the simplification c=1 is often done and it will be so in the following. Hence, the Edwards curves equation will be written These curves , discovered in 2007 by the mathematician Harlod M. Edwards and published in his paper [6], exhibit the special property concerning its addition. By taking a close look

at the equations:

$$P_1 = (x_1, y_1)$$
$$P_2 = (x_2, y_2)$$
$$R = P_1 + P_2 = \left(\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right)$$

we notice that no zero will appear at the denominator if we try to add twice the same point. That property contrasts with what happens in the simplified Weierstrass curves and allows the use of a unique instruction to add two points or double one point. This is called the unified operation property. From an efficiency point of view, the computation of the expression – which follow Bernstein's indications [1][2] and can be found in Appendix A – needs 12M+4S+3A which does not propose a real improvement in comparison with the simplified Weierstrass Curves. In fact, the benefit for adopting Edwards curves is its property of unified operation. From a security point of view, the fact that the Addition and Doubling operations are the same confers an advantage against SPA. Since their implementation and execution will be the same when the program runs, it makes it possible to hide from the user whether the addition or the doubling is being ran. A longer explanation will be given in the section about physical attacks.

2.4 Protocol

A protocol is a flow of defined actions designed to ensure a defined goal. In the case of this master thesis, the aim is the authentication of an entity. We picked up the Schnorr protocol for its simplicity of concept. It works basically as a sigma protocol with three steps: commitment, challenge, answer as illustrated in Figure 2.5.

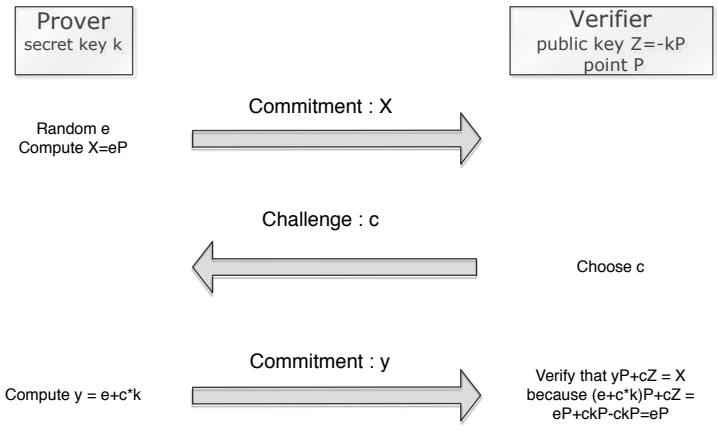

FIGURE 2.5: Schnorr Protocol for ECC.

2.5 Advantages of ECC

There are several factors which made this thesis implement elliptic curves based security systems. A list of points have to be raised in order to legitimate this choice. Firstly, the preference for a public key crypto-system rather than for a symmetric key is covered. Then, we explain the dominance of ECC over RSA.

2.5.1 Public Key vs Symmetric Key

By opting for ECC, we made the choice of public-key over symmetric-key cryptography. This is coherent regarding the context of this work and the comparison between these two options given in [21] and [14]. We present here a short summary of the relevant points for our system. One the one hand, a public-key scheme has the advantage of handling shorter keys and offering algorithms with higher throughputs. However, besides less good performances, public-key cryptography offers advantage that made us choose it. In a network involving several entities – such as the CAN bus of a car where a lot of chips would have to send encrypted data – the management of the keys for symmetric-key cryptography becomes tricky. First, the key distribution of a public-key can be done easily in comparison with the symmetric-key scheme. Furthermore, with in the symmetric case, the key should be changed often in comparison of public-keys that can be kept for years. Finally, the key distribution of the public-key is easier since no secret has to be known in advance.

2.5.2 Comparison to RSA

Table 2.1: NIST recommended key sizes (in bits).

Symmetric Key Size	RSA Key Size	ECC Key Size
80	1024	160
112	2038	224
128	3072	256
192	7680	384
256	15360	521

As RSA, ECC is a cryptographic tool which is not secured in the theoretical sense because there exist algorithm to recover the key. These two systems are rather based on the computational infeasibility of finding the key. Since their appearance, ECC is predicted as the successor of RSA due to its better efficiency. NIST has published a table of comparison between the key length of ECC and RSA which ensure the same level of security. The ECRYPT II report on keysizes [12] gives more detailed information on the subject. Table 2.1 exhibits an evident advantage of ECC over RSA on the plan of the key size. But this point also implies the involved amount of computation in order to encrypt or decrypt something. The longer is the key, the

	Active	**Passive**
Invasive	Laser Fault Injection	Bus probing
Non-Invasive	Clock Tampering	Power Analysis

TABLE 2.2: Classification of Physical Attacks : Examples.

longer is the time. For some applications as in the automotive world, the length of the key plays an important role since this amount of information has to be send on the CAN bus of the car and forms the bottleneck of the authentication process. As the CAN bus is a message-based system, the information sent should be kept as short as possible.

2.6 Physical Attacks

The final concern on which a focus is put in this work is hardware security. The previous sections related the basic information over elliptic curves, and the way their arithmetic works in order to give an intuition about the provided security. Afterwards, the NIST equivalence gave an estimate of the amount of security. But in real implementations, even a perfectly secure algorithm as the one-time-pad could be broken by an attack based on information from the physical world. The exploited leakage can originate from very different sources [16] :

- Timing
- Power consumption
- Electromagnetic Emission
- Fault injection (laser, clock, ...)

These few examples form a non-exhaustive list and more channels could be cited, but this gives a sufficient idea of the possibilities of attack. In fact, all the possible attacks can be classed following two characteristics [31] :

Invasive/Non-invasive Whether the package of the chip has to be opened or not.

Passive/Active Whether if the attacks tampers with the behavior of the chip or only listens to information.

Table 2.2 illustrates that classification with four cases.However, in spite of the multitude of possible attacks, this work will only study the resistance against the non-invasive and passive Power Analysis Attacks.

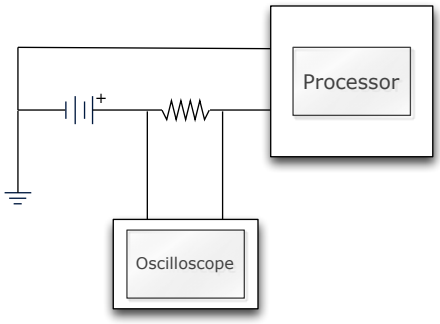

Figure 2.6: Setup for Power Analysis.

2.6.1 Power Analysis Attacks

As their name implies, those techniques base their approach on the study of the power consumed by the chip on which a crypto-system is ran. This information is recorded as traces and can be obtained thanks to an oscilloscope and a small resistor connected in series between the power supply and the chip as illustrated in Figure 2.6. A trace is basically a vector of values registering the values of the power consumption in the time. [15] This information can be utilized in different ways. In this work, we consider the Simple Power Analysis – or SPA – and the Differential Power Analysis – DPA – for reason of time. Besides that, we also chose these two attacks because they confer a reasonable power to the attacker regarding our application of the Schnorr Protocol in the automotive world. The attacker can possibly listen to computations of scalar multiplications with partial information on the input. However, the case of the template attack – where the attacker is allowed to perform any chosen computation to profile the hardware – seems too strong in this context.

2.6.2 Simple Power Analysis

This first technique is quite simple in the concept. On the trace of an execution of the algorithm under test, a visual analysis is applied to try to distinguish the internal execution of the processor. We say the leakage – if it exists – is dependent on the instruction.

The reader should note that there exist also more sophisticated techniques of SPA. For example, the template-based SPA attack used in [20] would lead to a more rigorous analysis. But in the cadre of this work, the SPA will be kept at the level of the visual inspection.

2.6.3 Differential Power Analysis

The Differential Power Analysis proposes a method able to crack noisy implementation thanks to statistical tools. In comparison with the SPA which relies on the

difference of instruction, the DPA relies on the data handled. We do not cover the subject exhaustively but following the scheme of [19], the idea of these attacks can be summarized in five steps. Suppose we want to attack an algorithm F which executes a computation on basis of a known information P and an unknown information k: F(P,k). The scheme of the attack is :

1. **Choose a point of attack :** We target an internal value which relies on the information k and P. For example, we take a bit.

2. **Measures power traces :** We take traces of the power consumption for known values P_i.

3. **Calculate hypothetical internal value :** We make a partial hypothesis on the secret key k. For example, we make an hypothesis on the two least significant bits which gives a spaces of four hypotheses. One of these hypotheses is the good one, the goal from now on is to find which one. On the basis of the known values P_i and for each of the four hypotheses, we compute the value of the targeted bit.

4. **Map the hypothetical internal values :** According to a chosen leakage model, we extrapolate the leakage of the taken traces at the target bit. For example, we take the Hamming Weight model. So for each hypothesis, we have the hypothetical value of the bit from last step for each trace and we use this information in the Hamming weight model to give the hypothetical leakage at that point.

5. **Compare hypothetical model with reality :** Following a statistical method, verify the hypothetical leakage. For example, we decide to add the traces where the Hamming Weight is one and subtract the traces with a Hamming weight zero.

   If the hypothesis – and the leakage model – is coherent with the reality, the additions and subtractions will have a constructive effect and let appear a peak in the trace resulting from the addition and subtraction of all the taken traces. If the hypothesis is false, the resulting trace will not have a peak since the additions/subtractions have a destructive effect.

2.7 Conclusion

In this chapter, we gave all the basic information concerning the Elliptic Curve Cryptography. The reader should now be able to get to the details of this implementation and the main issue around it.

Chapter 3

Design Choices

A lot of theoretical information has been presented in the previous chapter. In this chapter, we set the parameters of our implementation. In such a work, three main aspects are involved and we will explain each choice in the consecrated sections.

The first section describes our choice for the arithmetic layer. The second cover the used elliptic curves. And in the last section, we adopt the processor our implementation will run on.

3.1 Arithmetic Field

As arithmetic field, we chose to use the prime field $\mathbb{F}_{NIST-P192}$. This simple statement involves in fact 2 different choices for the design :

- The level of security: 192 bits of key

- The prime number NIST-P192 which we will call P_{192}.

192 bits of key. The choice follows the recommendations done on the key size in the ECRYPT II report [12]. For a ten-years protection[1] – or the legacy standard level – a security of 96 bits is advanced. Regarding the Table 2.1 given in Chapter 2, an 192-bit key ECC implementation fits the desired security.

Prime number: NIST-P192. The NIST-P_{192} ,or $2^{192}-2^{64}-1$ to give it explicitly, is a standard prime number recommended in the FIPS-186-3[25] publication of the NIST. The major advantage for using P_{192} resides in its simplified modular reduction for (big) integers. [2]. The following lines explain this property. Let us take A $\in \mathbb{N}$, with $A < P_{192}^2$. This can be written as

$$A = A_5 \times 2^{320} + A_4 \times 2^{256} + A_3 \times 2^{192} + A_2 \times 2^{128} + A_1 \times 2^{64} + A_0$$

[1] Ten years seems to be in line with the life-cycle of a car.

[2] To be used in the modular multiplication

where the A_i are integers of 64 bits. Hence, this equation can also be written as :

$$A = (A_5, A_4, A_3, A_2, A_1, 1_0)$$

To operate the reduction, four new numbers need to be define and to be added afterwards. First let's define these four integers in the two representations that were just presented :

$$\begin{array}{lll} S_0 = A_2 \times 2^{128} & +A_1 \times 2^{64} + A_0 & = (A_2, A_1, A_0) \\ S_1 = & A_3 \times 2^{64} + A_3 & = (\,0\,,\, A_3,\, A_3) \\ S_2 = A_4 \times 2^{128} & +A_4 \times 2^{64} & = (\,A_4,\, A_4,\, 0\,) \\ S_3 = A_5 \times 2^{128} & +A_5 \times 2^{64} + A_5 & = (A_5, A_5, A_5) \end{array}$$

And the final result equals :

$$\text{A'} = \text{A mod } P_{192} = S_0 + S_1 + S_2 + S_3 \text{ mod } P_{192}$$

3.2 Elliptic Curve

Edwards curve were chosen due to the advantage that can be taken in the SPA-resistance thanks to their property of unified operation. Since the addition and the doubling are the same, a SPA analysis will not be able to distinguish them. It allows to bypass the tricks usually used in the scalar multiplication such as the Montgomery Ladder and get a gain performance.

3.3 Processor

As explained before, the co-design done in this work relies on two main parts: a co-processor and an hardware acceleration. In this paragraph, the main criterion chosen to drive the selection of the soft-core are presented. The hardware design will be covered later on.

In order to operate an enlightened choice the task was divided in two steps. The first one was to decide wether the processor would consist in a 8-bit or a 32-bit architecture. The second one was to choose the soft-core itself within the adopted architecture.

3.3.1 Architecture

Within this decisional process, the cases of the 8-bit and 32-bit architecture were considered. While the 8-bit architecture proposes a popular low cost choice, the 32-bit provides a computational power way more important at the expense of power consumption and silicon area. Besides that, the throughput of information towards and from the memory within 32-bit architectures is also four times higher.

However, pure software implementations on 8-bit architecture exhibits already reasonable – but not sufficient since 800 ms is still too long for some applications – figures in the literature [18]. This confirms the trend where 8-bit are used with hardware acceleration and 32-bit architectures rely on their computational power.

Since we aim a co-design comparison, we opted for a 8-bit architecture.

3.3.2 Comparison

Table 3.1 gives the summary of the criteria we applied in order to choose our processor. From the four candidates, the LatticeMico8 and the 8051 seemed really interesting but we choose 8051 for two reasons. First, this standard processor really well documented. And secondly, the 8051 processor was already used in some comparable work about the implementation of ECC [18].

TABLE 3.1: Comparison of 8-bits processor.

Processor	Size (slices)	Communication	Open Source	Multiplier	Documentation
LatticeMico8	< 200	opt. UART, SPI, I2C	Yes	No	++
PacoBlaze	~ 200	in/out	Yes	No	-
PicoBlaze	<200	in/out	No	No	+
8051	600	serial, parallel	Yes	Yes	+

3.3.3 8051 processor

As a result of the comparison made to choose our hardware, the 8051 appeared to be the most interesting choice. A short description of its basic features is given in the next chapter.

The 8051, also called MCS-51, is a standard 8-bit processor developed by Intel in the 80's. This processor comes with a lot of convenient features as presented below :

- ◇ 4 parallel ports
- ◇ a serial port
- ◇ 2 timers

The Oregano IP [26] used in this thesis proposes a larger amount of parallel and serial ports for the communication since the author made the number of connections parametric. Nevertheless, in order to keep this work in a general environment, the initial number of parallel and serial ports of the 8051 is kept. The clock speed was arbitrarily set at a value of 12,5 MHz. "Maximum speed"

The 8051 basically works with 3 different memories. Harvard architecture: the code memory is separated from the data memory. Besides that fact, the processor also provides the control of two different RAM memory block: the internal and

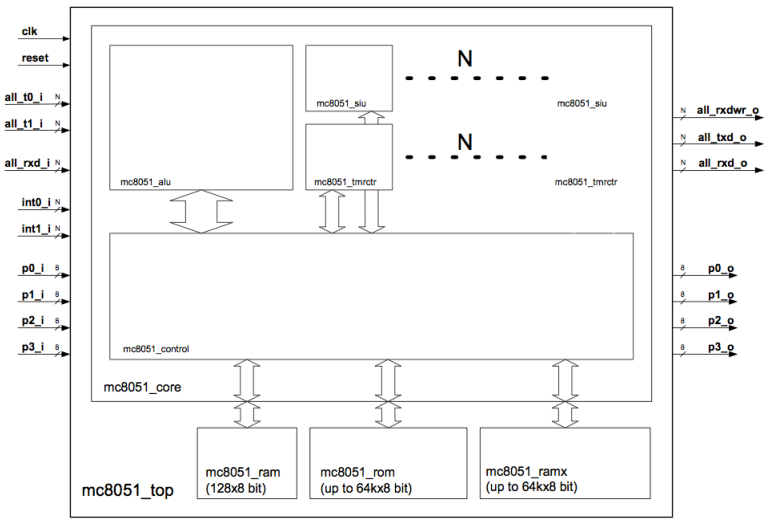

Figure 3.1: 8051 Architecture.

the external memories.The former one has a size of 256 bytes (8 bit-address) and is compulsory since it includes : - the Special Function Registers (SFR), used to configured the device : - the internal registers, as the accumulators, the usual R0...R7 - some variable from the code

The latter one can handle up to 65536 bytes (16 bit-address) and is not compulsory if all the variables can be handled inside the internal memory. A more detailed discussion about the different kinds of memory will be held later on in the chapter of the optimization done and the possible effect of the memory usage on the global performances.

3.4 Conclusion

To conclude this chapter, we have chosen to implement an authentication system based on Edwards curves in projective coordinates defined over the prime field $\mathbb{F}_{P_{192}}$. The implementation of this crypto-system is detailed in the following chapters.

Chapter 4

Software Implementation

In this chapter, we provide all the basic information concerning the software implementation such as the data representation and the algorithms. The first part explains the global architecture, the data representation and the memory management of our implementation. The following parts detail the functions implemented for each of our software layers, namely, the arithmetic layer, the elliptic-curve layer and the protocol layer. The signature of the functions are overviewed and some details about the C code given. Of course, a detailed analysis of the whole code will not be done for evident reasons of length but the entire sources can be found in Appendix **??**.

4.1 Structure and data

In the interest of reaching the heart of the implementation, three main points have first to be presented. The first one is the architecture of the code, the second one is the data handled all along and the third one covers the memory allocation.

4.1.1 Code Architecture

Since the final program intends to be of a considerable size, we have applied a structured manner of coding right from the beginning. An ECC implementation can basically be divided into three layers which should be independent each one from another[1]: the arithmetic layer, the elliptic-curve layer and the protocol layer. These layers ones are often represented in a pyramidal form like in Figure 4.1 to illustrate which layers another layer relies on. As in the OSI model, "a layer serves the layer above it and is served by the layer below it"[33]. These layers are described in the next sections.

[1] Meaning that if the signatures stay the same, the arithmetic field – for example – could be changed by a layer implementing the Binary Field Arithmetic

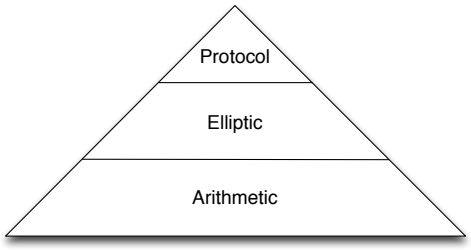

Figure 4.1: Layered Software Implementation.

4.1.2 Data structure

We chose to represent our variables using two high-level data structures: Fp and ECPOINT_PROJ. The former represents an element of $\mathbb{F}_{P_{192}}$ which by definition belongs to the range $[0, 2^{192} - 2^{64} - 1)$and so 192 bits are needed to store one of these. Considering the 8051 is an 8-bit processor, an array of 24 bytes/words is used

$word_{23}$	$word_{22}$	...	$word_1$	$word_0$

with the convention that the byte $word_{23}$ contains the most significant byte while the byte $word_0$ stores the least significant byte.

```
typedef struct {
        ELEMENT    e[NUMWORD];      // NUMWORD = 192/8 = 24
                                    // ELEMENT is a typedef unsigned char
}  Fp;
```

The ECPOINT_PROJ represents a projective point. We use this coordinate system to avoid the inversion needed in the affine coordinate algorithms. According to the definition given in the Edwards curve section, three elements of $\mathbb{F}_{P_{192}}$ – or Fp – are needed :

```
typedef struct
{
        Fp  x;
        Fp  y;
        Fp  z;
} ECPOINT_PROJ;
```

4.1.3 Memory Management

The last point to cover before beginning the description of the implementation in itself is the memory management to apply to the 8051. During the execution of a program, the processor handles data. This means it takes data from the memory, computes something with it and puts it back in the memory. As seen in Chapter 3, the 8051 owns three separated memories: one ROM and two RAMs. Here, only the internal and external RAMs are considered because they are the ones which contain

the data handled by the program: the global variables and the stack. The internal RAM is small and fast while the external RAM is bigger but slower. Hence, the programmer has to chose where to put which data, and choosing a memory model helps him to do so.

First, the different access times to the two RAMs are illustrated fusing exemplary assembler (asm) code fragments. Then, the different possibilities offered when one tries to compile a program are presented. Finally, a choice of model is made.

Access times. As often in the use of memory, bigger means slower and the 8051 makes no exception. A good way to estimate the difference between the two access times is to look at the code for both cases and compare them. In the case of a variable i located in the internal RAM, a piece of code to read the variable executes :

```
MOV R0,#LOW(i)
```

which takes two cycles. In the case of a variable j located in the external RAM, a piece of code to reads the variable executes :

```
MOV DPTR,#i
MOVX A,@DPTR
```

which takes five cycles.

Therefore, one can see the gain that can be obtained by putting the operands in the internal RAM rather than in the external RAM. Without considering the memory model, allocating a memory space – internal or external – to a variable can be done in the code thanks to the syntax :

- **char xdata i** to put the char i in the external data space,
- **char idata i** to put the char i in the internal data space.

But of course, it is not convenient to use this syntax for each variable. That is the purpose of the memory models.

Models. Rather than explicitly associate each variable with a memory space, memory models control the default case. When nothing is specified, the memory model applies its rule.

- **Small memory model :** tries to put all the data in the internal RAM.
- **Large memory model :** tries to put all the data in the external RAM.

However, the small memory model has the disadvantage that if the program cannot fit its variables in the internal memory – a case which is highly probable – the code will not be compiled. Therefore two approaches are possible. Either the programmer adopts the small memory model and puts manually variables in the external memory

until the internal variables fit into the internal memory. Or the programmer adopts the large memory model and puts manually as many variables as he can into the internal memory. The first method seems to be easier for the programmer's task and is thus applied in this implementation.

4.2 Arithmetic Layer

This layer describes the operations we implemented to operate on prime field elements. Special optimizations for the field $\mathbb{F}_{P_{192}}$ are as well introduced in the code. Some tricks, such as pre-computed values, could easily be replaced for other prime fields. However, the special reduction algorithm that was explained in Chapter 3 is specific to the NIST-P192 prime number and would require to seriously change the code for other prime numbers.

unsigned char BiggerOrEqual(Fp *a, Fp *b). This function is a auxiliary to the code and is used each time the symbol $<$ appears. It goes through the two inputs **a** and **b** to compare their value. If the value of a is bigger or equal to the value of b, one is returned, otherwise zero is returned. The comparison begins by the most significant bytes such that the process ends and returns the answer at the first met difference. Algorithm 4 clarifies the idea.

Algorithm 4 Bigger in Fp.

Input: Fp a,b
Output: 1 if a$\geq$ b, 0 otherwise

```
for i from 23 to 0 do
  if a[i]>b[i] then
    return 1
  else if a[i]<b[i] then
    return 0
  end if
end for
return 1
```

void Add(Fp *a, Fp *b, Fp *c). Taking two input operands **a,b** this function computes in a simple fashion their sum modulo p and puts the result in **c**.[2]. As the two inputs are supposed to belong to the range [0,p-1] their sum is necessarily bounded by 2p-2 and operate a modulo results in at most one subtraction of p. Nevertheless, since 2p-2 $> 2^{192}$ and only 192 bits are used to store field elements, special care should be taken to check for a possible overflow. This point is ensured by the verification of k at the end as shown in Algorithm 5.

The expression c = c-p is more complex and the two conditions are handled differently but ends up with the same result :

[2]From now on, the letter p will replace the number $2^{192} - 2^{64} - 1$ for reasons of concision.

Algorithm 5 Addition in Fp.

Input: Fp a,b
Output: Fp c

```
k = 0
for i from 0 to 23 do
  c[i] = a[i]+b[i]+k
  if overflow then
    k = 1
  else
    k = 0
  end if
end for
if k=1 or c>p then
  c = c-p
end if
```

- **k=0 and c>p :** This case simply uses the subtraction function.

- **k = 1 :** This case can not make use of the subtraction function since Fp is encoded on 192 bits and the overflow would need a 193rd bit. However adding a byte for one bit does not seem interesting, so a simple trick is used: the precomputed value SubConst=$2^{193} - p < 2^{193}$ is added to the value of c. After an overflow, the value c equals $a + b - 2^{193}$ and the addition with the SubConst directly gives the desired answer :$(a + b - 2^{193}) + (2^{193} - p) = a + b - p$.

void Sub(Fp *a, Fp *b, Fp *c). This function works with a similar flow than the addition except that a preliminary test is effectuated. Taking two inputs **a,b** it places the result of their subtraction in **c**. Algorithm 6 details the implementation which has two possible execution paths depending on :

- **a>b :** Since a-b>0, the subtraction is simply executed.

- **b>a :** Since a-b<0, the desired answer is a-b+p. So, the subtraction b-a is calculated since it gives a positive result without carry problems. Then, this result is subtracted to p such that the obtained computation is: p-(b-a)=a-b+p the correct answer.

void Mult(Fp *a, Fp *b, Fp *c). This function stores in **c** the result of **a** $\times$ **b** modulo p. This process involves two main steps, the first one being the multiplication in itself and the second one the reduction. The multiplication is computed by using the grade school algorithm from [10] while the reduction is executed as explained in Chapter 3 for the prime number NIST-P192.

[3]This subtraction is computed with a loop but it is not written here to be conciser.

Algorithm 6 Subtraction in Fp.

Input: Fp a,b
Output: Fp c

```
k = 0
if a>b then
  for i from 0 to 23 do
    c[i] = a[i]-b[i]-k
    if underflow then
      k = 1
    else
      k = 0
    end if
  end for
else
  for i from 0 to 23 do
    c[i] = b[i]-a[i]-k
    if underflow then
      k = 1
    else
      k = 0
    end if
  end for
  c = p-c [3]
end if
```

Algorithm 7 Multiplication in Fp.

Input: Fp a,b
Output: Fp c

```
x = vector of 48 bytes/words
x[i]=0 for 0 ≤ i ≤ 47
for i from 0 to 23 do
  U=0
  for j from 0 to 23 do
    (UV) = x[i+j]+a[i]*b[j]+U
    x[i+j] = V
  end forx[i+24]=U
end for
c=reduction(x)
```

4.3 Elliptic-Curve Layer

The elliptic-curve layer contains only two functions in the case of Edwards curves. The first one is the Point Addition of two points on the curve and the second one is the scalar multiplication.

void EC_Edwards_Add(ECPOINT_PROJ *p1,ECPOINT_PROJ *p2, ECPOINT_PROJ *res, Fp *d). This function take as input two points **p1** and **p2** in the projective coordinates system and computes the sum of this two points on the curve following the algorithm given in Appendix A. The fourth parameters is the parameter for the Edwards curve.

void EC_Edwards_Mult(Fp *k,ECPOINT_PROJ *p,ECPOINT_PROJ *res, Fp *d). This function takes as input a prime field element **k** and a point **p** and computes the scalar multiplication with the square-and-multiply algorithm presented in Chapter 2.

4.4 Protocol Layer

The protocol layer consists in the implementation of the high-level authentication scheme. In this work, we did not programmed this aspect of the system to rather focus on the Elliptic-Curve layer and the co-design comparison. But if it had to be done in future developments, the layer would involve the three main steps of the Schnorr protocol: commitment, challenge and verification. These operations basically rely on the scalar multiplication and the field arithmetic but make appear the communication between the two entities. In the automotive world, the communication between the verifier and the prover occurs on a CAN bus. The implementation should be able to handle such a channel of communication besides the ECC.

4.5 Results of the implementation

The implementation presented so far was successfully tested for a half-dozen of points and scalars by using the online Magma Calculator to verify the result. The scripts used to compute some values on the curves can be found in Appendix D. Concerning the time taken for an execution, we ran the target with a 12.5 MHz clock and obtained a duration around 29 seconds.

In this chapter we present a lot of functions with conditional executions of sub-functions. We note that here, because it becomes a problem in the security assessment.

4.6 Conclusion

Regarding the amount of code that already exists at this point of the task, it should be noted that this layered structure gives a real benefit. Besides the advantage

on the organization, the arithmetic layer could be change without problem if the naming stays the same. Concerning the performances, an obvious conclusion is that improvement can and must be done in order to lower the timing to a reasonable duration.

Chapter 5

Hardware Acceleration

This chapter handles the speed-up of the previously presented implementation. In order to increase the performance of the software application, a first analysis of the code will be executed. By doing this, it becomes clear where the bottleneck of the system is located. The second step consists in removing the bottleneck by adding hardware modules in order to accelerate a specific process. Different modules are considered and their impact shortly studied. After the hardware implementation, a second analysis of the bottleneck leads to some specific ASM optimization of the code. The chapter ends with the validation of the implemented co-design by running some tests.

5.1 Overview

In fine, a co-design results from the combination of software and hardware. As such, a co-design represents a trade-off between two extreme cases. On one side the implementation of the crypto-system could be a C code running on a standard processor i.e. a pure software implementation. It makes it relatively cheap to develop and really flexible in the maintenance. On the other side, the implementation could be a specific ASIC executing a precise task as it exist for AES for example. It makes it fast but difficult to change.

In this work, the adopted work flow proposes to start from the C version which offers a structure, profile the code, and transfer parts of the computation to the hardware. The process can be done iteratively to finally ends up with a trade-off between hardware and software. Where to place the boundary is no easy question and depends on goals and constraints. In this chapter we propose to transfer some computation to the hardware on basis of some analysis. Concerning the effects of those changes and a discussion what the best option is -if one exists - is left for Chapter 7.

5.2 Bottleneck Analysis

The profiling shown on Table 5.1 corresponds to the hierarchical study of the function EC_Edwards_Mult which computes the scalar multiplication k*P. For each sub-function called in a function, the number of call and the cycles corresponding to the execution of the sub-function are roughly estimated. This mode of presentation is used to help the reader verify the addition of times spent in each sub-function equals to the time of the caller function. And finally, this table reveals an highly expected fact: a majority ($\sim 95\%$) of the time consumed by the computation of a scalar multiplication is spent in the modular multiplication of two elements from $\mathbb{F}_{P_{192}}$. This is thus the first bottleneck that is going to be tackled by accelerating the multiplication.

Function	Sub-Functions	Cycles	# Appearances in Function
EC_Edwards_Mult_Soft		245M	
	EC_Edwards_Add	850k	~ 288 [1]
EC_Edwards_Add		850k	
	Mult	68.5k	12
	Sub	6k	4
	Add	5.1k	3

Table 5.1: Profiling of the C code.

In order to accelerate the multiplication, the Montgomery's multiplication [10] is a method often exploited. In this work, it was arbitrarily chosen to stuck to the grade school multiplication presented in Chapter 4. In that context, an obvious choice was made: increase the size of the multiplier. Since the main loop of the algorithm has a size which depends on the ratio:

$$\left(\frac{|| \text{ Operand of Multiplier } ||}{||\text{Element of } \mathbb{F}_{P_{192}}||} \right)^2$$

doubling the size of the multiplier would divide by four the number of loop cycles. However, other factors arise and lower this gain: the handling of the carry-propagation in C, the overhead created by the traveling of data between the processor and co-processor, ...

[1] Considering the fact that a bit '1' involves 2 point additions and a bit '0' one point addition, the average number of point additions for a uniformly distributed scalar equals to $192 \times (0.5 \times 1 + 0.5 \times 2) = 288$

5.3 Interface of communication

Now that the bottleneck is detected, the presented work flow plans the addition of hardware modules besides the soft-core. On the one hand, there is of course the module in itself. But in the other hand, the interface of communication between the processor and the co-processor is a key factor that will also directly affect the performance of the acceleration. Ideally this interface should stay the same for all the hardware modules such that the comparison between the modules is possible without effect of the interface. In reality, this principle is not applied for all the modules that will be presented. The interface of communication changed for simple reason that will be explained.

First one has to analyze the offered possibilities. As a reminder, the 8051 owns three ways of communication with the external components: the serial port, the parallel port and the memory. On that basis three cases were considered :

- Memory mapped

- Parallel port Communication

- Serial Communication

Right from the beginning, serial communication was dismissed because it would force the co-processor to have a block handling the serial communication and was considered to complicated. The two other interfaces are used in the following sections. The parallel port was first used because of the easiness of its development/programming. Using this mean of communication appeared to be a simple and working idea. With the time, it became obvious that the overhead created by the moves of the operands between the memory and the co-processor through the CPU represented a non-negligible amount of time. This point made us use a memory mapped interface to command the 192-bit co-processor. A posteriori, the memory mapped interface would not be more efficient in the cases of the 8-bit, 16-bit and 32-bit multipliers since these multiplications involves too many different operands in the code. In comparison, the 192-bit multiplier was successfully summarized as twelve different cases (two fixed locations to take the input from and a location to write the result). The smaller multiplications involve too many operands at different memory locations. One way to tackle this problem would be to move the operands at a precise memory location but it would also create an overhead. Another possibility is the writing in memory the location of the operands which means writing three times two bytes.

5.4 Basic blocks: 8x8, 16x16, 32x32

In line with what was said in the previous section, the first three modules that were implemented are built with a communication based on the parallel port. Figure 5.1 illustrates the configuration put in place. It makes use of three out of the four parallel ports :

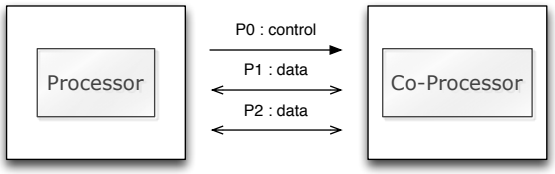

Figure 5.1: Parallel communication interface.

P_0 **:** sends command to the co-processor.

P_1, P_2 **:** transfer 2 bytes from/to the processor to/from the co-processor.

In reality, stricto sensu, no actual command is sent to the co-processor. The basic reason for that is the internal architecture of the modules. In order to be efficient in the re-use of the code and to be able to compare them, the same approach was used such that they all -8x8, 16x16 and 32x32- work on basis of Finite State Machine. The sending of commands on P_0 result in the sending of precise to switch from one state of the FSM to the next one. This point is more easily clarified by a quick look to Appendix C where the codes can be found.

Concerning the computation in itself, no special effort was made to improve – if possible – the Xilinx process to synthesize the * VHDL operation into FPGA elements. Actually, for the 8-bit and 16-bit multiplier, the synthesis uses directly the 18-bit hardware multiplier existing inside the Virtex2 FPGA. For the 32-bit version, the Xilinx tool employs four times this same block to construct the wanted multiplier.

Of course, this approach can not be continued for big multipliers since the synthesis of a 64-bit multiplier would burn more resources than available. Furthermore, even if synthesizing such a big block was possible, it would not be a good solution. It is preferable to develop an sequential architecture which re-use the hardware and takes a few cycles rather than use a huge combinatorial module which would compute the result in one clock cycle.

5.5 192-bits modular multiplication

In this section we cover the design of our bigger multiplier. Regarding the consideration done in the last section about the resource utilization. We intend to develop an architecture more efficient in resources utilization than what the VHDL tools would generate for a 192-bit multiplier. This section is divided in two parts. The first one covers the second interface which is now possible to implement as explained earlier. The second part cover the multiplier in itself.

5.5.1 Interface

The adopted memory mapped interface of communication is illustrated in Figure 5.2. The module fetches the byte of information directly from the memory. In the figure, we see the Multiplexer block which allows the memory access to the 8051 or the co-processor in an exclusive way: the two entities can not access the memory at the same time. The interface works basically as follows: when the processor wants the co-processor to compute a 192-bit multiplication, it sends a byte on the parallel port. This byte contains the information it has to start and where the input can be found and where to put the result. We are able to do all that in one byte because there are only twelve cases. Therefore the processor sends only the reference of the configuration – a number between zero and elven – and the memory location are hardcoded in the hardware module. This has the advantage to keep the overhead minimal.

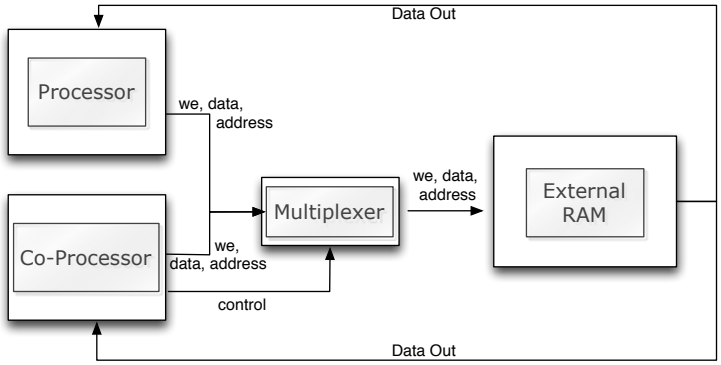

FIGURE 5.2: Memory Mapped Interface.

5.5.2 Multiplier

Concerning the multiplier, as it was related earlier, a special .... After researching on the subject , three architectures were seriously considered and are briefly overviewed to defend the choice made:

- Divide-and-Conquer [27]
- Broadcast [4]
- Horner Architecture [24]

Divide-and-Conquer. This method is the hardware equivalent of the Karatsuba-Ofman algorithm [13][2]. The goal being the computation of the multiplication between

[2] Actually, the Karatsuba-Ofman algorithm gives an additional trick to end up with only 3 multiplications. But this part is not covered here since it does not change the conclusion.

two n-bit operands A and B, the idea is to split the two operands in two $\frac{n}{2}$-bit. The multiplication then consists in the sum of four multiplications between two operands of $\frac{n}{2}$ bits as follows :

$$\begin{aligned} A &= A_1 2^{\frac{n}{2}} + A_0 \\ B &= B_1 2^{\frac{n}{2}} + B_0 \\ A \times B &= (A_1 2^{\frac{n}{2}} + A_0)(B_1 2^{\frac{n}{2}} + B_0) \\ &= A_1 B_1 2^n + A_1 B_0 2^{\frac{n}{2}} + B_1 A_0 2^{\frac{n}{2}} + A_0 B_0 \,. \end{aligned}$$

In the case of this work, four multiplications between two 96-bit operands would be required. A 96-bit multiplier is still too big to be synthesized on a FPGA but the method can be applied recursively such that sixty-four multiplications of 24-bit operands. From that short presentation, one can say that this method is greedy and would consume a lot of FPGA resources. Indeed, the Virtex2 used in the lab furnishes forty-eight 18x18 hardware multipliers. These one would allow to build twelve 24x24 multipliers since they are built from four of the 18x18 multipliers. Furthermore, the force of the Divide-and-Conquer resides in the parallelism. Such a parallelism is not possible in our case where the Xilinx IP memory must be accessed byte by byte.

As a final note, this method sound greedy and not obvious to implement since the addition of 64 multiplications have to be added

Broadcast. This multiplier divides the two n-bit operands in k word of n bits such that k.n=n. The computation consists in several round using k multipliers to get first the result $(A_{k-1}A_{k-2}...A_1A_0).B_0$, then $(A_{k-1}A_{k-2}...A_1A_0).B_1$, until $(A_{k-1}A_{k-2}...A_1A_0).B_{k-1}$. Besides that computation a accumulator adds the result of the multiplication shifted in order to end up with the result of AxB as follows :

$$A \times B = \sum_{i=0}^{k-1} ((A_{k-1}A_{k-2}...A_1A_0) \times B_i >> i\,p)\,.$$

According

Horner. This architecture process the multiplication of two n-bits operands with only shift and addition. This architecture proposes the recursive use of a basic block as illustrated in Figure 5.3. The basic idea is that one of the two operands is scanned bit by bit from right to left. For each bit scanned, an accumulator is multiplied by two – shifted one bit to the left –, then if the scanned bit is '1', the other operand is added to the accumulator, if the scanned bit is '0' , zero is added. This architecture is really light compared to the two others. However, if its execution time is

Chosen architecture After this comparison, we decided to adopt the Broadcast multiplier since it seems to be the best area/time trade-off of our three candidates. The module presented earlier is depicted with high-level blocks in Figure 5.4. We

$$XY = (\cdots((x_{n_X-1}Y)2 + x_{n_X-2}Y)2 + \cdots + x_1Y)2 + x_0Y$$

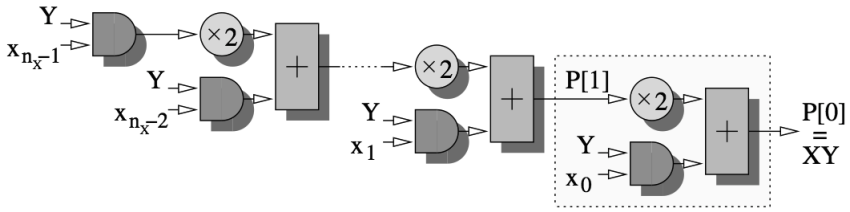

FIGURE 5.3: Horner's Rule-Based Architecture. Source :[3].

decided to divide the operand in 24 bytes. Since the RAM outputs byte, it seemed a advised choice. One operand is fetched in the memory by an FSM. And the other operand is not stored but treated on the fly: each cycle, we receive a byte from the memory and execute a 8×192 multiplication that we add to the accumulator as explained earlier. To that piece of hardware, we also added the hardware translation of the special reduction for NIST-P192 as presented in Chapter 2. The first step is to construct the four numbers defined by the concatenation of bits from the final result of the multiplication. The hardware equivalent of this is basically a re-wiring which is cheap in hardware resources. The second step consists in adding these four numbers. And finally, we subtract a number depending on the value of the obtained sum :

sum < p : sum ← sum

p < sum < 2p : sum ← sum-p

2p <sum < 3p : sum ← sum-2p

3p <sum < 4p : sum ← sum-3p .

After this reduction, the result is kept in the module and sent by the FSM when the module receives the command from the processor.

5.6 ASM optimization

As the last step in this co-design, after all the optimizations brought to the multiplication modulo p, the new profile of the Point Addition shows an interesting point of view on the current state. Indeed, Table 5.2 reveals the predictable fact that the trends reversed itself during the optimization process. The amount of time consumed by the multiplication represents at most 3 percent of the time.

The new bottleneck is now the Addition and Subtraction from the arithmetic layer. In fact, without too much effort, a big improvement can be brought to these two functions. When analyzing the ASM code obtained by compiling the C code, one

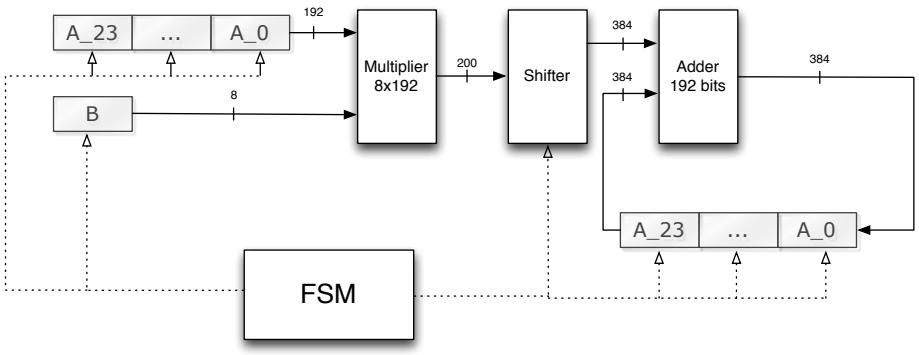

Figure 5.4: Schematics of the 192-bit Co-Processor.

Function	Sub-Functions	Cycles	# Appearances in Point Addition
	Mult_192	90	12
	Sub	6000	4
	Add	5100	3

Table 5.2: Profiling of the Point Addition.

can see that a lot of time is spent in the computation of the memory locations of the variables. Indeed, in order to access a byte contained in a vector which is owned by a structure, the program begins by the structure, then access the vector and finally the precise index inside this vector. The previous point slows down the implementation. If the previous C version is replaced by an optimized assembly version where the computation of the index is done on base of the programmer's knowledge, a good improvement can be achieved as illustrated by Table 5.3.

Function	Sub-Functions	Cycles	# Appearances in Point Addition
	Mult_192	90	12
	Sub	1100	4
	Add	1100	3

Table 5.3: Profiling of the Point Addition with the ASM optimization.

The 1100 cycles which appear in Table 5.3 results from the the mean of two possible executions of the Subtraction/Addition software. In one case, only the Subtraction is executed, in the other case another Subtraction is executed at the end of the function to operate a modulo. The two cases take respectively 800 and 1400 cycles. Assuming that the modulo is applied fifty percent of the time, we compute the average: 0.5*800+1400*0.5=1100.

5.7 Testing

Finally, after all these optimizations, an implementation with a reasonable timing was obtained and allowed to test more exhaustively the validity of the code. In order to check possible errors in the code, 25 thousands scalar multiplication were executed consecutively and the final result compared with a test vector. As the end result was the same, we draw the conclusion that no error occurred during these 25 thousands iterations. The used test vector were generated with the online version of the Magma Software[9] running the scripts from Appendix D.

5.8 Conclusion

The applied bottleneck analysis led us from a pure C software implementation to our final result through several types of steps. At first, the improvements were purely hardware and made us develop several co-processors. Then, we continued with the assembly optimization of the arithmetic layer. The final analysis shows that there is still place for improvement. Indeed, the addition and subtraction – which are supposed to be light operations – are each ten times longer than the multiplication.

Chapter 6

Security Assessment

In this chapter, an attempt to estimate the level of practical security ensured by the developed co-design will be directed. As theoretically introduced in the first background information chapter of this thesis, even perfectly secure scheme of encryption can be attacked in real-life cases. The so-called side-channel leakages can lead to the break of a system. In the following sections, the only channel of information that will be used is the power consumption.

In a first phase, traces of power consumption will be analyzed to crack the secret key of the implementation. Based on the found leakages, fixes will be put in place in order to prevent the attacks. At the end of the first phase, the implementation is believed to be resistant against a visual SPA.

In the second phase of this security assessment, the problematic of the Differential Power Analysis will be tackled in a more theoretical way. No real attack was lead but theoretic arguments are advanced to prove the DPA-resistance.

6.1 Experimental setup

As introduced in Chapter 2, the Power Analysis attack takes place thanks to an experimental setup. Traces of the power consumption are obtained thanks to the measure of the voltage drop on a resistor connected in series between the power supply and the target. Such an experimental setup consists in two entities that we briefly describe :

- The target

- The measurement tool.

The target. As a target we use the Sasebo G board [23] which proposes two Xilinx Virtex2 Pro: xc2vp7 and xc2vp30. The latter is the one on which we load our designs and take measures. This attack board is dedicated to the side-channel analysis. The FPGA receives the power supply through a 1Ω resistor connected in series. This value is convenient since the voltage dropped across the load has the same value

than the current which flows in it. Indeed according to the Ohm's law: V=RI = 1.I, in this case.

Besides the analysis, the board allows the user to connect any clock source. We choose 12.5 MHz in order to obtain a reasonably fast UART connection.

The measurement setup. To measure the traces an oscilloscope Tektronix DP0 7254 is used. Thanks to its bandwidth of 2.5 GHz and sample rate up to 40 GS/s we are able to receive traces of high quality on the computer stations to which it is connected.

6.2 Point of attack

According to [7], at least two points are good targets in the attack against the Schnorr Protocol. Referring to the terminology used in the description of the protocol, the two proposed attacks are :

- The scalar multiplication c*P
- The arithmetic multiplication k×c.

In this work, we focus entirely on the scalar multiplication and do not consider the arithmetic multiplication even if the threat is real.

6.3 Simple Power Analysis

As stated in the introduction, this first attempt to break the implementation consists in the visual analysis of power traces recorded from a computation. The secret key is unknown to the analyzer and this one tries to recover the full key or a maximum information about it. The analysis of the code is done on the coprocessor for 192-bit multiplication because the traces are smaller and it makes it easier to see patterns. One will see that it allows to find quickly the first weaknesses of the code. In fact, two threats to the SPA-resistance were found. Each of them is first exhibited and then resolved.

6.3.1 First Attack : Conditional arithmetic operations

The first weakness that is described here takes its origins in the arithmetic layer. It illustrates the statement from [28] saying that the side-channel resistance should be taken into account at all levels of abstraction. This case shows that an error in the arithmetic layer can lead to the break of the whole implementation.

On Figure 6.1, one sees the consumption traces of a Point Addition. Annotations[1] help the reader see the internal execution of the operation. The series of operations:

$$M, M, M, M, M, M, S, A, M, M, S, M, A, A, M, S, S, M, M$$

[1] M,A,S stand for Multiplication, Addition, Subtraction with the number of time they are executed consecutively.

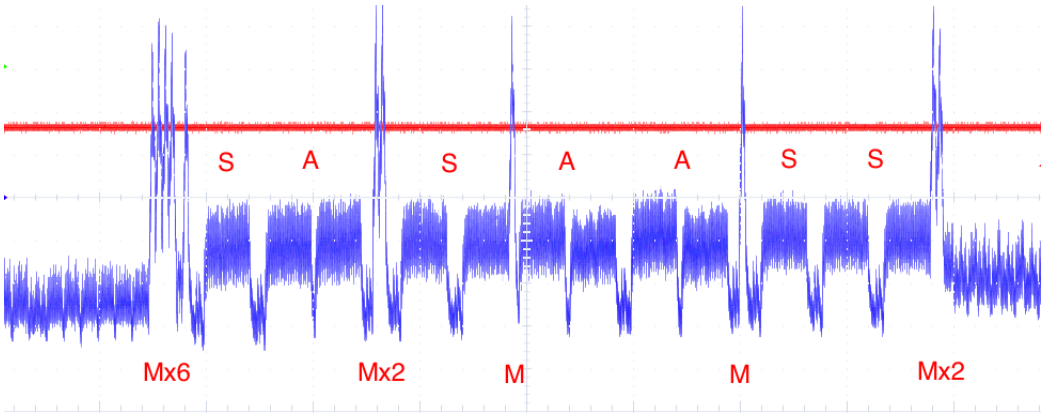

FIGURE 6.1: One Point Addition.

reflects exactly the C code. And in fact, this point is not a problem. Of course, it helps to distinguish the operations executed but as part of an invariant code – a code which executes always the same series of operations – this lead to no important leakage. The weakness is rather the variable execution of the Addition and Subtraction of the arithmetic layer. For the Subtraction, for example, as stated in Chapter 4, a conditional subtraction is operated at the end of the function in order to apply a modulo. That is the reason why the annotated Subtraction on the trace involves sometimes one and sometimes two sub-parts.

To fix this weakness one can make the arithmetic time-constant by modifying the C code. Of course, it leads to a decrease in the timing performance, but this is the price one has to pay to get closer to the SPA-resistance.

In the case of the Subtraction function, a dummy subtraction by the value 0 is inserted in Algortihm 6 and leads to the a modified version which is summarized in Algorithm 8. The same kind of modification is applied to the Addition operation and these changes lead to the desired invariant execution illustrated by the power trace on Figure 6.2.

Algorithm 8 Time-constant subtraction in $\mathbb{F}_p$.

Input: a,b in $\mathbb{F}_p$
Output: c = a-b mod p
  if a > b **then**
    c = a - b;
    c = c - 0;
  else
    c = b - a;
    c = p - c;
  end if

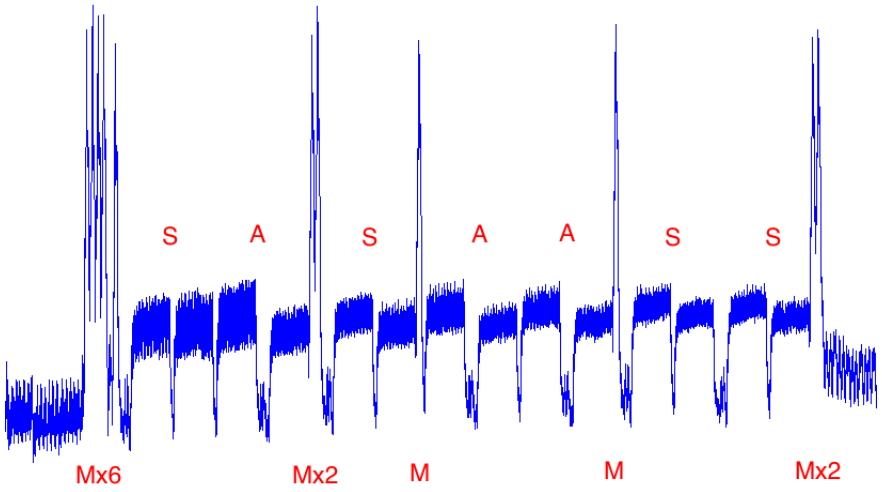

Figure 6.2: Invariant execution.

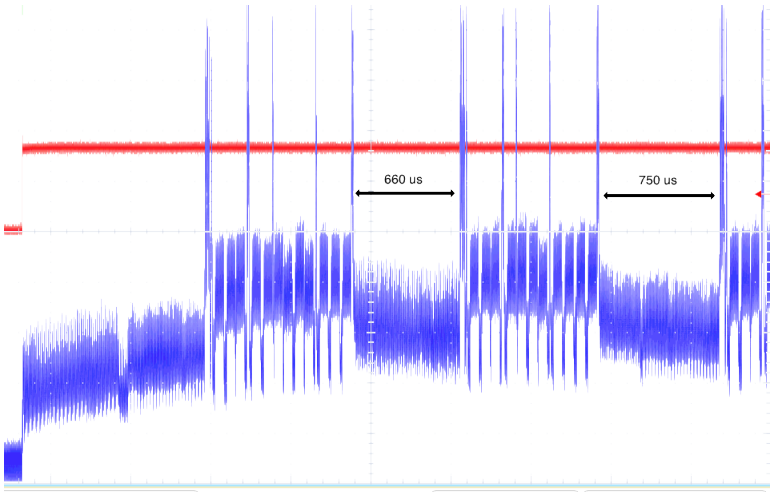

Figure 6.3: Unequal intervals.

6.3.2 Second Attack : Unequal intervals of time

The second weakness takes its roots in the Elliptic Layer. Figure 6.3 exhibits the existing problem. One can clearly distinguish that the gap between two Point Additions is not always the same. This information reflects perfectly the C code summarized by Figure 6.4. Some details of the implementation were removed but one can see that as explained in Chapter 5 the three coordinates of two points have to be put at a precise memory location to let the hardware compute their sum. Then the result can be taken from a precise memory location. These movements of memory space create unequal intervals when executed in a loop.

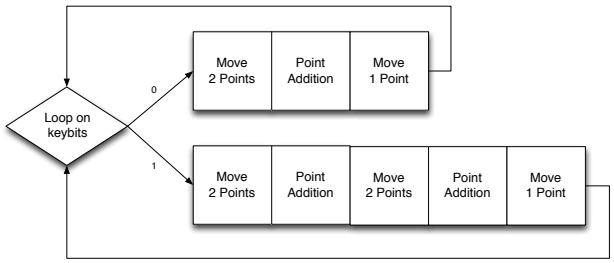

FIGURE 6.4: Graphical Representation of the Software.

In order to fix the problem, one could think of implementing a series of NOP[2] in assembly such that the intervals become of the same length. In fact, this trick would make equal intervals but the result would not be SPA-resistant. The series of NOP creates a visible gap in the power consumption as attested in Figure 6.5.

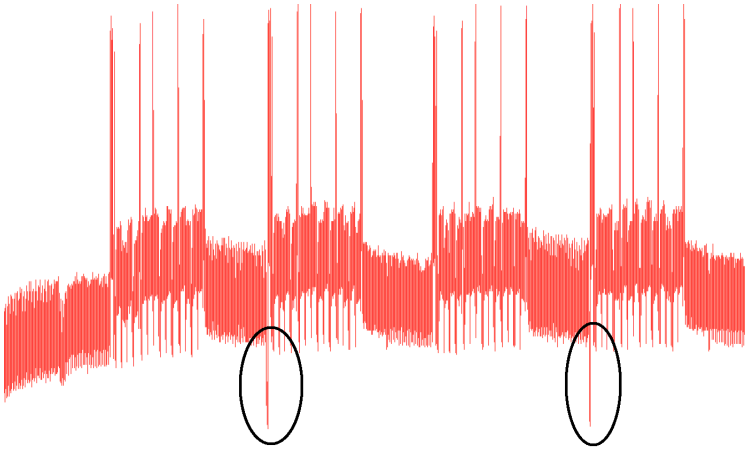

FIGURE 6.5: Effect of NOP's.

To avoid these unequal intervals without using NOPs we put in place a more regular structure where the same function is repeatedly called with always the same movement of memory as shown in the simplified piece of code presented in Listing 6.1.

If visually, Figure 6.6 seems to show that the SPA security is ensured, the reality is different. Indeed, the evaluation of the statement inside the IF instruction still creates unequal intervals. These differences are distinguishable by subtracting two different traces or superposing a slid version of the same trace. Figure 6.7 depicts the trace of two computations and their difference. One of the traces represents the processing of the bit sequence "01" and the other one the sequence "10".

[2]NOP is an instruction of the 8051 instruction set which dictates to do nothing i.e. no computation, no change in the register or memory contents, etc.

```
if(keybit==1)
{
        //  2*P1+P
        EC_Edwards_operation(...);
        EC_Edwards_operation(...);
}
else
{
        //2*P1
        EC_Edwards_operation(...);
}
```

Listing 6.1: Regular Sctructure

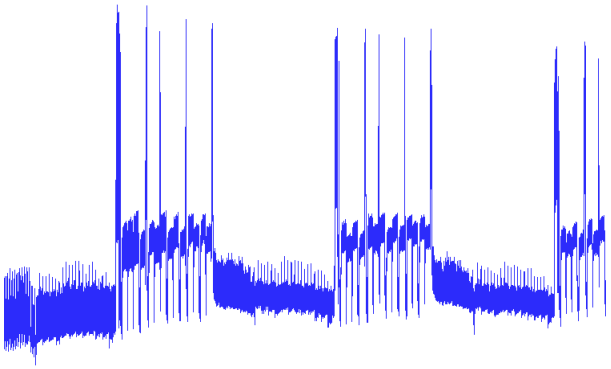

Figure 6.6: Visually Equal Intervals.

Another leakage – which is not illustrated here – resides in the BiggerOrEqual function. As said in Chapter 4 , the function stops as soon as the two compares inputs are different. Even if it is unsure if the leaked information is important, it still represents a weakness in the SPA-resistance.

To conclude this analysis, two security issues still exist. One of them is known to allow to discover the key if the attacker knows the program. In case where the attacker doe not know the program, he still sees the gaps but does not know if the longer gap is associate to a bit zero or a one. Hence, he ends up with two possible keys. He only has to try the two possibilities to discover the key.

In order to obtain the SPA-resistance, we should modify the implementation to make it even more regular. The final result should provide a data independent execution.

6.4 Differential Power Analysis

This section has the purpose to outline a way to protect our implementation against DPA attacks in future developments. We did not implement the proposed

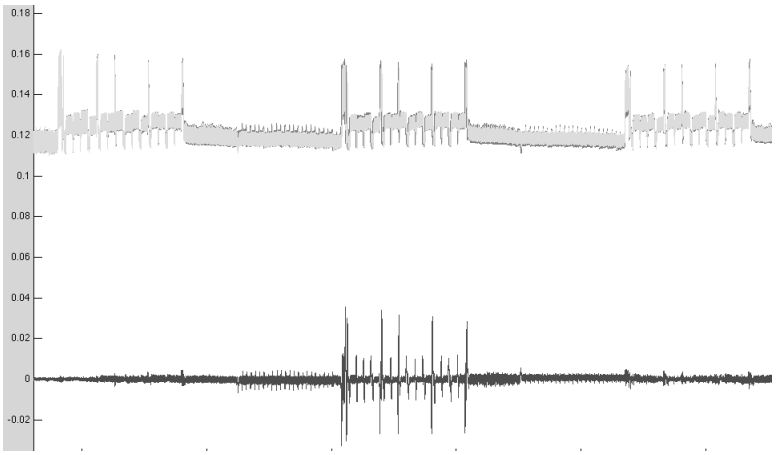

FIGURE 6.7: Up: Power traces of keys 0x5 and 0x6.
Down: Difference of the two traces.

method because as stated in the design flow [7] it makes little sense to look for a DPA-resistance if an implementation is not SPA-resistant. Nevertheless, we present the Randomized Projective Coordinates technique from Jean-Sebastien Coron [5] in order to give perspectives on the reachable security once the SPA-resistance is reached. Other techniques exist, but this method has the advantage to only use one scalar. The Point Blinding uses two scalar multiplications for example.

The Randomized Projective Coordinates technique is based on the fact that in the projective coordinates we use $(X, Y, Z) \sim (\lambda X, \lambda Y, \lambda Z)$[3]. In other words these two coordinates correspond to the same point and both can used to compute the scalar multiplication. Hence, it suffices to change the λ for each scalar multiplication and we get a DPA-resistant implementation. Indeed, the attack explained in Chapter 2 is based on the development of internal values from the point P and an hypothesis. But since, the target is not only dependent on the input, but also a random λ, the attack is not feasible anymore.

6.5 Conclusion

In this section, we cracked the implemented system by means of power analysis attacks. More precisely, SPA attacks were sufficient to completely break the implementation. Two security issues still remain and make our implementation insecure. Nevertheless, even though we lack time to implement them, ways to make it secure exist.

[3] But it also works for other projective coordinates systems.

Chapter 7

Comparison

In this chapter, we give a comparison of the co-designs developed on basis of four different aspects: the execution time, the area utilization, the energy consumption and the ensured security. For each one, we introduce their relevance in our context and compare their performances. We chose these four parameters to limit the scope of our work. Others specifications could be added such as the flexibility of the co-design, the code size, the memory utilization.

7.1 Execution time

This specification appears in the first place since it drove a major part of the work. The end-user allows a lot of importance to this performance in different context of the automotive world. If the authentication system serves as the key of the car, the time should be as short as possible since the user does not want to wait too long. For the car manufacturer this number also impact the utilization rate of a processor. If the task is light, other program can be ran in parallel.

Table 7.1: Speed of each version.

Version	Duration [cycles]	Time at 12.5Mhz [s]
C	362.5M	29
Assembly	412.5M	33
8-bit multiplier	512.5M	41
16-bit multiplier	262.5M	21
32-bit multiplier	150M	12
192-bit multiplier	8.1M	0.647

Table 7.1 gives the comparison of the speed between all the versions in seconds and in cycles. The time in cycles is the measure usually preferred since it is clock independent. Besides that, the time in seconds is shown to give an idea about when

run when the system runs at 12.5 Mhz. The time can be measured thanks to the C code in Appenix B. Here, due to an unexpected lack of time, the figures are extrapolated from the timing in Appendix E.

First, one sees the odd result that the assembly version ends up with a longer execution time than the C version. Thi result is misleading since usually the assembly program are faster than their C equivalent. What is named "Assembly version" designates a version where we wrote our own asm code to use the 8-bit multiplication instruction. We did that experiment because we thought that the C version could have suffered from a bad translation in asm. After analyzing the two codes it turns out that our asm multiplication is slower due to :

- It is not easy to access members of a vector in asm. So, we put the values first in a simple variable and then access the variable in asm. This intermediate decreases the efficiency.

- The compiler translates perfectly the C multiplication in asm. While, the C code puts the result in a 16-bit variable and then separate the result in two bytes thanks to masks and shifts, the compiler understands and puts directly the result in the separate C variables.

For the other figures, we see the expected result from the co-design process. There exists an improvement factor of 45 between the software implementation and our fastest co-design. Within the smaller co-processors, we see that the 8-bit co-processor decreases the performance of the system since it introduces an overhead without providing more computational power[1]. Then, we see that from a 8-bit coprocessor to a 16-bit co-processor a improvement factor two is achieved. But this factor decreases for a transition from a 16-bit to a 32-bit multiplier. Two reasons explain that decreasing improvement. The first one is the memory. The two inputs and the result of the co-processor have to be sent through the parallel port which creates an increasing over-head to the computation. The second factor is the C carry management which is not efficient. For the 192-bit co-processor the two problems are solved since we use a faster interface and the carry propagation is handled directly in hardware. The management of the reduction in hardware gives the benefit of not using the addition function which is still slow. To conclude, and by comparing to some existing 8-bit implementations [18], we can say that we have a rather inefficient software implementation but that the hardware acceleration helps to get finally with the 192-bit multiplier a usable implementation in a non real-time context.

7.2 Area utilization

The area utilization is a parameter which directly reflects the price of a hardware implementation. Indeed, the area utilization gives an idea about the silicon area consumed by the device. And this area can be directly translated into a price for the

[1] It was an expected result. Actually, we introduced the 8-bit co-processor to have a better set to assess the trend of the co-processors on the parallel interface.

manufacturer which is one of its most important parameter. Therefore, we chose to take it into account. Table 7.2 gives the resources utilization for five configurations since the assembly and software version have the same hardware. Basically four numbers are given: the number of slices, the number of Lookup tables, the number of flip-flops and the number of 18×18 hardware multipliers. We do not take into account the memory block as the RAMB16 because the memory configuration is fixed and not in the scope of this work.

A more detailed comparison is given by translating the resources into their transistor structures and using the transistor as common unit of measure. To estimate the structures we followed the indications given in [8] [11] and took 164 transistors for the 4-input LUT, 12 transistors for the flip-flop and 4536 transistors[2] for the 18-bit multiplier.

Version	Slices	LUT4	Flip-Flop	18x18 multipliers	Transistors Estimate
Without co-proc.	2066	3857	505	1	643k
8-bit multiplier	2171	4019	563	2	675k
16-bit multiplier	2216	4071	605	2	683k
32-bit multiplier	2298	4198	645	5	720k
192-bit multiplier	5074	9326	1455	13	1.6M

TABLE 7.2: Resources Utilization.

The numbers given in Table 7.2 indicates that when kept smaller than 16-bit, the hardware acceleration does not cost a big area. For an increase of ∼10% of the area, the hardware module can be implemented.The reason for this good performance is that 18-bit x 18-bit multiplier is provided. However, four 18-bit multipliers have to be combined to build for a 32-bit multiplier in a combinatorial fashion. In comparison, with a broadcast architecture, a 72-bit multiplier could be built and executed in a few cycles. These resukts justify our followed approach. The final point that we would like to highlight is the area taken by the hardware multiplier. According to our estimations, it represents a minor percentage of the design. What cost in all what is around the multiplier: the FSM, the registers, ...

7.3 Energy consumption

The third parameter we opted for is the energy consumption. From one part, it is – with the timing – the most important performance for the manufacturer and end-user. Table 7.3 gives the figures for the energy consumption of a scalar multiplication. The given numbers represent the energy consumed for a key uniformly distributed –

[2]Following the rule which says that n^2 full adders are used for a n-bit multiplier and counting 14 transistors by full adder[32].

meaning that on the 192 bits, 96 are zeros and other are ones. The method applied to find this information can be found in Appendix E.

In the context of the automotive world, the energy consumption is of big importance since the quantity of energy in car battery is limited. We can not let the electronics embedded in the car totally discharge the battery.

Version	Energy	Average Power	Peak Power
C	4.02 J	0.138W	0.148W
Assembly	4.68 J	0.139W	0.151W
8-bit multiplier	5.46 J	0.132W	0.145W
16-bit multiplier	2.58 J	0.122W	0.13W
32-bit multiplier	1.6 J	0.135W	0.146W
192-bit multiplier	0.119 J	0.179W	0.239W

Table 7.3: Figures for a Scalar Multiplication.

After analyzing Table 7.3, we notice the factor two between the energy consumption the 8-bit and 16-bit multiplier. It is an expected result since the two versions use nearly the same hardware – an hardware 18-bit multiplier is used in the two cases– but the version with a 16-bit multiplier is twice faster.[3]

For the 192-bit multiplier, we see two expected facts. The peak power is higher than for the other cases and this point can be easily noticed on the SPA traces from the previous chapter. But the speed-up operated thanks to this hardware acceleration allows to lower the energy consumption by a factor ∼35. We add to that point that our context of automotive application allows us to not worry about these high peaks. This is not the case in all application. In more limited application – RFID for example – peaks

7.4 Security

The last point we elected as a parameter is the hardware security. This issue has become one of the main topic in the last decade for embedded implementations. This section does not really include a comparison since all our versions finally use the same structure: the same scalar multiplication, the same point addition. They also have the same arithmetic layer except that the multiplier used inside the the grade school multiplication is different. Here, we only repeat the conclusion of the previous chapter. The final implementation does not exhibit the property of SPA-resistance since the scalar multiplication still has a key dependent execution and that the BiggerOrEqual function still is input dependent.

[3]Table 7.3 shows that the 16-bit hardware owns a lower power consumption than the 8-bit. It seems strange and since no experience was done to confirm that point, we can not draw a conclusion about that.

Chapter 8

Conclusion

As result of this master thesis, we ended up with the expected result that the hardware acceleration leads to a faster, and less consuming system at the expense of a bigger area. This comparison is illustrated by the quantitative ratio obtained. The hardware acceleration made us come from a pure software implementation to a co-design with a factor 45 on the speed, a gain factor 35 on the consumption and a loss factor 3 on the area. But it must be noticed that for a smaller price – 10 percent of the area – improvement factor around two to three in time and energy consumption can be obtained.

In the end, we finish with a set of non-SPA-resistant implementations. The fastest design – the 192-bit multiplier – takes 0.65 seconds when ran at 12.5MHz which is reasonable but not usable in real-time application. However, the final bottleneck analysis shows that there is still place for improvement since the addition and subtraction – which are supposed to be light – still take the major time of the computation.

In the future, this work could be continued and improved in several ways. First, on the security side, a more regular structure for the scalar multiplication could be implemented in order to get a data independent execution of the code and ensure the SPA-resistance. Then, the proposed measure to ensure the DPA-resistance could be added.

Besides the security, the comparison of co-design configurations could be continued in two directions. On the one hand, the effect of converting the addition and subtraction in the hardware could be studied. On the other hand, other version of multipliers could be implemented – such as the Horner architecture – to obtain more contrasted trade-offs. On the application side, the implementation of the protocol layer on a standard bus , such as a CAN bus,

Appendices

Appendix A

Elliptic curves algorithms

A.1 Simplified Weierstrass in affine coordinates

Algorithms taken from [10].

Algorithm 9 Addition

Input: $P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2)$
Output: $R = P_1 + P_2 = (X_3, Y_3, Z_3)$

$A \leftarrow Y_1 - Y_2$
$B \leftarrow X_1 - X_2$
$C \leftarrow inverse(B)$
$D \leftarrow AC$
$E \leftarrow D^2$
$X_3 \leftarrow E - X_1 - X_2$
$Y_3 \leftarrow D(X_1 - X_3) - Y_1$

Algorithm 10 Doubling

Input: $P = (X_1, Y_1, Z_1)$
Output: $R = 2P_1 = (X_3, Y_3, Z_3)$

$A \leftarrow X_1^2$
$B \leftarrow 3A$
$C \leftarrow A + a$
$D \leftarrow 2Y_1$
$E \leftarrow inverse(D)$
$F \leftarrow CE$
$X_3 \leftarrow F^2 - 2X_1$
$Y_3 \leftarrow F(X_1 - X_3) - Y$

A.2 Unified operation for Edwards Curves

Algorithm taken from [1].

Algorithm 11 Edwards Curves Addition/Doubling

Input: $P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2)$
Output: $R = P_1 + P_2$

$A \leftarrow Z_1 Z_2$
$B \leftarrow A^2$
$C \leftarrow X_1 X_2$
$D \leftarrow Y_1 Y_2$
$E \leftarrow dCD$
$F \leftarrow B - E$
$G \leftarrow B + E$
$X_3 \leftarrow [(X_1 + Y_1)(X_2 + Y_2) - C - D]AF$
$Y_3 \leftarrow AG(D - C)$
$Z_3 \leftarrow cFG = FG$ ▷Since c=1, according to the chosen simplification

Appendix B

Program timing measurement

```
#include <iostream>
#include <windows.h>
#include <time.h>

using namespace std;

int main()
{
    HANDLE hSerial;
    DCB dcbSerialParams = {0};
    char szBuff[72];
    DWORD dwBytesRead = 0;
    clock_t a,b,temp;
    szBuff =
    {3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    196,197,85,142,106,212,213,77,163,80,123,
    16,128,228,160,243,20,109,86,177,158,11,181,144,
     1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    hSerial = CreateFile("COM8",
    GENERIC_READ | GENERIC_WRITE,
    0,
    0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0);
    if(hSerial==INVALID_HANDLE_VALUE){
        if(GetLastError()==ERROR_FILE_NOT_FOUND)
        {
            std::cout << "Doesn't exist\n" ;
        }
        std::cout << "Other problem\n" ;
    }

    dcbSerialParams.DCBlength=sizeof(dcbSerialParams);
    if (!GetCommState(hSerial, &dcbSerialParams))
    {
```

```
        cout << "Doesn't get state\n";
    }
    dcbSerialParams.BaudRate=CBR_9600;
    dcbSerialParams.ByteSize=8;
    dcbSerialParams.StopBits=ONESTOPBIT;
    dcbSerialParams.Parity=NOPARITY;
    if(!SetCommState(hSerial, &dcbSerialParams)){
        cout << "Doesn't set state\n";
    }

    COMMTIMEOUTS timeouts={0};
    timeouts.ReadIntervalTimeout=50;
    timeouts.ReadTotalTimeoutConstant=50;
    timeouts.ReadTotalTimeoutMultiplier=10;
    timeouts.WriteTotalTimeoutConstant=50;
    timeouts.WriteTotalTimeoutMultiplier=10;
    if(!SetCommTimeouts(hSerial, &timeouts))
    {
        cout << "Timeout problem\n";
    }

    //Send projective coordinates
    for(int i=0;i<72;i++)
    {
            temp=clock();
            if(!WriteFile(hSerial, &(szBuff[i]), 1, &dwBytesRead, NULL))
            {
                cout << "Problem to write\n";
            }
            while((clock()-temp)<10);
    }
    cout << "Written\n";


    while(1) // Wait for a '.'
    {
        if(!ReadFile(hSerial, &(szBuff[0]), 1, &dwBytesRead, NULL))
        {
            cout << "Problem to read\n";
        }
        if(szBuff[0]=='.')
        {
            a=clock();
            break;
        }
    }

    szBuff[0]=0;

    while(1) // Wait for a '.'
    {
       if(!ReadFile(hSerial, &(szBuff[0]), 1, &dwBytesRead, NULL))
        {
            cout << "Problem to read\n";
        }
```

```
        if(szBuff[0]=='.')
        {
            b=clock();
            break;
        }
    }

    cout << "CLOCKS :"<<(b-a)<<endl;
    cout << "CLOCK_PER_SEC :"<<CLOCKS_PER_SEC<<endl;
    cout << "Finished" << endl;

    CloseHandle(hSerial);

}
```

Appendix C

VHDL Code of the 8x8 module

In this appendix, we give the code for the 8-bit multiplier. The other modules can be found on the attached CD. The 16-bit and 32-bit multipliers are not given since their structure are very similar to the 8-bit one. The 192-bit is not given here because the code is too long.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;


entity coproc8x8 is
    Port (      clk : in STD_LOGIC;
                        reset : in STD_LOGIC;
                        control : in  STD_LOGIC_VECTOR (7 downto 0);
                data : in  STD_LOGIC_VECTOR (15 downto 0);
                res : out  STD_LOGIC_VECTOR (15 downto 0)
          );
end coproc8x8;

architecture Behavioral of coproc8x8 is

signal M1 : std_logic_vector(7 downto 0);
signal M2 : std_logic_vector(7 downto 0);
signal mul : std_logic_vector(15 downto 0);
signal state : std_logic_vector(3 downto 0);
signal next_state : std_logic_vector(3 downto 0);

begin

mul <= M1*M2;

process (clk) begin

        if RISING_EDGE(clk) then

                if reset = '1' then
                                state<=X"0";
```

```
                                next_state<=X"0";
                else
                                state<=next_state;
                end if;

                if state=X"0" then
                        if control(7)='1' then
                                next_state<=X"1";
                        end if;
                elsif state=X"1" then
                                next_state <=X"2";
                                M1 <= data(7 downto 0);
                                M2 <= data(15 downto 8);
                elsif state=X"2" then
                        if control(6)='1' then
                                next_state<=X"3";
                        end if;
                elsif state=X"3" then
                        res<=mul;
                        next_state<=X"0";
                end if;

        end if;


end process;


end Behavioral;
```

Appendix D

Magma Scripts

D.1 Points on Edwards Curves

Given the x coordinate and d parameter of an Edwards curve, this script finds the corresponding point if it exists.

```
X:=FiniteField(2^192-2^64-1);
Z:=IntegerRing();
x:=X!3;
d:=22;
y:=X!Sqrt((1-x^2)/(1-d*x^2));
y:=Z!y;
y:Hex;
```

D.2 Scalar multiplications

Given a point **P**, the **d** parameter of an Edwards curve, a scalar **nbr** and a number of iterations **ite** this code computes the answer of $k^{ite} * P$. It is based on the piece of code for the Point Addition on Edwards curves from Bernstein's website [1] and the Square-And-Multiply algorithm from Chaper 2

```
nbr :=0x3DCF46ED302128736C0844766B41273BEB74600FF5984564;
XF:=FiniteField(2^192-2^64-1);
ZF:=IntegerRing();
d:=XF!22;
c:=XF!1;

X:=XF!2;
Y:=XF!1458560742465818495538825075188873665709837862246418407 23;
Z:=XF!1;

X1:=XF!2;
Y1:=XF!1458560742465818495538825075188873665709837862246418407 23;
Z1:=XF!1;

for ite:=1 to 25000 by 1 do
```

```
for i:=190 to 0 by -1 do
        if IsOdd(ShiftRight(nbr,i)) then
                val:=i-1;
                break i;
        end if;
end for;

for i:= val to 0 by -1 do

        R1:=X1; R2:=Y1; R3:=Z1;
        R4:=X1; R5:=Y1; R6:=Z1;
        R3:=R3*R6;
        R7:=R1+R2;
        R8:=R4+R5;
        R1:=R1*R4;
        R2:=R2*R5;
        R7:=R7*R8;
        R7:=R7-R1;
        R7:=R7-R2;
        R7:=R7*R3;
        R8:=R1*R2;
        R8:=d*R8;
        R2:=R2-R1;
        R2:=R2*R3;
        R3:=R3^2;
        R1:=R3-R8;
        R3:=R3+R8;
        R2:=R2*R3;
        R3:=R3*R1;
        R1:=R1*R7;
        R3:=c*R3;
        X1:=R1; Y1:=R2; Z1:=R3;

        if IsOdd(ShiftRight(nbr,i)) then
                R1:=X1; R2:=Y1; R3:=Z1;
                R4:=X; R5:=Y; R6:=Z;
                R3:=R3*R6;
                R7:=R1+R2;
                R8:=R4+R5;
                R1:=R1*R4;
                R2:=R2*R5;
                R7:=R7*R8;
                R7:=R7-R1;
                R7:=R7-R2;
                R7:=R7*R3;
                R8:=R1*R2;
                R8:=d*R8;
                R2:=R2-R1;
                R2:=R2*R3;
                R3:=R3^2;
                R1:=R3-R8;
                R3:=R3+R8;
                R2:=R2*R3;
                R3:=R3*R1;
                R1:=R1*R7;
```

```
                        R3:=c*R3;
                        X1:=R1; Y1:=R2; Z1:=R3;
                end if;
        end for;

        X:=XF!X1;
        Y:=XF!Y1;
        Z:=XF!Z1;

        X1:=XF!X1;
        Y1:=XF!Y1;
        Z1:=XF!Z1;

end for;

res:=ZF!X1;
res:Hex;
res:=ZF!Y1;
res:Hex;
res:=ZF!Z1;
res:Hex;
```

Appendix E

Data of Energy Consumption

Table E.1 make appear several figures for each co-design. The three first numbers result from the average of 20 measures for the computation of a scalar multiplication between a random point and the key 0x06. The key is intentionally short in order to record a trace of a reasonable size with the oscilloscope. In the order, these three first numbers are the average power, the peak power and the time taken for the computation.

The fourth number represents the energy consumption for an average key i.e. 96 zeros and 96 ones, and 288 point additions in total. For the 192-bit version, the energy consumption is obtained from a full computation which is not given here. For all the other versions, one proceeds as follows :

1. Since the traces recorded involves the scan of all the bit which are equal to zero and then three point addition (the key is 0x6), we make the assumption that the scan of the zeros is negligible compared to the three point addition. Such that, we divide the power consumption of the trace by three to find the power consumption of a point addition.

2. We multiply the consumption of one point addition by 288 to obtain the energy consumption of an average key.

The assumption which says the key scanning is negligible in comparison with the three point additions makes an error of at most five percent. Indeed, the worst case is the 32-bit version since it is the fastest implementation after the 192-bit version – for which we do not use this method. And the trace for the 32-bit version teaches us that the time taken to scan the key represents less than five percent of the total time. Since, the power of the key scanning is in the average of the trace, we draw the conclusion that the error on the energy consumption is also bounded by five percent. And this error is acceptable since we only want to monitor a trend.

Version	Physical quantity	Value
software	Average	0.0920A*1.5V = 0.138W
	Peak	0.0986A*1.5V = 0.148W
	Time	0.3033s
	Energy	4.02J
asm	Average	0.0929A*1.5V=0.1394W
	Peak	0.1004A*1.5V = 0.151W
	Time	0.35s
	Energy	4.68J
8x8_module	Average	0.0888A*1.5V = 0.1332W
	Peak	0.0969A*1.5V = 0.145W
	Time	0.4271s
	Energy	5.46J
16x16_module	Average	0.0816A*1.5V=0.1224W
	Peak	0.0864A*1.5V = 0.13W
	Time	0.2193
	Energy	2.58J
32x32_module	Average	0.0902A*1.5V=0.1353W
	Peak	0.0975A*1.5V = 0.146W
	Time	0.1231s
	Energy	1.6J
192x192_module	Average	0.1195*1.5V=0.1793W
	Peak	0.1591A*1.5V = 0.239W
	Time	0.0101s
	Energy	0.119J

Table E.1: Figures of Energy Consumption

Appendix F

C implementation

Due to length reasons, the code can be found on the attached CD.

Bibliography

[1] D. J. Bernstein. Edwards coordinates for elliptic curves. http://cr.yp.to/newelliptic/newelliptic.html.

[2] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *ASIACRYPT*, pages 29–50, 2007.

[3] J.-L. Beuchat and J.-M. Muller. Une famille d'algorithmes de multiplication modulaire. http://www.cipher.risk.tsukuba.ac.jp/ beuchat/Teaching/uqac_multiplication_modulo_m.pdf.

[4] D. Buell, J. Davis, and G. Quan. Reconfigurable computing applied to problems in communications security bibtex. In *Military and Aerospace Programmable Logic Devices*, 2002.

[5] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES*, pages 292–302, 1999.

[6] H. M. Edwards. A normal form for elliptic curves. In *Bulletin of the American Mathematical Society*, pages 393–422, 2007.

[7] B. Gierlichs and L. Batina. Power analysis on curve-based cryptography. https://www.cosic.esat.kuleuven.be/bcrypt/lecture%20slides/gierlichs.pdf.

[8] Grishman. Trends in high-performance computer architecture. http://cs.nyu.edu/courses/fall10/V22.0436-001/lecture18.html.

[9] C. A. Group. Magma calculator. http://magma.maths.usyd.edu.au/calc/.

[10] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004. ISBN-0: 387-95273-X.

[11] Y. Hu, S. Das, S. Trimberger, and L. He. Design, synthesis and evaluation of heterogeneous fpga with mixed luts and macro-gates. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, ICCAD '07, pages 188–193. IEEE Press, 2007.

[12] E. II. Yearly report on algorithms and keysizes, june 2011.

[13] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. pages 193–194. Doklady Akad. Nauk SSSR, 1962.

[14] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[15] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[16] F. Koeune and F.-X. Standaert. Foundations of security analysis and design iii. chapter A tutorial on physical security and side-channel attacks, pages 78–108. Springer-Verlag, 2005.

[17] N. Kolbitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[18] M. Koschuch, J. Lechner, A. Weitzer, J. Großschadl, A. Szekely, S. Tillich, and J. Wolkerstorfer. Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller. In *Proceedings of the 8th international conference on Cryptographic Hardware and Embedded Systems*, CHES'06, pages 430–444. Springer-Verlag, 2006.

[19] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN-10: 0-387-30857-1.

[20] M. Medwed. Template-based spa attacks on 32-bit ecdsa implementations. Master's thesis, Graz University of Technology, Austria, 2007.

[21] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[22] V. Miller. Use of elliptic cuves in cryptography. In *CRYPTO 85', advances in cryptology*, pages 417–426. Springer-Verlag, 1986.

[23] Morita Tech / AIST. Side-channel attack standard evaluation board sasebo. http://www.morita-tech.co.jp/SASEBO/en/board/sasebo.html.

[24] J.-M. Muller, J.-L. Beuchat, T. Miyoshi, and E. Okamoto. Horner's rule-based multiplication over fp and fpn̂: A survey. *International Journal of Electronics*, 95(7):669–685, July 2008.

[25] National Institute of Standards and Technology. Fips pub 186-3: Digital signature standard (dss). http://csrc.nist.gov/publications/drafts/fips_186-3/Draft-FIPS-186-3%20_March2006. pdf, 2006.

[26] Oregano Systems. 8051 ip core. http://www.oreganosystems.at/?page_id=96.

[27] G. Quan, J. P. Davis, S. Devarkal, and D. A. Buell. High-level synthesis for large bit-width multipliers on fpgas: a case study. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 213–218. ACM, 2005.

[28] K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede. Side-channel resistant system-level design flow for public-key cryptography. In *Great Lakes Symposium on VLSI*, pages 144–147. ACM, 2007.

[29] J.-M. Schmidt and M. Medwed. Fault attacks on the montgomery powering ladder. In *13th Annual International Conference on Information Security and Cryptology, Proceedings*. Springer, 2010.

[30] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, 1986. ISBN-0: 387-96203-4.

[31] F.-X. Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–44. Springer, 2009.

[32] T. Vigneswaran, B. Mukundhan, and P. Subbarami Reddy. A novel low power, high speed 14 transistor cmos full adder cell with 50% improvement in threshold loss problem. *World Academy of Science, Engineering and Technology*, 13:81–85, 2006.

[33] Wikipedia. Osi model. http://en.wikipedia.org/wiki/OSI_model.