On the security of pairing implementations

Ronan Lashermes

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

cea tech

École doctorale Sciences et Technologies de Versailles - STV - ED 539

Étude de la sécurité des implémentations de couplage

On the security of pairing implementations

THÈSE

présentée et soutenue publiquement le 29 Septembre 2014 pour le grade de

Docteur de l'Université de Versailles St-Quentin-en-Yvelines

(spécialité informatique)

par

Ronan Lashermes

Composition du jury

Rapporteurs :	**Antoine Joux**, UPMC **Jean-Pierre Seifert**, TU-Berlin
Examinateurs :	**Pierre-Alain Fouque**, Université Rennes 1 **Jacques Fournier**, CEA-Tech **Daniel Page**, University of Bristol
Directeur de thèse :	**Louis Goubin**, UVSQ

Laboratoire Parallélisme, Réseaux, Systèmes et Modélisation PRiSM – UMR 8144 | CEA-Tech

Uses a latex class derived from thesul.

Acronyms

ABE Attribute-Based Encryption. 10, 41
AES Advanced Encryption Standard. 2, 3, 5, 47

BDH Bilinear Diffie-Hellman. 42, 49, 127, 128, 131
BLS Barreto-Lynn-Scott. 34, 35
BN Barreto-Naehrig. 6, 32, 34, 35, 38, 43, 44, 48, 57, 60, 61, 63, 72, 109, 122, 123, 125, 131, 133, 134, 136, 150

CDH Computational Diffie-Hellman. 42, 127
CM Complex Multiplication. 33

DDH Decisional Diffie-Hellman. 42, 127
DEM Dupont-Enge-Morain. 33
DES Data Encryption Standard. 2
DFA Differential Fault Attack. 6
DLP Discrete Logarithm Problem. 4, 5, 34, 42, 43, 123, 127, 128

ECB Electronic Code Book. 5
ECC Elliptic Curves Cryptography. 3, 4
ECDLP Elliptic Curve Discrete Logarithm Problem. 3–5, 34, 42, 43, 123, 127, 128
EM Electromagnetic. 6, 46, 48–52, 57, 68, 78, 92, 94, 101, 109, 110, 129–132, 139, 148, 150, 151

FA Fault Attack. 46, 59, 78, 79, 101, 110, 129, 151
FE Final Exponentiation. 38, 44, 79–82, 84, 87, 92, 98, 100, 101, 107, 109, 110, 141, 142, 144, 145, 148, 150, 151
FIB Focused Ion Beam. 6

GMV Galbraith-McKee-Valença. 34

HIBE Hierarchical Identity-Based Encryption. 10, 40, 41

IBE Identity-Based Encryption. 4, 6, 10, 39–41, 49, 59, 126, 131

KSS Kachisa-Schaefer-Scott. 34, 35

MA Miller Algorithm. 44, 79, 87, 101, 110, 141, 150
MNT Miyaji-Nakabayashi-Takano. 33–35

PBC Pairing-Based Cryptography. 4, 6, 10, 42, 109, 110, 127, 128, 150
PC Personal Computer. 48, 101, 102, 131
PKC Public-key cryptography. 1–4
PKG Private Key Generator. 39–41, 126, 127
PKI Public-Key Infrastructure. 39, 126

RAM Random Access Memory. 47, 49, 52, 94, 131
RNS Residue Number System. 16
ROM Read-Only Memory. 49, 131

SCA Side-Channels Analysis. 6, 46, 101, 110, 129, 140, 151

USB Universal Serial Bus. 48

XOR eXclusive OR. 13, 47

List of Figures

List of Tables

Contents

Chapter 1

General introduction

The ubiquity of digital devices in our lives raises the question of the control of the data we generate. As revealed by Edward Snowden [GE13], numerous states worldwide have set up surveillance systems to globally harvest and analyse the data transiting through the internet. Above all, these revelations have shown the necessity to protect personal data against various entities, being (rogue) states, criminal organizations, advertisers... Cryptology is the science which is used to allow or to fight the surveillance. Cryptology is the science of secrets and has two sides: cryptography to protect data and cryptanalysis to defeat cryptography.

Contents

Under the term cryptography are gathered the techniques used to protect data at rest or in transit. Data protection implies at least one of the following features.

- Confidentiality: limit access to the data to the intended recipients only.
- Authenticity: ensure the identity of the communicating entities.
- Integrity: prevent the data from being altered.

In addition to these features, the widespread use of cryptography in modern communications is at the origin of the emergence of additional desired features.

- Privacy protection: the identity of the participants should be protected if they want to, as well as any data or actions associated with an identity.
- Usability: the cost to use security should be the lowest possible for all users (or they won't use it).
- Scalability: as more and more devices use cryptography, the complexity of cryptographic techniques employed should scale well with the number of devices using them.

Cryptanalysis on the contrary aims at preventing one or several of the features presented above. This goal can be achieved mathematically by exploiting a weakness in the algorithm or the assumptions it relies on. But it can also be achieved using the so called physical attacks which can exploit weaknesses of the algorithm during its execution.

1.1 Introduction to cryptography

Cryptography intends to protect data with at least one of the features described previously. To share secret data between two parties, they may agree on a secret whose knowledge is the only requirement to access the data. This scheme describes the family of the so-called symmetric-key algorithm. Sometimes, the two parties are not able to securely exchange a shared secret off-line but must do it through an insecure channel of communication. In this case, public-key algorithms are used (also called asymmetric-key algorithms). In cryptography, the key is the only secret value (when a secret is required) which prevent adversaries from altering the protected data. There is often a great temptation to hide the algorithm used to protect data but as stated by the Kerckhoff's principle [Ker83] this should not be the case. The principle recommends that the security of the system should rely on a changing parameter called the key and not on an invariant scheme. If not respected, if the algorithm is discovered all instances become trivially decipherable.

1.1.1 Symmetric-key cryptography

In a symmetric-key algorithm, the data is protected by a secret key shared only among the intended parties. The same key is used for both the encryption and the decryption. These algorithms offer a good compromise between the security offered, the size of the key and the efficiency of their computation. Advanced Encryption Standard (AES) [FIP01], Data Encryption Standard (DES) [FIP77], Twofish [SKW$^+$98], Serpent [ABK98] are examples of symmetric key algorithms.

Symmetric-key algorithms are either stream ciphers or block ciphers. In a stream cipher, the plaintext bytes are XORed with pseudorandom bytes derived from the key. The pseudorandom bytes form the keystream. The seed of the keystream is the symmetric-key, and this scheme is secure if no adversary can make a better prediction than a random guess for the next key byte knowing the previous keystream. Stream ciphers are often fast but the users have to be cautious when using them (*e.g.* they must not use the same key twice). In a block cipher, the plaintext is split into blocks with a fixed size. The blocks are then combined with a key with repetitive calls to a round function. How to chain blocks in order to encrypt messages with a bigger size than the block size can be a tricky problem and has to be done carefully, it is called mode of operation. Block ciphers can be achieved with several schemes, the most famous ones being Feistel networks (like the DES) or the substitution-permutation networks (like the AES).

The problem with a symmetric-key algorithm is to securely share the secret key among participants who can be far from each other and without any secure channel of communication among them.

1.1.2 Public-key cryptography

The public-key cryptography intends to overcome the latter difficulty of sharing a secret over an insecure channel of communication. The modern concept of Public-key cryptography (PKC) is due to Diffie and Hellman in 1976 [DH76]. Yet at that time, no algorithm was proposed to

achieve the features described in that paper. In 1978, the first public-key algorithm, RSA, was proposed in [RSA78]. It is still widely used mainly for encryption and signatures.

At the same level of security with respect to a symmetric-key algorithms, the public key is often much larger [BBB+06] (e.g. the AES 128-bits security is considered equivalent to the one of RSA 3072-bits).

RSA

The encrypting algorithm based on RSA works as follows:

1. Randomly choose p and q two different large prime numbers.
2. Compute $n = p \cdot q$.
3. Compute $\phi(n) = (p-1) \cdot (q-1)$.
4. Choose e such that $gcd(e, \phi(n)) = 1$ and $e < \phi(n)$.
5. Compute d such that $e \cdot d \equiv 1 \mod \phi(n)$.

The pair (n, e) is the public key. (d, p, q) is the private key. In order to encrypt a message M, one has to compute $C \equiv M^e \mod n$ where (n, e) is the public key of the intended recipient. In order to decrypt, the recipient computes $M \equiv C^d \mod n$.

The security of this system relies on two problems (not equivalent):

- The RSA problem: knowing the n and e previously defined, for a given C find M such that $M^e \equiv C \mod n$.
- The factorization problem: knowing n, find p and q.

Due to the efficiency of the algorithms to break RSA (cf. Section 2.9), the size of the keys does not size well with the security. The size of the public key for the 128-bits security level is 3072-bits [BBB+06]. For the 256-bits security level, the key size (for n) goes up to 15360-bits!

Elliptic Curves Cryptography

Other widespread techniques for PKC used Elliptic Curves, these techniques are called Elliptic Curves Cryptography. The use of elliptic curves for cryptography has been initiated in 1985 by Koblitz [Kob87] and Miller [Mil86b]. The cryptanalysis algorithms for Elliptic Curves Cryptography (ECC) are less efficient than for RSA, resulting in more interesting key sizes. At the 256-bits security level, the key size is 512-bits.

The main difference between RSA and ECC lies in the groups used by the algorithms. In RSA, the group formed by the integers modulo n is used whereas in ECC the group is an abelian group formed by the points on an algebraic curve. More details on ECC are given in Chapter 2.

An example of the use of ECC is the Diffie-Hellman key exchange over elliptic curves. Alice and Bob agree on an elliptic curve and on a point P on this curve. Alice chooses her secret a and transmits $[a]P = \underbrace{P + P + \cdots + P}_{a \text{ times}}$ to Bob. Bob chooses his secret b and transmits $[b]P$ to Alice. Now Alice can compute $[a]([b]P) = [ab]P$ and Bob can compute $[b]([a]P) = [ab]P$: they have both agreed on the same key $[ab]P$. The security of this scheme relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP) problem (detailed in Section 2.9.1) which states that knowing $[x]P$ and P, one cannot recover x.

Pairing-Based Cryptography

Initially, pairings have been used as a cryptanalysis method in order to attack ECC. A (very) short insight is provided below on how it works. Details are given in Chapter 2. Let G_1 and G_2 be groups of points on elliptic curves and G_T be a subgroup of a finite field such that e is a pairing (in particular $e : G_1 \times G_2 \to G_T$ is bilinear). Then the ECDLP problem of finding a when knowing $P \in G_1$ and $[a]P \in G_1$ can be transformed in the following way: let $x = e(P, Q)$ for some $Q \in G_2$, then $e([a]P, Q) = e(P, Q)^a = x^a$. a can be found with a Discrete Logarithm Problem (DLP) knowing x and x^a. This method is called the MOV [MOV93] or the FR [FR94] attack. The size of the field of which G_T is a subgroup depends on the original elliptic curve and more precisely on the embedding degree of the extension field for the coordinates of the points on the curve. As a consequence an elliptic curve used for ECC must have a big embedding degree. In 2000, Joux [Jou00] proposed to use curves with a moderate embedding degree in order to allow novel protocols such as a tripartite one-round Diffie-Hellman protocol. A year later, Boneh and Franklin [BF01] proposed an Identity-Based Encryption (IBE) scheme based on pairings. Since then, numerous novel protocols have been proposed which are based on pairings. Symmetric-key algorithms ensures data confidentiality and previous PKC allowed to ensure authenticity. Now with PBC, schemes exist that ensure the scalability, usability and the privacy protection of participants. Yet even if they allow new schemes otherwise impossible to do, pairings have the same key size efficiency as RSA.

1.2 Introduction to cryptanalysis

The goal of cryptanalysis is the opposite of cryptography, the opponent tries to prevent the data protection. The advances in cryptanalysis are of the utmost importance to the cryptographer since it allows him to improve the primitives used for cryptography. Details on some cryptanalysis techniques are proposed in Chapter 3.

1.2.1 Classical cryptanalysis

Cryptanalysis is often (but not always *e.g.* signature forgery, ciphertext decryption, reverse engineering) used to recover the secret key from a cryptographic algorithm. In our examples, let F be the cryptographic algorithm, k the secret key, P the plaintext (clear message) and $C = F(P, k)$ the ciphertext. As stated by Kerckhoff's principle [Ker83], the attacker should know F. According to its possibilities, the other parameters can be known or not. Among the possibilities, the attack can be

- ciphertext only: the attacker knows only C (in addition to F),
- known plaintext: the attacker has pairs of corresponding (P, C),
- chosen plaintext: the attacker knows the ciphertexts corresponding to chosen plaintexts,
- chosen ciphertext: the attacker knows the plaintexts corresponding to chosen ciphertexts,
- related-key: the attacker knows pairs of (P, C) (chosen or not) for two different keys with a known relation between them.

Attacks are always possible on an algorithm, for example with the brute force method (the attacker knows (P, C) and he tries all k until $C = F(P, k)$). But in practice, an attack is

achievable only if the attacker possesses the required computational power. The computational cost is often a mix of computation time, memory consumption and data $(P, C\dots)$ required. The designer of a cryptosystem dedicates his efforts so that no attack is more efficient than the brute force attack.

Brute force attack The brute force attack is the most basic one and the most expensive in computing resources. The attacker knows a pair (P, C) and she tries all keys k in the keyspace until he finds a match $C = F(P, k)$. In order to be secure, the keyspace must be big enough so that the time to explore it is orders of magnitude bigger than what is practically achievable. As specified in [BBB+06], is is estimated possible to brute force a key at the 80-bits security level.

Statistical cryptanalysis Historically, a lot of ciphers were not properly dealing with the patterns occurring in natural languages. A toy example is Caesar's cipher where each letter in the alphabet is replaced with the letter 3 positions later in lexicographic order. The problem in this scheme is that the frequency of the symbols is conserved with this transformation. As a consequence the letter 'e', the most frequent letter in English is transformed in a symbol which would have then the biggest number of apparitions. A frequency analysis can reveal which symbol corresponds to the letter 'e'. More generally, these attacks are possible when a pattern in the plaintexts is reproduced in the ciphertexts; it can happen even with modern algorithms (*e.g.* AES in Electronic Code Book (ECB) mode).

Differential cryptanalysis The differential cryptanalysis method intends to study the difference $C_2 - C_1 = F(P_2) - F(P_1)$ when $P_2 - P_1$ is known to the attacker. Discovered by Biham and Shamir [BS91] in the academic community, it is particularly efficient against symmetric-key ciphers. It is possible to adapt cryptographic algorithms in order to make them secure against this kind of attack. Yet differential cryptanalysis is still used in the context of fault attacks where a known fault created during the execution of the algorithm allows to exploit the difference in the ciphertexts.

Reduction to cryptographic problems In order to make a proof of security of an algorithm, the designer usually shows the equivalence between breaking the crypto algorithm and solving a well-known mathematical problem believed to be hard. These particular mathematical problems (or cryptographic problems) are for example the factorization problem, the DLP, the ECDLP... Interestingly the true difficulty to solve most of the problems with cryptographic interest is not well known [Imp95]. For example both the factorization problem and the DLP are NP hard but not NP complete. Details about the cryptographic problems used in Pairing Based Cryptography are given in Section 2.9.1.

1.2.2 Physical attacks

Even when an algorithm is considered secure mathematically, the computation of the algorithm on a chip can allow an attacker to retrieve the secret. The interaction of the computation with its physical environment allows the attacker to access intermediate values in the computation, invalidating the black-box model required for the security of the algorithm. These attacks are heavily implementation dependent.

Physical attacks can be divided into several families

- side-channel attacks (non-invasive attacks): the leaked secret information (timing, power consumption, EM radiation...) is only observed by the attacker,
- fault attacks (semi-invasive attacks): a fault is injected during the computation (with a clock glitch, a laser pulse, an EM pulse...) and alter the behaviour of the algorithm,
- invasive attacks: the computing chip is permanently altered (*e.g.* with Focused Ion Beam (FIB)) in order to probe a value or modify it.

Side-Channels Analyses (SCAs)

Side-channel attacks use the observations made by an attacker when the computing chip evaluates the cryptographic algorithm. A classic algorithm is the Square-and-Multiply algorithm used to compute an exponentiation. In this algorithm, at each iteration an additional multiplication is performed for each bit at 1 in the exponent. If the attacker is able to monitor the power consumption of the chip, she can observe that sometimes an iteration of the algorithm takes longer and the corresponding power consumption is higher. As a consequence, she can link these iterations with a bit at 1 in the exponent and finally retrieve the whole exponent even if it is a secret.

Fault attacks

In a fault attack, the attacker modifies the cryptographic algorithm with an external apparatus such as a laser, power glitches, clock glitches or with EM radiations. Such a perturbation can modify a data (modification of a value) or even the control flow (*e.g.* an instruction skip such as a branch skip). Often this perturbation has to be controlled as to ensure a minimum modification of the data (usually attackers prefer when the fault is a single-bit modification) and in order to avoid the destruction of the chip. These faults are often leveraged with differential cryptanalysis, it is the so called Differential Fault Attack (DFA). In this thesis we will implement fault attacks on pairing algorithms.

1.3 In this thesis

In this thesis we will focus on fault attacks on PBC. Following an intense research effort in the last decade, the time to compute a pairing is now reasonable (comparable to an RSA decryption) and even possible on low performance devices such as smartcards. Before a widespread use, the security of pairing implementations must be analysed, in particular their resistance to fault attacks. We focus on fault attacks because they are algorithm-specific and because they are extremely powerful to recover a secret. Previous works [PV06, WS07, EM09] have dealt with the subject but were not successful to propose an attack against Barreto-Naehrig (BN) curves, the best candidates for the 128-bit security level. Additionally, these papers analysed the resistance of pairings using small characteristic fields, now deprecated by recent works [Jou13, GKZ14]. A modern pairing protocol would probably use a BN curve with a large characteristic field and an asymmetric pairing. Contrary to what is claimed in [CKM14], protocols do exist that use asymmetric pairing (notably an IBE scheme proposed in the full version of [BF01]).

In Chapter 2, the mathematical background required to construct a pairing is explained as well as a detailed description of the mathematical security offered by pairings. In Chapter 3, precisons on fault attacks are given with a detailed description of our experimental set-up used to implement the fault attacks. The example of the calibration step is used as an illustration

of the different parameters at stake when doing a fault injection. The security of the Miller algorithm is examined in Chapter 4, first with a description of previous works on the matter followed by some refinements, notably a study of the efficiency of the countermeasures. Finally practical experiments explore the feasibility of the fault attacks proposed. The security of the final exponentiation (for large characteristic fields) is explored in Chapter 5 where we propose both theoretical fault attacks as well as practical experiments. Finally the feasibility of a fault attack on a complete pairing is studied in Chapter 6, where we demonstrate for the first time that such an attack is experimentally possible.

The following papers have been published, accepted or are currently submitted in the course of this thesis:

- **A DFA on AES Based on the Entropy of Error Distributions** [LRD+12] by Lashermes, Reymond, Dutertre, Fournier, Robisson, Tria at FDTC 2012,
- **Inverting the Final Exponentiation of Tate Pairings on Ordinary Elliptic Curves Using Faults** [LFG13] by Lashermes, Fournier, Goubin at CHES 2013,
- **Practical Validation of Several Fault Attacks against the Miller Algorithm** [EMFG+14] by El Mrabet, Fournier, Goubin, Lashermes and Paindavoine at FDTC 2014 (accepted),
- **A survey of Fault attacks in Pairing Based Cryptography** by El Mrabet, Fournier, Goubin, Lashermes in special issue of Cryptography and Communications (submitted).

Chapter 2

The design of a pairing based crypto-system

Where we describe the mathematical background to construct a pairing.

Contents

The inner workings of pairings are going to be described in this chapter. The theory of how to compute various pairings will be detailed. In Appendix A, an implementation of the algorithms can be found both in a functional representation and in C language.

2.1 Mathematical background

In this section are reminded some basic notions of Algebra in order to have a consistent notation through the rest of this document.

2.1.1 Algebra basics

Some definitions of simple algebraic structures are provided.

Definition 2.1.1 *Monoid*
*Let S be a set (finite or infinite). Let $\cdot$ be a binary operation defined on this set. We say that this $(S, \cdot)$ is a **monoid** if it satisfies the three properties:*

- *Closure: $\forall a, b \in S$, $a \cdot b \in S$.*
- *Associativity: $\forall a, b, c \in S$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*
- *Identity element: $\exists e \in S$ such that $\forall a \in S$ we have $e \cdot a = a \cdot e = a$.*

Definition 2.1.2 *Group*
*Let $(M, \cdot)$ be a monoid. We say that $(M, \cdot)$ is a **group** if $\forall a \in M$, $\exists b \in M$ such that $a \cdot b = b \cdot a = e$ (e is the identity element). b is called the inverse of a and is often noted a^{-1} (or $-a$ if the binary operation is $+$).*
We say that M is commutative (or abelian) if it has the additional property $\forall a, b \in M$, $a \cdot b = b \cdot a$.

Definition 2.1.3 *Ring*
*Let $(G, +)$ be an abelian group with the binary operation $+$ and corresponding identity element 0. We say that $(G, +, \cdot)$ is a **ring** with the two binary operations $+$ and $\cdot$ if the following properties are satisfied:*

- *$(G, +)$ is an abelian group.*

- *Closure for* $\cdot$*:* $\forall a, b \in G,\ a \cdot b \in G$
- *Associativity for* $\cdot$*:* $\forall a, b, c \in G,\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- *Identity element for* $\cdot$*:* $\exists 1 \in G$ *such that* $\forall a \in G$ *we have* $1 \cdot a = a \cdot 1 = a$
- *Distributivity:* $\forall a, b, c \in G,\ a \cdot (b + c) = a \cdot b + a \cdot c$ *and* $(a + b) \cdot c = a \cdot c + b \cdot c$.

We say that the ring G *is commutative if it has the additional property* $\forall a, b \in G,\ a \cdot b = b \cdot a$.

Definition 2.1.4 *Field*

Let $(R, +, \cdot)$ *be a commutative ring. We say that* $(R, +, \cdot)$ *is a* ***field*** *if it has the additional property of multiplicative inverse:* $\forall a \in R \backslash \{0\}$, $\exists a^{-1} \in R$ *such that* $a \cdot a^{-1} = a^{-1} \cdot a = 1$

Therefore a field $(F, +, \cdot)$ *consists of two abelian groups:* $(F, +)$ *and* $(F \backslash \{0\}, \cdot)$.

Definition 2.1.5 *Ideal*

Let $(R, +, \cdot)$ *be a commutative ring. We say that* $I \subset R$ *is an* ***ideal*** *if the following properties are satisfied:*

- $(I, +)$ *is a subgroup of* $(R, +)$.
- $\forall i \in I$, $\forall r \in R$ *then* $r \cdot i \in I$ *and* $i \cdot r \in I$

Definition 2.1.6 *Product of sets*

Let $S_1, S_2, \ldots, S_n$ *be arbitrary sets. We can construct a new set noted*

$$S_1 \times S_2 \times \ldots \times S_n$$

by simply taking the set of vectors $(s_1, s_2, \ldots, s_n)$ *with* $s_i \in S_i, \forall i \in [\![1, n]\!]$.

2.1.2 Functions

Definition 2.1.7 *Function*

Let P *and* I *be two arbitrary sets. A* ***function*** $f : P \to I$ *maps for all elements of* P *onto at most one element of* I*. One element* $i \in I$ *such that* $i = f(p), p \in P$ *is called the image of* p*. Similarly* p *is said to be a preimage of* i.

Definition 2.1.8 *Injective function*

A function $f : P \to I$ *is said to be* ***injective*** *if each element of* I *has at* ***most*** *one preimage. An injective function is also called a one-to-one function (different from a one-to-one correspondence, see definition later).*

Definition 2.1.9 *Surjective function*

A function $f : P \to I$ *is said to be* ***surjective*** *if each element of* I *has at* ***least*** *one preimage.*

Definition 2.1.10 *Bijective function*

A function $f : P \to I$ *is said to be* ***bijective*** *if it is injective and surjective, i.e. each element of* I *has exactly one preimage. A bijective function is also called a one-to-one correspondence.*

Definition 2.1.11 *Domain and codomain*

Let $f : P \to I$ *be a function. The set of inputs of* f *is called the* ***domain*** *and the set of outputs is called the* ***codomain***.

Definition 2.1.12 *Kernel*
Let $f : P \to I$ be a function. The kernel of f is the set:

$$\ker(f) = \{p \in P | f(p) = 0\}.$$

2.1.3 Equivalence relations and classes

Definition 2.1.13 *Equivalence relation*
A binary relation $\sim$ on a set S is said to be an equivalence relation if it has the following properties $\forall a, b, c \in S$:

- *Reflexivity: $a \sim a$.*
- *Symmetry: if $a \sim b$ then $b \sim a$.*
- *Transitivity: if $a \sim b$ and $b \sim c$ then $a \sim c$.*

Definition 2.1.14 *Equivalence class*
*Let $\sim$ be an equivalence relation on the set S. Let a be an element of S, then the **equivalence class** of a under $\sim$ is the set $X_a = \{b \in S | a \sim b\}$. We note that $X_a = X_b$ if and only if $a \sim b$.*

Proposition 2.1.15 *Partition of a set*
The set of equivalence classes of a set S under $\sim$ is a partition of S. It means that every element of S is in one and only one equivalence class.

2.2 Finite fields

2.2.1 Definitions

Definition 2.2.1 $\mathbb{Z}/n\mathbb{Z}$
Let $a, b \in \mathbb{Z}$. Let $\sim$ be the equivalence relation defined by $a \sim b \iff a \equiv b \mod n$. Then $\mathbb{Z}/n\mathbb{Z}$ is defined as the commutative ring of the equivalence classes under $\sim$.
As a shortcut, $\mathbb{Z}/n\mathbb{Z}$ is often noted as the ring of elements $\{0, 1, 2, \cdots, n-1\}$ with the modular addition and multiplication.

Definition 2.2.2 $\mathbb{Z}/p\mathbb{Z}$
The ring $\mathbb{Z}/p\mathbb{Z}$ is a field if and only if p is a prime. This field has exactly p elements and is noted $\mathbb{F}_p$.
***Proof:** Let $a \in \mathbb{Z}/p\mathbb{Z}$, then $\exists b \in \mathbb{Z}/p\mathbb{Z}$ such that $a \cdot b \equiv 1 \mod p$ if and only if $gcd(a, p) = 1$. Iff p is prime, this property is verified for all elements of $\mathbb{Z}/p\mathbb{Z}$.*

Definition 2.2.3 *Extension fields over $\mathbb{F}_p$*
A finite field of cardinality q is noted $\mathbb{F}_q$. In particular the field $\mathbb{Z}/p\mathbb{Z}$ can be noted $\mathbb{F}_p$. Finite fields can be created with a cardinal a power of a prime in the following way. Let $f(x) \in \mathbb{F}_p[x]$ be an irreducible polynomial of degree d with coefficients in $\mathbb{F}_p$. $\mathbb{F}_p[x]/f(x)$ forms a finite field with p^d elements with additions and multiplication $\mod f(x)$. The new field $\mathbb{F}_p[x]/f(x)$ is an extension field (cf. Definition 2.3.1) of $\mathbb{F}_p$ and an element can be represented as a vector with d coordinates in $\mathbb{F}_p$. This extension field can be noted $\mathbb{F}_{p^d}$ with $f(x)$ implicit. Details on extension fields are given in Section 2.3.

We call binary fields the fields of the form $\mathbb{F}_{2^n}$ for some n, ternary fields the ones $\mathbb{F}_{3^n}$ and large characteristic fields the ones $\mathbb{F}_{p^n}$ with p a large prime.

2.2.2 Computations over finite fields

Until now, the construction of finite fields has been introduced in its mathematical form. Yet, we are interested in the actual computation of pairings and therefore in the operations over finite fields. A computation is done on a computing unit which imposes some data structures. We are interested in microcontrollers which deal with data word per word. Let B be the base value for data handling ($B = 2^{32}$ on a 32-bit computing unit). In order to achieve a strong enough security level, often $p > B$. The number of machine words to store a value in $\mathbb{F}_p$ is noted $M = \lceil \log_2(p)/\log_2(B-1) \rceil$. As an example for the 128-bits security level, p is of size approximatively 256 bits which is $M = 9$ machine words (if $B = 2^{32}$).

As a consequence, every element $X \in \mathbb{F}_p$ can be represented as

$$X = \sum_{i=0}^{n-1} B^i x_i.$$

In most algorithms, data are handled in the vector-based representation where $(x_0, x_1, \ldots, x_{n-1})$ represents X. The i coordinate of X in this representation is noted either $X[i]$ or x_i. The algorithms used to manipulate data in $\mathbb{F}_p$ should be optimized for this vector representation. Examples in C are shown in Appendix A.2.

Modular addition

The implementations of additions in a finite field depend on the characteristic of the field and have to take into account the modulo operation. In binary fields, a modular addition (and modular subtraction as well) is equivalent to a simple eXclusive OR (XOR). There is no carry propagation and no modular reduction step. In ternary fields, the addition is still quite simple since an addition in $\mathbb{F}_{3^n}$ consists of n parallel additions modulo 3: subtraction and multiplication by two in $\mathbb{F}_{3^n}$ are easy too (the carry propagation is limited); details on such implementations can be found in [HPS02].

In prime fields, modular additions and subtractions are more complex. An addition is divided into two steps: first a classic addition with carry propagation and then a reduction step in order to have our result within the field range.

The algorithm for a modular addition in $\mathbb{F}_p$ (Algorithm 1) requires binary addition, subtraction and comparison algorithms (cf. Code A.2).

Algorithm 1: Modular addition in $\mathbb{F}_p$.

Data: $a, b \in \mathbb{F}_p$
Result: $c = a + b \mod p \in \mathbb{F}_p$
$a' \leftarrow a \in \mathbb{Z}$;
$b' \leftarrow b \in \mathbb{Z}$;
$c' \leftarrow a' + b' \in \mathbb{Z}$;
if $c' < p$ **then**
  $c \leftarrow c' \in \mathbb{F}_p$;
else
  $c \leftarrow c' - p \in \mathbb{F}_p$;
end
return c

Modular multiplication

There is an extensive literature on finite field multipliers. Performing the modular multiplication $A \times B \bmod N$ on two large integers A and B (i.e. with $n = |N|$ large) is often the most time-critical operation in a pairing calculation. Lots of variants exist in order to perform multiplications in $\mathbb{F}_p$, some of them are presented below.

The "first generation" of optimised multipliers involved calculating the multiplication $U = A \times B$ and then performing the reduction that can be written as:

$$U \bmod N = U - qN \text{ with } q = \left\lfloor \frac{U}{N} \right\rfloor \tag{2.1}$$

The large number multiplications were first optimised using one of the following methods.

- *The Schoolbook multiplication* consists in successive additions of partial products using, for example, a classic double and add algorithm where the second operand is processed bit-wise or word-wise. The reduction step is either done after each addition of partial products or at the very end. For small characteristic fields, this approach is often used because they can easily be interleaved with reduction steps.
- *The Booth multiplication* [Boo51] considers that instead of an addition for each '1' in the second operand, we only need one addition and one subtraction for each group of '1' which is particularly useful when the digits can be organised into groups of '1's.
- *The Karatsuba multiplication*, instead of multiplying two large numbers of size n bits, uses three multiplications on data of size $\frac{n}{2}$, plus some additions. More generally, in the Toom-Cook algorithm, the operands are divided into k parts instead of only two.

Early techniques for accelerating the reduction phase were based on one of the following techniques:

- *Brickwell's method* [Bri82] where "delayed carry save adders" were used to do the n-bit reduction.
- *Sedlak's method* [Sed88] where the reduction is done using large carry look-ahead adders and large shifters.

More recent techniques involve the interleaving between partial multiplications and reductions. The interleaving is mainly based on the fact that the operands A and/or B are accessed by bits or words. The two main approaches, based on Barrett's or Montgomery's principles [MVOV97], are hereby introduced. Another possibility is Balkely's modular multiplication [Bla83] but one of the operands is accessed bit per bit which is too slow.

Barrett reduction In [Bar87], Barrett observes that the quotient q from Equation (2.1) can be rewritten as $q = \left\lfloor \frac{\frac{U}{2^{n-1}} \cdot \frac{2^{2n}}{N}}{2^{n+1}} \right\rfloor$ which can in turn be estimated by $\hat{q} = \left\lfloor \frac{\left\lfloor \frac{U}{2^{n-1}} \right\rfloor \left\lfloor \frac{2^{2n}}{N} \right\rfloor}{2^{n+1}} \right\rfloor$. The factor $\frac{2^{2n}}{N}$ needs to be calculated only once.

Montgomery reduction and multiplication Montgomery's technique [Mon85] is based on the following observation: given an integer R such that $\gcd(R, N) = 1$ and $N' = -\frac{1}{N} \bmod R$, then the following equivalence holds:

$$UR^{-1} \mod N \equiv \frac{U + (UN' \mod R)N}{R}.$$

Thus, when in the Montgomery domain, with R as the residue, the reduction simply comes down to adding a multiple of N. The multiplication can be done in the Montgomery domain and be interleaved with the reduction. [KKAKJ96] gives a comparison between various implementations of the Montgomery multiplication. These implementations vary by how the multiplication and reduction steps are interleaved. Of the 5 algorithms analysed in the latter paper, the "Coarsely Integrated Operand Scanning" (CIOS) methods was declared the best. Due to the easy scaling of this algorithm when p (the modulus) is changed, this algorithm has been chosen for the modular multiplication in our case (cf. Code A.4 an Algorithm 2).

Algorithm 2: Modular Montgomery multiplication in $\mathbb{F}_p$. Word based algorithm. $r \cdot r^{-1} - p \cdot p' = 1$ with r the Montgomery residue.

Data: $a, b \in \mathbb{F}_p$
Result: $c = a \cdot b \cdot r^{-1} \mod p \in \mathbb{F}_p$
for $i \leftarrow 0$ **to** $M + 1$ **do**
  $t[i] \leftarrow 0$;
end
for $i \leftarrow 0$ **to** $M - 1$ **do**
  $C \leftarrow 0$;
  for $j \leftarrow 0$ **to** $M - 1$ **do**
    $(C, S) \leftarrow t[j] + a[j] \cdot b[j] + C$;
    $t[j] \leftarrow S$;
  end
  $(C, S) \leftarrow t[M] + C$;
  $t[M] \leftarrow S$;
  $t[M + 1] \leftarrow C$;
  $m \leftarrow t[0] \cdot p'[0] \mod B$;
  $(C, S) \leftarrow t[0] + m \cdot p[0]$;
  for $j \leftarrow 1$ **to** $M - 1$ **do**
    $(C, S) \leftarrow t[j] + m \cdot p[j] + C$;
    $t[j - 1] \leftarrow S$;
  end
  $(C, S) \leftarrow t[M] + C$;
  $t[M + 1] \leftarrow S$;
  $t[M] \leftarrow t[M + 1] + C$;
end
if $t[0 \,..\, M - 1] < p$ **then**
  $c \leftarrow t[0 \,..\, M - 1]$;
else
  $c \leftarrow t[0 \,..\, M - 1] - p$;
end
return c

Comparison of the multiplication algorithms Table 2.1 shows a comparison among some of the above mentioned algorithms, where n is the size of the operands. Even if the asymptotic complexity is important as field sizes increase with security requirements, one has to evaluate the particular cost of each implementation on a given platform.

A comparison between Barrett and Montgomery reductions can be found in [BGV94], with an advantage to Montgomery for modern security requirements since the algorithm scales better

Table 2.1: Asymptotic complexities of modular multiplication algorithms

Algorithm	*Complexity*
Schoolbook	$\mathcal{O}(n^2)$
Montgomery	$\mathcal{O}(n^2)$
Karatsuba	$\mathcal{O}(n^{1.585})$
Toom-Cook	$\mathcal{O}(n^{\frac{\log(2k-1)}{\log k}})$

with the field size. Some comparisons on variants of Barrett techniques can be found in [D+98]. The modular multiplication algorithm is the bottleneck in terms of performances when computing a pairing since it is expensive and called repeatedly (> 10000 times, cf. Table 3.1).

Residue Number System (RNS)

Latest implementations tout the use of RNS ([CDF+11],[YFCV11]). The principle of RNS is to decompose numbers into a vector of smaller numbers. In order to calculate $U \mod N$, we choose a vector (called basis) $\mathbb{B} = \{b_0, b_1, \cdots, b_k\}$, with b_i pairwise co-primes and $\prod_{i=0}^{k} b_i > N$, and we write $U_{\mathbb{B}} = (U_0 = U \mod b_0, U_1 = U \mod b_1, \cdots, U_k = U \mod b_k)$. Modular operations can hence be performed on the smaller U_i independently from each other. In an RNS setting, the choice of the basis is critical in order to improve performances.

Even if RNS lowers the cost of the multiplication, the reduction step is still expensive. In [CDF+11] the authors use a "lazy" reduction [Sco07] which requires more memory though.

A major disadvantage of RNS is the difficulty (and circuit cost) of computing an inverse. As recent advances [BT13] have been made regarding this problem, there is hope that it will not remain difficult in the future.

Variations on the RNS representation has also been proposed in [BIP05].

2.3 Extension fields

2.3.1 Definitions

Definition 2.3.1 *Extension field.*
The extension field L of a field K is a field which contains K as a subfield.

Definition 2.3.2 *Extension degree*
Let L be an extension field of K. Then L is a vector space over K. The dimension of this vector space is called the extension degree. If this degree is finite, the extension is called finite.

For efficiency reasons when computing pairings, we prefer to use extension degrees of the form $2^i \cdot 3^j$. It allows to build this extension as a tower of extensions of degree 2 (called quadratic extensions) and of degree 3 (called cubic extensions) which have simpler operation formulae. The resulting scheme is called a tower extension of finite fields.

In order to build our extension fields, we use the rupture field of an irreducible polynomial (cf. Definition 2.3.3).

Definition 2.3.3 *Rupture field of an irreducible polynomial $P(X)$*
A rupture field of the irreducible polynomial $P(X) \in K[X]$ is the extension field $L = K[s]$ where $s \notin K$ is a root of $P(X)$. It is the smallest extension of K containing s. L is isomorphic to $K[X]/(P(X))$. The extension degree is equal to the degree of $P(X)$.

Example $\mathbb{C} = \mathbb{R}[X]/(X^2+1)$

An element of the rupture field can be seen as a polynomial in s. In our example, an element $z \in \mathbb{C}$ can be written $z = x + i \cdot y$ with $x, y \in \mathbb{R}$.

The additive and multiplicative laws are easily computed on a rupture field. The addition is simply the polynomial addition (term to term addition in K). The multiplication result of two elements $x, y \in L = K[a]$ can be seen as the remainder of $x \cdot y$ as polynomial in the Euclidean division by the polynomial $P(X)$. The inverse can be computed with the extended Euclidean algorithm.

Definition 2.3.4 *Algebraic closure*
The algebraic closure of a finite field $\mathbb{F}_p$ is the field containing all the extension fields of $\mathbb{F}_p$ and is noted $\overline{\mathbb{F}_p}$.

$$\overline{\mathbb{F}_p} = \bigcup_{i=1}^{\infty} \mathbb{F}_{p^i}. \tag{2.2}$$

Definition 2.3.5 *Characteristic of a field*
The characteristic of a field is the smallest integer n such that $[n]1 = \underbrace{1+1+\ldots+1}_{n \text{ times}} = 0$ where 1 and 0 are sum and product identities. The characteristic of an extension field $\mathbb{F}_{p^k}$, with p prime, is p.

Definition 2.3.6 *Roots of unity*
Let K be any field, then the n^{th} roots of unity in K form a group noted $\mu_n = \{x \in K | x^n = 1\}$.

In the following, the formulae to compute the additions, squarings, multiplications and inverses in quadratic and cubic equations are presented. The corresponding algorithms and C codes are proposed in the Appendix A. Adaptations have to be made for specific platforms based on the requirements in memory, computing resources, latency *etc.*

2.3.2 Computations on quadratic extensions [BGDM+10]

A quadratic extension is an extension of degree 2. In our computation, we write $L = K/(X^2 - \beta)$. If we call u a root of $X^2 - \beta$ (an irreducible polynomial), an element $z \in L$ can be noted $z = z_0 + z_1 \cdot u$ with $z_0, z_1 \in K$. Since u is a root of $X^2 - \beta$, $u^2 = \beta$.

We note the cost of an addition, a multiplication, a squaring and an inverse in the field K with A_K, M_K, S_K and I_K respectively. The relative costs of these operations depends on the implementations. For quick evaluations, we usually consider $M_K = 10A_K$, $S_K = 0.9M_K$ (cf. Section 2.3.4 and Section 3.2.2) and $I_K \gg M_K$.

Addition algorithm

The addition is simply a polynomial addition (cf. Algorithm 9 in Appendix A). The cost is $A_L = 2A_K$.

Multiplication algorithm

The multiplication algorithm uses the Karatsuba method to limit the number of K multiplication (multiplication in the field K) calls (cf. Algorithm 10 in Appendix A).

$$(x_0 + x_1 u) \cdot (y_0 + y_1 u) = (x_0 y_0 + x_1 y_1 \beta) + ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1) \cdot u. \tag{2.3}$$

The cost is $M_L = 4M_K + 5A_K$.

Squaring algorithm

Squaring is a particular case of multiplication (cf. Algorithm 11 in Appendix A).

$$(x_0 + x_1 u)^2 = (x_0^2 + x_1^2\beta) + \left((x_0 + x_1)^2 - x_0^2 - x_1^2\right) \cdot u. \tag{2.4}$$

The cost is $S_L = 3S_K + M_K + 4A_K$.

Inverse algorithm

The inverse algorithm (cf. Algorithm 12 in Appendix A) is based on the fact that

$$(x_0 + x_1 \cdot u) \cdot (x_0 - x_1 \cdot u) = x_0^2 - x_1^2 \cdot u^2 = x_0^2 - x_1^2 \cdot \beta. \tag{2.5}$$

Which can equally be written as

$$(x_0 + x_1 \cdot u)^{-1} = \frac{x_0 - x_1 \cdot u}{x_0^2 - x_1^2 \cdot \beta}. \tag{2.6}$$

The cost is $I_L = I_K + 3M_K + 2S_K + A_K$. As a summary, the costs of the operations for the quadratic extension are shown on Table 2.2.

Table 2.2: Cost for the quadratic extension.

	I_K	S_K	M_K	A_K
I_L	1	2	3	1
S_L	0	3	1	4
M_L	0	0	4	5
A_L	0	0	0	2

2.3.3 Computations on cubic extensions [BGDM+10]

A cubic extension is an extension of degree 3. In our computation, we redefine $L = K/(X^3 - \xi)$. If we call v a root of $X^3 - \xi$ (an irreducible polynomial), an element $x \in L$ can be noted $x = x_0 + x_1 \cdot v + x_2 \cdot v^2$ with $x_0, x_1, x_2 \in K$. Since v is a root of $X^3 - \xi$, $v^3 = \xi$. We note the cost of an addition, a multiplication, a squaring and an inverse in the field K with A_K, M_K, S_K and I_K respectively.

Addition algorithm

Again, the addition is a polynomial addition (cf. Algorithm 13 in Appendix A). The cost is $A_L = 3A_K$.

Multiplication algorithm

The multiplication algorithm uses the Karatsuba method to limit the number of K multiplication calls (cf. Algorithm 14 in Appendix A).

$$\begin{aligned}(x_0 + x_1 v + x_2 v^2)(y_0 + y_1 v + y_2 v^2) =& x_0 y_0 + x_1 y_2 \xi + x_2 y_1 \xi \\ &+ (x_0 y_1 + x_1 y_0 + x_2 y_2 \xi) v \\ &+ (x_0 y_2 + x_1 y_1 + x_2 y_0) v^2 \\ =& \left((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2\right) \xi + x_0 y_0 \\ &+ \left((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1 + x_2 y_2 \xi\right) v \\ &+ \left((x_0 + x_2)(y_0 + y_2) - x_0 y_0 - x_2 y_2 + x_1 y_1\right) v^2\end{aligned} \tag{2.7}$$

The cost is $M_L = 8M_K + 15A_K$.

Squaring algorithm

The squaring can be a bit optimized compared to the multiplication (cf. Algorithm 15 in Appendix A).

$$\begin{aligned}(x_0 + x_1 v + x_2 v^2)^2 =& x_0^2 + 2x_1 x_2 \xi \\ &+ (x_2^2 \xi + 2x_0 x_1) v \\ &+ (x_1^2 + 2x_0 x_2) v^2\end{aligned} \tag{2.8}$$

The cost is $S_L = 3S_K + 5M_K + 6A_K$.

Inverse algorithm

The inverse is y, the solution to the equation

$$(x_0 + x_1 \cdot v + x_2 v^2) \cdot (y_0 + y_1 \cdot v + y_2 \cdot v^2) = 1. \tag{2.9}$$

Which gives

$$t = \xi^2 x_2^3 - 3\xi x_0 x_1 x_2 + \xi x_1^3 + x_0^3 \tag{2.10}$$

$$y_0 = \frac{x_0^2 - \xi x_1 x_2}{t} \tag{2.11}$$

$$y_1 = \frac{\xi x_2^2 - x_0 x_1}{t} \tag{2.12}$$

$$y_2 = \frac{x_1^2 - x_0 x_2}{t} \tag{2.13}$$

The algorithm is shown in Algorithm 16. The cost is $I_L = I_K + 3S_K + 13M_K + 5A_K$. As a summary, the costs of the operations for the cubic extension are shown on Table 2.3.

2.3.4 Example: constructing tower extensions for $\mathbb{F}_{p^{12}}$

The field $\mathbb{F}_{p^{12}}$ is often used in BN curves (cf. Section 2.5.7). Since $12 = 2 * 2 * 3$, it can be constructed with a tower extension from quadratic and cubic extensions. There are three possibilities for this tower extension. For each possibility, we can evaluate the cost of the operations in $\mathbb{F}_{p^{12}}$ in terms of $\mathbb{F}_p$ operations calls. This evaluation is done by considering Table 2.2 and Table 2.3 as matrices and by multiplying them. The non commutativity of matrices implies that the different tower extensions have different costs.

Table 2.3: Cost for the cubic extension.

	I_K	S_K	M_K	A_K
I_L	1	3	13	5
S_L	0	3	5	6
M_L	0	0	8	15
A_L	0	0	0	3

Tower $12 = 2 \cdot 2 \cdot 3$

The tower extension is constructed as follows:

$$\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - \beta) \tag{2.14}$$
$$\mathbb{F}_{p^4} = \mathbb{F}_{p^2}[X]/(X^2 - \xi) \tag{2.15}$$
$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^4}[X]/(X^3 - \gamma) \tag{2.16}$$

The costs of the operations for this tower extension are represented in Table 2.4.

Table 2.4: Cost for the tower extension $2 \cdot 2 \cdot 3$.

	$I_{\mathbb{F}_p}$	$S_{\mathbb{F}_p}$	$M_{\mathbb{F}_p}$	$A_{\mathbb{F}_p}$
$I_{\mathbb{F}_{p^{12}}}$	1	35	246	511
$S_{\mathbb{F}_{p^{12}}}$	0	27	101	249
$M_{\mathbb{F}_{p^{12}}}$	0	0	128	300
$A_{\mathbb{F}_{p^{12}}}$	0	0	0	12

Tower $12 = 2 \cdot 3 \cdot 2$

The tower extension is constructed as follows:

$$\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - \beta) \tag{2.17}$$
$$\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[X]/(X^3 - \xi) \tag{2.18}$$
$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[X]/(X^2 - \gamma) \tag{2.19}$$

The costs of the operations for this tower extension are represented in Table 2.5.

Table 2.5: Cost for the tower extension $2 \cdot 3 \cdot 2$.

	$I_{\mathbb{F}_p}$	$S_{\mathbb{F}_p}$	$M_{\mathbb{F}_p}$	$A_{\mathbb{F}_p}$
$I_{\mathbb{F}_{p^{12}}}$	1	29	200	402
$S_{\mathbb{F}_{p^{12}}}$	0	27	101	241
$M_{\mathbb{F}_{p^{12}}}$	0	0	128	310
$A_{\mathbb{F}_{p^{12}}}$	0	0	0	12

Tower $12 = 3 \cdot 2 \cdot 2$

The tower extension is constructed as follows:

$$\mathbb{F}_{p^3} = \mathbb{F}_p[X]/(X^3 - \beta) \tag{2.20}$$
$$\mathbb{F}_{p^6} = \mathbb{F}_{p^3}[X]/(X^2 - \xi) \tag{2.21}$$
$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[X]/(X^2 - \gamma) \tag{2.22}$$

The costs of the operations for this tower extension are represented in Table 2.6.

Table 2.6: Cost for the tower extension $3 \cdot 2 \cdot 2$.

	$I_{\mathbb{F}_p}$	$S_{\mathbb{F}_p}$	$M_{\mathbb{F}_p}$	$A_{\mathbb{F}_p}$
$I_{\mathbb{F}_{p^{12}}}$	1	27	189	386
$S_{\mathbb{F}_{p^{12}}}$	0	27	101	234
$M_{\mathbb{F}_{p^{12}}}$	0	0	128	330
$A_{\mathbb{F}_{p^{12}}}$	0	0	0	12

As a consequence, the best tower extension depends on the relative costs of the different operations in $\mathbb{F}_p$. The choice of the tower extension must be decided according to each specific application. In our own implementation, we choose the tower extension $2 \cdot 3 \cdot 2$.

2.4 Elliptic curves

In this section a practical presentation of elliptic curves is given. For more complete explanations, the reader is encouraged to read [BSS99, Sil09].

2.4.1 Definitions

Definition 2.4.1 *Elliptic curve*

An elliptic curve over a field K is a projective nonsingular algebraic curve of genus 1*. It can be written as the cubic curve*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.23}$$

together with the point at infinity 0_∞*.*

If the field characteristic is not 2 (division by 2 possible), we can define

$$y' = y + \frac{a_1x}{2} + \frac{a_3}{2}. \tag{2.24}$$

Then

$$y'^2 = (y + \frac{a_1x}{2} + \frac{a_3}{2})^2 \tag{2.25}$$
$$= y^2 + a_1xy + a_3y + \frac{a_1^2}{4}x^2 + \frac{a_1a_3}{2}x + \frac{a_3^2}{4} \tag{2.26}$$
$$= x^3 + a_2x^2 + a_4x + a_6 + \frac{a_1^2}{4}x^2 + \frac{a_3^2}{4} + \frac{a_1a_3}{2}x \tag{2.27}$$
$$= x^3 + \left(a_2 + \frac{a_1^2}{4}\right)x^2 + \left(a_4 + \frac{a_1a_3}{2}\right)x + a_6 + \frac{a_3^2}{4} \tag{2.28}$$
$$= x^3 + b_2x^2 + b_4x + b_6, \tag{2.29}$$

with

$$b_2 = a_2 + \frac{a_1^2}{4}$$
$$b_4 = a_4 + \frac{a_1 a_3}{2}$$
$$b_6 = a_6 + \frac{a_3^2}{4}.$$

Additionally, if the field characteristic is not 3 (division by 3 allowed), a new simplification is possible with the variable change

$$x' = x + \frac{b_2}{3}. \tag{2.30}$$

Then the curve equation can be rewritten

$$y'^2 = x'^3 + Ax' + B. \tag{2.31}$$

Which can be seen by observing that

$$x'^3 + Ax' + B = \left(x + \frac{b_2}{3}\right)^3 + A\left(x + \frac{b_2}{3}\right) + B \tag{2.32}$$
$$= x^3 + b_2 x^2 + \left(A + \frac{b_2^2}{3}\right)x + \frac{b_2^3}{3^3} + A\frac{b_2}{3} + B. \tag{2.33}$$

Therefore

$$A = b_4 - \frac{b_2^2}{3}$$
$$B = b_6 - \frac{b_2 b_4}{3}.$$

The elliptic curve equation $E : y^2 = x^3 + ax + b$ is called the short Weierstrass equation. The discriminant of this curve is

$$\Delta_E = -16(4a^3 + 27b^2) \tag{2.34}$$

and must be different than 0. When K is a finite field ($\mathrm{card}\,(K) = p$), the cardinality of $E(K)$ is finite and can be bounded by Hasse's theorem

$$|\mathrm{card}\,(E(K)) - (p+1)| \leq 2\sqrt{p}. \tag{2.35}$$

This theorem shows that the cardinality of $E(K)$ is of the same order of magnitude as the cardinal of K.

2.4.2 Operations on an elliptic curve

The points on the elliptic curve $E(K)$ (including 0_∞) form an abelian group. The formulae for the group operations are given below (see [Yu11] for another description). From now on we are in the specific case of a short Weierstrass equation (field characteristic not equal to 2 or 3): $E : y^2 = x^3 + ax + b$.

In practice, there is a trade-off to make between memory consumption and computation duration and whether the designer chooses or not to store intermediate values in memory for later reuse.

Coordinate systems

A point on the curve can be expressed in various coordinate systems, each one has specific costs in terms of curve operations. The three coordinate systems mostly used are:

- Affine coordinates: the system implicitly used until now, (x, y) is a point on the curve if it satisfies $y^2 = x^3 + ax + b$.
- Projective coordinates: $(x : y : z)$ can be mapped to the affine point $(\frac{x}{z}, \frac{y}{z})$ if $z \neq 0$. The points with $z = 0$ represent the point at infinity 0_∞. The elliptic curve equation becomes $zy^2 = x^3 + axz^2 + bz^3$.
- Jacobian coordinates: $(x : y : z)$ can be mapped to the affine point $(\frac{x}{z^2}, \frac{y}{z^3})$ if $z \neq 0$. The points with $z = 0$ represent the point at infinity 0_∞. The elliptic curve equation becomes $y^2 = x^3 + axz^4 + bz^6$.

Projective and jacobian coordinates introduce more redundancy in a point representation, several points in these coordinates are equivalent, they map to the same affine point. Other possible coordinate systems include Edwards coordinates [BL07], Hessian coordinates [JQ01], Jacobi quartic coordinates [BJ02]. For some calculations, mixed representations can be useful. A mixed addition is an addition of one point in jacobian coordinates (for example) with one in affine coordinates. It can be performed by taking the jacobian addition formulae and setting $Z = 1$ for the affine point. The following formulae are given for affine and jacobian coordinates with points on the short Weierstrass equation

$$E : y^2 = x^3 + ax + b. \tag{2.36}$$

The cost of an addition, a multiplication, a squaring and an inverse in the field K are noted A_K, M_K, S_K and I_K respectively. The costs of point addition and point doubling are noted PA_K and PD_K.

Formulae for point addition

The point $P_3 = P_1 + P_2$ with $P_1 \neq \pm P_2$ is computed. Geometrically, the line passing through P_1 and P_2 intersect E at exactly one other point: $-P_3$, the mirror image of P_3 by the X axis.

Affine coordinates P_1, P_2 and P_3 have coordinates (X_1, Y_1), (X_2, Y_2) and (X_3, Y_3) respectively. Let $l(x, y)$ be the line passing through P_1, P_2 and $-P_3$. Then

$$l(x, y) : y = \lambda(x - X_1) + Y_1, \tag{2.37}$$

with $\lambda = \frac{Y_2 - Y_1}{X_2 - X_1}$ ($X_2 \neq X_1$ since $P_2 \neq \pm P_1$). Let $R(X_3, -Y3) = -P_3$, the R satisfies

$$\begin{cases} y &= \lambda\left(x - X_1\right) + Y_1 \\ y^2 &= x^3 + ax + b. \end{cases} \tag{2.38}$$

Therefore

$$\left(\lambda\left(x - X_1\right) + Y_1\right)^2 = x^3 + ax + b \tag{2.39}$$

$$x^3 - \lambda^2 x^2 + \left(a + 2\lambda^2 X_1 - 2\lambda Y_1\right) x + \left(b - \lambda^2 X_1^2 + 2\lambda X_1 Y_1 - Y_1^2\right) = 0. \tag{2.40}$$

This equation of degree 3 has three known solutions: X_1, X_2 and X_3. As a consequence, Equation (2.39) is equivalent to

$$(x - X_1)(x - X_2)(x - X_3) = 0. \tag{2.41}$$

By identification of the degree 2 term,

$$X_1 + X_2 + X_3 = \lambda^2 \tag{2.42}$$

Finally,

$$\begin{cases} X_3 &= \lambda^2 - X_1 - X_2 \\ Y_3 &= -\left(\lambda\left(X_3 - X_1\right) + Y_1\right) \end{cases} \tag{2.43}$$

are the formulae for point addition in affine coordinates. The cost is $PA_K = I_k + S_K + M_K + 7A_K$.

Jacobian coordinates The previous equations (Equation (2.43)) are reused with a change of variables.

$$\frac{X_3}{Z_3^2} = \left(\frac{\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}}{\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}}\right)^2 - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2}, \tag{2.44}$$

$$\frac{Y_3}{Z_3^3} = \left(\frac{\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}}{\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}}\right) \cdot \left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Y_1}{Z_1^3}, \tag{2.45}$$

which gives

$$\frac{X_3}{Z_3^2} = \frac{\left(Y_2Z_1^3 - Y_1Z_2^3\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right) \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^2}{Z_1^2Z_2^2\left(X_2Z_1^2 - X_1Z_2^2\right)^2},$$

$$\begin{aligned} \frac{Y_3}{Z_3^3} =& \frac{\left(Y_2Z_1^3 - Y_1Z_2^3\right) \cdot Z_2^2X_1\left(X_2Z_1^2 - X_1Z_2^2\right)^2}{Z_1^3Z_2^3\left(X_2Z_1^2 - X_1Z_2^2\right)^3} \\ &- \frac{\left(Y_2Z_1^3 - Y_1Z_2^3\right) \cdot \left[\left(Y_2Z_1^3 - Y_1Z_2^3\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right) \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^2\right]}{Z_1^3Z_2^3\left(X_2Z_1^2 - X_1Z_2^2\right)^3} \\ &- \frac{Y_1Z_2^3\left(X_2Z_1^2 - X_1Z_2^2\right)^3}{Z_1^3Z_2^3\left(X_2Z_1^2 - X_1Z_2^2\right)^3}. \end{aligned}$$

Z_3 is chosen in order to simplify the formulae.

$$Z_3 = Z_1Z_2\left(X_2Z_1^2 - X_1Z_2^2\right) \tag{2.46}$$

giving for X_3

$$X_3 = \left(Y_2Z_1^3 - Y_1Z_2^3\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right) \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^2, \tag{2.47}$$

and for Y_3

$$Y_3 = \left(Y_2Z_1^3 - Y_1Z_2^3\right) \cdot Z_2^2X_1 \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^2 \tag{2.48}$$

$$- \left(Y_2Z_1^3 - Y_1Z_2^3\right) \cdot \left[\left(Y_2Z_1^3 - Y_1Z_2^3\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right) \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^2\right] \tag{2.49}$$

$$- Y_1Z_2^3 \cdot \left(X_2Z_1^2 - X_1Z_2^2\right)^3. \tag{2.50}$$

When using shared intermediate values, the cost is $PA_K = 4S_K + 14M_K + 6A_K$. The interest of jacobian coordinates lies in the fact that there is no inversion in the computation.

Formulae for point doubling

The point $P_3 = [2]P_1$ is computed. Geometrically, the tangent in P_1 intersect the elliptic curve at one other point $-P_3$, the mirror image of P_3 by the X axis.

Affine coordinates P_1 and P_3 have coordinates (X_1, Y_1) and (X_3, Y_3) respectively. By differentiating Equation (2.36), the equality

$$2y\frac{dy}{dx} = 3x^2 + a \tag{2.51}$$

is verified. λ is defined as the differential at point P_1.

$$\lambda = \frac{dy}{dx}(P_1) = \frac{3X_1^2 + a}{2Y_1}. \tag{2.52}$$

Finally

$$\begin{cases} X_3 &= \lambda^2 - 2X_1 \\ Y_3 &= -\left(\lambda\left(X_3 - X_1\right) + Y_1\right). \end{cases} \tag{2.53}$$

The cost is $PD_K = I_K + 2S_K + M_K + 9A_K$.

Jacobian coordinates From the previous equations (Equation (2.53)),

$$\frac{X_3}{Z_3^2} = \left(\frac{3\left(\frac{X_1}{Z_1^2}\right)^2 + a}{2\frac{Y_1}{Z_1^3}}\right)^2 - 2\frac{X_1}{Z_1^2}, \tag{2.54}$$

which gives

$$\frac{X_3}{Z_3^2} = \frac{\left(3X_1^2 + aZ_1^4\right)^2 - 8X_1Y_1^2}{4Y_1^2Z_1^2}. \tag{2.55}$$

Similarly,

$$\frac{Y_3}{Z_3^3} = \frac{3\left(\frac{X_1}{Z_1^2}\right)^2 + a}{2\frac{Y_1}{Z_1^3}} \cdot \left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Y_1}{Z_1^3},$$

$$\begin{aligned} \frac{Y_3}{Z_3^3} =& \frac{4Y_1^2X_1\left(3X_1^2 + aZ_1^4\right)}{8Y_1^3Z_1^3} \\ &- \frac{\left(3X_1^2 + aZ_1^4\right) \cdot \left[\left(3X_1^2 + aZ_1^4\right)^2 - 8X_1Y_1^2\right]}{8Y_1^3Z_1^3} \\ &- \frac{8Y_1^4}{8Y_1^3Z_1^3}. \end{aligned}$$

By choosing

$$Z_3 = 2Y_1Z_1, \tag{2.56}$$

we get

$$X_3 = \left(3X_1^2 + aZ_1^4\right)^2 - 8X_1Y_1^2 \tag{2.57}$$

and

$$\begin{aligned} Y_3 =& 4Y_1^2X_1\left(3X_1^2+aZ_1^4\right)-8Y_1^4 \\ & -\left(3X_1^2+aZ_1^4\right)\cdot\left[\left(3X_1^2+aZ_1^4\right)^2-8X_1Y_1^2\right]. \end{aligned}$$

When using shared intermediate values, the cost is $PD_K = 7S_K + 5M_K + 13A_K$. The interest of jacobian coordinates lies in the fact that there is no inversion in the computation.

Formulae for line equation

The line equation is the formula to compute $l_{P_1,P_2}(x,y)$, the line passing through P_1 and P_2 evaluated in (x,y). This equation is required to compute the Miller algorithm (cf. Section 2.6). For the line equation, $P_1 \neq \pm P_2$.

Affine coordinates We have already seen in Section 2.4.2 (Point addition in affine coordinates) that

$$l_{P_1,P_2}(x,y) = y - Y_1 - \frac{Y_2-Y_1}{X_2-X_1}\cdot(x-X_1). \tag{2.58}$$

Jacobian coordinates The formula for jacobian coordinates is obtained with a change of variables (Z_3 is the Z-coordinate of $P_3 = P_1 + P - 2$).

$$l_{P_1,P_2}(x,y,z) = \frac{yZ_1^3Z_3 - Y_1z^3Z_3 + \left(Y_2Z_1^3 - Y_1Z_2^3\right)\cdot\left(X_1Z_1z^3 - xzZ_1^3\right)}{Z_3z^3Z_1^3}. \tag{2.59}$$

Formulae for tangent equation

The tangent equation is $l_{P_1,P_1}(x,y)$, the tangent line through point P_1 evaluated in (x,y). This equation is obtained if $P_1 = P_2$. This equation is required to compute the Miller algorithm (cf. Section 2.6).

Affine coordinates Similarly to the line equation,

$$l_{P_1,P_1}(x,y) = y - Y_1 - \frac{3X_1^2+a}{2Y_1}\cdot(x-X_1)\,3 \tag{2.60}$$

Jacobian coordinates With a change of variables, the equation

$$l_{P_1,P_1}(x,y,z) = \frac{2Y_1\left(yZ_1^3 - Y_1z^3\right) - z\left(3X_1^2+aZ_1^4\right)\cdot\left(xZ_1^2 - X_1z^2\right)}{2Y_1Z_1^3z^3} \tag{2.61}$$

is obtained in jacobian coordinates.

Formulae for vertical equation

The vertical equation is $v_{P_1}(x,y)$ the vertical line passing through P_1 and evaluated in (x,y). This equation is obtained if $P_2 = -P_1$ and is required to compute the Miller algorithm (cf. Section 2.6).

Affine coordinates The equation obtained is not unexpected for a vertical line:

$$v_{P_1}(x, y) = x - X_1. \tag{2.62}$$

Jacobian coordinates With a change of variables,

$$v_{P3}(x, y, z) = \frac{xZ_1^2 - X_1 z^2}{z^2 Z_1^2}. \tag{2.63}$$

In practice, the line evaluation and the point addition operations can be combined since they share lots of intermediate values (and the same for tangent evaluation and point doubling). The algorithms for these compound operations are shown in Code A.10 and in Code A.9 (in Appendix A).

2.4.3 r-torsion

Let $E(\mathbb{F}_q)$ be an elliptic curve, let r be an integer such that $gcd(r, q) = 1$ and $r|\text{card}\,(E(\mathbb{F}_q))$. The points of order a divisor of r form a group, $\{P|[r]P = 0_\infty\}$, noted $E(\mathbb{F}_q)[r]$. The r-torsion of E is the group $E(\overline{\mathbb{F}_q})[r]$ and its notation is often shortened by $E[r]$.

The smallest integer k such that $E[r] \subset E(\mathbb{F}_{q^k})$ is called the embedding degree of E with respect to r. It is the smallest (positive) integer which satisfies $r|q^k - 1$. If r is prime and $r \not| q - 1$ then $E[r] \subset E(\mathbb{F}_{q^k}) \Leftrightarrow r|q^k - 1$ [BK98].

2.4.4 Twists of elliptic curves

In this section, we write $\mathbb{F}_q = \mathbb{F}_{p^n}$ for some integer n and p prime.

Definition 2.4.2 *j-invariant*
Let $E(\mathbb{F}_q) : y^2 = x^3 + ax + b$ be an elliptic curve. We note the j-invariant of E

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}. \tag{2.64}$$

Proposition 2.4.3 *Let $E_1(\mathbb{F}_q)$ and $E_2(\mathbb{F}_q)$ be two elliptic curves, if $j(E_1) = j(E_2)$ then there is an isomorphism between $E_1(\overline{\mathbb{F}_q})$ and $E_2(\overline{\mathbb{F}_q})$.*

The construction of this isomorphism in the case $j \neq \{0, 1728\}$ is done as follows. Let $j = j(E_1) = j(E_2)$ be the j-invariant of curves $E_1 : y^2 = x^3 + a_1x + b_1$ and $E_2 : y^2 = x^3 + a_2x + b_2$. Then for $i \in \{1, 2\}$ we have

$$b_i^2 = a_i^3 \left(\frac{4}{27}\right) \cdot \left(\frac{1728}{j} - 1\right). \tag{2.65}$$

As a consequence, $\exists D \in \overline{\mathbb{F}_q}$ such that $D^2 = \frac{a_1}{a_2}$ and $D^3 = \frac{b_1}{b_2}$. So E_2 can be rewritten $E_2 : y^2 = x^3 + D^2a_1x + D^3b_1$. Finally the isomorphism ϕ we are looking for maps $(x, y) \in E_1(\overline{\mathbb{F}_q})$ to $\phi(x, y) = (Dx, D^{3/2}y) \in E_2(\overline{\mathbb{F}_q})$.

The question now is: does this isomorphism ϕ hold for all extensions $\mathbb{F}_{q^n}$? In the previous example, if D is a square then $\phi : E_1(\mathbb{F}_q) \to E_2(\mathbb{F}_q)$ is an isomorphism (as shown previously). Yet if D is not a square in $\mathbb{F}_q$, $D^{3/2} \in \mathbb{F}_{q^2}$. As a consequence we have now an isomorphism $\phi_2 : E_1(\mathbb{F}_q) \to E_2(\mathbb{F}_{q^2})$. E_1 is called the twist of degree 2 of E_2.

As a conclusion, a curve E has a twist of degree d according to the rules:

- $d = 2$ if $j(E) \notin \{0, 1728\}$,
- $d = 4$ if $j(E) = 1728$ $(b = 0)$,
- $d = 6$ if $j(E) = 0$ $(a = 0)$.

2.5 Pairings

With the elliptic curves defined, we can now describe pairings. The descriptions below are mainly inspired from [Mil04, Gal05, Eng13, Riv10].

2.5.1 Divisors

In this section, $E(\mathbb{F}_q) : y^2 = x^3 + ax + b$ is an elliptic curve with defining polynomial $p_E(x, y) = x^3 + ax + b - y^2$ (*i.e.* $(x, y) \in E(\mathbb{F}_q) \Leftrightarrow p_E(x, y) = 0$). More details on the subject of divisors can be found in [Sil09].

Definition 2.5.1 *Function field $K(E)$*

The function field $K(E)$ is the field of rational functions $E(K) \to K$. A rational function $f : E(K) \to K$ can be written as $f = \frac{f_1}{f_2}$ with $f_1, f_2 \in K[x, y]$.

$f, g \in E(K)$ are said equivalent iff $f_1 g_2 - g_1 f_2 = h \cdot p_E$ with $h \in K[x, y]$. Indeed, $\forall P \in E(\mathbb{F}_q)$, $f_1(P)g_2(P) - g_1(P)f_2(P) = h(P)p_E(P) = 0$ therefore $f(P) = g(P)$.

Definition 2.5.2 *Uniformizer and order of a rational function*

The uniformizer of a curve E at point $P \in E(K)$ is a generator of the ideal $\{f \in K(E) | f(P) = 0\}$. The uniformizer is unique up to a constant in $\overline{K}^*$.

Let $f \neq 0 \in K(E)$ be a non-zero rational function, the order of f at point P is the unique integer n such that $f = gu^n$ where u is a uniformizer of E at P and $g(P) \in K^*$. This integer n is noted $ord_P(f)$. If $ord_P(f) > 0$, then $f(P) = g(P)u(P)^n = 0$ since $u(P) = 0$. f has a zero at P. If $ord_P(f) \geq 0$ f is said to be regular or defined at P. If $ord_P(f) < 0$, f has a pole at P ($f(P)$ is undefined or is divided by 0).

Definition 2.5.3 *Divisor*

A divisor D on E is the formal sum

$$D = \sum_{P \in E} n_P(P), \tag{2.66}$$

where $n_P \in \mathbb{Z}$. There is a finite number ($= card(E)$) of n_P. The support of D is $supp(D) = \{P | n_P \neq 0\}$. The degree of D is

$$deg(D) = \sum_{P \in E} n_P. \tag{2.67}$$

The divisors of a curve E form a group $Div(E)$ with the natural law

$$\sum_{P \in E} n_P(P) + \sum_{P \in E} n'_P(P) = \sum_{P \in E} (n_P + n'_P)(P). \tag{2.68}$$

The divisors of degree 0 form a subgroup of $Div(E)$: $Div^0(E) = \{D \in Div(E) | deg(D) = 0\}$. Let $f \in \overline{K}(E)^*$ be a rational function, we can associate the divisor $div(f)$ to f in the following way

$$div(f) = \sum_{P \in E} ord_P(f)(P). \tag{2.69}$$

$div(fg) = div(f) + div(g)$ and $div(f/g) = div(f) - div(g)$ for $f, g \in \overline{K}(E)^*$. If $div(f) = 0$ then $f \in \overline{K}$ is constant. As a consequence, $div(f)$ determines f up to a constant in $\overline{K}$.

Definition 2.5.4 *Principal divisor*
A divisor D is said to be principal if a $f \in \overline{K}(E)$ exists such that $D = div(f)$. We can create an equivalence relation saying that $D_1 \sim D_2 \Leftrightarrow D_1 - D_2$ is principal. The equivalence classes of divisors with this relation form a group called the Picard group.

Proposition 2.5.5 *Let $f \in \overline{K}(E)^*$ be a rational function, then [Sil09]*

$$deg(div(f)) = 0. \tag{2.70}$$

Proposition 2.5.6 *Let $E(\mathbb{F}_q)$ be an elliptic curve. Let $D = \sum_P n_P(P)$ be a degree 0 divisor on E. Then*

$$\exists f \in \overline{\mathbb{F}_q}(E)^* | D = div(f) \Leftrightarrow \sum_{P \in E(\mathbb{F}_q)} [n_P]P = 0_\infty. \tag{2.71}$$

Definition 2.5.7 *Rational function of a divisor*
Let f be a rational function and $D = \sum_P n_P(P)$ with $deg(D) = 0$ such that $supp(div(f)) \cap supp(D) = \varnothing$. Then we can define

$$f(D) = \prod_P f(P)^{n_P}. \tag{2.72}$$

If $g = cf$ for some $c \in \overline{K}^*$ then for all degree 0 divisors, $f(D) = g(D)$. $f(D)$ depends only on D and $div(f)$.

Proposition 2.5.8 *Weil reciprocity law*
Let $f, g \in \overline{\mathbb{F}_q}(E)$ be two rational functions such that $supp(div(f)) \cap supp(div(g)) = \varnothing$. Then

$$f(div(g)) = g(div(f)). \tag{2.73}$$

2.5.2 Definitions

Definition 2.5.9 *Pairing*
Let G_1 and G_2 be two abelian groups and let G_T be a commutative multiplicative group. A pairing is an application $e : G_1 \times G_2 \to G_T$ with the following properties:

- *Non-degeneracy: let $P \in G_1$ and $Q \in G_2$, if $\forall P \in G_1, e(P, Q) = 1$ then $P = 0$ and if $\forall Q \in G_2, e(P, Q) = 1$ then $Q = 0$.*
- *Bilinearity: let $P, P_1, P_2 \in G_1$ and $Q, Q_1, Q_2 \in G_2$ then*

$$e(P, Q_1 + Q_2) = e(P, Q_1)e(P, Q_2)$$
$$e(P_1 + P_2, Q) = e(P_1, Q)e(P_2, Q)$$

  As a consequence, $\forall a, b \in \mathbb{Z}$, $e([a]P, [b]Q) = e(P, Q)^{ab}$.
- *Efficiency: $\forall P \in G_1$ and $\forall Q \in G_2$, the pairing $e(P, Q)$ is efficiently computable.*

Definition 2.5.10 *Pairing types*
According to the relation between G_1 and G_2, we define types for pairings.

- *Type 1 (symmetric pairing): there are efficiently computable isomorphisms $\phi_1 : G_1 \to G_2$* ***and*** *$\phi_2 : G_2 \to G_1$ (with the possibility $G_1 = G_2$).*
- *Type 2: there is an efficiently computable isomorphism $\phi_1 : G_1 \to G_2$* ***exclusive or*** *$\phi_2 : G_2 \to G_1$.*
- *Type 3: there are no efficiently computable isomorphisms between G_1 and G_2.*

Type 2 and Type 3 pairings are called asymmetric. The type of the pairing influences the cost of the computation and the protocols that can be used with it.

2.5.3 Weil pairing

Proposition 2.5.11 *If $D \neq 0$ is a divisor of degree 0 on an elliptic curve E, then there is a unique point P on E such that $D \sim (P) - (0_\infty)$.*

Let $E[r]$ be the r-torsion of an elliptic curve E and let k be the embedding degree of $E(\mathbb{F}_q)$ with respect to r. Let D_1 and D_2 be two divisors of degree 0 on E with $supp(D_1) \cap supp(D_2) = \varnothing$, $rD_1 \sim 0$ and $rD_2 \sim 0$. It means that rD_1 and rD_2 are principal divisors, *i.e.* $\exists f_1 | div(f_1) = rD_1$ and $\exists f_2 | div(f_2) = rD_2$ (another notation is $f_1 = f_{r,P_1}$ where P_1 is the unique point such that $D_1 \sim (P_1) - (0_\infty)$). The Weil pairing can be defined (among other equivalent definitions [Eng13]) as the application

$$\begin{aligned} e_W : E[r] \times E[r] &\to \mu_r \subset \mathbb{F}_{q^k} \\ (P_1, P_2) &\mapsto \frac{f_1(D_2)}{f_2(D_1)}, \end{aligned} \tag{2.74}$$

where $D_1 \sim (P_1) - (0_\infty)$ and $D_2 \sim (P_2) - (0_\infty)$. First we can see that $e_W(P_1, P_2) \in \mu_r$, according to Definition 2.5.7 and Proposition 2.5.8:

$$\frac{f_1(D_2)^r}{f_2(D_1)^r} = \frac{f_1(rD_2)}{f_2(rD_1)} = \frac{f_1(div(f_2))}{f_2(div(f_1))} = \frac{f_1(div(f_2))}{f_1(div(f_2))} = 1. \tag{2.75}$$

If instead of $D_2 \sim (P_2) - (0_\infty)$, $D_2' \sim (P_2) - (0_\infty)$ is used, then $\exists g \in \mathbb{F}_{q^k}^*(E) | D_2' = D_2 + div(g)$. In this case,

$$div(f_2') = rD_2' = rD_2 + r \cdot div(g) = div(f_2) + r \cdot div(g) \tag{2.76}$$

which induces that $f_2' = f_2 g^r$. Finally by Weil's reciprocity law,

$$\frac{f_1(D_2')}{f_2'(D_1)} = \frac{f_1(D_2) f_1(div(g))}{f_2(D_1) g(D_1)^r} = \frac{f_1(D_2) f_1(div(g))}{f_2(D_1) g(div(f_1))} = \frac{f_1(D_2)}{f_2(D_1)}. \tag{2.77}$$

The bilinearity on the left operand can be shown as follows (the right bilinearity can be shown similarly). Let $P_3 = P_1 + P_2$ and let $g \in \mathbb{F}_{q^k}^*(E)$ be the rational function such that $D_3 \sim (P_3) - (0_\infty), D_3 = D_1 + D_2 + div(g)$ (possible thanks to Proposition 2.5.6). As a consequence, if $div(f_1) = r(P_1) - r(0_\infty)$ and $div(f_2) = r(P_2) - r(0_\infty)$ then

$$\begin{aligned} div(f_1 f_2 g^r) &= r(P_1) + r(P_2) - 2r(0_\infty) + r(P_3) - r(0_\infty) - r(P_1) - r(P_2) + 2r(0_\infty) \\ &= r(P_3) - r(0_\infty) \\ &= div(f_3). \end{aligned} \tag{2.78}$$

Let $D_Q \sim (Q) - (0_\infty)$ be a divisor and $f_Q \in \mathbb{F}_{q^k} * (E)$ such that $div(f_Q) = r(Q) - r(0_\infty)$. D_Q, D_1, D_2, D_3 all have disjoint supports. According to Definition 2.5.7

$$f_Q(D_3) = f_Q(D_1) f_Q(D_2) f_Q(div(g)). \tag{2.79}$$

Then

$$\begin{aligned} e_W(P_3, Q) &= \frac{f_3(D_Q)}{f_Q(D_3)} && (2.80)\\ &= \frac{f_1(D_Q) f_2(D_Q) g(D_Q)^r}{f_Q(D_1) f_Q(D_2) f_Q(div(g))} && (2.81)\\ &= \frac{f_1(D_Q)}{f_Q(D_1)} \frac{f_2(D_Q)}{f_Q(D_2)} \frac{g(D_Q)^r}{f_Q(div(g))} && (2.82)\\ &= \frac{f_1(D_Q)}{f_Q(D_1)} \frac{f_2(D_Q)}{f_Q(D_2)} \frac{g(D_Q)^r}{g(D_Q)^r} && (2.83)\\ &= e_W(P_1, Q) e_W(P_2, Q). && (2.84) \end{aligned}$$

2.5.4 Tate pairing

The Tate pairing is defined with the same parameters $E, \mathbb{F}_q, r, k$ as the Weil pairing. $\mathbb{F}_{q^k}$ is the minimal extension such that $E[r] \subseteq E(\mathbb{F}_{q^k})$.

The Tate pairing e_T is defined as

$$\begin{aligned} e'_T : E[r] \times E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k}) &\to \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^r \\ (P_1, P_2) &\mapsto f_1(D_2) \end{aligned} \tag{2.85}$$

with $D_2 \sim (P_2) - (0_\infty), P_1 \notin supp(D_2)$ and $div(f_1) = r(P_1) - r(0_\infty)$.

The bilinearity can be shown in a similar manner as for the Weil pairing. Let $P_1, P_2, P_3 = P_1 + P_2 \in E[r]$ and $Q \in E(\mathbb{F}_{q^k})$, then $f_3(D_Q) = f_1(D_Q) f_2(D_Q) g(D_Q)^r$ where $D_3 \sim (P_3) - (0_\infty), D_3 = D_1 + D_2 + div(g)$. But the value $g(D_Q)^r$ is in the same equivalence class as 1 in $\mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^r$, so $e'_T(P_3, Q) = e'_T(P_1, Q) e'_T(P_2, Q)$.

For cryptographic applications, managing classes of equivalence instead of values is not handy. For that purpose, we define the mapping

$$\begin{aligned} \pi : \mathbb{F}_{q^k} * /(\mathbb{F}^*_{q^k})^r &\to \mu_r \subset \mathbb{F}_{q^k} \\ x &\mapsto x^{\frac{q^k-1}{r}}. \end{aligned} \tag{2.86}$$

Now the reduced Tate pairing defined by

$$\begin{aligned} e_T : E[r] \times E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k}) &\to \mu_r \subset \mathbb{F}_{q^k} \\ (P_1, P_2) &\mapsto \pi(f_1(D_2)) = f_1(D_2)^{\frac{q^k-1}{r}} \end{aligned} \tag{2.87}$$

maps equivalent elements to the same value in μ_r.

2.5.5 Ate pairing [HSV06]

Definition 2.5.12 *Supersingular curve*

A curve $E(\mathbb{F}_q)$ ($\mathbb{F}_q$ of characteristic p or $q = p^k$) is said to be supersingular if $card(E(\mathbb{F}_q)) \equiv 1 \mod p$. A curve which is non supersingular is called ordinary.

Definition 2.5.13 *Trace of E*

We call trace of the Frobenius endomorphism of E (or trace of E) the value t such that

$$t = q + 1 - card\left(E(\mathbb{F}_q)\right). \tag{2.88}$$

Let $\pi_p((x,y)) = (x^p, y^p)$ be the Frobenius endomorphism on E. A curve E is supersingular iff its trace is a multiple of the characteristic of $\mathbb{F}_q$.

Supersingular curves are possible only with $k \leq 2$ for large characteristic fields, $k \leq 4$ for binary fields and $k \leq 6$ for ternary fields [CFA+05]. When a bigger k is required, one must choose an ordinary curve instead.

The Ate pairing is a refinement of the Tate pairing which aims at being faster to compute. The Ate pairing uses the fact that the Frobenius endomorphism π_p has two eigenvalues 1 and p in $E\left(\mathbb{F}_{p^k}\right)[r]$. Let P and Q be the respective eigenvectors ($\pi_p(P) = P, \pi_p(Q) = [p]Q$), then

$$E\left(\mathbb{F}_{p^k}\right)[r] = < P > \times < Q >$$

where $< X >$ is the group generated by X. Additionally, if $k > 1$ then $P \in E\left(\mathbb{F}_p\right)[r]$.

The Ate pairing is defined as follows. Let $P \in G_1 = E(\mathbb{F}_q)[r] \cap \ker\left(\pi_p - [1]\right)$, $Q \in G_2 = E(\mathbb{F}_q)[r] \cap \ker\left(\pi_p - [p]\right)$ and $T = t - 1$ then

$$e_A(Q,P) = f_{T,Q}(P)^{\frac{p^k-1}{r}} \tag{2.89}$$

is a pairing called the Ate pairing. In this equation, $f_{T,Q} \in \mathbb{F}^*_{p^k}(E)$ such that $div(f_{T,Q}) = T(Q) - ([T]P) - (T-1)(0_\infty)$.

Let $N = gcd(T^k - 1, p^k - 1)$ and $T^k - 1 = LN$, then

$$e_T(Q,P)^L = f_{T,Q}(P)^{\frac{c(p^k-1)}{N}}, \tag{2.90}$$

where $c = \sum_{i=0}^{k-1} T^{k-1-i}p^i \equiv kp^{k-1} \mod r$ if $r \nmid L$. This equation establishes a link between the Tate and the Ate pairing which is used to ensure the properties of the Ate pairing (proofs are given in [HSV06]). According to Hasse's theorem (Equation (2.35)) $T \approx \sqrt{p} \approx \sqrt{r}$. Since in the Tate pairing, $\log_2(r)$ defines the number of iterations in the Miller algorithm (cf. Section 2.6), the Ate pairing greatly reduces the number of iterations.

2.5.6 Optimal Ate (OATE) pairing [Ver10]

The Optimale Ate (OATE) pairing [Ver10] improves the Ate pairing by minimizing the number of iterations in the Miller algorithm used to compute $f_{\lambda,Q}(P)$. In [Ver10], the author shows that there is an optimal value λ (so that we obtain a non-degenerate pairing) and how to compute it. In the case of BN curves (cf. Section 2.5.7), the OATE pairing is given by

$$e_O(Q,P) = \left(f_{\lambda,Q}(P) \cdot M\right)^{\frac{p^k-1}{r}}, \tag{2.91}$$

where $\lambda = 6x+2$ (x the BN curve parameter), $M = l_{Q_3,-Q_2}(P) \cdot l_{-Q_2+Q_3,Q_1}(P) \cdot l_{Q_1-Q_2+Q_3,[\lambda]Q}(P)$ and $Q_i = \pi_{p^i}(Q) = (x_Q^{p^i}, y_Q^{p^i})$ (**note:** the use of a twisted curve must be accounted for in this frobenius computation). The $l_{A,B}(C)$ are the line equations and will be detailed in Section 2.6. A generalization and a reformulation of optimal pairings can be found in [Hes08].

2.5.7 Families of elliptic curves for pairings

The elliptic curves must have a certain structure in order to be both computable and secure when used for pairings. The key feature is to have a moderate embedding degree: a lower embedding degree weakens the security of the curve while a larger degree makes the computations impracticable.

If E is a random curve with a subgroup of prime order r, then with high probability $k \approx r$ whereas we want $k \ll r$. To construct a pairing-friendly curve, one wants to specify the embedding degree k, a prime p and an integer r and find a curve $E(\mathbb{F}_{p^k})$ with an r order subgroup. Some curves have some common criteria which allows to classify them as members of the same curve family. If a method exists where $p = p(x)$ and $r = r(x)$ define a curve when $p(x)$ is prime, then the curves created when x varies form a family. More details can be found in [FST10] and in the following.

Families of curves only exist when creating ordinary curves. The construction is based on the Complex Multiplication (CM) method. A sparse family is a family where the x values defining valid curves grow exponentially. In the other case, it is a complete family. The CM method tries to find a couple (x, y) such that

$$Dy^2 = 4h(x)r(x) - (t(x) - 2)^2, \tag{2.92}$$

where $h(x)$ is a cofactor with $\text{card}\left(E(\mathbb{F}_{p^k})\right) = h(x)r(x)$ and $t(x)$ is the trace ($\text{card}\left(E(\mathbb{F}_{p^k})\right) = p(x)^k + 1 - t(x)$).

The parameter $\rho = \frac{\log p}{\log r}$ is introduced and compares the sizes of the field and of the subgroup on the elliptic curve used for pairing computations. A curve is better if ρ is the closest possible to 1.

"Orphan curves"

Some constructions allow to find elliptic curves with a small k and do not belong to any family, *i.e.* the construction methods work for one single x each time.

Cocks and Pinch The Cocks and Pinch method starts from Equation (2.92), chooses $r(x)$, finds $t(x)$ and $h(x)$ compatible with their definitions and finds y in Equation (2.92). The Cocks and Pinch method usually produces curves with $\rho \approx 2$.

Dupont-Enge-Morain (DEM) curves [DEM05] The DEM method first fixes D and y and then tries to find $t(x)$ and $r(x)$ simultaneously, finally find $h(x)$ according to Equation (2.92). Here again the constructed curves have $\rho \approx 2$.

Miyaji-Nakabayashi-Takano (MNT) curves [MNT01]

The strategy behind the construction of MNT curves is to choose $t(x), h(x)$ then compute $r(x)$ satisfying the conditions and finally solve Equation (2.92). This method constructs a sparse family of curve with possible embedding degrees $3, 4, 6$ (and can be extended to support embedding degrees $10, 12$). The parametrization of MNT curves depends on k. For $k = 3$,

$$\begin{aligned} t(x) &= \pm 6x - 1 \\ p(x) &= 12x^2 - 1. \end{aligned} \tag{2.93}$$

For $k = 4$,

$$\begin{aligned} t(x) &= -x \text{ or } x+1 \\ p(x) &= x^2 + x + 1. \end{aligned} \tag{2.94}$$

For $k = 6$,

$$\begin{aligned} t(x) &= \pm 2x + 1 \\ p(x) &= 4x^2 + 1. \end{aligned} \tag{2.95}$$

Galbraith-McKee-Valença (GMV) curves [GMV07]

The GMV curves construction method is inspired from the MNT method but focuses on the use of a prescribed cofactor. The cofactor h represents the ratio between the number of points on the curve and the order of the subgroups used for the pairing:

$$\text{card}\,(E(\mathbb{F}_q)) = h \cdot r. \tag{2.96}$$

In order to have an efficient scheme ($\rho \approx 1$), the cofactor h is desired small. Once the cofactor and the embedding degree $(3, 4, 6)$ have been fixed, the GMV method gives expressions for $p(x)$ and $t(x)$. The explicit formulae can be found in Tables $4, 5, 6$ in [GMV07].

Barreto-Naehrig (BN) curves [BN06]

The BN curves are particularly efficient for the 128-bit security level and the embedding degree 12. They are efficient because $\rho \approx 1$ and because for $\log(p) \approx 256$, both the ECDLP in $E(\mathbb{F}_p)$ and the DLP in $\mathbb{F}_{p^k}$ have a 128-bit security level. BN curves form a complete family derived from the MNT family. A BN curve is defined by the x value, of which $t(x), r(x), p(x)$ are derived.

$$\begin{aligned} t(x) &= 6x^2 + 1, \\ r(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1, \\ p(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1. \end{aligned} \tag{2.97}$$

Curves are defined for a x where both $p(x)$ and $r(x)$ are prime numbers.

Kachisa-Schaefer-Scott (KSS) curves [KSS08]

KSS curves are used for high security level curves and the method to construct them is inspired by the method by Brezing and Weng [BW05]. Possible embedding degrees $8, 16, 18, 32, 36, 40$ are reported in [KSS08]. An example is provided below for a security level close to 192-bit with an embedding degree $k = 18$ (the base field should be 512-bit wide).

$$\begin{aligned} t(x) &= (x^4 + 16x + 7)/7, \\ r(x) &= x^6 + 37x^3 + 343, \\ p(x) &= (x^8 + 5x^7 + 7x^6 + 37x^5 + 188x^4 + 259x^3 + 343x^2 + 1763x + 2401)/21. \end{aligned} \tag{2.98}$$

This formula (for $k = 18$) gives $\rho = 4/3$.

Barreto-Lynn-Scott (BLS) curves [BLS03]

BLS curves are intended for high security levels (greater than 128-bit) and particularly the 256-bit security level. To reach this security level, the embedding degree $k = 24$ is chosen for a base field 640-bit wide. For $k = 24$, the curve parameters are shown below.

$$\begin{aligned} t(x) &= x + 1, \\ r(x) &= x^8 - x^4 + 1, \\ p(x) &= (x-1)^2(x^8 - x^4 + 1)/3 + x. \end{aligned} \tag{2.99}$$

This parametrization gives $\rho \approx 1.25$ ($512 \cdot \rho = 640$ and $512 \cdot \rho \cdot k = 15360$). Finally in [Sco11], Michael Scott compared the speed of various implementations based on curves Cocks-Pinch, MNT, BN, KSS and BLS. The results show that the BN curves have the highest security to computation time ratio.

2.6 Miller algorithm

The Miller algorithm proposed by Victor Miller in [Mil86a] is the main algorithm in order to compute a pairing. It uses a recurrence relation in order to find a rational function $f_{n,P}$ such that $div(f_{n,P}) = n(P) - ([n]P) - (n-1)(0_\infty)$.

Definition 2.6.1 *Line function*

Let l_{P_1,P_2} be the rational function such that $l_{P_1,P_2}(x, y) = 0$ denotes the line passing through P_1 and P_2 ($P_1, P_2 \neq 0_\infty$, if $P_1 = P_2$ the tangent is used). On an elliptic curve, this function has three zeros: P_1, P_2 and $-(P_1 + P_2)$. As a consequence,

$$div(l_{P_1,P_2}) = (P_1) + (P_2) + (-(P_1 + P_2)) - 3(0_\infty). \tag{2.100}$$

Let v_P be the rational function such that $v_P(x, y) = 0$ denotes the vertical line passing through P. On an elliptic curve, this function has two zeros: P and $-P$. As a consequence,

$$div(v_P) = (P) + (-P) - 2(0_\infty). \tag{2.101}$$

The Miller algorithm relies on the following relation, $\forall n, m \in \mathbb{N}$

$$\begin{aligned} div(f_{n+m,P}) &= (n+m)(P) - ([n+m]P) - (n+m-1)(0_\infty), \\ &= (n)(P) - ([n]P) - (n-1)(0_\infty) \\ &+ (m)(P) - ([m]P) - (m-1)(0_\infty) \\ &+ ([n]P) + ([m]P) + (-[n+m]P) - 3(0_\infty) \\ &- ([n+m]P) - (-[n+m]P) + 2(0_\infty). \end{aligned} \tag{2.102}$$

Or using line and vertical functions,

$$div(f_{n+m,P}) = div(f_{n,P}) + div(f_{m,P}) + div(l_{[n]P,[m]P}) - div(v_{[n+m]P}). \tag{2.103}$$

This relation implies a relation between the rational functions:

$$f_{n+m,P} = f_{n,P} f_{m,P} \frac{l_{[n]P,[m]P}}{v[n+m]P}. \tag{2.104}$$

This recurrence relation can be completed by the fact that $div(f_{0,P}) = div(f_{1,P}) = 0$ which means that both functions are constant and can be chosen equal to 1. From the recurrence relation, the following particular cases are important:

$$\begin{aligned} f_{i+1,P} &= f_{i,P} \cdot \frac{l_{[i]P,P}}{v_{[i+1]P}} \\ f_{2i,P} &= f_{i,P}^2 \cdot \frac{l_{[i]P,[i]P}}{v_{[2i]P}}. \end{aligned} \tag{2.105}$$

The Miller algorithm (Algorithm 3) allows to compute $\forall n \in \mathbb{N}$ the value $f_{n,P}(Q)$ with an algorithm inspired from the Square and Multiply algorithm (cf. Code A.11).

Algorithm 3: Miller algorithm for the Tate pairing

Data: $r = (r_n \dots r_0)_2$, $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$;
Result: $f_{r,P}(Q) \in \mathbb{G}_3$;
$T \leftarrow P$;
$f \leftarrow 1$;
for $i = n - 1$ **to** 0 **do**
  $f \longleftarrow f^2 \times \frac{l_{T,T}(Q)}{v_{[2]T}(Q)}$;
  $T \leftarrow [2]T$;
  if $r_i = 1$ **then**
    $f \longleftarrow f \times \frac{l_{T,P}(Q)}{v_{T+P}(Q)}$;
    $T \leftarrow T + P$;
  end
end
return f

Several optimizations can (or must) be used in the algorithm.

1. Last iteration: for a Tate pairing, r is prime (and $\neq 2$) and therefore odd. During the last iteration, the last operation is the computation of $f = f_{r,P}(Q)$. But $div(f_{r,P}) = div(f_{r-1,P}) + div(v_P)$ which induces that the last operation on f should be replaced with
$$f \longleftarrow f \times v_P(Q).$$
2. Denominator elimination: the final exponentiation will map all values in a strict subfield of $\mathbb{F}_{p^k}$ to 1 which removes the need to compute these values. This is particularly useful when using twists of elliptic curve since it becomes possible to simplify all vertical line evaluations.
3. r with low Hamming Weight: as for a standard Square-and-Multiply algorithm, the computation can be shortened by using an r value with a low Hamming Weight (or with efficiently grouped bits for Booth's method).

2.7 Final exponentiation for Tate-like pairings

The final exponentiation for Tate-like pairings is the exponentiation by a factor $\frac{p^k-1}{r}$ in order to map elements into $\mu_r \subset \mathbb{F}_{p^k}^*$. This is a large exponent applied to a data in the full field $\mathbb{F}_{p^k}$, as a consequence the computation can be quite time consuming.

2.7.1 Basic method

A method to efficiently compute this final exponentiation when k is even has been shown in [SBC+09]. We write $k = 2d$ which induces that

$$\frac{p^k - 1}{r} = \left(p^d - 1\right) \cdot \frac{p^d + 1}{\Phi_k(p)} \cdot \frac{\Phi_k(p)}{r}, \tag{2.106}$$

where $\Phi_k(X)$ is the k^{th} cyclotomic polynomial. The aim is to express the exponent with as many frobenius endomorphisms as possible.

Definition 2.7.1 *Frobenius endomorphism*

The Frobenius endomorphism is defined as the application $x \in \mathbb{F}_q \mapsto x^{char(\mathbb{F}_q)}$. Let $a, b \in \mathbb{F}_{p^k}$ be two elements in a field of characteristic p. Then

$$(a + b)^p = a^p + b^p. \tag{2.107}$$

More generally for $n \in \mathbb{N}$,

$$(a + b)^{p^n} = ((a + b)^p)^{p^{n-1}} = (a^p + b^p)^{p^{n-1}} = \ldots = a^{p^n} + b^{p^n}. \tag{2.108}$$

Since every element $a \in \mathbb{F}_{p^k}$ can be written as $a = \sum_{i=0}^{k-1} a_i \cdot \omega^i$, where $a_i \in \mathbb{F}_p$ and $(1, \omega, \omega^2, \ldots, \omega^{k-1})$ form a basis of $\mathbb{F}_{p^k}$ as a k vector space over $\mathbb{F}_p$, it can be seen that

$$a^{p^n} = \sum_{i=0}^{k-1} a_i^{p^n} \cdot \omega^{i \cdot p^n} = \sum_{i=0}^{k-1} a_i \cdot \omega^{i \cdot p^n}. \tag{2.109}$$

Definition 2.7.2 *Cyclotomic polynomial*

The k^{th} cyclotomic polynomial $\Phi_k(X)$ is the minimal monic polynomial with the roots equal to the k^{th} roots of unity. As a consequence, $\Phi_k(X) | X^k - 1$. The first cyclotomic polynomials are shown in Table 2.7.

Table 2.7: Cyclotomic polynomials

k	$\Phi_k(x)$
1	$x - 1$
2	$x + 1$
3	$x^2 + x + 1$
4	$x^2 + 1$
5	$x^4 + x^3 + x^2 + x + x1$
6	$x^2 - x + 1$
7	$x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$
8	$x^4 + 1$
9	$x^6 + x^3 + 1$
10	$x^4 - x^3 + x^2 - x + 1$
11	$x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$
12	$x^4 - x^2 + 1$

The final exponentiation is composed of three exponentiations, two easy ($p^d - 1$ and $\frac{p^d+1}{\Phi_k(p)}$) since they use Frobenius endomorphisms and one difficult ($\frac{\Phi_k(p)}{r}$). Additionally, the element $a' = a^{p^d-1}$ is called unitary [SB04] which makes all subsequent inversions "free" (equivalent to a conjugation).

The inversion of a unitary element is equivalent to a conjugation. Indeed, since $k = 2d$, we have $\mathbb{F}_{p^k} = \mathbb{F}_{p^{d^2}}$. We can write $x + iy$ an element of $\mathbb{F}_{p^k}$ with $x, y \in \mathbb{F}_{p^d}$ and i^2 a quadratic non-residue in $\mathbb{F}_{p^d}$. First, it can be seen that

$$(x + iy)^{p^d-1} = \frac{(x + iy)^{p^d}}{x + iy} = \frac{x - iy}{x + iy}. \tag{2.110}$$

Let $a + ib$ be an element of $\mathbb{F}_{p^k}$ such that $a + ib = (x + iy)^{p^d-1}$ with $a, b \in \mathbb{F}_{p^d}$.

$$(a + ib) \cdot (a - ib) = a^2 - i^2b^2 = \frac{x - iy}{x + iy} \cdot \frac{x + iy}{x - iy} = 1, \tag{2.111}$$

hence the name "unitary" (the norm is equal to 1). Finally the previous equation shows that

$$(a + ib)^{-1} = a - ib. \tag{2.112}$$

The hard exponentiation (exponent $h = \frac{\Phi_k(p)}{r}$) is performed by expressing the exponent in base p : $h = \sum_{i=0}^{n-1} a_i p^i$. Then, computing f^h is the same as computing $f^{a_{n-1}p^{n-1}} \cdot \ldots \cdot f^{a_1 p} \cdot f^{a_0}$ or equivalently

$$f^h = \left(f^{p^{n-1}}\right)^{a_{n-1}} \cdot \ldots \cdot (f^p)^{a_1} \cdot f^{a_0}. \tag{2.113}$$

The various f^{p^i} are first computed and then a multi-exponentiation algorithm is used to get f^h. We have proposed a graphical representation of the complete algorithm for BN curves ($k = 12$) on Figure 2.1, the code can be found in appendix (Code A.13). This representation was the base for one of the proposed fault attacks (Section 5.2).

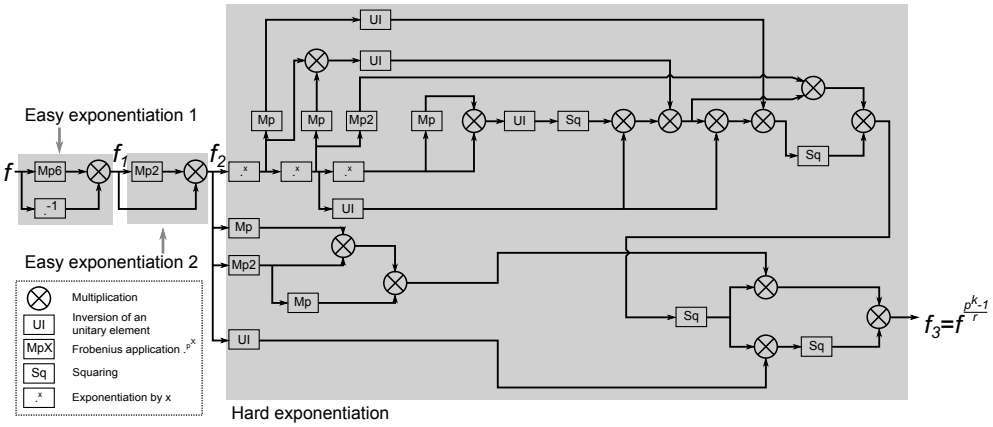

Figure 2.1: Algorithm for the FE in $\mathbb{F}_{p^{12}}$. x is a public parameter of the curve.

2.7.2 Other methods

The previous method can be optimized. In [AKL+11], the authors propose to compress the representation of elements after the two easy exponentiation. Indeed at that instant an element

$f^{(p^d-1)\cdot\frac{p^d+1}{\Phi_k(p)}} \in G_{\Phi_d}\left(\mathbb{F}_{p^2}\right)$ can be represented by $\phi(k)$ (Euler's totient function) elements in $\mathbb{F}_p$. This compressed representation allows fast squaring formulae.

Another possibility is the used of a Square-and-Multiply algorithm for the exponentiation but with more than 3000 iterations, this algorithm would be time consuming.

2.8 Protocols for PBC

The main interest of pairings is in that they allow new protocols, new cryptographic possibilities easily achievable in practice in real world applications. In this section some of those new opportunities are presented in order to show the potential behind these applications.

2.8.1 One round tripartite key exchange

The One round tripartite key exchange as proposed by Joux [Jou00] was the first proposal of a constructive use of pairings. The goal is to exchange one key among three participants in just one communication round.

In a Type 1 setting, the users agree on the public parameters e, G_1, G_2 where G_1, G_2 are the two groups such as $e : G_1 \times G_1 \to G_2$ is a cryptographically sound pairing. They also agree on a generator P such that $< P >= G_1$.

Let the three users be Alice, Bob and Charlie. They all choose a secret key (respectively $a, b, c \mod char(G_1)$) and broadcast their public keys (respectively $[a]P, [b]P, [c]P$) in one communication round. They now can agree on a common secret key $e(P, P)^{abc}$ in the following manner:

- Alice computes $e([b]P, [c]P)^a = e(P, P)^{abc}$.
- Bob computes $e([a]P, [c]P)^b = e(P, P)^{abc}$.
- Charlie computes $e([a]P, [b]P)^c = e(P, P)^{abc}$.

2.8.2 Identity-Based Encryption (IBE) [BF01]

An IBE scheme allows to simplify one of the biggest issues with public-key cryptography, the key distribution. A Public-Key Infrastructure (PKI) using an IBE scheme is less complex and is easier to scale up compared to traditional schemes (with certificates).

In an IBE scheme, the public key *is* the identity of the entity. As a consequence, the associated private-key cannot be computed by this entity but has to be generated by the Private Key Generator (PKG). The decryption is possible only if the correct private key is known.

A simplified version of the Boneh-Franklin IBE [BF01] is described by four algorithms: Setup, Extract, Encrypt, Decrypt. This protocol uses a Type 1 pairing.

Setup: The PKG generates the common parameters for the pairing computations. It chooses e, G_1, G_2 two groups of order r such that $e : G_1 \times G_1 \to G_2$ is a pairing. It chooses $P \in G_1$, a random generator of G_1. It chooses two hash functions, $H_1 : \{0,1\}^* \to G_1^*$ and $H_2 : G_2 \to \{0,1\}^n$. The PKG picks a random $s \in \mathbb{Z}_r$, its private key (called the master key), and computes $P_{PUB} = [s]P$ the global public key.

The public parameters are finally $\{r, n, G_1, G_2, e, P, P_{pub}, H_1, H_2\}$.

Extract: The extract algorithm provides a user with its private key. Let a user Alice have $ID =$ "Alice" $\in \{0,1\}^*$, the PKG then computes the public identity point $Q_A = H_1(\text{"Alice"})$ and the associated private key $d_A = [s]Q_A$.

Encrypt: If the user Bob wants to send a message $M \in \{0,1\}^n$ to Alice, he uses the Encrypt algorithm.

- He computes $Q_A = H_1(\text{"Alice"})$.
- He chooses a random nonce k.
- He computes $g_A = e(Q_A, P_{PUB}) \in G_2^*$.
- Finally he computes the ciphertext $C = \{[k]P, M \oplus H_2(g_A^k)\}$ and sends it to Alice.

Decrypt: When Alice wants to decrypt the ciphertext $C = \{U, V\}$ with $U \in G_1, V \in \{0,1\}^n$, she uses the Decrypt algorithm.

- She computes $e(d_A, U) = e([s]Q_A, [k]P) = e(Q_A, P)^{sk} = e(Q_A, [s]P)^k = e(Q_A, P_{PUB})^k = g_A^k$.
- She gets the message $M = V \oplus H_2(g_A^k)$.

Since she is the only user with knowledge of d_A, she is the only one able to decrypt the message.

In the previous scheme, all private keys are revoked if the PKG changes the master key s (and thus P_{PUB}). In this case, all users need to get their new private key. Another way to limit the validity of a key is to include the date in the public key. If Bob sends a message to Alice by using the public key "Alice/2014". Alice can decrypt the message only if she has the private key associated with "Alice/2014". If she has the private key for "Alice/2013" she cannot decrypt it.

2.8.3 Hierarchical Identity-Based Encryption (HIBE)

A Hierarchical Identity-Based Encryption (HIBE) is a development of the IBE scheme were the entities involved form an organizational hierarchy [BBG05, LW10]. A node in this hierarchy can delegate secret keys to its descendants and can only decrypt messages for these descendants. The identity ID of a node is a vector $ID = (I_1, \ldots, I_k) \in \left(\mathbb{Z}_p^*\right)^k$. A hash function can be used to transform an arbitrary string into a value in $\mathbb{Z}_p^*$.

A description of the HIBE proposed in [BBG05] is summarized below. The HIBE scheme has the same 4 algorithms as presented in Section 2.8.2. A Type 1 pairing is used.

Setup: The setup is performed for a maximum depth l (depth of the hierarchy). The pairing $e : G_1 \times G_1 \to G_2$ is used where $p = char(G_1)$. The root PKG (node at level 0) selects a random generator $g \in G_1$ and a secret random value $a \in \mathbb{Z}_p$. It then sets $g_1 = g^a$ and picks random elements $g_2, g_3, h_1, \ldots, h_l \in G_1$. The master public key is set to $mk = g_2^a$. Finally the public parameters are $(g, g_1, g_2, g_3, h_1, \ldots, h_l, mk)$.

Extract: The private key d_{ID} of the entity $ID = (I_1, \ldots, I_k) \in \left(\mathbb{Z}_p^*\right)^k$ of depth $k < l$ is, for a random $r \in \mathbb{Z}_p$:

$$d_{ID} = \left(g_2^a \cdot \left(h_1^{I_1} \cdot \ldots \cdot h_k^{I_k} \cdot g_3\right)^r, g^r, h_{k+1}^r, \ldots, h_l^r\right) \in G_1^{2+l-k}. \tag{2.114}$$

The private key d_{ID} can be generated incrementally by the ascendant of the node. If $ID_p = (I_1, \ldots, I_{k-1})$ with private key $d_{ID_p} = (a_0, a_1, b_k, \ldots, b_l)$ has a child node $ID_c = (I_1, \ldots, I_{k-1}, I_k)$ the private key of the child can be computed as

$$d_{ID_c} = \left(a_0 \cdot b_k^{I_k} \cdot \left(h_1^{I_1} \cdot \ldots \cdot h_k^{I_k} \cdot g_3\right)^t, a_1 \cdot g^t, b_{k+1} \cdot h_{k+1}^t, \ldots, b_l \cdot h_l^t\right), \tag{2.115}$$

where $t \in \mathbb{Z}_p$ is random.

Encrypt: In order to encrypt a message $M \in G_2$ for the entity $ID = (I_1, \ldots, I_k)$, one needs to pick a random $s \in \mathbb{Z}_p$ and then he can produce the ciphertext

$$CT = \left(e(g_1, g_2)^s \cdot M, g^s, \left(h_1^{I_1} \cdot \ldots \cdot h_k^{I_k} \cdot g_3\right)^s\right) \in G_2 \times G_1^2. \tag{2.116}$$

Decrypt: To decrypt the ciphertext $CT = (A, B, C) \in G_2 \times G_1^2$, the node with private key $d_{ID} = (a_0, a_1, b_{k+1}, \ldots, b_l)$ computes

$$M = A \cdot \frac{e(a_1, C)}{e(B, a_0)}. \tag{2.117}$$

It comes from the fact that

$$\frac{e(a_1, C)}{e(B, a_0)} = \frac{e\left(g^r, \left(h_1^{I_1} \cdot \ldots \cdot h_k^{I_k} \cdot g_3\right)^s\right)}{e\left(g^s, g_2^a \cdot \left(h_1^{I_1} \cdot \ldots \cdot h_k^{I_k} \cdot g_3\right)^r\right)} = \frac{1}{e(g, g_2)^{sa}} = \frac{1}{e(g_1, g_2)^s}. \tag{2.118}$$

2.8.4 Attribute-Based Encryption (ABE)

IBE and HIBE are members of the functional encryption family. A functional encryption scheme allows the correct user to compute a function of a ciphertext. In IBE, the function is $F(ct) = pt$ iff ID is correct.

An Attribute-Based Encryption (ABE) scheme is inspired from an HIBE scheme where nodes in the hierarchy do not represent entities but instead they represent attributes, *i.e.* keywords representing an authorization policy (*e.g.* an entity can be a "doctor", with specialisation "surgeon" etc...). Relations (AND or OR) are specified between nodes and encoded in the key (Key Policy ABE or KP-ABE [GPSW06]) or in the ciphertext (Ciphertext Policy ABE or CP-ABE [Wat11]) which describe the set of attributes necessary to decipher the ciphertext.

An example of KP-ABE is presented below.

Setup: The setup is done in a Type 1 setting. The pairing $e : G_1 \times G_1 \to G_2$ is used where $p = char(G_1)$. A generator g of G_1 is chosen by the PKG as well as a random element $a \in \mathbb{Z}_p$, the master key. The set of possible attributes is fixed (size n) and noted U. The public parameters are then $(g, e(g, g)^a, H_1 = g^{h_1}, \ldots, H_n = g^{h_n})$. Where $h_i \in \mathbb{Z}_p$ is a secret random element for each $i \in U$.

Extract: In order to create a key according to a policy, the PKG splits a into shares $\{\lambda_i\}$. For example, if the user is required to have attributes 1 AND 2 to decipher the ciphertext, the key is split into $\{a - z, z\}$ for $z \in \mathbb{Z}_p$ a random element.

$$SK = \left\{ g^{\frac{\lambda_i}{h_i}} \right\}_i .$$

Encrypt: The message M to encrypt is in G_2. To encrypt, an entity needs to choose a random $s \in \mathbb{Z}_p$ and produces the ciphertext

$$CT = (M \cdot e(g,g)^{as}, S, \{H_i^s\}_{i \in S \subseteq U}).$$

The ciphertext size grows with the number of possible attributes.

Decrypt: In order to decrypt the ciphertext $CT = (A, B, \{C_i\})$, the user must compute $M = A/e(g,g)^{as}$. The shares can be recovered with $e(g^{\frac{\lambda_i}{h_i}}, C_i) = e(g,g)^{\lambda_i \cdot s}$. The shares may then be combined to obtain $e(g,g)^{as}$.

2.9 Cryptanalysis of pairing based cryptography

Recently progress has been made regarding the cryptanalysis of some pairing algorithms. We will not describe all of them into details, a whole thesis would be needed for that. But the results will be presented as well as insights of the methods used. Before that, a description of the common cryptographic problems (relevant to the context of PBC) is given in order to recall the assumptions currently made with PBC.

2.9.1 Cryptographic problems

A cryptographic problem is a mathematical problem believed to be (computionally hard to solve and used for cryptographic purposes.

DLP: The Discrete Logarithm Problem (DLP) states that for $g, g^a \in \mathbb{F}_p$ known, it is hard (no polynomial algorithm in the bit size of g) to recover a.

ECDLP: The Elliptic Curve Discrete Logarithm Problem (ECDLP) is a variant of the DLP but on an elliptic curve. It states that for $P, [a]P \in E(\mathbb{F}_p)$ known, it is hard to recover a.

The ECDLP can be linked to the DLP thanks to pairings. Let $P, [a]P \in E(\mathbb{F}_{p^k})[r]$ where k is the embedding degree. Let $X = e(P,P)$, then $X^a = e(P,[a]P)$. a can be found by solving the DLP or the ECDLP [MOV93, FR94]. As a consequence, k should have a correct size to balance the security problems when using pairings.

DDH: The Decisional Diffie-Hellman (DDH) assumption states that for a generator $g \in \mathbb{F}_p$ and given random $a, b \in \mathbb{Z}_p$, the value g^{ab} is undistinguishable from a random value g^c. If the DDH is false, then an attacker is able to decide if $X = g^c$ or $X = g^{ab}$.

CDH: The Computational Diffie-Hellman (CDH) assumption states that given $g, g^a, g^b \in \mathbb{F}_p$, it is hard to compute g^{ab}.

BDH: The Bilinear Diffie-Hellman (BDH) is an assumption about symmetric pairings. Let $e : G_1 \times G_1 \to G_2$ be a Type 1 pairing, the BDH assumption states that knowing $P, [a]P, [b]P, [c]P \in G_1$, it is hard to compute $e(P, P)^{abc}$.

co-BDH: The co-Bilinear Diffie-Hellman problem is similar to the BDH but for asymmetric pairings. Let $e : G_1 \times G_2 \to G_T$, the co-BDH assumption states that knowing $P_1, [a]P_1, [b]P_1 \in G_1$ and $P_2, [a]P_2, [c]P_2 \in G_2$, it is hard to compute $e(P_1, P_2)^{abc}$.

2.9.2 Cryptanalysis and PBC

The security of the DLP for pairings

Most of the cryptanalysis effort against PBC has been devoted to solving the DLP (since k is small in PBC). Let $q = p^k$ and $n = \lceil \log_2(q) \rceil$. The general discussion about the algorithms to solve the DLP is inspired from [Ste].

The generic algorithm to solve the DLP (but it also works for the ECDLP) is the Pollar-ρ algorithm [Pol78] which has an asymptotic cost in $\sqrt{\pi n/2}$. This algorithm can be parallelized for a speed-up linear in the number of cores. For the ECDLP, the Pollar-ρ algorithm is the best known and so the security of the ECDLP of the group $E(\mathbb{F}_q)[r]$ is approximately $\lceil \log_2(r)/2 \rceil$, therefore r needs to be at least 256-bit wide at the 128-bit security level.

For the DLP, better algorithms than Pollard-ρ exist which are sub-exponential. The index calculus method consists, in a simplified explanation, in creating a factor base of $< g > \subseteq \mathbb{F}_q$ of small and irreducible elements and then to express an element g^a in $< g >$ as a product of these factors. Computing the logarithm of g^a can be done by computing the logarithms of the factors. The generic index calculus method has asymptotic cost $L_q[1/2, \sqrt{2} + o(1)]$ where $L_q[s, c] = \exp\left(c(\ln q)^s (\ln \ln q) 1 - s\right)$.

Several algorithms have been derived from the index calculus method. When p is a big prime, the Number Field Sieve has an asymptotic cost $L_q\left[1/3, \sqrt[3]{64/9}\right]$. When $p \in \{2, 3\}$, until recently the basic Function Field Sieve had asymptotic cost $L_q\left[1/3, \sqrt[3]{32/9}\right]$ but in [Jou13], Joux proposed an algorithm in $L_q\left[1/4 + \epsilon, c\right]$. More recently again, in [BGJT13], a quasi polynomial algorithm was proposed for the fields of characteristic 2 or 3. In the light of these late developments, the binary and ternary fields should now be avoided.

At the time this thesis is written, the DLP record is for the field $\mathbb{F}_{2^{9234}} = \mathbb{F}_{(2^{18})^{513}}$ where a logarithm has been computed in January 2014 in approximately 400000 core hours by Granger, Kleinjung and Zumbragel (in Jan 2014) [JOP].

Computation time versus security

From the asymptotic complexities of the cryptanalysis algorithms, hypotheses can be made on the size of the fields in order to compute a pairing for a given security level. With the recent new algorithms for solving the DLP, it is now difficult to have one unique asymptotic complexity. Instead, for each field on which one wants to solve the DLP, the attacker will borrow some steps from all these similar algorithms. As of today, the best algorithm to solve the DLP in fields of interest has complexity $L(1/4, c)$ where c is a constant with $c \geq \left(\frac{\omega}{8}\right)^{1/4}$ where ω is the matrix multiplication constant. There is a quasi-polynomial algorithm (better asymptotic complexity) but the computation time is longer for fields of interest. The estimation of the algorithm complexity of the DLP has been done for ternary fields $\mathbb{F}_{3^{6 \cdot 509}}$ in [AMORH13b], $\mathbb{F}_{3^{6 \cdot 1429}}$ in [AMORH13a] and for the binary field $\mathbb{F}_{2^{4 \cdot 1223}}$ in [GKZ14]. These fields provide a security of

respectively 2^{82}, 2^{96} and 2^{59}, values to be compared to the level of security they were previously supposed to provide: 2^{128}, 2^{192} and 2^{128} respectively.

Yet notoriously, binary and ternary pairings are faster due to simplified operations over the fields $\mathbb{F}_2$ and $\mathbb{F}_3$. As a consequence, one must compare the computation times of pairings over these fields with respect to the security they provide as shown on Table 2.8.

Table 2.8: Computation time vs security level.

Field	Computation time ($\cdot 10^3$ clock cycles on a PC)	Security level
$\mathbb{F}_{2^{1223}}$	17400 [ALH10]	2^{59} [GKZ14]
$\mathbb{F}_{3^{509}}$	15100 [BLTMR$^+$09]	2^{82}[AMORH13b]
$\mathbb{F}_p$ (BN curve)	1177 [ABLR13]	2^{128}

Small characteristic fields have a lower security level for a higher computing cost, they should now be avoided. The problem is that these fields were used in order to get supersingular curves which are mandatory for some protocols. Now that only prime fields can be used, to get supersingular curves, the embedding degree $k = 2$ must be used resulting in inefficient computations. To have efficient schemes, some protocols have to be rethought in order to work with Type 3 pairings on ordinary curves (cf. Definition 2.5.10).

2.10 Conclusion

In this chapter, we have presented the mathematical construction of pairings and proposed a consistent notation which will be reused through this thesis. We have seen that pairings rely on elliptic curves and finite fields. We have seen that a pairing computation can be split in two main algorithms, namely the Miller Algorithm (MA) and the Final Exponentiation (FE). According to the latest cryptanalytical results, our implementation is based on a twisted Ate pairing, on BN curves with $k = 12$, on a large characteristic field, with the tower extension $2 \cdot 3 \cdot 2$. A state-of-art implementation should now use these parameters at the 128-bit security level but replace the twisted Ate pairing by an OAte pairing (but the fault exploitation is a bit harder for the OAte).

Chapter 3

Setting up fault attacks against PBC

Where we detail the theoretical and practical methods for fault attacks.

Contents

The notion of security of a cryptographic algorithm has several dimensions. This security can be mathematical - how much effort (computing power, money...) is required to "break" the algorithm. This is the field of the cryptanalysis and recent developments have been made regarding this domain concerning Pairing Based Cryptography (as presented in Section 2.9.2). The security of a cryptographic algorithm should also be evaluated with respect to physical attacks. These notions of security should always be enlightened by a description of the power of the attacker. Whether an attack is achievable by a non specialist at home or by a governmental agency does not imply the same degree of vulnerability for the algorithm. Our focus on fault attacks, and the set-up we used, is justified by their relatively low cost and their high efficiency.

3.1 Physical attacks

When evaluating the security of an algorithm, one needs to consider the context of its execution. If the execution is performed on a remote server or a hand-held device does not imply the same threat models. In particular, it allows to assert potential vulnerabilities through physical attacks.

A physical attack consists in physically tampering (observing or altering) with a device to retrieve sensitive information.

3.1.1 Physical attack techniques

Physical attacks are divided into several families, the two most common ones being Side-Channels Analyses (SCAs) and Fault Attacks (FAs). SCAs are passive attacks where the attacker measures data leaking information. Different side-channels are listed below.

- Timing [KSWH98, Koc96]: the duration of the computation may depend on the secret.
- Power consumption [KJJ99]: the power consumption depends on the data handled at that time.
- EM emission [AARR03]: the EM emission depends on the data handled at that time and is localised on the chip allowing to filter irrelevant signals.
- Sound [GST13]: the sound caused by voltage regulation circuits is a low-bandwidth image of the power consumption that can be measured at distance.

Fault Attacks (FAs) are semi-invasive attacks where the attacker disrupts the normal behaviour of the algorithm (*i.e.* creates a fault) in order to make the chip leak some information. Some fault injection techniques are listed below.

- Clock glitches [ADN+10]: a particular clock period is shortened as to create set-up time violations in the registers.
- Voltage and Temperature [ZDC+12]: outside the nominal values for the temperature and the power voltage, glitches can occur that can leak sensitive data.
- Laser and light fault injection [SA03]: a chip may be vulnerable to laser fault injections where a laser interacts with the logic gates or the memory to change the data.
- EM fault injection [DDRT12]: an EM pulse sent onto a chip can modify its behaviour. Details on this technique are provided later since we used this fault injection technique.

A third family exists, the invasive attacks where the chip is modified (*e.g.* with a Focused Ion Beam and micro-probing [HNT+13]) but they won't be considered in this document since there are still considered high-end attacks for well-funded adversaries.

3.1.2 Fault models

Our analysis focuses on fault attacks on pairings since fault attacks are more algorithm-specific than side-channel analyses. Fault attacks are more complex to implement but are potentially more dangerous, due to their high efficiency and the difficulty to circumvent them. When doing a fault injection attack, fault models must be considered with respect to the kind of faults that are injected.

Bit-level faults

A first fault effect can be the modification of a data, in Random Access Memory (RAM) or in a register. The fault can create one or several bit-flips, *i.e.* the bit values are changed from 0 to 1 or from 1 to 0. In this case, the fault is modelled by the XOR function. The fault can also be a stuck-at (0 or 1) fault. A stuck-at 0 (respectively 1) fault leaves unchanged the bit if its value is 0 (respectively 1) or flips it if its value is 1 (respectively 0).

The two most common fault models at bit-level are single-bit faults, where a single bit of data is modified or single-word faults (a single word is modified, the size of the world depends on the machine word size which is commonly 8, 16, 32 or 64 bits).

Instruction skips

A higher-level fault effect is the instruction skip. This fault model can be used when injecting faults on a microcontroller. The effect of the fault is the skip of one instruction in the program. The instruction skip can create a fault on the data (*e.g.* an arithmetic instruction has been skipped), or a fault on the control flow (*e.g.* to skip a loop test causes the program to exit the loop).

The instruction skip model is particularly useful when attacking a pairing, since it allows to exit the Miller algorithm whenever we want.

3.1.3 Examples of fault attacks

The first fault attack against a cryptographic algorithm was proposed by Boneh *et al.* in [BDL97]. They showed that one random fault on an implementation of an RSA-CRT (Chinese Reminder Theorem) was enough to totally recover the secret key. Later Biham and Shamir in [BS97] showed that the same principles apply to other algorithms. Since then fault attacks have become a domain of study and are extensively studied on various algorithms. Historically, practical research on the effect of faults on integrated circuits mainly comes the space industry where lasers have been used to simulate the effect of cosmic radiations on space-ready chips [Hab65]. In the context of fault attacks One of the most studied algorithm is the AES. Its structure allows an easy study of the fault attacks. Our work [LRD+12] includes a study of the fault models on the AES, which is not adaptable to pairings. Yet the method used (adopting a system point of view) was the origin of the fault attack proposed in Section 5.2.

3.2 Setting-up the EM bench for injecting faults

In this section, we describe the apparatus, the target and the calibration for our fault injection. The Electromagnetic fault injection bench is similar to the one described in [DMM+13, MDH+13]. *The set-up of the EM bench was a laboratory team work, mainly with Amine Dehbaoui and Nicolas Moro (his contribution will be given in his thesis).*

3.2.1 Device Under Test

The targeted chip is an STM32F100RB, a 32-bit microcontroller implemented in a CMOS 130 nm technology embedding an ARM Cortex-M3 core (with the Thumb-2 instruction set) and running at 56 MHz (hence a clock cycle of ≈17.8 ns). This chip is embedded on a STM32VLDISCOVERY board [STM] which was used with an external power source (and not the Universal Serial Bus (USB) power). This chip is a modern microcontroller in the medium performance range and has no dedicated cryptographic functionality or accelerator which means that our twisted Ate pairing takes ≈16 s to compute, 9 s for the Miller algorithm and 7 s for the final exponentiation (as a comparison our code also implements an OAte pairing (which uses the same low-level routines) that runs in 17 ms on one core of an Intel Core i5 2430M).

The Thumb-2 instruction set possesses 32-bit and 16-bit instructions. For example, two NOP instructions can be fetched in one clock cycle. The core has a three stage pipeline (FETCH, DECODE, EXECUTE). It is suspected that the EM injection disrupts the FETCH stage by modifying the data on the bus (the data can be a memory value but also an instruction). The result can either be assimilated to an instruction skip (the instruction microcode has been modified which replaces the instruction with another) or to a data modification.

The chip has not been designed as a secure chip (in particular there is no shielding) but it bears some basic sensors for monitoring power and clock glitches which trigger hardware interrupts. These sensors are active during the experiments and are able to detect the EM pulses if they are too powerful. Several types of interrupts are possible (Hard fault, Bus fault, Usage Fault and Memory Management) and they give details about the detected effect of the EM fault if they are raised. The board is underpowered at 2.8 V instead of 3.3 V in order to increase the sensitivity of the chip to the EM pulses.

The chip is linked to the controlling computer through an ST-Link (JTAG-equivalent) connection. This connection is managed via Keil's μvision UVSOCK library [ARM13]. It allows us to access the internal state of the microcontroller at user-defined breakpoints.

3.2.2 Targeted program

In order to realize our fault attacks, we created our own software implementation of a pairing computation, inspired from the Miracl library [Cer12] (the structure of the program has been kept similar but we have redeveloped everything). Having our own library allowed us to have complete control of the targeted program with easier possibilities to modify the program. This library allows to compute the Tate and Ate pairings with parameters taken from [BGDM+10] using algorithms described in Chapter 2. The curve used is a BN curve at the 128-bit security level. The software implementation is ≈ 1800 lines of C code long. The memory consumption on the microcontroller is 7.2 kB of RAM and 13.6 kB of ROM when compiled without optimizations. The computation time is ≈ 16 s. A profiling of the same code done on a Personal Computer (PC) gives us the ratio of the computation times $Mult(\mathbb{F}_p)/Add(\mathbb{F}_p) = 9.7$. As a comparison, in [GL09] the authors used a MSP430 microcontroller at 8 MHz, computing an Optimal Ate

pairing on a BN curve in 14.7 s, using 4.7 kB of RAM and 32.3 kB of Read-Only Memory (ROM)

Table 3.1: Number of calls to $\mathbb{F}_p$ primitive operations in our implementation.

# calls	$\mathbb{F}_p$ additions	$\mathbb{F}_p$ subtractions	$\mathbb{F}_p$ multiplications	$\mathbb{F}_p$ inversions
Targeted Ate pairing	70538	41491	23020	1
Optimal Ate pairing	50811	30023	15993	1

3.2.3 Targeted protocol

Our attack scenario supposes that a pairing running in a protocol where one of the two input points is the secret is targeted. It has been claimed in [CKM14] that such a protocol does not exist for asymmetric pairings. Yet in the full version of [BF01], the authors remark that their scheme is compatible with asymmetric parings by replacing the BDH assumption by the co-BDH assumption in the security proof. Their IBE scheme is a relevant example of a protocol with an asymmetric pairing where one of the two input points is a secret (in the decryption phase). If an attacker is able to find this secret point, he is able to impersonate the target.

3.2.4 Apparatus

The apparatus for the EM fault injection is composed of several components as illustrated in Figure 3.1. The targeted chip sends a trigger signal at the desired instant of injection. This trigger is detected by the pulse generator which creates a pulse into the EM probe (a coil antenna located at the surface of the chip), which in turn injects a fault into the chip. The pulse generator has rising and falling edges of 2 ns. A fault is created when the EM pulse reaches the Power Ground Network of the chip. A precise explanation of the physical phenomena at stake are out of the scope of this thesis, see [Deh11] (in French) for more details.

A computer is used to control the operations and the parameters (noted in capital letters) of the experiment: the X and Y locations of the EM probe with respect to the chip, the amplitude (AMP) of the pulse generated by the pulse generator, the pulse width (WIDTH), and the delay (DELAY) between the trigger sent by the chip and the fault injection.

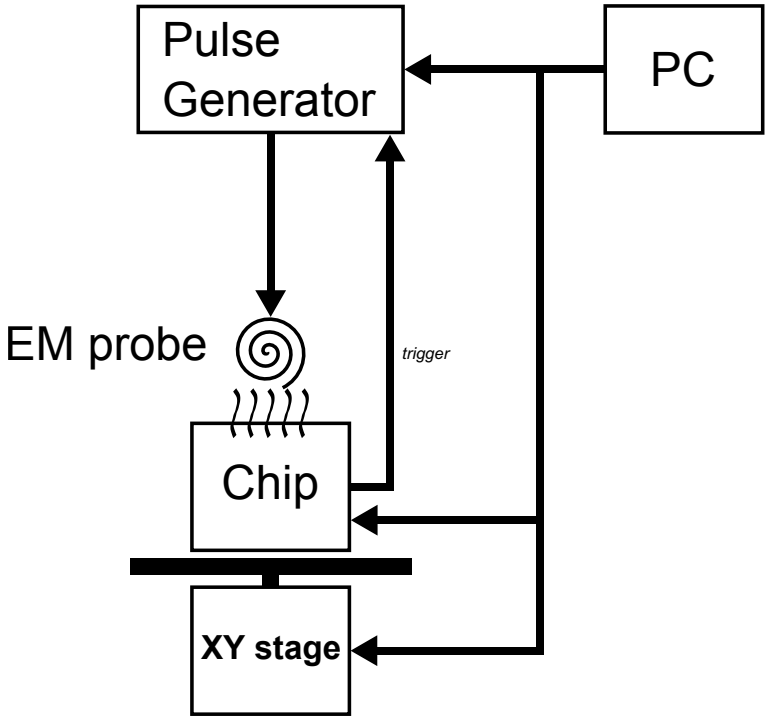

Figure 3.1: EM bench scheme for fault injection.

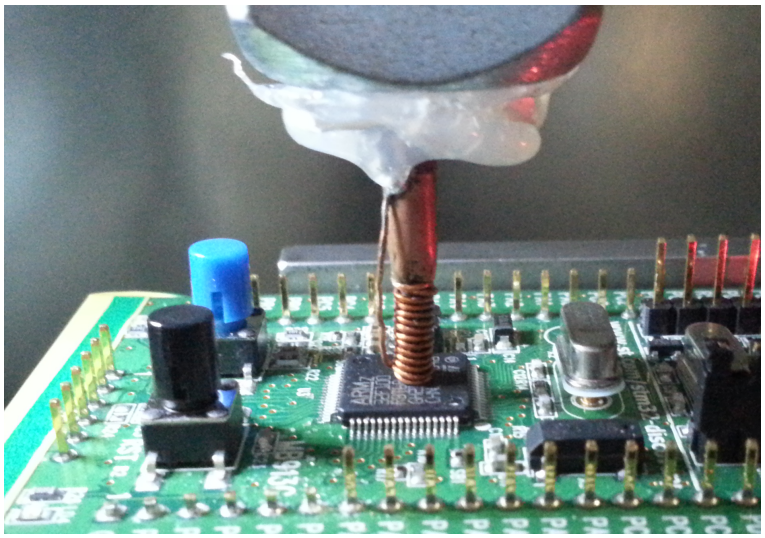

Figure 3.2: Zoom on the EM probe.

The diameter of the probe is 1 mm but the XY stage controlling the relative position between the targeted chip and the probe has a 1 µm resolution. The computer allows us to automatize the experiments allowing the exploration of the effect of changing various parameters. In particular, we automatized the XY stage to perform automatic XY scans of the chip, where a fault injection is tried at each point of a surface of the chip, defined by the experimenter, with the desired resolution ($\geq$ 1 µm).

3.2.5 Preliminary experiments

The first goal of the attack is to find a set of parameters (XY location, AMP, WIDTH, DELAY) that creates any fault on the device.

Changing the pulse width WIDTH in the range from 10 ns to 100 ns had have no consequence on the results, so we keep this value at 10 ns throughout all the experiments. A possible reason for this may be that the chip is mainly vulnerable to the pulse edges, thanks to the voltage regulation in the chip and the physical laws governing the EM coupling.

Our first experiment was to spatially localize a vulnerable spot on the chip. A vulnerable spot is a location where we are able to inject an fault undetected by the chip or if the chip raises an interrupt. A location where a fault raises an interrupt can often be used to inject an undetected fault. To avoid detection, it is often enough to decrease the intensity of the EM pulse. In this section, we do not care about the kind of undetected fault that we create. In this case, an undetected fault means that one of the chip registers or one of the counters (see below) has been modified. An explanation of the precise effects of the EM pulses on the chip which induce faults is not covered by this thesis. Indeed for that purpose, one would require a privileged access to the internal states of the targeted chip (and to the pipeline stages in particular) through an access to the "test mode". Without this access, we can only make assumptions (as in Section 5.3).

To find the right set-up, a dummy algorithm (cf. Code 3.1, a loop with 3 counters) has been first used to have a faster execution and an easier interpretation of the effects of the faults.

Code 3.1: Dummy algorithm for EM injection

```
int main(void)
{
	int i; //loop counter

	//3 dummy counters
	int ct1 = 0;
	int ct2 = 0;
	int ct3 = 0;

	Hardware_Init();

	for(i = 0; i < 256; i++)
	{
		GPIOB->BRR = GPIO_Pin_8;//trigger down (no effect if already down)

		ct1++;
		ct2 += 2;
		ct3 += 3;

		if(i == 254)
		{
			GPIOB->BSRR = GPIO_Pin_8; //trigger up
			__NOP();//force a NOP instruction to clear the pipeline and account for the 80ns
			    minimum delay between trigger and fault injection
			__NOP();
			__NOP();
			__NOP();
			__NOP();
		}
	}

	__NOP();//breakpoint here to read data after fault injection
	__NOP();

	return 0;
}
```

First a coarse scan of the chip is made where faults are injected randomly on the chip. When one interrupt is raised, we fix all parameters and then vary them independently. Between each fault injection, the chip is reset so as to have a clean state, the previous fault does not influence the next result.

First we realize a coarse XY mapping of the chip's responses to fault injections (cf. Figure 3.4). Some areas create chip crashes, *i.e.* locations where we are not able to communicate with the chip after the fault injection, forcing us to reprogram it. Some zones raise interrupts, some create undetected faults on data in RAM or in registers. Finally the fault injection has no observable effect on some areas of the chip. The vulnerable surface is illustrated in Figure 3.3, and is compared to the size of the chip and to the diameter of the EM probe.

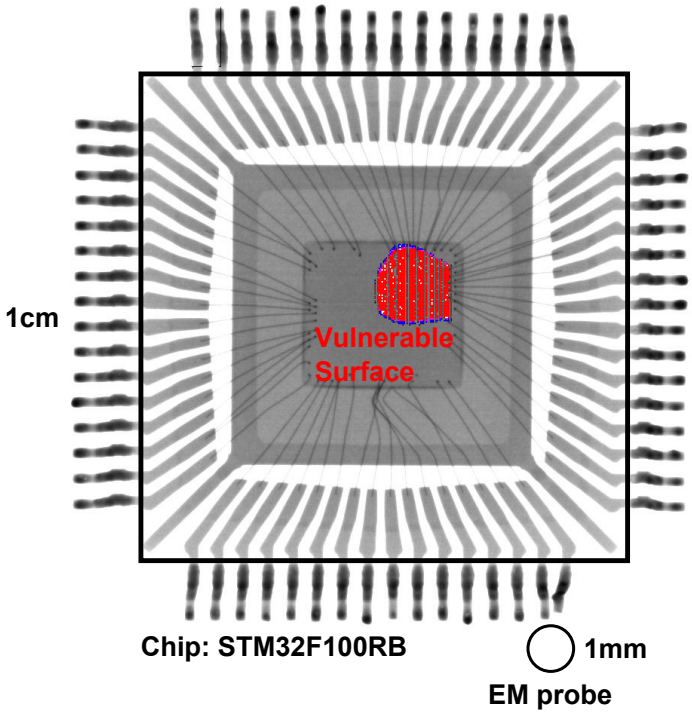

Figure 3.3: Scales for the chip, the probe and the vulnerable surface.

The coarse map allows then to realize a fine-grained XY mapping (cf. Figure 3.6) at a location susceptible to create undetected faults after the fault injection.

Usually when an interrupt is raised, it is enough to reduce the amplitude of the pulse to get an undetected fault as can be seen on Figure 3.7.

Once the XY coordinates have been fixed, we vary the DELAY parameter (relative delay between the trigger raised by the chip and the EM pulse) between 300 ns and 550 ns (cf. Figure 3.8).

At this point the effects of all fault injections are observed, which allows us to build a model of the effect of the faults on the computation. The adequate delay (DELAY) is chosen according to the desired effect.

Finally, the amplitude of the pulse generator (AMP) is tuned until the created fault is undetected by the chip. We preferred to use a negative amplitude (down to -210 V) rather than

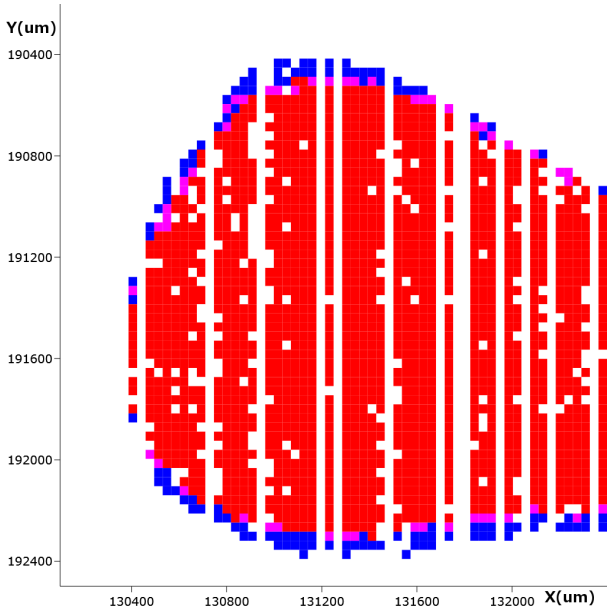

Figure 3.4: Coarse XY mapping with AMP = −210 V. Red (light) if the fault injection raised an interrupt, blue (dark) when an undetected fault has been created.

a positive one since the chip seems to be more sensitive to a negative pulse (cf. Figure 3.9).

These preliminary experiments allowed us to determine that for this chip, in order to obtain exploitable faults, the parameters X =131 200 µm, Y =191 500 µm, WIDTH =10 ns are the best ones to use, independently from the targeted software. Additionally, we have seen that the DELAY and AMP parameters must be modified for each desired effect onto the computation, for each targeted software.

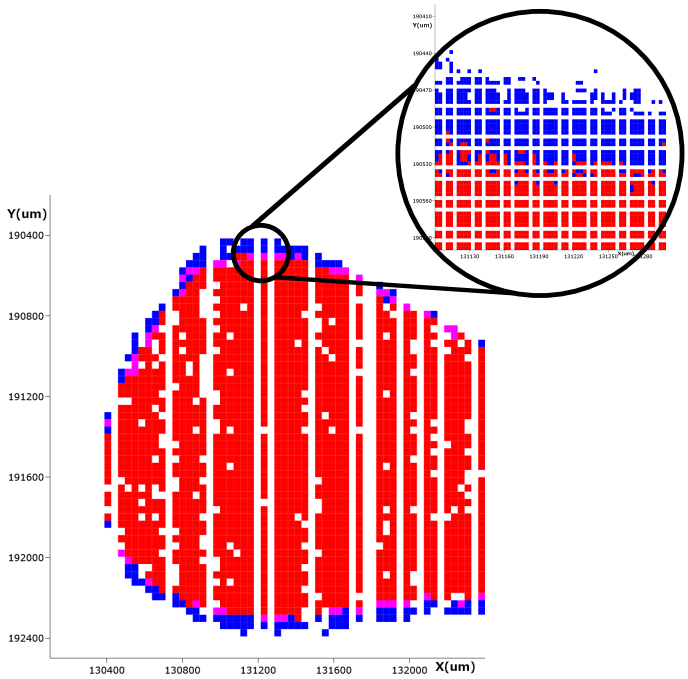

Figure 3.5: Coarse and fine-grained XY mappings scales.

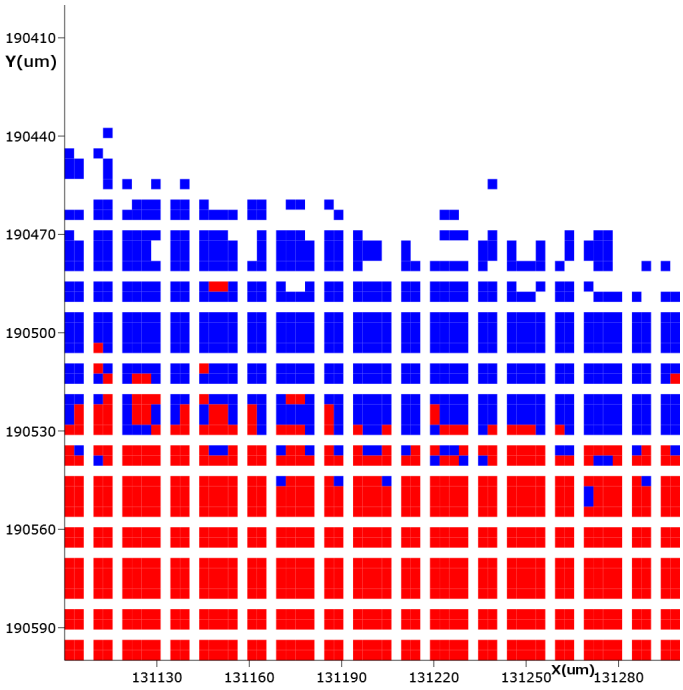

Figure 3.6: Fine-grained XY mapping. Red (light) if the fault injection raised an interrupt, blue (dark) when an undetected fault has been created.

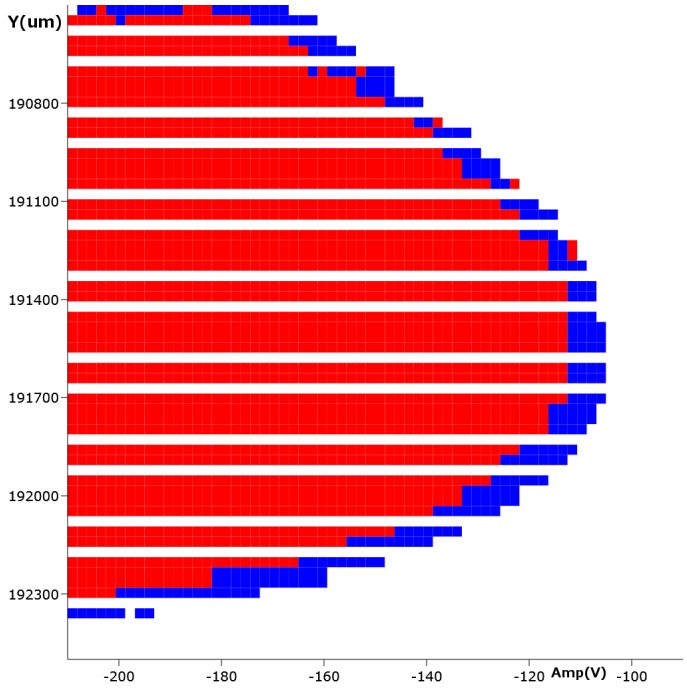

Figure 3.7: Y coordinate versus AMP at X =131 200 µm. Red (light) if the fault injection raised an interrupt, blue (dark) when an undetected fault has been created.

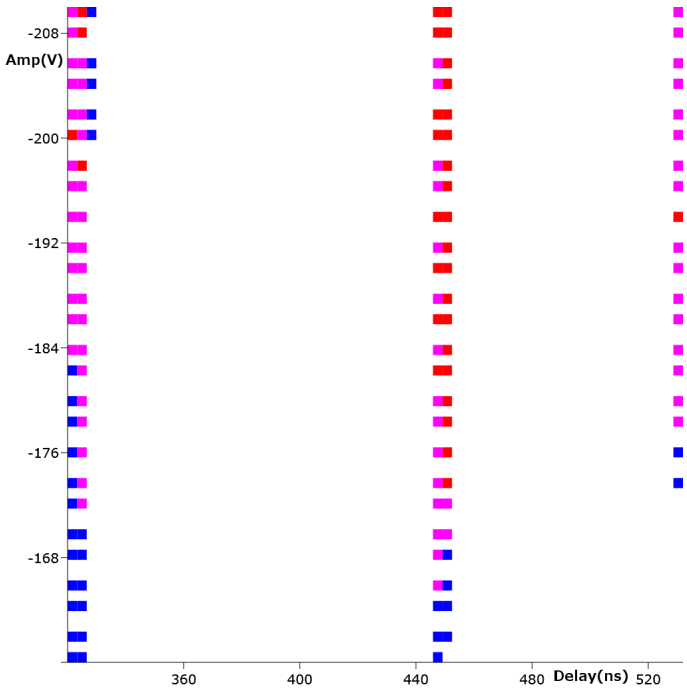

Figure 3.8: AMP versus DELAY. Red (light) if the fault injection raised an interrupt, blue (dark) when an undetected fault has been created.

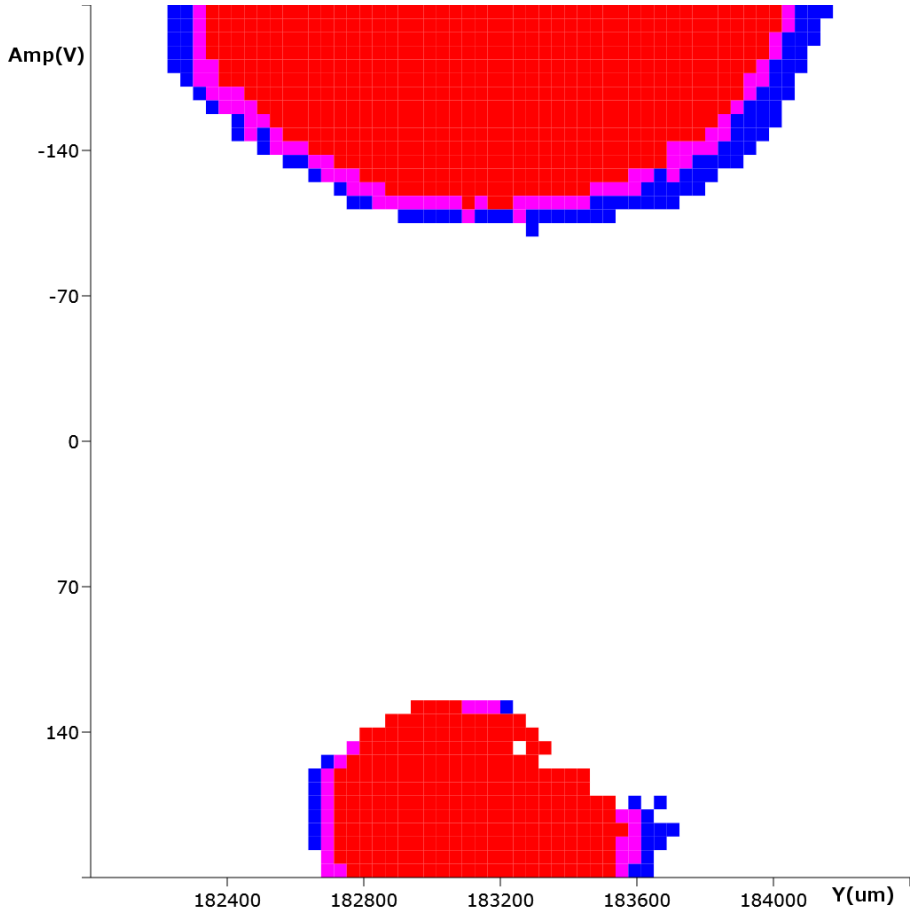

Figure 3.9: AMP versus Y coordinate. Red (light) if the fault injection raised an interrupt, blue (dark) when an undetected fault has been created. The X scale is different since this experiment was done during a campaign different from the others.

3.3 Conclusion

With a properly calibrated EM bench, it is now possible to target pairing algorithms. Our fault injection method has a low cost and is easily set-up (almost plug-and-play). We have characterized our targeted chip and evaluated its sensitivity to EM fault injections. We have identified a vulnerable surface where we will perform all our fault injections. We noticed that the pulse width has no effect on the faults created. Our successive attacks will lead us to a practical inversion of a complete pairing (twisted Ate on BN curves).

Chapter 4

Fault attacks on the Miller algorithm

Where the security of the Miller algorithm with respect to FAs is analysed.

Contents

In this chapter, the security of the Miller algorithm is evaluated independently of the final exponentiation (*i.e.* it is considered that the result of the Miller algorithm is easily available). In practice, the final exponentiation can be simple (with small characteristic fields) or even non existent, *e.g.* the Weil pairing. Even for Tate-like pairings with a large prime field, the resistance of the Miller algorithms to fault attacks is of interest in order to eventually protect the whole pairing computation. The Miller algorithm is first analysed theoretically and in a second time some of these theoretical attacks are tested in practice. We focus on the instruction skip fault model as presented in Section 3.1.2 since this model allows to simulate all the fault models proposed against pairings in the literature.

Throughout this document, an attack on the Miller algorithm means that one of the two inputs point is the secret sought while the other is public and therefore known to the attacker. The two possible cases are

- P is secret, Q is public,
- Q is secret, P is public.

These assumptions represent protocols where the pairing computation involves a secret (*e.g.* the decryption algorithm in the IBE scheme in Section 2.8.2). Yet sometimes, the pairing computation itself does not handle any secret (*e.g.* Joux's tripartite key exchange in Section 2.8.1). In this case, a physical attack on the pairing cannot break the protocol. Still, if classical cryptanalysis

allowed to reverse a pairing, even protocols where no secret is handled by the pairing computation would be put at risks.

Indeed, let F be an algorithm allowing to find the secret Q from P and $e(P,Q)$. Supposing a tripartite key exchange protocol as presented in Section 2.8.1. The attacker, by listening to the communications, can know $P, [a]P, [b]P$ and $[c]P$. In order to find the secret shared key, she can perform the following steps.

- She computes $z = e([a]P, [c]P)$.
- With F, she can find a point Q such that $e(P,Q) = z$.
- Finally, she computes $e([b]P, Q) = e(P,Q)^b = z^b = e(P,P)^{abc}$, the secret key.

For our practical experiments, a particular setting has been chose. Our choice was a twisted Ate pairing on a BN curve (cf. Section 2.5.7) at the 128-bit security level with parameters taken from [BGDM+10]. In this setting, the point P is stored in affine coordinates and the point Q is stored in jacobian coordinates. A BN curve is defined by the parametrized values $t(x)$, $r(x)$ and $p(x)$ such that x fully defines the curve, with $r(x)$ and $p(x)$ prime integers. In our case $x =$ `0x3FC0100000000000` and $\mathbb{F}_{p^{12}}$ is constructed through the following tower extension:

$$\begin{aligned}\mathbb{F}_{p^2} &= \mathbb{F}_p[u]/(u^2 - \beta),\\ \mathbb{F}_{p^6} &= \mathbb{F}_{p^2}[v]/(v^3 - u),\\ \mathbb{F}_{p^{12}} &= \mathbb{F}_{p^6}[w]/(w^2 - v),\end{aligned}$$

where $\beta = -5$ is a quadratic non-residue in $\mathbb{F}_p$, u is a cubic non-residue in $\mathbb{F}_{p^2}$, and v is a quadratic non-residue in $\mathbb{F}_{p^6}$.

The base for $\mathbb{F}_{p^{12}}$ as a vector space over $\mathbb{F}_{p^2}$ is then $(1, w, w^2, w^3, w^4, w^5)$ and the base for $\mathbb{F}_{p^{12}}$ over $\mathbb{F}_p$ is $(1, w, w^2, w^3, w^4, w^5, w^6, w^7, w^8, w^9, w^{10}, w^{11})$, and for $R \in \mathbb{F}_{p^{12}}$, we denote

$$R = \sum_{i=0}^{11} R_i \cdot w^i, \tag{4.1}$$

with $w^2 = v \in \mathbb{F}_{p^6}, w^6 = u \in Fsq$ and $w^{12} = \beta \in \mathbb{F}_p$.

The BN curve E that we use as an example is defined by the equation $Y^2 = X^3 + 5$ and the twisted curve E' is given by $Y^2 = X^3 - 5/u = X^3 - u$. We know that $Q \in E(\mathbb{F}_{p^{12}})$ and $P \in E(\mathbb{F}_p)$. However we use the degree 6 (called sextic) twisted curve for the representation of Q: we simplify the notation by denoting Q as the point $(x_q : y_q) \in E'(\mathbb{F}_{p^2})$ instead of $(x_q \cdot w^2 : y_q \cdot w^3) \in E(\mathbb{F}_{p^{12}})$.

In the Miller algorithm, to simplify the notation as usually done in the literature, the tangent evaluation at point P is noted $h_1(P)$ instead of $l_{T,T}(P)/v_{[2]T(P)}$ while the line evaluation is noted $h_2(P)$ instead of $l_{Q,T}(P)/v_{T+Q}(P)$.

4.1 Theoretical fault attacks on the Miller algorithm

In this section, we propose a review of the theoretical fault attack,s which have been proposed against the Miller algorithm (Algorithm 3), adapted to our implementation. First are presented two families of fault attacks on the Miller algorithm which allow to recover the value $h_1(P)$. Then the method to find the secret from $h_1(P)$ is proposed. Finally we analyse the countermeasures allowing to protect the Miller algorithm.

4.1.1 Data-flow attacks on the Miller algorithm

A data-flow attack is an attack where the injected fault affects a data value but not the course of the algorithm.

Whelan *et al.* fault attack [WS07]

Such an attack has been proposed by Whelan *et al.* in [WS07]. The authors propose a complete fault attack on an η pairing (on small characteristic curves) but their proposition against the Miller algorithm with ordinary elliptic curves in $\mathbb{F}_{p^2}$ is more pertinent now that small characteristic fields are deprecated.

In [WS07], the authors show that with a sign-change fault attack (where the sign of one of the coordinates of the line evaluation result is changed), they are able to reverse a Weil pairing by observing the ratio of a faulted result over the correct one. This ratio gives a system which is then solved. Yet this method works only with a Weil pairing with a simple final exponentiation $p-1$ (added to allow the denominator elimination optimization) and they show that it does not work for a Tate pairing with final exponentiation $(p^2-1)/r$ since they are not able to reverse the final exponentiation.

A conclusion drawn by the authors is that Tate pairings are immune to fault attacks thanks to the final exponentiation. They argue that the only way to bypass this protection is by a fault attack on the final exponentiation combined with a fault on the Miller loop, therefore requiring double faults.

An adaptation of this attack for symmetric pairings with a large prime field and $k = 2$ has been proposed in [CKM14].

Variant with a controlled-add fault model

In our implementation, sign-change faults are impossible since our finite field elements representations are always positive values (the sign change becomes $-x = p - x$). Yet it is possible to create a fault on a data which allows to revert the Miller algorithm. If the fault value is unknown, we would like the value to have a manageable entropy which is what the term "controlled" stands for. We propose a fault attack which is able to invert the Miller algorithm on a BN curve, $k = 12$ and with mixed coordinates for the input points.

In this attack, the secret point is Q and P is known to the attacker. We inject a fault during an addition in $\mathbb{F}_p$. The latter operation requires a multi-word addition algorithm on the 32-bit chip which is the reason why the fault value is limited to ≈ 32-bits. It is possible to use a fault on the modular addition to recover $h_1(P)$ if the attacker knows the point P. In this case, he can target the evaluation of one of the coordinates (R_0, R_3 or R_4, cf. Equation (4.1)) of $R = h_1(P)$ during the last iteration of the Miller algorithm. For example the value R_0 is computed with an algorithm ending with the following pseudo-C code ($t_0 \in \mathbb{F}_{p^2}$):

```
t0 = t0 + t0; //fast modular doubling
R0 = t0 * YP; //P = (XP : YP)
```

The attacker can recover $h_1(P)$ by injecting a known fault e on the modular addition giving $t_0^* = t_0 + e$. This fault is propagated onto R_0:

$$R_0^* = t_0^* \cdot YP = (t_0 + e) \cdot YP = R_0 + e \cdot YP = R_0 + \Delta_{R_0}. \tag{4.2}$$

Since e and YP are known to the attacker, Δ_{R_0} is known as well. At the last iteration of the Miller algorithm, we have:

$$f_{K,Q}(P) = f_1^2 \times h_1(P) \tag{4.3}$$

If the attacker is able to inject a known fault Δ_{R_0} in $h_1(P)$, he recovers

$$f_{K,Q}(P)^* = f_1^2 \times (h_1(P) + \Delta_{R_0}). \tag{4.4}$$

As he knows $f_{K,Q}(P)^*$, $f_{K,Q}(P)$ and Δ_{R_0}, he can find $h_1(P)$:

$$h_1(P) = \frac{f_{K,Q}(P) \times \Delta_{R_0}}{f_{K,Q}(P)^* - f_{K,Q}(P)}. \tag{4.5}$$

If P is the secret and Q is known, it is possible to obtain the same result with a fault on the last operation computing R_3 which is a modular subtraction.

4.1.2 Control-flow attacks on the Miller algorithm

Another fault model is when a fault is injected to corrupt the flow of the program (*e.g.* on branch-like instructions).

Attack on the Duursma-Lee algorithm [DL03]

The Duursma-Lee algorithm [DL03] is a variant of the Miller algorithm optimized for fields of characteristic 3. Since these fields should not be used anymore, the description here has mainly an interest as an historical context. Indeed, the fault attack proposed in [PV06] was the first one against a pairing. This attack is a control-flow attack. The parameters below are taken from this paper.

The Duursma-Lee algorithm (Algorithm 4) computes a pairing over supersingular elliptic curves over $\mathbb{F}_q$ with $q = 3^m$ and $k = 6$. Curves with equation

$$E : y^2 = x^3 - x + b, \tag{4.6}$$

where $b = \pm 1$. The tower field used is $\mathbb{F}_{q^3} = \mathbb{F}_q[\rho]/(\rho^3 - \rho - b)$ and $\mathbb{F}_{q^6} = \mathbb{F}_{q^3}[\sigma]/(\sigma^2 + 1)$. $G_1 = E(\mathbb{F}_q)[r]$ (same r as defined in Section 2.4.3), $G_2 = \mu_r \subset \mathbb{F}_{q^6}^*$.

In [PV06], the authors propose an attack where a faulty and a correct execution of the pairing computation are performed. The faulty computation has an additional iteration in the Miller loop. In this case, the ratio of the faulty result over the correct result (discarding the final exponentiation) gives the value

$$g_{m+1} = -y_P^{3^{m+1}} \cdot y_Q \cdot \sigma - \mu_{m+1}^2 - \mu_{m+1} \cdot \rho - \rho^2, \tag{4.7}$$

where $\mu_i = x_P^{3^i} + x_Q^{3^{m-i+1}} + b$.

Algorithm 4: The Duursma-Lee algorithm.

Data: $P = (x_P, y_P) \in \mathbb{G}_1$ and $Q = (x_Q, y_Q) \in \mathbb{G}_2$.
Result: $e(P, Q) \in \mathbb{G}_3$.

$f \leftarrow 1$;
for $i = 1$ **to** m **do**
  $x_P \leftarrow x_P^3$, $y_P \leftarrow y_P^3$;
  $\mu \leftarrow x_P + x_Q + b$;
  $\lambda \leftarrow -y_P y_Q \sigma - \mu^2$;
  $g \leftarrow \lambda - \mu\rho - \rho^2$;
  $f \leftarrow f \cdot g$;
  $x_Q \leftarrow x_Q^{1/3}$, $y_Q \leftarrow y_Q^{1/3}$;
end
return f^{q^3-1};

Since $\forall z \in \mathbb{F}_{3^m}, z^{3^m} = z$ and by decomposing g_{m+1} into $\mathbb{F}_q$, it is possible to find the secret P knowing Q.

The authors then extend their fault model to make their attack work when the attacker is only able to force the algorithm to perform a random number of iterations. By repeating the injection process, results with a consecutive number of iterations are rapidly found and can be exploited.

The next phase is to invert the final powering by the factor $q^3 - 1$. Let S be the result of the Miller algorithm and R the pairing result, *i.e.* $R = S^{q^3-1}$. R has several preimages by the final exponentiation but S^*/S can be distinguished by its particular form:

$$\frac{R^*}{R} = \left(\frac{S^*}{S}\right)^{q^3-1} = g_{m+1}^{q^3-1}, \tag{4.8}$$

where R^* and S^* are the faulty results. Inverting the final exponentiation can be done in two steps, first by finding any correct root g of $R = g^{q^3-1}$, then by deriving the correct answer g_{m+1} from g. It is easy to find the roots of $R = X^{q^3-1}$ by remarking that this equation is equivalent to

$$X^{q^3} - R \cdot X = 0, \tag{4.9}$$

which is a linear equation in X. Then the special relations that g_{m+1} must satisfy are used (details in [PV06]) to find the correct preimage by the final exponentiation.

An adaptation of this attack for symmetric pairings with a large prime field and $k = 2$ has been proposed in [CKM14].

El Mrabet's fault attack [EM09]

In [EM09], El Mrabet extends the work by Page *et al.* in [PV06] to attack the Miller algorithm in a more general setting. The author shows that by obtaining two results of the Miller algorithm with a consecutive number of iterations, the secret can be recovered. The explanations below are given for the particular case of our implementation (Twisted Ate pairing, $k = 12$ BN curve, with mixed coordinates) and by comparing the correct result with a faulty result without the last iteration. But it can be generalised for all Tate-like pairings and coordinate systems. Additionally, it is enough for the attacker to find two faulty results that have executed a consecutive number of iterations (one has τ iterations and the other one has $\tau + 1$ iterations).

In the Ate pairing, the last iteration is a tangent evaluation only:

$$f_{K,Q}(P) = f_1^2 \times h_1(P). \tag{4.10}$$

Thus if we skip the last iteration, we obtain

$$f_{K,Q}(P)^* = f_1. \tag{4.11}$$

Finally, h_1 is simply

$$h_1(P) = \frac{f_{K,Q}(P)}{(f_{K,Q}(P)^*)^2}. \tag{4.12}$$

This method can be quite generally applied with various curves or coordinate systems.

Exit after the first iteration attack

A particular case of the previous attack is when the attacker is able to obtain the faulty result of the Miller algorithm if he exits the loop after the first iteration.

In this case,

$$f_{K,Q}(P)^* = h_1(P) \times h_2(P) \tag{4.13}$$

or

$$f_{K,Q}(P)^* = h_1(P), \tag{4.14}$$

depending on K. No other value is needed to recover the secret as demonstrated in Section 4.1.3. This way only one faulty execution of the Miller algorithm is required to find the secret.

Bae *et al.* fault attack [BMH13]

In [BMH13], Bae *et al.* propose another control-flow fault attack by skipping the addition step at the last iteration of a Tate pairing (there is no addition step at the last iteration of an Ate pairing since the Miller index K is even). They skip the addition step by targeting the *if* instruction with a fault attack. In the Tate pairing, the last iteration is

$$f_{K,Q}(P) = f_1^2 \times h_1(P) \times h_2(P). \tag{4.15}$$

By skipping the addition step, they have

$$f_{K,Q}(P)^* = f_1^2 \times h_1(P). \tag{4.16}$$

Finally, $h_2(P)$ can be found with

$$h_2(P) = \frac{f_{K,Q}(P)}{f_{K,Q}(P)^*} \tag{4.17}$$

The secret can be recovered with $h_2(P)$ as shown in Section 4.1.3.

4.1.3 How to find the secret

The equations proposed in this section are derived from the previous works [PV06, EM09], with refinements to encompass our particular cases.

Finding the secret knowing $h_1(P)$ (Ate pairing)

Now that we have seen how the value of $h_1(P)$ is recovered with fault attacks, we are going to illustrate how the secret point is derived from the latter value. Q is represented in the degree 6 twisted curve. Q is represented as the point $(x_q : y_q) \in E'(\mathbb{F}_{p^2})$ instead of $(x_q \cdot w^2 : y_q \cdot w^3) \in E(\mathbb{F}_{p^{12}})$. Additionally, the point Q (and therefore T) is represented in jacobian coordinates $(X_Q : Y_Q : Z_Q)$ which corresponds to the affine representation $(X_Q/Z_Q^2 : Y_Q/Z_Q^3)$. The attacker knows $h_1(P)$ for the Ate pairing with (cf. Section 2.4.2, Formulae for line equation):

$$\begin{aligned} h_1(P) =&(3X_T^3 - 2Y_T^2) \cdot w^6 + 2Y_T Z_T^3 y_p \cdot w^3 \\ &- 3X_T^2 Z_T^2 x_p \cdot w^4, \\ =&R_0 + R_3 \cdot w^3 + R_4 \cdot w^4, \end{aligned} \tag{4.18}$$

with $R_0, R_3, R_4 \in \mathbb{F}_{p^2}$ (since $w^6 = u \in \mathbb{F}_{p^2}$) recovered through identification and $T = [i]Q$ for some i known to the attacker. For the Ate pairing R_0, R_3, R_4 provide a system in $\mathbb{F}_{p^2}$:

$$\begin{cases} R_0 &= (3X_T^3 - 2Y_T^2) \cdot u, \\ R_3 &= 2Y_T Z_T^3 y_p, \\ R_4 &= -3X_T^2 Z_T^2 x_p. \end{cases} \tag{4.19}$$

First, if P (for the Ate pairing) is the secret and Q is known, P can trivially be obtained with this system since $T = [i]Q$ is known to the attacker and the system is linear in P coordinates:

$$\begin{cases} x_p &= \frac{-R_4}{3X_T^2 Z_T^2}, \\ y_p &= \frac{R_3}{2Y_T Z_T^3}. \end{cases} \tag{4.20}$$

When the secret point is Q while P is known, the solution is barely more complex. In this case, the system yields the univariate polynomial

$$\frac{R_0^2}{\beta} \cdot Z_T^{12} + \left(4\frac{R_0}{u}\lambda_2^2 - 9\lambda_3^3\right) \cdot Z_T^6 + 4\lambda_2^4 = 0 \tag{4.21}$$

with $\lambda_2 = \frac{R_3}{2y_p}$ and $\lambda_3 = -\frac{R_4}{3x_p}$. This polynomial can be solved on $\mathbb{F}_{p^2}$ providing candidate values for Z_T. Once we know Z_T, we use it into the initial system to obtain X_T and Y_T. The points which do not lie on the curve are eliminated. Finally, the possibilities for $Q = [i^{-1}]T$ are computed. $h_1(P)$ is a sparse vector, out of its 12 coordinates, only 6 are not equal to 0.

Finding the secret knowing $h_1(Q)$ (Tate pairing)

In this section, the computations are done for a twisted Tate pairing: $T = [i]P$ for some i, $Q \in E'(\mathbb{F}_{p^2})$, $P \in E(\mathbb{F}_p)$. The point P and Q are switched compared to the Ate pairing. It is easier to find P knowing $h_1(Q)$ than to find Q knowing $h_1(P)$ because there are fewer unknowns in the first case.

The attacker can recover $h_1(Q)$ with

$$h_1(Q) = \left(3X_T^3 - 2Y_T^2\right) - \left(3X_T^2 Z_T^2 x_q\right) \cdot w^2 + \left(2Y_T Z_T^3 y_q\right) \cdot w^3, \tag{4.22}$$

$$h_1(Q) = R_0 + R_2 w^2 + R_3 w^3, \tag{4.23}$$

with $R_0, R_2, R_3 \in \mathbb{F}_{p^2}$ recovered through identification and $T = [i]P$ for some i known to the attacker (*e.g.* obtained by monitoring the timing of the computation).

We focus on the case where P is the secret and Q is known. The system of equation can be rewritten as

$$\begin{cases} 3X_T^3 & = R_0 + 2Y_T^2 \\ X_T^2 & = -\frac{R_2}{3x_q} Z_T^{-2} = \lambda_2 Z_T^{-2} \\ Y_T & = \frac{R_3}{2y_q} Z_T^{-3} = \lambda_3 Z_T^{-3}. \end{cases} \tag{4.24}$$

From this system, we can obtain the equation

$$R_0^2 \cdot Z_T^{12} + \left(4R_0\lambda_3^2 - 9\lambda_2^3\right) \cdot Z_T^6 + 4\lambda_3^4 = 0. \tag{4.25}$$

This equation on $\mathbb{F}_p$ can be solved, for example with Sage [S+12]. Once we know Z_T, we use it into the initial system to obtain X_T and Y_T. The points which do not lie on the curve are eliminated. Finally, the possibilities for $P = [i^{-1}]T$ are computed. We perform the Miller loop for all possibilities that lie on the curve to determine the correct solution.

Finding the secret knowing $h_2(Q)$ (Tate pairing)

In some cases (notably the *if* instruction skip in [BMH13]), the attacker can recover the value $h_2(Q)$. From this value, he can recover the secret. We know that

$$h_2(Q) = ((Y_T - Y_P Z_T^3)X_P - Y_P Z_R) + (Y_P Z_T^3 - Y_T)x_q \cdot w^2 + y_q Z_R \cdot w^3. \tag{4.26}$$

where $R = T + P$ and as a consequence $Z_R = Z_T(X_T - X_P Z_T^2)$. There are 5 unknowns in $\mathbb{F}_p$ and $h_2(Q)$ provides 5 equations on $\mathbb{F}_p$. If needed, the curve equations for T and P may be added. This system can finally be solved with a Gröbner basis [AL94].

A special case occurs when the known value $h_2(Q)$ is the line evaluation in the last iteration as shown in [BMH13]. Since $[r]P = 0_\infty$, $h_2(Q)$ has a simplified form in the last iteration

$$h_2(Q) = Z_P^2 x_Q - X_P, \tag{4.27}$$

where $P \in E(\mathbb{F}_p), Q \in E'(\mathbb{F}_{p^2})$. If Q is secret and P known, x_Q can be trivially recovered and two solutions are possible for y_Q (with the curve equation). If P is secret and Q is public with $x_Q = xq_1 \cdot u + xq_0$ known, and writing $h_2(Q) = R_1 \cdot u + R_0$, the system

$$\begin{cases} R_1 = & Z_P^2 xq_1 \\ R_0 = & Z_P^2 xq_0 - X_P \end{cases} \tag{4.28}$$

is found which easily gives Q.

Finding the secret knowing $h_1(Q) \cdot h_2(Q)$ (Tate pairing)

Another possible case is if the Miller loop iteration evaluates both $h_1(Q)$ and $h_2(Q)$, if the corresponding bit in r is equal to 1. The attacker can then recover

$$h_1(Q) \cdot h_2(Q). \tag{4.29}$$

We know that

$$h_1(Q) = \left(3X_T^3 - 2Y_T^2\right) - \left(3X_T^2 Z_T^2 x_q\right) \cdot w^2 + \left(2Y_T Z_T^3 y_q\right) \cdot w^3 \tag{4.30}$$

and

$$h_2(Q) = ((Y_D - Y_P Z_D^3)X_P - Y_P Z_R) + (Y_P Z_D^3 - Y_D)x_q \cdot w^2 + y_q Z_R \cdot w^3, \tag{4.31}$$

where $D = [2]T$, $R = D + P$ and as a consequence $Z_R = Z_D(X_D - X_P Z_D^2)$. From the equations for doubling a point,

$$\begin{cases} X_D &= 9X_T^4 - 8X_T Y_T^2 \\ Y_D &= 12Y_T^2 X_T^3 - 8Y_T^4 - 3X_T^2(9X_T^4 - 8X_T Y_T^2) \\ Z_D &= 2Y_T Z_T. \end{cases} \tag{4.32}$$

We have five unknown values $X_P, Y_P, X_T, Y_T, Z_T \in \mathbb{F}_p$ and the knowledge of the product $R = h_1(Q) \cdot h_2(Q)$ by the attacker provides the system

$$\begin{cases} P_i(X_P, Y_P, X_T, Y_T, Z_T) &= R_i, i \in \{0, 2, 3, 4, 5, 6, 8, 9, 10, 11\} \\ X_P^3 + 5 - Y_P^2 &= 0 \\ X_T^3 + 5 \cdot Z_T^6 - Y_T^2 &= 0, \end{cases} \tag{4.33}$$

with the $R_i \in \mathbb{F}_p$ obtained through the identification of the $\mathbb{F}_p$ terms over $\mathbb{F}_{p^{12}}$. The system can be solved by computing the Gröbner basis [AL94] or by the resultant method and directly gives the value of X_P and Y_P.

4.2 Practical fault attacks on the Miller algorithm

Using the EM bench presented in Section 3.1, we validated that the attacks, described theoretically until now, can be implemented in practice.

For that purpose, a first fault attack has been set-up experimentally where a fault was injected on the modular addition as described in Section 4.1 (Controlled-add). Since the exact same experiment has been done in order to invert the final exponentiation, with details in Chapter 5, we do not detail this experiment here.

The goal of the second fault attack is to exit the Miller loop at whatever iteration we want, with an EM fault injection. We focus on a faulty Miller algorithm where the last iteration is skipped. The targeted implementation is a twisted Ate pairing where the final exponentiation has been removed with parameters taken from [BGDM+10]. In our implementation, computations are done in the Montgomery domain, therefore the intermediate values must be multiplied by $1/Res$, the inverse of the Montgomery residue, to convert them back to the canonical domain.

$p =$ 0x2370FB049D410FBE4E761A9886E502417D023F40180000017E80600000000001,
$r =$ 0x2370FB049D410FBE4E761A9886E502411DC1AF70120000017E80600000000001,
$Res =$ 7E922DFB33891CBDAC545D44FBCF03594F0453F57FFFFF58A7D5FFFFFFFFF9,

the curve equation is $E : Y^2 = X^3 + 5$, $E' : Y^2 = X^3 + 5 \cdot Z^6$ (same equation but in jacobian coordinates), the public point is

$$P = (\texttt{0x38009F84045BBBC1BE5D7EBE2AE3CC1AD2DB2A342856477FD090951DFF430A1},\\ \texttt{0x7401C9670C5C62BC083614A6080C25025B9BBA7C49D46A9AEB7077CC58CA36E})$$

and finally Q is the unknown secret point that we want to recover. For verification purposes, the secret point is:

$$Q = (\texttt{0xA1CF585585A61C6E9880B1F2A5C539F7D906FFF238FA6341E1DE1A2E45C3F72} \cdot u +\\ \texttt{0x19B0BEA4AFE4C330DA93CC3533DA38A9F430B471C6F8A536E81962ED967909B5},\\ \texttt{0xEE97D6DE9902A27D00E952232A78700863BC9AA9BE960C32F5BF9FD0A32D345} \cdot u +\\ \texttt{0x17ABD366EBBD65333E49C711A80A0CF6D24ADF1B9B3990EEDCC91731384D2627}),$$

The size of r is 254 bits. First a correct execution is performed providing $f_{K,Q}(P)$, the correct result of the Miller algorithm.

$$\begin{aligned} f_{K,Q}(P) =& \texttt{0x1ED7E66141E83841515DDB0AD3D1236AF729D546877379983A2F738820D6BBF6} \cdot uv^2w \\ &+ \texttt{0x1DD740F718413FF626D3D01CB0E1D2DF144BD80E1DF936A032A493F2CDEAF9EB} \cdot v^2w \\ &+ \texttt{0x1D7FFC434B8997573F7D15A3327B6B400DBE23B368D3A83AA003817208308A89} \cdot uvw \\ &+ \texttt{0x12017F8ABD965D6EF6B6AE2EDAC74A6C097337362DF6251918FCDA8FED9A77BA} \cdot vw \\ &+ \texttt{0x11AEF08EFACCF3AC542618DC4C06C0A2DA29DC0FBD202CAABD37F08679840D} \cdot uw \\ &+ \texttt{0x22EC3BE513BD8CD2DFD8EABED0F1D263CF15534DF36352EC67F4DE8D16FE8A15} \cdot w \\ &+ \texttt{0x1AD2F1E2F3BBE5557653668643EDD3D4D9DF07E40F138858C13B8D9D8D16E332} \cdot uv^2 \\ &+ \texttt{0x3FE3B7F3CDDEE95CBB83675F1DDFC94404E04F8AE33AB5523CC62EFFB82C0B6} \cdot v^2 \\ &+ \texttt{0x1F03E9624FCC8289E594184E5BEFB56D9185372A7A5C3F3F24E98B9A3D61372E} \cdot uv \\ &+ \texttt{0x53EF8A26C5493348EE3482022E9177732389E9DAF5DDA117BA03F9DB6546FB5} \cdot v \\ &+ \texttt{0x16E00488E04BEE3C44A8FC0EF8AE1B7D111942AB5C873E898E46E82A82EE4162} \cdot u \\ &+ \texttt{0x860E5EA3B58E3F38FF8EE8FA779D195C1A3DBF9DA52BCC591D4B8FC7AD40921}. \end{aligned}$$

To facilitate the experiments the code has been modified. First a trigger is defined by software, we set a pin of the board to its high state at the desired moment for the fault injection. This trick is often used in the context of "white box" chip certifications. Yet for a proper black-box approach, the trigger is often raised thanks to pattern recognition techniques in the power consumption trace, or using the IO lines once a command has been sent. These latter techniques are quite complex and do not seem necessary in our experiments to prove the feasibility of fault attacks on pairings.

It is important to note that there is a jitter of around 80 ns (equivalent to 5 instructions) between the chip trigger and the EM pulse. This jitter, added to the complexity of the microcontroller pipeline, makes it difficult to link the time of the pulse injection (controlled with the DELAYparameter of the pulse generator) and the instruction impacted by the pulse.

Another addition to the code is the use of dummy counters (6 of them) which are placed in the code and are used as "snow-steps". We can easily infer where the fault injection occurred in the program by checking these counters. When removing the counters, the same results are observed but it becomes more difficult to find the correct parameters for the fault injection.

The last code modification is the "isolation" of the loop counter decrementation. Practically, it means that the loop is done with a *while* instead of a *for* and *NOP* instructions are added around the decrement instruction. There again this modification is not essential but allows a better reproducibility of the faults. The reproducibility is the probability to have a second identical fault when all parameters of the fault injection are fixed to the values producing the first fault.

The resulting code of the Miller loop is shown in Code 4.1.

Code 4.1: Miller algorithm for EM injection

```
i = pmngr->T_ate.bit_len - 2;
while( i > -1)
//for(i = pmngr->T_ate.bit_len - 2; i > -1; i--)
{
	ct1++;
	ct2 +=2;
	ct3 += 3;


	//f <- f^2*lTT(P); T <- 2T
	bnpair_dbl_leval(XT,YT,ZT,xp, yp, XT, YT, ZT, &l00, &l10, &l11);
	zzn12_sqr(f, f);  //f = f^2
	zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lTT(P)

	if(bnpair_get_sen_bit(&(pmngr->T_ate), i, TRUE))
	{
		bnpair_add_leval(XQ, YQ, XT, YT, ZT, xp, yp, XT, YT, ZT, &l00, &l10, &l11);
		zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lTQ(P)
	}
	else if(bnpair_get_sen_bit(&(pmngr->T_ate), i, FALSE))
	{
		bnpair_add_leval(XQ, &YQneg, XT, YT, ZT, xp, yp, XT, YT, ZT, &l00, &l10, &l11);
		zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lT(-Q)(P)
	}

	ct4 += 4;
	ct5 += 5;

	if(i == 1)
	{
		//trig up
		GPIOB->BSRR = GPIO_Pin_8;
```

```
            __NOP();
            __NOP();
            __NOP();
            __NOP();
            __NOP();
            __NOP();
            __NOP();
            __NOP();

            //trig down
            GPIOB->BRR = GPIO_Pin_8;
        }

        ct6 += 6;

        __NOP();
        __NOP();
        i--;
        __NOP();
        __NOP();
}
```

When injecting faults on this program at a vulnerable spot (cf. Section 3.2), with an amplitude AMP of $-140\,\mathrm{V}$ and a DELAY of $652.1\,\mathrm{ns}$, we are able to perform a loop skip. The resulting faulty value $f_{K,Q}(P)^*$ is

$$\begin{aligned} f_{K,Q}(P)^* =& \texttt{0x59C2611469286172DADDD0C75AA153892F88C99C37C077984D14DA9FEE41974} \cdot uv^2w \\ &+ \texttt{0x8470FA960068D635E751872ED5D023BA9D5299E928B8E268CF98253585BEEAC} \cdot v^2w \\ &+ \texttt{0x2665E826CD91E0600ED4D4FACFFAA04BFF2F840637C3DDA0E9779B979C83E7} \cdot uvw \\ &+ \texttt{0xB3BA94E4358629A81054738B7205E4813772FC2216C1F87CC93E7C3A822B2F3} \cdot vw \\ &+ \texttt{0xD73A86EE3F4CED82C65B11DC042E5F6DA7A17B20FB5DE2B235B4D2E220B0BA3} \cdot uw \\ &+ \texttt{0xD1DBA02BBD58DB8D43580B9A7A43DD01904B91209535DAFAD67226598F72BFE} \cdot w \\ &+ \texttt{0x1562D210D63067F742830F87D80F5AD0FBE081B796D72406A6E4514EBAC73996} \cdot uv^2 \\ &+ \texttt{0x1270571A206637E9130A2639AB72DD446115D1845DBEA68F4752E9BE79E8D91A} \cdot v^2 \\ &+ \texttt{0x7444FF71B099E4883991AF01C2DCD04A788B1CBC80CD6F314F3FDA805EEFD26} \cdot uv \\ &+ \texttt{0x74B8623EE4E3FBA7C4D239C370209EABF93A58776DF845C5121BC1C23553897} \cdot v \\ &+ \texttt{0xD7491706187F2BC1903D6C6D1504E805961645185C1098A956CA645177887F2} \cdot u \\ &+ \texttt{0x7A7DD4FF50D99B968DAD3A1F636205EFD6E89D270E74F3571B37BBC959D476E}. \end{aligned}$$

Using Sage [S+12], it now possible to compute $h_1(P) = \frac{f_{K,Q}(P)}{f_{K,Q}(P)^{*2}}$. Since Sage performs the multiplications in the canonical domain, the values $f_{K,Q}(P)$ and $f_{K,Q}(P)^*$ are multiplied by $1/Res$ for the domain conversion from Montgomery domain.

$$\begin{aligned}
h_1(P) =& \texttt{0} \cdot uv^2w \\
&+ \texttt{0} \cdot v^2w \\
&+ \texttt{0xA17D142DEECC8668A1C3BBFD12385544D2761AD4FCD0F85DCEF561A7F3297F4} \cdot uvw \\
&+ \texttt{0xAE2480DC8DA9E2154111EE78DF038649D73E44A92D9CED2229D99973D3D039A} \cdot vw \\
&+ \texttt{0} \cdot uw \\
&+ \texttt{0} \cdot w \\
&+ \texttt{0x13ED9D2A9F479B20BEE61C47CEFC680A6B7C1A72687FA1B279671A65D039359F} \cdot uv^2 \\
&+ \texttt{0x207B4F67D5556DF71847B3AA322216D07242A3EFE379FB9393EFDF4A2F9E6644} \cdot v^2 \\
&+ \texttt{0} \cdot uv \\
&+ \texttt{0} \cdot v \\
&+ \texttt{0x1A2EB33142FB757AFFC903D58FB8FD81FFEEDDFED3EEC780DACF371899A15F6E} \cdot u \\
&+ \texttt{0x229EB28B4DE041C0DEAC9E673D2E452E16C334B90E7836CF4FAB01365CDC4DA7}.
\end{aligned}$$

There is a small difference between our result and the one in Section 4.1.3 ($h_1(P)$). As an optimization used in [BGDM$^+$10], the value h_1 and h_2 actually computed are different from the ones in Section 4.1.3 by a factor $w^3/2$. This factor is removed thanks to the final exponentiation (true because $p \equiv 1 \mod 12$).

By identification of $h_1(P) * w^3/2$, we have

$$\begin{aligned}
R_0 =& \texttt{0x1A2EB33142FB757AFFC903D58FB8FD81FFEEDDFED3EEC780DACF371899A15F6E} \cdot u \\
&+ \texttt{0x229EB28B4DE041C0DEAC9E673D2E452E16C334B90E7836CF4FAB01365CDC4DA7},
\end{aligned}$$

$$\begin{aligned}
R_3 =& \texttt{0xA17D142DEECC8668A1C3BBFD12385544D2761AD4FCD0F85DCEF561A7F3297F4} \cdot u \\
&+ \texttt{0xAE2480DC8DA9E2154111EE78DF038649D73E44A92D9CED2229D99973D3D039A},
\end{aligned}$$

$$\begin{aligned}
R_4 =& \texttt{0x13ED9D2A9F479B20BEE61C47CEFC680A6B7C1A72687FA1B279671A65D039359F} \cdot u \\
&+ \texttt{0x207B4F67D5556DF71847B3AA322216D07242A3EFE379FB9393EFDF4A2F9E6644}.
\end{aligned}$$

By following the method detailed in Section 4.1.3, we find

$$\begin{aligned}
\lambda_2 =& \texttt{0x2366729ABF911E2F87FB619C79E234E262FC1CAA5A4352B6375EA4BB9D292615} \cdot u \\
&+ \texttt{0x20F907AF96EF828E954D9E5AC743BEFA920B82A67B52098331DE112FAAE5886C},
\end{aligned}$$

and

$$\begin{aligned}
\lambda_3 =& \texttt{0x1CCC714BBD7E86B364296680973B8A3E03D8E119F52ACA1B5608AC88BA9798CC} \cdot u \\
&+ \texttt{0xFC8E8997F935ED120F77A4C6EBA3D058EA891ABC2CAC24A385803C9ACB333F}.
\end{aligned}$$

The resolution is done with Sage in the canonical domain: first the variable ZT is created ($K \leftrightarrow \mathbb{F}_{p^2}$), then the equation is described and we use the *factor* function to find candidates for ZT.

```
V.<ZT> = PolynomialRing(K,'ZT')
eq = ZT**12 * R0**2/b + (4*l2**2*R0 / u - 9 * l3**3)*ZT**6 + 4*l2**4
eq.factor()
```

outputs

```
(22768538455231441734762147682206374539318810765353153731604238943777027601 02*u +
25391390370628065234940011363273403684188195654829971521301033156056343061 40) *
(ZT + 10428241657308962104082982504843566435697400903459790544896653265759467637196*u +
10427293109238011858454456392960279739069824933641038801462396529486461334143) *
(ZT + 10739127499087857558585590660374074644496836973490210456495351435436698599613*u +
10205260986791663445986430096307066540926230265019352769017117055050243524402) *
(ZT + 12020386895876364170186338617218813030114097013135739940433679312313565888620*u +
57263879830998559060033101535620703548505706712960018844778459882300255458 37) *
(ZT + 12073096958214511369068770740263386885194663516013084467094561377033689873793*u +
20354235334345124180233177678268548029262491641659082934529893142885047290 38) *
(ZT + 12812593967885066116007479568447385656172600396455294839383278768768672890 07*u +
11551696030711320737773568344753653329561178507460986615239121913296004268348) *
(ZT + 14749309637614621666155940330653910950019578061538808015840065103239354957906*u +
44788730036918075399831199427449961860756595937233508845392710668202179785 65) *
(ZT + 39574720761886169086879175472352626304421745851712530326838316030825323731 20*u +
13995145500968615859733370519671794712710588937018429206325403665827717517875) *
(ZT + 40101821385267641075703496702798364855227410880485975593447136678026565358293*u +
10304181051303272371753378133936579160786267429888335615300546991886196701076) *
(ZT + 52914415353152707191710976271245750511400011276941270432830415446795236473 00*u +
58253080476114648317702581911915829747106078361649847307612759250659787225 11) *
(ZT + 56023273770941661736737057826550830799394371977245469548817397143567546097 17*u +
56032759251651164193022318945383697765670131675432986983159964506297609127 70) *
(ZT + 64707695811203451953950649576083038052552263182885375122128216626769352640 76*u +
12462716642672524276477774160787134541996074097806947094915385843774966063181) *
(ZT + 95597994532827830823616233298903457103816117828957999875655713174392869828 37*u +
35678523917306040012789141267115149736407640033773904048630071363412561837 32)
```

We have 12 candidates and the penultimate one is the correct one. using a candidate value for ZT, we can compute XT and YT with the System (4.19).

$$\begin{aligned} Z_T =& \texttt{0x1522A79D50862EE4F5E15BBE4508099DDBC97BD69CDAC90BE87E0D66F843B8B5} \cdot u \\ &+ \texttt{0x7E355385C8C905DE59D62026275D206A0218C8665E9753604169A273A2E1FB4}, \end{aligned}$$

$$\begin{aligned} Y_T =& \texttt{0x1BA1077796333A1D4B1A4A0626D14159499BC332649EC507B9732F724DA8EF96} \cdot u \\ &+ \texttt{0x24F9081D80EC0249871E41DFC8641E5CEFB029B6C4400E78C02DF2B2BFB9D44}, \end{aligned}$$

and two candidates are possible for X_T, only one of them being the correct one:

$$\begin{aligned} X_T =& \texttt{0x1373060C1068C26ECFFCED0216C1C43571CC85CD7D6FEE883BFCBAE2DC82EE5B} \cdot u \\ &+ \texttt{0x12DFF3CEE2926615F66070D3A46153A9CD13BB20966E56B67E5924509B04C17E}. \end{aligned}$$

At the penultimate iteration, we know that $T = [3x^2]Q$ where x is the BN curve parameter.

$$3x^2 = \texttt{0X2FA047E8030000000000000000000000}$$

We find j such that $3x^2 \cdot j \equiv 1 \mod r$.

$$j = \texttt{0X11CA435310DCA0483B8859E203C000005FA01800000000006}$$

Finally we can compute $Q = [j]T$. The Miller algorithm has been reverted with a correct execution and a faulty one.

4.3 Countermeasures to protect the Miller algorithm

Since fault attacks on the Miller have already been studied, countermeasures have been proposed in the literature. In this section, we review the proposed countermeasures and analyse their efficiency. *This work is the result of a collaboration with Nadia El Mrabet and Marie Paindavoine, accepted in [EMFG+14].*

4.3.1 Countermeasures in the literature

Some countermeasures were proposed to circumvent the attacks described in Section 4.1. We present here those countermeasures [EMPV12] and we shall discuss about the efficiency of some of them in Section 4.3.2.

1. A naive countermeasure consists in computing the pairing twice. Namely, one can compute $e_1 = e(P, Q)$, $e_2 = e(P, Q)$ and check if $e_1 = e_2$. Moreover, the bilinearity of pairings allows the computation of $e_1 = e(P, Q)$ and $e_2 = e(\alpha P, \beta Q)$, with α, β being random integers and from there we could check if $e_2 = e_1^{\alpha\beta}$.

2. One can also check the intermediate results, this will prevent an attacker from injecting a fault in the intermediate computation. One can check if the intermediate points T are elements of the elliptic curve or if the value f_1 in the Miller algorithm is an element of $\mathbb{F}_{p^k}$.

3. In [GS11], the authors propose a countermeasure suited for their very particular attack model. The countermeasure consists in verifying if the Miller loop performed more than two iterations.

4. Robust codes for fault detection were analysed in [OGS07] for fields of characteristic 3. The necessity of a robust loop counter is also examined.

More popular and elegant countermeasures (since the impact on the performances is often lower) use randomization and blinding in the Miller algorithm.

5. **Coordinates randomization** is proposed in [KTH+06] where the inputs of the pairing computation are randomized. In Jacobian coordinates, the point $P = (X_P : Y_P : Z_P)$ is equivalent to the point $(\lambda^2 X_P : \lambda^3 Y_P : \lambda Z_P)$, for a non zero integer λ. The countermeasure consists in modifying the coordinates of P before the pairing computation. As a consequence, the values in the equations occurring during the Miller algorithm are different. The same argument holds for projective coordinates in general.

6. **Miller variable blinding**: Another countermeasure proposed in [Sco05] consists in randomizing the intermediate Miller function f. For every Miller iteration, the function f is multiplied by a random λ in a strict subfield of $\mathbb{F}_{p^{12}}$. This countermeasure does not influence the result of pairing, as the final exponentiation, which is present in the majority of pairings, maps all these elements onto 1.

7. **Additive blinding**: Using the bilinearity as proposed by Page *et al.* in [PV06], one can also randomize the input point using additive blinding. For a random point $M \in E(\mathbb{F}_{p^k})$, we have $e(P, Q) = \frac{e(P,Q+M)}{e(P,M)}$ or equivalently $e(P, Q) = e(P, Q + M) \cdot e(P, -M)$.

8. **Multiplicative blinding**: As described in [PV06], the multiplicative blinding uses the fact that the points P and Q verify $e(\alpha P, \beta Q) = e(P, Q)^{\alpha\beta}$. Choosing α, β such that $\alpha \cdot \beta = 1 \pmod r$, then $e(\alpha P, \beta Q) = e(P, Q)$.

9. Shirase *et al.* [STO08] proposed a countermeasure specific to fields of characteristic 3 and not applicable in our case where they add a randomization constant during the line evaluation.

Note that the first two blinding countermeasures (5 and 6) were initially proposed in the context of Side-Channel Analysis and not Fault Attacks. Nevertheless, they have been suggested to be efficient to thwart fault attacks too [EMPV12].

4.3.2 Evaluating the countermeasures

In this section, we show that some blinding countermeasures resented above and initially suggested against Side-Channel Analysis cannot actually be used as a protection against fault attacks.

Coordinates randomization

As proposed in [KTH+06], initially against side-channels analyses, instead of executing the Miller Loop with $Q = (X_Q : Y_Q : Z_Q) \in E'(\mathbb{F}_{p^2})$ (represented in jacobian coordinates), we execute it with

$$Q = (\lambda^2 X_Q : \lambda^3 Y_Q : \lambda Z_Q),$$

where $\lambda \in \mathbb{F}_{p^2}$ is a random blinding value. We note $h_1^{(\lambda)}, h_2^{(\lambda)}, f_{K,P}^{(\lambda)}$ the line evaluations and the output of the Miller algorithm when performed with this blinding. When we include the λ in the tangent equation, we find that it is possible to factor it. For the doubling step the equation becomes $h_1^{(\lambda)} = \lambda^{24i} h_1$, and for the addition step it becomes $h_2^{(\lambda)} = \lambda^{9i+12} h_2$, where i is an integer related to the number of iterations. The value of i can be found, but since this value has no influence on the result of the attack we remove it. Thus the result of the Miller loop is $f_{K,Q}^{(\lambda)} = \lambda^a \cdot f_{K,Q}$ for some integer a. In order to perform the fault attack, we need two different executions of the Miller algorithm. Thus we use two different blinding values, one for each execution. But actually, this adds only one unknown into the system:

$$\begin{aligned} h_1(P)^{(\lambda)} =& \frac{f_{K,Q}^{(\lambda_1)}(P)}{f_{K,Q}^{(\lambda_2)*}(P)^2} \\ =& \frac{\lambda_1^a \cdot f_{K,Q}(P)}{\lambda_2^b \cdot f_{K,Q}^*(P)^2} = \frac{\lambda_1^a}{\lambda_2^b} \cdot h_1(P). \end{aligned}$$

We call $L = \frac{\lambda_1^a}{\lambda_2^b}$ the new unknown and hence have:

$$\begin{aligned} h_1(P)^{(\lambda)} =& L \cdot ((3X_T^3 - 2Y_T^2) \cdot w^6 + 2Y_T Z_T^3 y_p \cdot w^3 \\ & - 3X_T^2 Z_T^2 x_p \cdot w^4), \\ h_1(P)^{(\lambda)} =& R_0^{(\lambda)} + R_3^{(\lambda)} w^3 + R_4^{(\lambda)} w^4. \end{aligned}$$

By identification of the decomposition in $\mathbb{F}_{p^2}$, we obtain the system

$$\begin{aligned} R_0^{(\lambda)} =& \quad L \cdot R_0 \\ R_3^{(\lambda)} =& \quad L \cdot R_3 \\ R_4^{(\lambda)} =& \quad L \cdot R_4. \end{aligned} \tag{4.34}$$

To this system, we add the equation derived from the fact that T lies on the curve E:

$$Y_T^2 = X_T^3 + 5Z_T^6. \tag{4.35}$$

The system (4.34) and the equation (4.35) can be solved for the Ate pairing. To be solved, the resulting system requires the computation of the Gröbner basis which provides an equation of degree 12 in Z_T only. An example of this attack is given in Appendix B.

Miller's variable blinding

In this countermeasure, initially again side-channels analyses, the value of the line evaluation is multiplied by a random element L of $\mathbb{F}_{p^2}$ for all iterations as suggested by Scott [Sco05]. Thus, we have:

$$h_1^{(\lambda)} = L \cdot h_1. \tag{4.36}$$

As can be seen immediately, this countermeasure can be bypassed in exactly the same way as the previous one with the system (4.34) and equation (4.35). Since the two previous countermeasures do not work against fault attacks, it is probable that they are unable to thwart side-channel analyses too, these countermeasures simply do not introduce enough unknowns into the system.

Additive blinding

This countermeasure seems to be efficient as long as the mask is truly random at each pairing execution. Yet the mask has to be properly used. For example, let P be the secret point during the pairing computation $e(P, Q)$ (Q is public). In order to protect this computation with an additive blinding, one has to compute $e(P, Q) = e(P, Q + M)e(P, -M)$, where M is a secret random mask. If the computation $e(P, Q) = e(P + M, Q)e(-M, Q)$ is performed instead, an attacker can find the secret point with the following scheme.

1. Fault attack on $e(-M, Q)$ in order to recover $-M$.
2. Fault attack on $e(P + M, Q)$ in order to recover $P + M$.
3. $P = P + M - M$.

Similarly if P is public and Q is the secret point, the additive blinding must be $e(P, Q) = e(P + M, Q)e(-M, Q)$. The computation of a pairing with an additive blinding can benefit from an optimization where the two pairings are done in a shared loop. If the attacker is able to replay the mask, the scheme becomes unsafe. The attacker can use the following methods in this case, presented for a Tate pairing.

Additive blinding with calculations in separate Miller Loops For this scheme, the device computes two pairings separately: $e(P, Q + M)$ and $e(P, M)$. Let M be a point that lies on the twisted curve and Q be the public point. Suppose we perform the fault attack against the first pairing $e(P, Q + M)$ at an iteration with only a doubling step. The Miller accumulator point T ($T = [i]P$) has coordinates $(X_T : Y_T : Z_T)$. As in Section 4.1.3, we thus obtain through identification of the decomposition in the basis of $h_1(Q)$ in $\mathbb{F}_{p^{12}}$ over $\mathbb{F}_{p^2}$:

$$R_0 = 3X_T^3 - 2Y_T^2, \quad R_2 = -3X_T^2 Z_T^2 X_{Q+M}, \quad R_3 = 2Y_T Z_T^3 Y_{Q+M}. \tag{4.37}$$

For the second pairing evaluation $e(P, M)$, we obtain through identification at a only doubling step (the Miller's accumulator point is now S, $S = [j]P$, and has coordinates $(X_S : Y_S : Z_S)$):

$$R'_0 = 3X_S^3 - 2Y_S^2, \quad R'_2 = -3X_S^2 Z_S^2 X_M, \quad R'_3 = 2Y_S Z_S^3 Y_M. \tag{4.38}$$

As T and S lie on the curve E we have

$$Y_T^2 = X_T^3 + 5Z_T \quad \text{and} \quad Y_S^2 = X_S^3 + 5Z_S. \tag{4.39}$$

And as M and $Q + M$ lie on the twisted curve E', we have

$$Y_M^2 = X_M^3 - u \quad \text{and} \quad Y_{Q+M}^2 = X_{Q+M}^3 - u. \tag{4.40}$$

Using identification in the basis of $\mathbb{F}_{p^2}$ over $\mathbb{F}_p$, we can rewrite all these systems as polynomials with coefficients in $\mathbb{F}_p$ in order to use the `groebner_basis()` Sage method. Once we have the Gröbner basis for lexicographic order ($\cdots < X_T < Y_T < Z_T$) of the ideal generated by systems (4.37), (4.38), (4.39) and (4.40) we can solve the system since the last polynomial is in Z_T only, the first before last in Y_T and the second before last in X_T.

Additive blinding with calculations in the same Miller Loop An optimization for the computation of the product of two pairings suggested in [Sco05] consists in using only one Miller loop, both pairings sharing the same Miller variable.

Algorithm 5: Miller algorithm for the computation of a product of Tate pairings

Data: $r = (r_n \dots r_0)_2$, $P \in \mathbb{G}_1(\subset E(\mathbb{F}_p))$ and $M, Q + M \in \mathbb{G}_2(\subset E(\mathbb{F}_{p^k}))$;
Result: $f_{(r,P)}(M) \cdot f_{(r,P)}(Q + M) \in \mathbb{G}_3(\subset \mathbb{F}_{p^k}^*)$;
$T \leftarrow P$;
$f_1 \leftarrow 1$;
for $i = n - 1$ **to** 0 **do**
  $f_1 \longleftarrow {f_1}^2 \times h_1(M) \times h_1(Q + M)$, $h_1(x)$ is the equation of the tangent at the point T;
  $T \leftarrow [2]T$;
  if $r_i = 1$ **then**
    $f_1 \longleftarrow f_1 \times h_2(M) \times h_2(Q + M)$, $h_2(x)$ is the equation of the line (PT);
    $T \leftarrow T + P$;
  end
end
return f_1

We suppose that we stop the process at an iteration with only the doubling step. The ratio of the faulty value with the correct one is equal to the product of the two tangent line evaluations $R = h_1(M) \cdot h_1(Q + M)$. We have the following system:

$$\begin{cases} R_0 &= (4Y_T^2 Z_T^6 Y_{Q+M} Y_M)u + 4Y_T^4 + 9X_T^6 - 12X_T^3 Y_T^2 \\ R_2 &= (6X_T^2 Y_T^2 Z_T^2 - 9X_T^5 Z_T^2)(X_M + X_{Q+M}) \\ R_3 &= (6X_T^3 Y_T Z_T^3 - 4Y_T^3 Z_T^3)(Y_{Q+M} + Y_M) \\ R_4 &= 9X_T^4 Z_T^4 (X_{Q+M} X_M) \\ R_5 &= -6X_T^2 Y_T Z_T^5 (X_M Y_{Q+M} + X_{Q+M} Y_M). \end{cases} \tag{4.41}$$

As previously, equations (4.39) and (4.40) hold.

We were unable to solve this system with Sage, but Magma [BCP97] was successful using the Gröbner basis since the last polynomial was in Z_T only, the first before last in Z_T and Y_T and the second before last in Z_T and X_T.

Multiplicative blinding

Since the secret point P is blinded in the computation of $e([\alpha]P, Q)^{\beta}$, the attacker is able to recover only the point $[\alpha]P$, which does not reveal P thanks to the ECDLP. As a consequence, the security of the computation relies on the ability of the device to keep secret the values α and β but not on the pairing computation itself.

Loop protection

In order to counteract most fault attacks, a simple loop protection can be enough. Robust counters, additional counters and a verification at the end of the computation that the correct number of iterations has been performed are possible "cheap" solutions. A similar protection against the *if* instruction skip fault model can be to count the number of h_2 computations and verify that it is equal to the Hamming weight of the Miller index (r in Algorithm 3) minus one.

4.4 Conclusion

In this chapter, we have reviewed the fault attacks on the Miller algorithm proposed in the literature. We have shown that they have a common structure, *i.e.* they first try to obtain a line evaluation h_1 or h_2 and then in a second time deduce the secret point. They mainly differ by their fault model. Even if all the fault attacks have been proposed for a particular pairing with a particular field, we have seen (as already noticed by [EM09]) that they can, in fact, be generalized with other settings. We have validated that the proposed fault attacks are achievable in practice using EM fault injections against a software implementation of a twisted Ate pairing on a Cortex-M3 ARM microcontroller. To our best knowledge, this is the first practical attack against a Miller algorithm. Finally we have reviewed the countermeasures aiming at preventing the FAs and we have shown that two of them (the coordinates randomization and the Miller variable blinding), based on blinding techniques, should not be used on their own, depending on the attack scenario.

Chapter 5

Fault attacks on the Final Exponentiation

Where the security of the Final Exponentiation with respect to FAs is analysed.

Contents

Now that we have studied fault attacks on the Miller Algorithm (MA), demonstrated that they are feasible and studied the corresponding countermeasures, we present fault attacks aiming at inverting the Final Exponentiation algorithm when computing a pairing. A difference between the fault attacks presented in Section 5.1 and in Section 5.2 is that the first one uses a mathematical point of view whereas the second one uses a "system" point of view.

5.1 A fault attack to reverse the final exponentiation in 3 separate faults [LFG13]

As mentioned in [WS07], the FE in Tate-like pairings is a complex calculation. We show how precisely chosen faults can help in finding the critical intermediate values to finally reverse the entire exponentiation. To simplify the fault exploitation, we consider that the fault value is comprised between 0 and $2^l - 1$.

Our work is based on the algorithms proposed by Scott *et al.* in [SBC+09]. It focuses on FE in fields with an even embedding degree. We shall write $d = k/2$. The optimisation technique described in [SBC+09], still widely used in pairing implementations, is based on the decomposition of the FE into three stages. As $\frac{p^k-1}{r}$ can be rewritten as $\frac{p^k-1}{r} = \left(p^d - 1\right) \cdot \frac{p^d+1}{\Phi_k(p)} \cdot \frac{\Phi_k(p)}{r}$ where $\Phi_k(p)$ is the k-th cyclotomic polynomial (cf Definition 2.7.2). The FE can be performed as a succession of three exponentiations. Two are "easy" (with $p^d - 1$ and $\frac{p^d+1}{\Phi_k(p)}$) since they rely on exponentiations to the power p^n for some n and can hence be computed with the help of the Frobenius endomorphism (cf Definition 2.7.1) which has a low computational cost. The last step is the so-called "hard exponentiation" (because it cannot rely on the use of the Frobenius) and is the exponentiation to the power $\frac{\Phi_k(p)}{r}$. For example, with $k = 12$, we have

$$\frac{p^{12} - 1}{r} = \left(p^6 - 1\right) \cdot \left(p^2 + 1\right) \cdot \frac{p^4 - p^2 + 1}{r}. \tag{5.1}$$

Let f be a random value in $\mathbb{F}^*_{p^k}$, symbolizing the result of a Miller Loop. We name these intermediate results of each exponentiation

$$f_1 = f^{p^d-1} \; ; \; f_2 = f_1^{\frac{p^d+1}{\Phi_k(p)}} \text{ and } f_3 = f_2^{\frac{\Phi_k(p)}{r}}. \tag{5.2}$$

The attacker knows the result f_3 and wants to recover f. Note that f_1, f_2 and f_3 belong to different subgroups of $\mathbb{F}^*_{p^k}$. Since $f \in \mathbb{F}^*_{p^k}$, the following equations hold

$$f^{p^k-1} = 1 \; ; \; f_1^{p^d+1} = 1 \; ; \; f_2^{\Phi_k(p)} = 1 \text{ and } f_3^r = 1. \tag{5.3}$$

Thus $f_1 \in \mu_{p^d+1}$, $f_2 \in \mu_{\Phi_k(p)}$ and $f_3 \in \mu_r$. These subgroups have sizes $p^d + 1$, $\Phi_k(p)$ and r respectively. As an example for $k = 12$, f_1 contains ≈ 1536 bits of entropy, f_2 contains ≈ 1024 bits of entropy and f_3 contains ≈ 256 bits of entropy.

5.1.1 Recovering f_1

In this section we shall show how a fault on the intermediate value f_1 can help to retrieve its value.

Extracting a candidate

Lemma 5.1.1 *Let $\mathbb{F}_{p^k} = \mathbb{F}_{p^d}[w]/(w^2 - v)$ be the construction rule for the $\mathbb{F}_{p^k}$ extension field. v is a quadratic nonresidue in $\mathbb{F}_{p^d}$ and is a public parameter. Let $x \in \mathbb{F}_{p^k}$ be such that $x = g + h \cdot w$ with $g, h \in \mathbb{F}_{p^d}$. Then $x^{p^d+1} = g^2 - v \cdot h^2 \in \mathbb{F}_{p^d}$.*

Proof We have $x^{p^d} = g - h \cdot w$ since $x^{p^d} = (g + h \cdot w)^{p^d} = g^{p^d} + h^{p^d} \cdot w^{p^d} = g + h \cdot (-w)$. As a result $x^{p^d+1} = x^{p^d} \cdot x = (g - h \cdot w) \cdot (g + h \cdot w) = g^2 - w^2 \cdot h^2 = g^2 - v \cdot h^2$ since $w^2 = v$. ∎

Let $f_1 = g_1 + h_1 \cdot w$ with $g_1, h_1 \in \mathbb{F}_{p^d}$. We have

$$f_1^{p^d+1} = f_3^r = 1. \tag{5.4}$$

Thus by Lemma 5.1.1

$$g_1^2 - v \cdot h_1^2 = 1. \tag{5.5}$$

But Equation (5.4) holds only because $f_1 \in \mu_{p^d+1}$. Let $e \in \mathbb{F}_{p^d}$ be a fault injected on f_1 (say during the multiplication producing f_1 or during the loading of f_1 for the second "easy" exponentiation, cf. Figure 5.1) such that the faulty value f_1^* equals

$$f_1^* = f_1 + e \notin \mu_{p^d+1}. \tag{5.6}$$

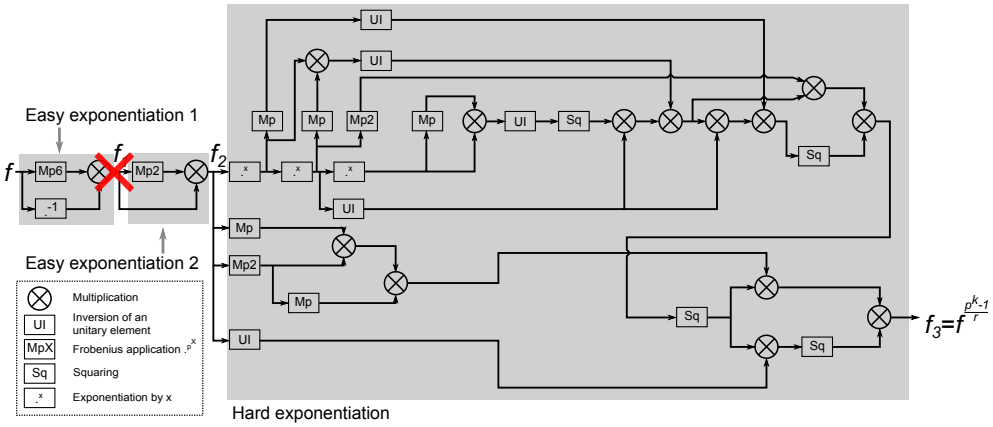

Figure 5.1: First fault location in the FE.

We consider that the fault e occurs only on the g_1 component[1] (which is compatible with our fault model if $2^l < p^6$), *i.e.*

$$f_1^* = (g_1 + e) + h_1 \cdot w. \tag{5.7}$$

$(f_1^*)^{p^d+1}$ can be computed by the attacker using the measured faulty result f_3^* since r is public knowledge

$$(f_1^*)^{p^d+1} = (f_3^*)^r \in \mathbb{F}_{p^d}. \tag{5.8}$$

Using Lemma 5.1.1, Equation (5.5) and Equation (5.7) we have

$$\begin{aligned}(f_1^*)^{p^d+1} &= (g_1 + e)^2 - v \cdot h_1^2 \\ &= g_1^2 - v \cdot h_1^2 + 2 \cdot e \cdot g_1 + e^2 \\ &= 1 + 2 \cdot e \cdot g_1 + e^2.\end{aligned}$$

Finally, g_1 can be written as:

[1] If on h_1, the same argumentation can be done.

$$g_1 = \frac{(f_1^*)^{p^d+1} - 1 - e^2}{2 \cdot e}. \tag{5.9}$$

Two possible values for h_1 can hence be calculated using Equation (5.5):

$$h_1^+ = \sqrt{\frac{g_1^2 - 1}{v}} \; ; \; h_1^- = -\sqrt{\frac{g_1^2 - 1}{v}}. \tag{5.10}$$

Verifying the candidates

The two candidates $f_1^+ = g_1 + h_1^+ \cdot w$ and $f_1^- = g_1 + h_1^- \cdot w$ can thus be verified by checking if $(f_1^+)^{\frac{p^d+1}{r}} = f_3$ or $(f_1^-)^{\frac{p^d+1}{r}} = f_3$. If the value of e is unknown, the attacker must guess the injected fault. For each guess, two candidates are computed and checked. A candidate is equal to the correct f_1 only when the correct e is guessed.

In our fault model, $0 < e < 2^l$ thus $2^l - 1$ attempts have to be made to find f_1 with 100% certainty. At this stage one may wonder what is the chance that the attacker finds a valid f_1 candidate (and an error value) which fits all his observations but is not equal to f_1 (*i.e.* a false positive). The f_1 candidate is noted f_{1c} and the corresponding guessed error is e_c.

$$f_{1c}^{p^d+1} = 1 \tag{5.11}$$

$$(f_{1c} + e_c)^{p^d+1} = (f_3^*)^r. \tag{5.12}$$

But the attacker observes $f_3 = f_1^{\frac{p^d+1}{r}}$ and $f_3^* = (f_1 + e)^{\frac{p^d+1}{r}}$. The question is what is the probability that $f_{1c} \neq f_1$ but that

$$f_3 = f_{1c}^{\frac{p^d+1}{r}} \tag{5.13}$$

$$f_3^* = (f_{1c} + e_c)^{\frac{p^d+1}{r}}. \tag{5.14}$$

Using Equation (5.11), the probability that Equation (5.14) is verified can be inferred as being equal to $1/r$ for a random f_{1c} in μ_{p^d+1}. Indeed we already know that $f_{1c}^{\frac{p^d+1}{r}}$ is in μ_r and $1/r$ is the probability that one random element in μ_{p^d+1} maps to a fixed value f_3 in μ_r. Similarly, from Equation (5.12), we can deduce that the probability for Equation (5.13) to be verified is equal to $1/r$ for a random f_{1c} in $\mathbb{F}_{p^k}^*$ since $(f_3^*)^r = (f_{1c} + e_c)^{p^d+1} \in \mu_{p^d-1}$. Thus $f_3^* \in \mu_{r\cdot(p^d-1)}$ and $(f_3^*)^r$ has r preimages in $\mu_{r\cdot(p^d-1)}$. As a consequence, the probability that we obtain the correct preimage is $1/r$.

We can combine these two probabilities and evaluate the probability of having an incorrect candidate for f_1 that matches the attacker's observations. The probability that a random candidate satisfying Equation (5.11) and Equation (5.12) also satisfies Equation (5.13) and Equation (5.14), corresponding to the observations of the attacker, is equal to $1/r^2$. In the case $k = 12$, typically $r \approx 2^{256}$, the probability of finding a valid candidate which is not equal to f_1 is $1/2^{512}$.

Hence we have shown how a fault injected on f_1 can be used to recover the latter's value, with a high probability, using the correct output f_3 and the faulty one f_3^* of the FE.

5.1.2 Recovering f

Knowing the value of f_1, we shall now see how to recover f.

Extracting a candidate

The strategy consists in using similar equations to the ones used previously and to include the new information about f_1 obtained by the attacker.

Lemma 5.1.2 *Let $f = g + h \cdot w$, $f^{-1} = g' + h' \cdot w$ and $f_1 = g_1 + h_1 \cdot w$.*
Then $\frac{g_1-1}{v \cdot h_1} = \frac{h'}{g'} = -\frac{h}{g} \Leftrightarrow f_1 = f^{p^d-1}$.

1: $f_1 = f^{p^d-1} \Rightarrow \frac{g_1-1}{v \cdot h_1} = \frac{h'}{g'} = -\frac{h}{g}$.

Proof

$$f_1 = \bar{f} \cdot f^{-1} = (f - 2 \cdot h \cdot w) \cdot f^{-1} = f \cdot f^{-1} - 2 \cdot h \cdot w \cdot f^{-1} = 1 - 2 \cdot h \cdot w \cdot (g' + h' \cdot w.)$$

Thus

$$g_1 = 1 - 2 \cdot h \cdot h' \cdot w^2 = 1 - 2 \cdot h \cdot h' \cdot v$$
$$h_1 = -2 \cdot h \cdot g'.$$

Finally

$$\frac{g_1 - 1}{v \cdot h_1} = \frac{-2 \cdot h \cdot h' \cdot v}{-2 \cdot h \cdot v \cdot g'} = \frac{h'}{g'}.$$

Moreover

$$g' = \frac{g}{g^2 - v \cdot h^2}$$
$$h' = \frac{-h}{g^2 - v \cdot h^2}.$$

So

$$\frac{g_1 - 1}{v \cdot h_1} = -\frac{h}{g}.$$

∎

2: $\frac{g_1-1}{v \cdot h_1} = \frac{h'}{g'} = -\frac{h}{g} \Rightarrow f_1 = f^{p^d-1}$.

Proof We write

$$\bar{f} \cdot f^{-1} = (g - h \cdot w) \cdot (g' + h' \cdot w) \tag{5.15}$$
$$= g \cdot g' - v \cdot h \cdot h' + (g \cdot h' + h \cdot g') \cdot w. \tag{5.16}$$

with

$$g' = \frac{g}{g^2 - v \cdot h^2} = \frac{1}{g\,(1 - v \cdot K^2)} \text{ and } h' = \frac{-h}{g^2 - v \cdot h^2} = \frac{1}{h\,(v - 1/K^2)}. \tag{5.17}$$

As a consequence:

$$g \cdot g' - v \cdot h \cdot h' = \frac{1 + v \cdot K^2}{1 - v \cdot K^2} = \frac{v \cdot h_1^2 + g_1^2 - 2 \cdot g_1}{v \cdot h_1^2 - g_1^2 + 2 \cdot g_1 - 1} = \frac{2 \cdot g_1 \cdot (g_1 - 1)}{2 \cdot (g_1 - 1)} = g_1.$$

And

$$g \cdot h' + h \cdot g' = \frac{K}{1 - v \cdot K^2} - \frac{1}{K \cdot (v - 1/K)} = \frac{2 \cdot K}{1 - v \cdot K^2}$$
$$= \frac{2 \cdot (g_1 - 1) \cdot h_1}{v \cdot h_1^2 - g_1^2 + 2 \cdot g_1 - 1} = \frac{2 \cdot (g_1 - 1) \cdot h_1}{2 \cdot (g_1 - 1)} = h_1.$$

∎

In the following, let K be the known value (known because we know g_1 and h_1 from f_1 found previously) $K = \frac{g_1 - 1}{v \cdot h_1} = -\frac{h}{g}$.
As a consequence, the knowledge of f_1 allows to find random preimages by taking a random $g \in \mathbb{F}_{p^d}$ and choosing $h = -K \cdot g$.

To recover f, the attacker creates a new fault $e_2 \in \mathbb{F}_{p^d}$ during the inversion in the first easy exponentiation (cf. Figure 5.2).

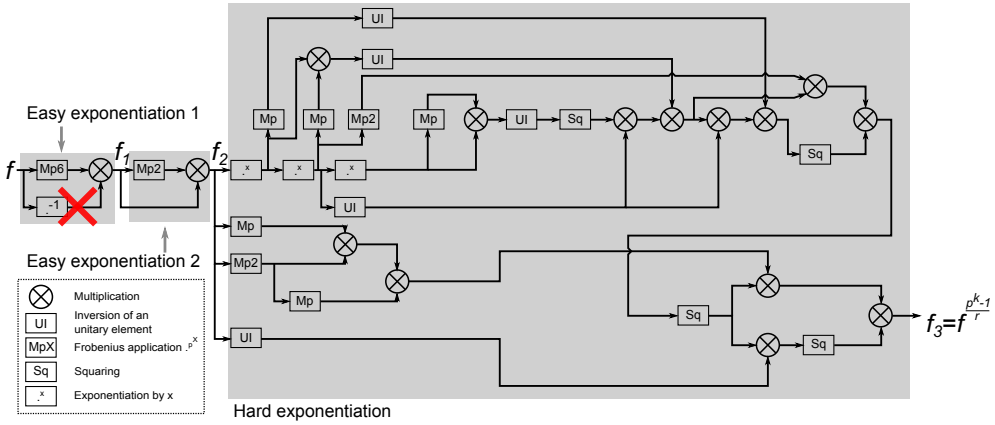

Figure 5.2: Second fault location in the FE.

Then

$$f_1 = f^{p^d - 1} = \bar{f} \cdot f^{-1} \text{ and } f_1^* = \bar{f} \cdot (f^{-1} + e_2).$$

Let Δ_{f_1} be the difference: $\Delta_{f_1} = f_1^* - f_1 = \bar{f} \cdot e_2$. Since $e_2 \in \mathbb{F}_{p^d}$, we can write $\Delta_{f_1} = \Delta_{g_1} + \Delta_{h_1} \cdot w$ with

$$\Delta_{g_1} = e_2 \cdot g \text{ and } \Delta_{h_1} = -e_2 \cdot h.$$

As f_1^* is not in μ_{p^d+1} with a high probability equal to $(1 - \frac{1}{p^d - 1})$, the attacker can compute $(f_1^*)^{p^d+1} = (f_3^*)^r \in \mathbb{F}_{p^d}$.

In this case

$$(f_1^*)^{p^d+1} = (g_1 + \Delta_{g_1})^2 - v \cdot (h_1 + \Delta_{h_1})^2$$
$$= (g_1 + e_2 \cdot g)^2 - v \cdot (h_1 - e_2 \cdot h)^2.$$

which gives the quadratic equation (using the relation $h = -g \cdot K$)

$$g^2 \cdot e_2^2 \cdot (1 - v \cdot K^2) + g \cdot 2 \cdot e_2 \cdot (g_1 - v \cdot K \cdot h_1) + 1 - (f_1^*)^{p^d+1} = 0. \tag{5.18}$$

We then solve this equation to obtain two solutions for g:

$$g^+ = \frac{v \cdot K \cdot h_1 - g_1 + \sqrt{(g_1 - v \cdot K \cdot h_1)^2 - (1 - v \cdot K^2) \cdot \left(1 - (f_1^*)^{p^d+1}\right)}}{e_2 \cdot (1 - v \cdot K^2)}$$
$$g^- = \frac{v \cdot K \cdot h_1 - g_1 - \sqrt{(g_1 - v \cdot K \cdot h_1)^2 - (1 - v \cdot K^2) \cdot \left(1 - (f_1^*)^{p^d+1}\right)}}{e_2 \cdot (1 - v \cdot K^2)}.$$

h can be computed with g and K: $h = -g \cdot K$. Thus we have two potential candidates for f.

Verifying the candidates

Even if e_2 is unknown, this procedure gives two candidates by guessing e_2. Now, whether this guess is correct or wrong, every potential candidate f_c has the following property: $f_c^{p^d-1} = f_1$ and therefore $f_c^{\frac{p^k-1}{r}} = f_3$. The attacker has thus found several valid preimages of f_3 and has to decide which is the correct one.

By checking whether $(\bar{f}_c \cdot (f_c^{-1} + e_2))^{\frac{p^d+1}{r}}$ is equal to the faulty result f_3^* allows to eliminate one of the two candidates for this guess of e_2. We finally obtain one candidate for each guessed e_2 and this candidate satisfies all observations made by the attacker. Finally we obtain a set of candidates of the same size as the set of possible error values.

The attacker has then to generate a third fault e_3, different from e_2, at the same location as the last one and intersect the two sets of candidates to find the correct one. Unfortunately, this intersection does not necessarily contain only one element. We can evaluate the size of this intersection set.

First we can neglect the probability that a random element of $\mathbb{F}_{p^k}^*$ maps to f_1 (the probability is $1/(p^d+1)$). Equation (5.18) outputs one f candidate f_{c1} by guessing $e_2 = 1$. Then the set of candidates is $\{f_{c1}, f_{c2}, \ldots, f_{c(2^l-1)}\}$ with f_{ci} corresponding to the guess $e_2 = i$. If we replace the product $g \cdot e_2$ by $\frac{g}{i} \cdot (i \cdot e_2)$ in Equation (5.18), we can see that the previous set can be rewritten as $\{f_{c1}, \frac{f_{c1}}{2}, \ldots, \frac{f_{c1}}{2^l-1}\}$.

Similarly with e_3, Equation (5.18) outputs one f candidate f'_{c1} by guessing $e_3 = 1$ and then $f'_{ci} = f'_{c1}/i$. The second set of candidates is $\{f'_{c1}, \frac{f'_{c1}}{2}, \ldots, \frac{f'_{c1}}{2^l-1}\}$.

Let e_{2t} and e_{3t} be the two faults truly injected. Since the correct value f is in the two sets of candidates, first equal to f_{c1}/e_{2t} then equal to f'_{c1}/e_{3t}, we have

$$f = \frac{f'_{c1}}{e_{3t}} = \frac{f_{c1}}{e_{2t}}. \tag{5.19}$$

Writing $a = \frac{e_{2t}}{e_{3t}}$, Equation (5.19) can be transformed into $f'_{c1} = f_{c1}/a$. The second set of candidates can be rewritten as $\{\frac{f_{c1}}{a}, \frac{f_{c1}}{2a}, \ldots, \frac{f_{c1}}{(2^l-1)a}\}$. Thus a same candidate is in the two sets each time the equation

$$a \cdot i = j \tag{5.20}$$

is satisfied with $i, j \in [\![1, 2^l - 1]\!]$. In our fault model, we can take e_{2t} and e_{3t} as elements in $\mathbb{N}$ and the number of solutions to this equation becomes $\left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor$.

Proof Let e_{2t} and e_{3t} be in our fault model: $0 < e_{2t}, e_{3t} < 2^l - 1$ and $p >> 2^l$. Let $a = \frac{e_{2t}}{e_{3t}} \in \mathbb{F}_p$, we want to find the number of pairs (i, j) solutions to Equation 5.20: $a \cdot i = j$ with $i, j \in [\![1, 2^l - 1]\!]$.

We can write

$$\frac{e_{2t}}{e_{3t}} = \frac{j}{i}.$$

This fraction can be rewritten as $\frac{u}{v}$, reducing it to lowest terms:

$$u = \frac{e_{2t}}{gcd(e_{2t}, e_{3t})}$$
$$v = \frac{e_{3t}}{gcd(e_{2t}, e_{3t})}.$$

All pairs solutions to Equation (5.20) can be written as $(k \cdot u, k \cdot v)$, $k \in \mathbb{N}^+$. The conditions $i, j \in [\![1, 2^l - 1]\!]$ are equivalent to $k \leq \frac{2^l-1}{u}$ and $k \leq \frac{2^l-1}{v}$ which combined give $k \leq \frac{2^l-1}{max(u,v)}$.

From the definition of u and v, we have: $max(u, v) = \frac{max(e_{2t}, e_{3t})}{gcd(e_{2t}, e_{3t})}$.

Finally, we have a solution for each integer k in the range

$$[\![1, (2^l - 1) \cdot \frac{gcd(e_{2t}, e_{3t})}{max(e_{2t}, e_{3t})}]\!]$$

The upper bound gives us the number of possible solutions to our Equation (5.20).

Finally the size of the intersection, which also contains the correct candidate, is

$$\#intersection = \left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor, \tag{5.21}$$

and the number of wrong candidates is

$$\#intersection - 1 = \left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor - 1. \tag{5.22}$$

The intersection of the sets of candidates obtained with e_2 and with e_3 contains at least one element if we get the two guesses correct once.

The computational cost of recovering f is low since the attacker has to use the procedure to recover a candidate through Equation (5.18) only once per fault injected with guesses $e_2 = 1$ and $e_3 = 1$.

Then she stores the corresponding candidates and computes the ratio $a = f_{c1}/f'_{c1}$. Finally she solves Equation (5.20), trying all $i \in [\![1, 2^l - 1]\!]$ and checking that $a \cdot i \in [\![1, 2^l - 1]\!]$, which provides e_{2t} and e_{3t} (the only solutions if there is no wrong candidate). With e_{2t}, he computes $f = f_{c1}/e_{2t}$. The memory used in the recovery of f is just one element of $\mathbb{F}_{p^k}$ per fault actually injected.

We cannot avoid the occurrence of wrong candidates. In order to conclude our attack we must have a unique candidate which satisfies all our observations.

If more than one candidate is contained in the intersection of the two sets then other faults must be generated at the same location until one candidate only matches all the observations of the attacker.

5.1.3 Summary of our fault attack on the Tate pairing's FE

At least four executions of the **same** pairing on the computing device are required to perform our attack.

1. The computation is executed normally. The attacker stores f_3 the correct result of the exponentiation.
2. A first fault is created on f_1 according to Section 5.1.1. The attacker memorizes f_3^*, a first faulty result. f_1 is found using Equation (5.9) and Equation (5.5).
3. A second fault e_2 is created during the inversion in the first easy exponentiation according to Section 5.1.2. The attacker stores f_3^*, the faulty result and extracts a candidate f_{c1} for f guessing $e_2 = 1$ with Equation (5.18) and Lemma 5.1.2.
4. Similarly to the previous step, a third fault $e_3 \neq e_2$ is created. With the faulty result f_3^*, the attacker extracts a new candidate f'_{c1} for f guessing $e_3 = 1$. The value $a = f_{c1}/f'_{c1}$ is then computed. A pair (i, j) solution to the equation $a * i = j$ with $i, j \in [\![1, 2^l - 1]\!]$ allows him to compute $f = f_{c1}/j$.

If several pairs (i, j) are found, more faults may be needed to ensure the uniqueness of the candidate for f. The important feature of this scheme is that only one fault per execution is needed to recover f, no double or triple faults.

5.1.4 Simulation of our attack

This attack scheme has been experimentally checked with Sagemath [S+12] in $\mathbb{F}_{p^{12}}$ with parameters identical to [BGDM+10]. Our fault model was the injection of a random e with $0 < e < 2^l$.

For a random $f \in \mathbb{F}_{p^k}^*$, we simulated 1000 fault injections for "f_1 recovery" with a random fault $e \in [\![1, 2^{10} - 1]\!]$ and we made $2^{10} - 1$ guesses on the fault value per injection. As a result, f_1 was correctly found for every fault injection and no wrong candidate was observed.

Similarly, we simulated "f recovery" knowing f_1. Two different errors in $[\![1, 2^l - 1]\!]$ were injected for 100 fault injections, first for $l = 7$ and then for $l = 10$. The number of wrong candidates reached, in average, 4.87 for $l = 7$ and 5.66 for $l = 10$. These examples show that even when we "loosen" the constraints on the possible errors (from 2^7 to 2^{10}) the number of wrong candidates, on average, does not increase dramatically. But of course, the computational cost of the attack increases with 2^l.

5.1.5 Countermeasures

So far in the literature, most countermeasures proposed against fault attacks on pairings focus on protecting the MA for the good reason that it has been the main target of the fault attacks [PV06, OGS07, GS11]. With our attack on the FE, we hope that other efficient countermeasures shall be proposed by the community in addition to the suggestions made below.

Inversion of unitary elements: In some implementations, an efficient countermeasure is already present. Indeed since normally $f_1 \in \mu_{p^d+1}$, this element is called "unitary" and has the following property: $f_1^{-1} = \bar{f_1}$. As a consequence, all inversions besides the first one (necessary to compute f_1) are replaced by a simple conjugation which has a far lower computational cost. As a consequence a fault injected on f_1 cannot be exploited since the resulting output is not

equal to the expected value $(f_1^*)^{\frac{p^d+1}{r}}$. The conclusion is that implementations should ensure that the inversions of unitary elements are always replaced with conjugations. Additionally, the use of a Boolean variable stating if the element is unitary and deciding which code (inversion or conjugation) is used for the inversion of an element should be avoided since this could then become a target in order to allow our fault injection. As an example, this latter Boolean variable is implemented in the classic Miracl library [Cer12].

Compressed representation: A generalization of the previous countermeasure is to use a compressed representation of the elements during the exponentiation as shown in [NBS08, AKL$^+$11]. The effect is similar to the previous countermeasure. A fault attack on an implementation with the compressed representation would have to be specifically designed in order to work.

Checking subgroup membership: It is possible to deter this attack by checking the subgroup membership of intermediate values. As an example, f_1 should be in μ_{p^d+1}. To check this membership (checking $f_1^{p^d+1} = 1$), one has to compute $f_1^{p^d+1}$ at the price of a conjugation and a multiplication in $\mathbb{F}_{p^k}^*$. Similarly it should be possible to check that $f_2^{\Phi_k(p)} = 1$ and $f_3^r = 1$.

5.2 A fault attack with multiple faults during an execution

The fault attack presented in Section 5.1 has the major disadvantage of not working when some optimization techniques, such as replacing the inversion by a conjugation for unitary elements, are used. In this section, we devise a new fault attack which overcomes this difficulty. On the other hand, this fault attack forces the attacker to inject multiple faults during the same execution of the pairing algorithm which is difficult experimentally. This fault attack has been successfully simulated using Sagemath [S+12].

The same notation than in Section 5.1 is used.

First a correct execution is performed and the correct result f_3 is stored in memory.

5.2.1 First faulty execution

A first known fault e_1 is injected on y_1 after one of the unitary inversion as shown on Figure 5.3 and the faulty result f_3^* is used.

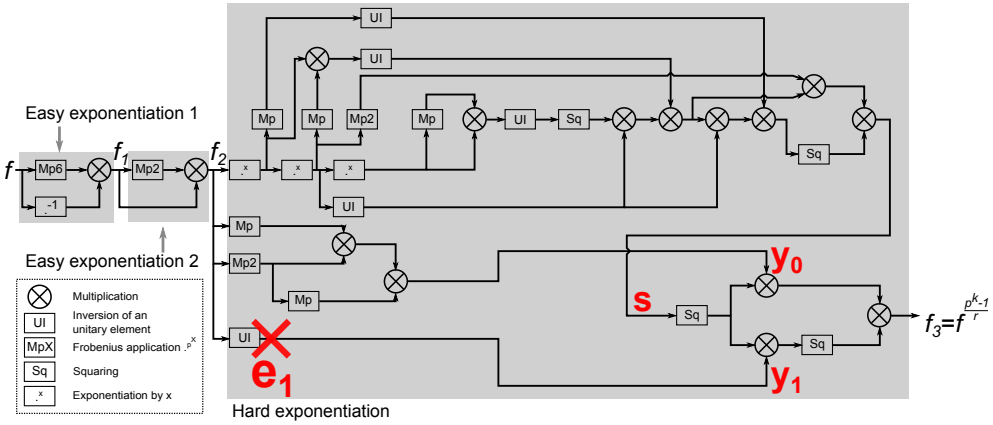

Figure 5.3: e_1 fault location.

In the FE algorithm, the result f_3 is computed as

$$f_3 = s^3 y_1^2 y_0, \tag{5.23}$$

where $s, y_1, y_0 \in \mathbb{F}_{p^k}^*$ are 3 intermediate values following the notations on Figure 5.3. The faulty value is

$$f_3^* = s^3 (y_1 + e_1)^2 y_0. \tag{5.24}$$

The differential Δ_3 can be defined as

$$\Delta_3 = f_3^* - f_3 = s^3 y_0((y_1 + e_1)^2 - y_1^2) = s^3 y_0(2e_1 y_1 + e_1^2). \tag{5.25}$$

Since Δ_1 can be computed by the attacker, the second degree equation

$$\Delta_3 y_1^2 - 2e_1 f_3 y_1 - f_3 e_1^2 = 0 \tag{5.26}$$

can be used to compute y_1. Two solutions are possible

$$y_1^+ = e_1 \frac{f_3 + \sqrt{f_3 f_3^*}}{\Delta_3}, \tag{5.27}$$

$$y_1^- = e_1 \frac{f_3 - \sqrt{f_3 f_3^*}}{\Delta_3}. \tag{5.28}$$

Since the unitary inversion (complement) is an involutary function (it is its own inverse), it is possible to compute $f_2 = UI(y_1)$, the intermediate value before the faulted unitary inversion.

5.2.2 Second and third faulty executions

During the second faulty execution, two known faults are injected: e_2 after the Frobenius application p^2 in the second easy exponentiation and e_3 at the same location as e_1 as shown on Figure 5.4. The result is $f_3^{*(e_2,e_3)}$.

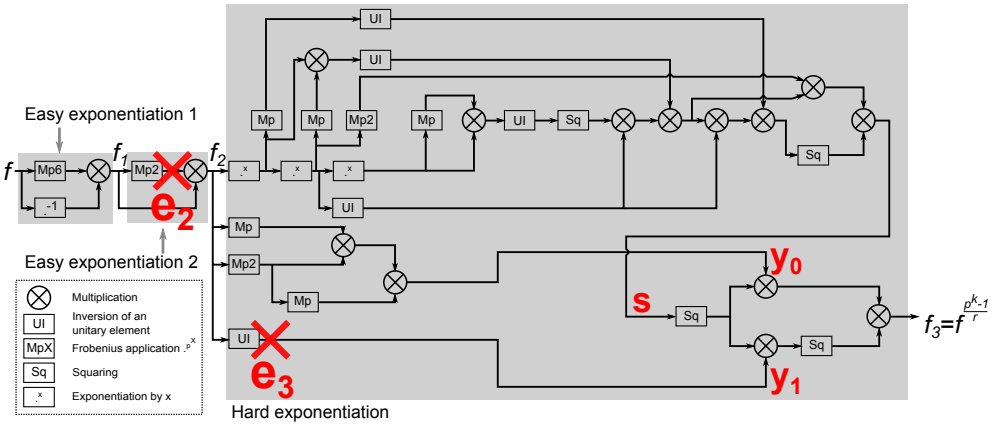

Figure 5.4: e_2,e_3 fault locations.

During the third faulty execution, we inject the exact same e_2 fault but not e_3 ($e_3 = 0$). The result is $f_3^{*(e_2)}$. e_3 is used with $f_3^{*(e_2)}$ and $f_3^{*(e_2,e_3)}$ as in Section 5.2.1 to find the value f_2^*, the intermediate value affected by e_2. We have

$$\begin{aligned} f_2 &= f_1^{p^2} \cdot f_1, \\ f_2^* &= (f_1 + e_2)^{p^2} \cdot f_1. \end{aligned}$$

Therefore the differential is

$$\Delta_2 = f_2^* - f_2 = f_1 \cdot e_2^{p^2}, \tag{5.29}$$

since the Frobenius application is linear. Finally,

$$f_1 = \frac{\Delta_2}{e_2^{p^2}}. \tag{5.30}$$

5.2.3 Fourth, fifth, sixth and seventh faulty executions

During the fourth faulty execution, 3 faults must be injected during the same computation. e_4 is injected after the Frobenius application p^6 in the first easy exponentiation, e_5 is injected in the second easy exponentiation at the same location as e_2 and e_6 is injected after an unitary inversion as e_1 and e_3. The result is $f_3^{*(e_4,e_5,e_6)}$.

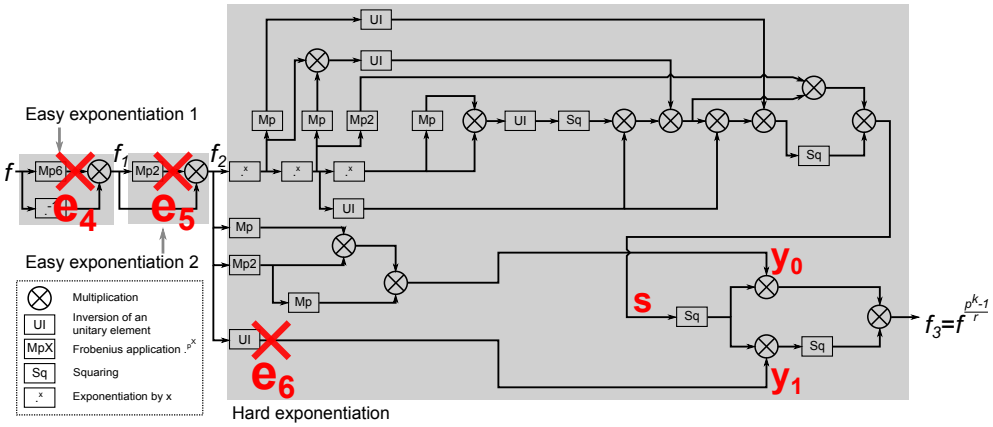

Figure 5.5: e_4,e_5,e_6 fault locations.

3 other faulty executions are required to exploit the faults: the attacker needs $f_3^{*(e_4)}, f_3^{*(e_4,e_6)}$ and $f_3^{*(e_4,e_5)}$. The values $f_3^{*(e_4)}$ and $f_3^{*(e_4,e_6)}$ are combined to obtain the value f_2', the f_2 value faulted by e_4 only (cf. Section 5.2.1). $f_3^{*(e_4,e_5)}$ and $f_3^{*(e_4,e_5,e_6)}$ are combined to obtain the value $f_2^{*'}$ the f_2 value faulted by both e_4 and e_5 (cf. Section 5.2.1). Finally, f_2' and $f_2^{*'}$ are combined to obtain f_1^* (cf. Section 5.2.2). We have

$$f_1 = f^{-1} \cdot f^{p^6}, \tag{5.31}$$

$$f_1^* = f^{-1} \cdot (f + e_4)^{p^6}. \tag{5.32}$$

Therefore the differential is

$$\Delta_1 = f_1^* - f_1 = f^{-1} \cdot e_4^{p^6}, \tag{5.33}$$

since the Frobenius application is linear. Finally,

$$f = \left(\frac{\Delta_1}{e_4^{p^6}} \right)^{-1}. \tag{5.34}$$

The final exponentiation has been reverted with 4 executions: a correct execution, a single fault execution, a double fault execution and a triple fault execution.

Achieving a multiple fault injection is hard but not impossible. Even if this fault attack has not been implemented in practice there is no reason that it cannot be with the proper apparatus. The principal property of this fault attack is that it works even with the use of complements as the inversion of unitary elements.

5.3 Practical fault attack to reverse the final exponentiation

Out of the two fault attack schemes that have been theoretically proposed in Section 5.1 and in Section 5.2, we tested in practice the simplest one in terms of attack bench set-up, *i.e.* the one where only one fault is required per pairing execution. In this section we describe the practical implementation of the this fault attack described in Section 5.1. The same experimental bench than the one described in Section 3.1 has been used. We used our own implementation of the Final Exponentiation (FE) following [SBC+09] with two modifications. First an instruction has been added to trigger the EM pulse. Finally and most importantly, all inversions of unitary elements use a full inversion algorithm rather than a simple conjugation. The use of a conjugation is enough to thwart our attack in this case.

The computation targeted here is the sequel of the Miller computation described in Section 4.2. The same parameters are used. At the end of the Miller algorithm, we have

$$\begin{aligned} f_{K,Q}(P) =& \texttt{0x1C3495395E01A778222B97BB540E5F64AD202595CC4C8DBDF965879343AA6243} \cdot uv^2w \\ &+ \texttt{0x1A31A168303794A99DD4CBB77A9378F41194902CBF94361EEE704D098D6F0EDD} \cdot v^2w \\ &+ \texttt{0x18EC610841767A5A4C957EC6C591D108B277A1378BC6D12AA2B70C80FBCFAA59} \cdot uvw \\ &+ \texttt{0x1BF23B9011E3E4E2E5711317B590F539EB4C0B56E01325E854F1105764A8EE82} \cdot vw \\ &+ \texttt{0x133E283BB4E3BD7FE09860524DC9D0005C82DD44D4C66D240E23C11DFF88E6B} \cdot uw \\ &+ \texttt{0x2870AD9940C6561B872AE50F4BFFFDCE18B2D571D7DDFA6AE7B246488021D8} \cdot w \\ &+ \texttt{0xE6CFD83127E6FA7316C8281EC44610BEDCD57769E9354B6D1281F2700A07709} \cdot uv^2 \\ &+ \texttt{0xDAC2B8C241F9EC596C72D4826111A3DB8051BE0FDDFF1FA0CB56852E87705DC} \cdot v^2 \\ &+ \texttt{0x1EAF6B3724B2609E1D7D29D6A7FB77623B0F1058A937ADE3323203D204A8F902} \cdot uv \\ &+ \texttt{0x86979540B0695A387EA320D8DE23CDDE97F7D33F334FD1BB1B66D93304F69F6} \cdot v \\ &+ \texttt{0x715C601A606825CC05325FFFAF66CF2C33B65214026E39427A8FFEB6C81FA86} \cdot u \\ &+ \texttt{0x1C55FA868ED54113E888AEF97E2C85A6943372B644A1F121781F7D7C8FF0CF7B}. \end{aligned}$$

This value is the input of the targeted final exponentiation, it is the secret value we are looking for.

5.3.1 f_1 recovery

First a correct FE execution is performed and f_3 (in Montgomery domain) is stored in order to check candidates

$$\begin{aligned} f_3 =& \texttt{0x107B6DEBA4224A6FF32AF42145AB199247D30B9B9A342F705F876B729A914E52} \cdot uv^2w \\ &+ \texttt{0x25B3E826C194C498437FF665BC4296C98755C1E1812F306401F164360B4F141} \cdot v^2w \\ &+ \texttt{0xF1A27EE43ED397ECE006DA44ACC3CC103D167C38E599ED541E9AC21E4D27CE} \cdot uvw \\ &+ \texttt{0x1A047A93C9474B9A682F6E83D77BF71D4974E747FA1EEA0F537C29F86AFA5242} \cdot vw \\ &+ \texttt{0xC3DDBCB702500E72E5003837B2791546E5E48E20AE3FC906832C4F2E0AED99} \cdot uw \\ &+ \texttt{0xD2C59DD194F73FB18D68DCD43B53F70E8B6E29F3CE8DD7D1765EA33F2979AA5} \cdot w \\ &+ \texttt{0xE385EE323F62D53BD7FD6251F9435995BAFE561B6E2247E7DE8D4887FAF86C9} \cdot uv^2 \\ &+ \texttt{0x191374D0D088E2C246D733962DB971BE20BAD4ABD41922DFEAA72D93E7E79333} \cdot v^2 \\ &+ \texttt{0x1B24611FA59F35A1C219DE54EB03652CBD5F51EC4F4EDD8B398A5CFB782D35CB} \cdot uv \\ &+ \texttt{0x698E099C239D2F4CB495B39A94EB827BC4EF14B3F2CA20D8DBE697567A2ABD1} \cdot v \\ &+ \texttt{0x1E504941E0F704DE5B1DA39A07273D03874D4E816CCF5D27B4F74CB146F10F2D} \cdot u \\ &+ \texttt{0x1557366DD77CA58DA26355C369FD42F229FB23F980A6D1E20E2DE6F66CE26424}. \end{aligned}$$

Then the first fault is injected. The target is a modular addition during the multiplication which results in f_1. The targeted addition is late in the computation so that it affects f_1 at the minimum. At first the program execution is stopped just when f_1 is computed, just after the fault. f_1^* is read and compared with f_1 (in the Montgomery domain).

$$\begin{aligned} f_1 =& \texttt{0x5FDB9C25D0A17B85870EDEC4FB82798D5F782BE6D2EDD7F46D235E4C27A83AC} \cdot uv^2w \\ &+ \texttt{0x100E3FCA9DA43B3F137657875BC67D960470CB437549158EFE5946F5417BADBA} \cdot v^2w \\ &+ \texttt{0x214F891704ED6FA1E2CA42E6CA92EF9F08A398589C72872CDDF9F11F2B83665E} \cdot uvw \\ &+ \texttt{0xCC20BA9813398E00BAC2D479D913AABBE1C5FA8BEEF3358064A12DB4CADD3E5} \cdot vw \\ &+ \texttt{0x184782481D0AE4ABA7036FF91A5BC02653F5C3717890D29133FC3D1DA154DCC4} \cdot uw \\ &+ \texttt{0x6E2C6FDCE2A0320F1AB77E6C0B2C587167E5B3A78D6D19144816AF96F9E8B0} \cdot w \\ &+ \texttt{0x364CC00E69D5CF101FAFBC6E2CEC34B910D1D5B5042E20E1FBCAA7FCBEE5F18} \cdot uv^2 \\ &+ \texttt{0x93AA709B531E68831A9741FC4163016819CA4B8C37789FF2542B692A33C2836} \cdot v^2 \\ &+ \texttt{0xC32573BA5F40D87DF21D2F0DDBA8E9A1C7AD4B60DB9302664978A3CF782ECFB} \cdot uv \\ &+ \texttt{0x1F6740570FEFC9CC18D832EEF87C99BCF34A8F4A12A4A424C71031FD56D818F1} \cdot v \\ &+ \texttt{0xA65168A71E32CCCC056B016431D70195F23FEDA97C620D236CB43864DF136E4} \cdot u \\ &+ \texttt{0x18D0A39786F8F2397BD3A2A7E15E8C8CB2FE0BE1493AED603396CD896EBCDA28}, \end{aligned}$$

and

$$\begin{aligned} f_1^* =& \texttt{0x5FDB9C25D0A17B85870EDEC4FB82798D5F782BE6D2EDD7F46D235E4C27A83AC} \cdot uv^2w \\ &+ \texttt{0x100E3FCA9DA43B3F137657875BC67D960470CB437549158EFE5946F5417BADBA} \cdot v^2w \\ &+ \texttt{0x214F891704ED6FA1E2CA42E6CA92EF9F08A398589C72872CDDF9F11F2B83665E} \cdot uvw \\ &+ \texttt{0xCC20BA9813398E00BAC2D479D913AABBE1C5FA8BEEF3358064A12DB4CADD3E5} \cdot vw \\ &+ \texttt{0x184782481D0AE4ABA7036FF91A5BC02653F5C3717890D29133FC3D1DA154DCC4} \cdot uw \\ &+ \texttt{0x6E2C6FDCE2A0320F1AB77E6C0B2C587167E5B3A78D6D19144816AF96F9E8B0} \cdot w \\ &+ \texttt{0x364CC00E69D5CF101FAFBC6E2CEC34B910D1D5B5042E20E1FBCAA7FCBEE5F18} \cdot uv^2 \\ &+ \texttt{0x93AA709B531E68831A9741FC4163016819CA4B8C37789FF2542B692A33C2836} \cdot v^2 \\ &+ \texttt{0xC32573BA5F40D87DF21D2F0DDBA8E9A1C7AD4B60DB9302664978A3CF782ECFB} \cdot uv \\ &+ \texttt{0x1F6740570FEFC9CC18D832EEF87C99BCF34A8F4A12A4A424C71031FD56D818F1} \cdot v \\ &+ \texttt{0xA65168A71E32CCCC056B016431D70195F23FEDA97C620D236CB43864DF136E4} \cdot u \\ &+ \texttt{0x18D0A39786F8F2397BD3A2A7E15E8C8CB2FE0BE1493AED603396CD8}\underline{8B75E6E10}. \end{aligned}$$

The effect of this fault is to replace `96EBCDA28` by `8B75E6E10` in the last coordinate of f_1. During a campaign (with carefully chosen parameters) of 1014 pulses emitted by the EM bench, 397 (39%) pulses induced an interrupt and 164 (16%) undetected faults were created on the least significant word. No fault was created on the other words, apart from the carry propagation due to the faulted least significant word.

The values obtained are

- Correct value (no fault): $0x96EBCDA28(45\%)$,
- $0x96EBCDA29(0.5\%)$,
- $0x8B75E6E10(10\%) \approx 0x96EBCDA28/2$,
- $0x8F75F792B(4\%)$,
- $0x8FFFFFFFF(1\%)$.

It is difficult to exactly explain what the exact effect of the electromagnetic pulses is since we do not have inside knowledge of the inner workings of the chip. We guess that in this case, the pulse affects the bus when data are retrieved from the RAM. The synchronisation may be altered which leads to 'shifts' within a word, or to the use of the bus pre-charge value of $0xFFFFFFFF$.

In order to obtain the faulted value $0x8B75E6E10$, an AMP of $-180\,\mathrm{V}$ and a DELAY of 323 ns have been used.

The fault value in the Montgomery domain is

$$diff = 0x8B75E6E10 - 0x96EBCDA28 = -3076418584. \tag{5.35}$$

In order to use it, we convert it to the canonical domain

$$\begin{aligned} e &= diff/Res \\ &= \texttt{0XEEBC50653A984CDCBE1D6D4F3C7F9D20B160EE882B7E27A407E41CA8A5E5053}. \end{aligned} \tag{5.36}$$

The same experiment is then repeated with the exact same parameters but the final result f_3^* is read this time.

$$\begin{aligned} f_3^* =& \texttt{0x19F8793DFF653171A5F9171EEFD67C1A29DC47C1AD132596562D3826B385132C} \cdot uv^2w \\ &+ \texttt{0x16C005CEE7FD9C8826DBF28F32360B2FE9EA2CAFCD3E961C7DA81DABBE7214A5} \cdot v^2w \\ &+ \texttt{0x13885A829278B96AF6E55A3DCEA663CE8DEFB5FE8A2C29A6C5C4F7D00ACECDE6} \cdot uvw \\ &+ \texttt{0x1B42DE1BE00913EE043F0AD81232FD764DBA2274ECD11F172468BE56B851B924} \cdot vw \\ &+ \texttt{0x11C16ECA0AC99DEA91442AADCD45B294A7937EBCECE18545E890193DA46D08F9} \cdot uw \\ &+ \texttt{0xE8767BE11C51951AE16CC85AB5242CA61C929BA6F43FC83FF3A86D9BED1FBF2} \cdot w \\ &+ \texttt{0x110F46519BA585532F19CF940A1FF9B39CDCCBCEBABB2C31A04FD2C38E313B26} \cdot uv^2 \\ &+ \texttt{0x105FCC4E58D211810C9618E55EDB293701CD8BC11F1A7BDBB996F6B2E7B8B4C0} \cdot v^2 \\ &+ \texttt{0xFA22B66770741E868F3EB91C01B024C2F9C1E4B1AB26FA5DDDFE0611475E9F7} \cdot uv \\ &+ \texttt{0x14242EC2A173FDE09CE0B827249EAF6E669975E14FBD32FECB7C3AE39D8E5E55} \cdot v \\ &+ \texttt{0x1F82C0F38B3043FEA738F839F1C8EC4675637DE159E46417F92178FB67FFDC16} \cdot u \\ &+ \texttt{0x8F2259B09127C8D16A0F5E66C6B4AD59E70305618F857DDE26B208DB9CCCF35}. \end{aligned}$$

The values f_3 and f_3^* are converted into the canonical domain for the fault exploitation. g_1 can now be computed (cf. Section 5.1)

$$g_1 = \frac{(f_3^*)^r - 1 - e^2}{2e}, \tag{5.37}$$

giving

$$\begin{aligned} g_1 =& \texttt{0x9EDCFE012D677FB155D6C0AE210EBC7762C9F676072AD46807B31997697FC8E} \cdot uv^2 \\ &+ \texttt{0x9DAE376D2225F2526799EBA325CAE6E021D5F777D4FA8474C229BAFBC28B441} \cdot v^2 \\ &+ \texttt{0x15E3A5496D59266091FDD24814B54BB239328C2D8DD764340A8FA5BE966A0EFE} \cdot uv \\ &+ \texttt{0xA426C8E955FC3B8DED16A9FB4A047BED94C6344DB89DE72B63E71C0F841FB37} \cdot v \\ &+ \texttt{0x2249DDA136999431017A04432EAA77342D12738F94AE6FFC656A1FC58E564F5D} \cdot u \\ &+ \texttt{0xEE7973B50A3182A75C4A1111DFE571FC4341A9DE101179D692F3F394F1F5CD0}. \end{aligned}$$

h_1 is computed with

$$h_1 = \pm\sqrt{\frac{g_1^2 - 1}{v}}, \tag{5.38}$$

giving (by checking $f_1^{\frac{p^{12}+1}{r}} = f_3$)

$$\begin{aligned} h_1 =& \texttt{0x13C2194E28244891C2E205FCE4A61F45B3E8446BFC8EDB56832FA0902C36F611} \cdot uv^2 \\ &+ \texttt{0x7578D74939C16BDD9A7C2008B6BC17922927C087995FC07D50F7B2ECDF41BD4} \cdot v^2 \\ &+ \texttt{0x220D21C23C03BBB464AD4F51A54037E92B95C26C068A9EBAF26C5B9D6178EAAD} \cdot uv \\ &+ \texttt{0x160908931D6414EEC95E9DFFBCB8D6E233F023B247D3EC6DE0642C506FE05446} \cdot v \\ &+ \texttt{0x1293850D06DD8020114A696F2E8BA22A9BDD7A8396A47DC3443214E78A51D52A} \cdot u \\ &+ \texttt{0x15AD42DC9E1B5C93B00F5FDE644119487E64739E5BCA8293A0ABE4E65CB13664}. \end{aligned}$$

f_1 has been found.

5.3.2 f recovery

As a preliminary computation, the value $K = \frac{g_1 - 1}{v \cdot h_1}$ is evaluated.

$$\begin{aligned} K =& \texttt{0x329406F47FF1ABB9A91D8EABC3934F87F6618E7C1D744AE4767A5856E97A74A} \cdot uv^2 \\ &+ \texttt{0x2110273BDEAFE5DBF27A7FB4EB878D3AD8BE284DC619DA63DC97ABD89B50F677} \cdot v^2 \\ &+ \texttt{0x20416690C819B91EBBED917ECE2648225EBCB500582BD87C4068B047EE05A06} \cdot uv \\ &+ \texttt{0x1C8B980C2D002A2D3181C44D4537343BF924751420D37D12FDE1ED5EA25C2731} \cdot v \\ &+ \texttt{0x31EA64FF05DA96DEA36DE317C474C8230B57219C9EF445F272BA69F18706AB} \cdot u \\ &+ \texttt{0xFDD114E4EA57E1742D5D5A936D1161424CA584C986E3561D6BB55B6EC2CA200}. \end{aligned}$$

A second fault attack is performed targeting a modular addition during the computation of the inverse of f in the first easy exponentiation.

First we stop the computation and read the value of f^{-1} just after the fault in order to observe the effect of the fault injection.

The following parameters produce interesting results: AMP between $-210\,\mathrm{V}$ and $-170\,\mathrm{V}$, DELAY between 322 ns and 325 ns. As previously, the last 33-bits of the targeted value are potentially changed by the fault attack.

During a particular campaign of 4579 fault injections, the values obtained were

- Correct value (no fault): $0xEC99E3A76(55\%)$,
- $0xF010AA837(13\%)$,
- $0xF08949F52(2\%)$,
- $0xEC99E3A77(0.8\%)$,
- $0xEFFFFFFFF(1\%)$.

The fault value in the Montgomery domain is

$$diff = 0xF010AA837 - 0xEC99E3A76 = 929852865. \tag{5.39}$$

In order to use it, we convert it to the canonical domain

$$\begin{aligned} e &= diff/Res \\ &= \texttt{0XFF547F01541833433371770A616487E643B90FF2309A944DE9C3BA74DC687A0}. \end{aligned} \tag{5.40}$$

After the observation of the faulty final exponentiation, we find (in the Montgomery domain)

$$\begin{aligned} f_3^* =& \texttt{0xBCB00049BBE26E67341289442140CEA59A2CE8493C62FFC982383490535BF70} \cdot uv^2w \\ &+ \texttt{0x175044754CA1FCF2F9F4AE9F78B7C2E6FE9842873F4D37CF42D0A667120C6220} \cdot v^2w \\ &+ \texttt{0x1CEF6C98ED0E47FBA62EE22E81DC47C68C400476D7A07525C5054073D1AA01D} \cdot uvw \\ &+ \texttt{0x1B84EC5111DEEF381903B58938B4C5F25D5AFEA9E63EE8C4F42B7FFDDBC527B6} \cdot vw \\ &+ \texttt{0x188418D32276E6D2F1B4080EA58780663427BF47D1E5C0B112F1FA58CE7F129B} \cdot uw \\ &+ \texttt{0x146B334E26F48B3ECC0BBFAC6EA2BF9E662102C451066B53A5EDCA9BEE2E33B6} \cdot w \\ &+ \texttt{0x87DB5011959F0D4BD6117650425045 8FA3C23305940754BD48B9127A3E68133} \cdot uv^2 \\ &+ \texttt{0x168A22E93506C43BC9EFACFAA265350D634221187FF0EF97CF32DEE754B9B8A8} \cdot v^2 \\ &+ \texttt{0x7021A5E532CD10CC1D6E3B8EBEF39C431F4A30616F79F283B2A1D20503E8C5} \cdot uv \\ &+ \texttt{0x1A715E6FE1676A8B5D6E9EEFF9E2A32AAE5B76D7F4BEF729FF539F76A13206EF} \cdot v \\ &+ \texttt{0xBE636CE76F835C2893ECC2E63E03E3451EB28C590F6B0A20C1A897F6B168DE6} \cdot u \\ &+ \texttt{0x176516F5FBACD069531C8DD2E315C801E0F9AFF51805A30E26FE02B9E3EFD8EF}. \end{aligned}$$

As in Section 5.1.2, two candidates are found for g with Equation (5.18).
Among them,

$$\begin{aligned} g =& \texttt{0xE6CFD83127E6FA7316C8281EC44610BEDCD57769E9354B6D1281F2700A07709} \cdot uv^2 \\ &+ \texttt{0xDAC2B8C241F9EC596C72D4826111A3DB8051BE0FDDFF1FA0CB56852E87705DC} \cdot v^2 \\ &+ \texttt{0x1EAF6B3724B2609E1D7D29D6A7FB77623B0F1058A937ADE3323203D204A8F902} \cdot uv \\ &+ \texttt{0x86979540B0695A387EA320D8DE23CDDE97F7D33F334FD1BB1B66D93304F69F6} \cdot v \\ &+ \texttt{0x715C601A606825CC05325FFFAF66CF2C33B65214026E39427A8FFEB6C81FA86} \cdot u \\ &+ \texttt{0x1C55FA868ED54113E888AEF97E2C85A6943372B644A1F121781F7D7C8FF0CF7B} \end{aligned}$$

is the correct solution. h is computed with $h = -g \cdot K$.

$$\begin{aligned} h =& \texttt{0x1C3495395E01A778222B97BB540E5F64AD202595CC4C8DBDF965879343AA6243} \cdot uv^2 \\ &+ \texttt{0x1A31A168303794A99DD4CBB77A9378F41194902CBF94361EEE704D098D6F0EDD} \cdot v^2 \\ &+ \texttt{0x18EC610841767A5A4C957EC6C591D108B277A1378BC6D12AA2B70C80FBCFAA59} \cdot uv \\ &+ \texttt{0x1BF23B9011E3E4E2E5711317B590F539EB4C0B56E01325E854F1105764A8EE82} \cdot v \\ &+ \texttt{0x133E283BB4E3BD7FE09860524DC9D0005C82DD44D4C66D240E23C11DFF88E6B} \cdot u \\ &+ \texttt{0x2870AD9940C6561B872AE50F4BFFFDCE18B2D571D7DDFA6AE7B246488021D8}. \end{aligned}$$

Finally, the value $f_{K,Q}(P)$ has been found and the final exponentiation reverted with an experimental fault injection.

5.4 Conclusion

We have proposed two theoretical fault attacks on the final exponentiation, proving for the first time that the complex FEs can be inverted by a fault attack. We have validated in practice one of the schemes, therefore proving its experimental feasibility. We have seen that it is easy to thwart the fault attack proposed in Section 5.1 by replacing the inversion of unitary elements by a conjugation. But this countermeasure is not efficient against the scheme in Section 5.2. In this case, a solution would be to use redundancy or error-detecting code in order to detect a value modification. With the final exponentiation inverted, the way is open towards a fault attack on complete pairings.

Chapter 6

Fault attacks on the complete pairing

Where we demonstrate a fault attack on a complete pairing.

Contents

Previously, fault attacks on the Miller algorithm or on the final exponentiation have been presented independently, and tested in practice separately. None of these, alone, is an effective threat against pairing computations. One needs to combine these fault attacks in order to completely invert a pairing. In this section we first propose strategies to attack the whole pairing and then a practical implementation of one of the proposed schemes is detailed.

6.1 Theoretical complete fault attacks with the instruction skip fault model

In these fault attacks, we consider that the attacker is able to skip chosen instructions during the pairing computation. We must loosen the constraints on our requirements and allow double faults, two faults during the same execution (which as already been done in [TK10, BGdSG+14]). Experimentally, it may be difficult to inject several faults during one computation. This difficulty depends on the delay between injections and on the equipment used. That is why we prefer attack schemes with a limited number of injections per execution even if it implies a bigger number of executions. *The search for new possible fault attack schemes on complete pairings has been done in collaboration with Hélène Le bouder.*

6.1.1 Complete fault attack with a loop skip

The principle behind this attack is to retrieve intermediate values in the algorithm. Since the attacker observes the output, it is easier to retrieve the intermediate value near the end of the computation at first and then to include this information to progressively go up the algorithm.

The first step in this attack is to retrieve the correct result of the Miller algorithm using the fault attack presented in Section 5.1. Once the attacker knows the correct result of the pairing $e(P,Q)$, the fault e_1 is used to recover f_1 and then a second fault e_2 is used to find f, the result of the Miller algorithm.

Now, in order to find P the secret point (or Q), the attacker has to fault both the Miller algorithm and the final exponentiation during the same execution.

In the instruction skip model, the attacker can skip the loop iteration she wants, and as a consequence she can exit from the loop at the desired iteration. In this case, the fault (instruction skip) is perfectly reproducible and will have the exact same effect on the computation.

Fault e_3 on the last Miller loop iteration

If the loop skip e_3 occurs on the last iteration, a complete attack scheme would be the following (with indexed executions).

1. No faults: obtains $f_3 = e(P,Q)$.
2. e_1: allows to obtain f_1.
3. e_2: obtains f the correct Miller algorithm result.
4. e_3: obtains f_3^* a faulty Miller result after a correct final exponentiation.
5. (e_3, e_1): obtains f_1^*.
6. (e_3, e_2): obtains f^* the faulty Miller result.

Finally with f and f^*, the attacker can recover the secret as shown in Section 4.2. This fault attack requires 6 executions with 3 single-faulted executions and 2 double-faulted ones. To make it work, the fault e_3 must be repeated in several executions.

The executions 1,2,3 are necessary to find the correct result of the Miller algorithm following the fault attack scheme described in Section 5.1. The fault e_3 corresponds to a loop skip, the faulty Miller algorithm has its last iteration skipped. The executions 4,5,6 combine the loop skip with the fault attack scheme on the FE presented in Section 5.1. It works only because the fault e_3, the last iteration skip, gives exactly the same value $f_{K,Q}(P)^*$. These 6 executions provides both $f_{K,Q}(P)$ and $f_{K,Q}(P)^*$ allowing to use one of the fault attack presented in Section 4.1.2.

This complete attack scheme on pairings has been implemented in practice as described in Section 6.2.

Fault e_3 on the first Miller loop iteration

If the attacker is able to exit the loop after the first iteration, she can reduce the number of required executions to only 3.

1. e_3: obtains f_3^* a faulty Miller result after a correct final exponentiation.
2. (e_3, e_1): obtains f_1^*.
3. (e_3, e_2): obtains f^* the faulty Miller result.

This fault attack requires 3 executions with 1 single-faulted execution and 2 double-faulted ones. In this scheme, e_3 corresponds to the case where the attacker is able to exit the loop after the first iteration. e_3 must be exactly repeated through executions 1,2,3. These 3 executions allow to invert the final exponentiation as in Section 5.1 providing the value $f_{K,Q}(P)^*$. This single value allows to invert the MA as shown in Section 4.1.2 (Exit after the first iteration attack).

6.1.2 Other possibilities

Another possible scheme (on a Tate pairing) is to combine the *if* instruction skip by [BMH13] presented in Section 4.1.2 (by Bae *et al.*) and the fault attack to revert the final exponentiation (cf. Section 5.1). The resulting scheme is similar to the one presented in Section 6.1.1 (in 6 executions) but e_3 becomes an *if* skip instead of a loop skip.

The FE fault attack presented in Section 5.2 can also be combined with a loop skip on the Miller loop but it requires 4 faults injected during one pairing execution.

Despite our efforts, no complete fault attack able to avoid double faults was found. Such an attack would increase the experimental ease to implement a fault attack able to revert a complete pairing algorithm.

Another strategy could be to use combined attacks, where the attacker perform a FA while measuring the power consumption for example. This way a FA may be required only on the Miller algorithm or on the Final Exponentiation and the SCA allows to recover the missing information.

6.2 A practical complete fault attack on pairings

As a first demonstration of the possibility to invert a pairing, the attack scheme presented in Section 6.1.1 (in 6 executions) has been implemented with EM pulses (using the platform from Section 3.2).

As a reminder, the 6 following executions have to be made.

1. No faults: obtains $f_3 = e(P, Q)$.
2. e_1: obtains f_1.
3. e_2: obtains f the correct Miller algorithm result.
4. e_3: obtains f_3^* a faulty Miller result after a correct final exponentiation.
5. (e_3, e_1): obtains f_1^*.
6. (e_3, e_2): obtains f^* the faulty Miller result.

6.2.1 The particularities of double fault injections

In order to inject a double fault, *i.e.* two faults during one execution, our EM bench had to be slightly modified. Indeed, between the two fault injections, the pulse generator must be reconfigured to match the parameters required for the second fault injection. We are missing a trigger signal which notifies the controlling PC that the first fault injection has been performed so that the PC can send the new parameters to the pulse generator. In practice we plugged the trigger signal to the PC sound card with the help of a small adaptation circuit (cf Figure 6.1).

In order to be able to see the trigger signal with the PC, the up time of the trigger has been lengthened to be above 50 ms (detectable with a 44 100 Hz sampling). We initially hoped that the duration of the reconfiguration would be less than the execution time between the two faults but the maximal baud rate between the PC and the pulse generator is 9600 bauds. We had to modify the library to add a delay between the Miller algorithm and the final exponentiation. A better equipment may allow to remove this modification.

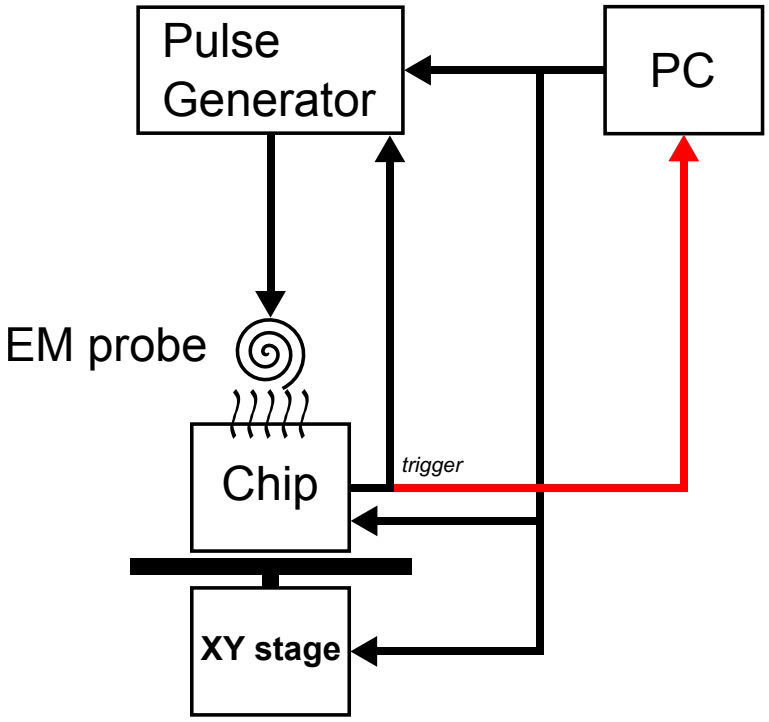

Figure 6.1: Upgraded EM bench scheme.

As a conclusion, in the double fault mode, the chip sends two trigger signals per execution. The trigger signals are used by the PC to send the next configuration to the pulse generator.

6.2.2 Reverting a pairing in practice

Executions 1,2,3

The practical experiments for recovering the correct result of the Miller algorithm (with executions 1,2,3) have already been presented in Section 5.3. The same experiments have been used, with the same parameters. The public point P is

$$P = (\texttt{0x38009F84045BBBC1BE5D7EBE2AE3CC1AD2DB2A342856477FD090951DFF430A1},\\ \texttt{0x7401C9670C5C62BC083614A6080C25025B9BBA7C49D46A9AEB7077CC58CA36E}),$$

and Q is the unknown secret point that we want to recover

$$Q = (\texttt{0xA1CF585585A61C6E9880B1F2A5C539F7D906FFF238FA6341E1DE1A2E45C3F72} \cdot u+\\ \texttt{0x19B0BEA4AFE4C330DA93CC3533DA38A9F430B471C6F8A536E81962ED967909B5},\\ \texttt{0xEE97D6DE9902A27D00E952232A78700863BC9AA9BE960C32F5BF9FD0A32D345} \cdot u+\\ \texttt{0x17ABD366EBBD65333E49C711A80A0CF6D24ADF1B9B3990EEDCC91731384D2627}).$$

We find the result of the Miller algorithm as in Section 5.3

$$\begin{aligned} f_{K,Q}(P) =& \texttt{0x1C3495395E01A778222B97BB540E5F64AD202595CC4C8DBDF965879343AA6243} \cdot uv^2w \\ &+ \texttt{0x1A31A168303794A99DD4CBB77A9378F41194902CBF94361EEE704D098D6F0EDD} \cdot v^2w \\ &+ \texttt{0x18EC610841767A5A4C957EC6C591D108B277A1378BC6D12AA2B70C80FBCFAA59} \cdot uvw \\ &+ \texttt{0x1BF23B9011E3E4E2E5711317B590F539EB4C0B56E01325E854F1105764A8EE82} \cdot vw \\ &+ \texttt{0x133E283BB4E3BD7FE09860524DC9D0005C82DD44D4C66D240E23C11DFF88E6B} \cdot uw \\ &+ \texttt{0x2870AD9940C6561B872AE50F4BFFFDCE18B2D571D7DDFA6AE7B246488021D8} \cdot w \\ &+ \texttt{0xE6CFD83127E6FA7316C8281EC44610BEDCD57769E9354B6D1281F2700A07709} \cdot uv^2 \\ &+ \texttt{0xDAC2B8C241F9EC596C72D4826111A3DB8051BE0FDDFF1FA0CB56852E87705DC} \cdot v^2 \\ &+ \texttt{0x1EAF6B3724B2609E1D7D29D6A7FB77623B0F1058A937ADE3323203D204A8F902} \cdot uv \\ &+ \texttt{0x86979540B0695A387EA320D8DE23CDDE97F7D33F334FD1BB1B66D93304F69F6} \cdot v \\ &+ \texttt{0x715C601A606825CC05325FFFAF66CF2C33B65214026E39427A8FFEB6C81FA86} \cdot u \\ &+ \texttt{0x1C55FA868ED54113E888AEF97E2C85A6943372B644A1F121781F7D7C8FF0CF7B}. \end{aligned}$$

Execution 4

The practical experiment to obtain a faulty Miller result (execution 4) has already been presented in Section 4.2. The exact same protocol has been used here but instead of stopping the execution of the pairing after the Miller algorithm, we let it run its course until the end and we read (in the Montgomery domain)

$$\begin{aligned} f_3^{*(e_3)} =& (f_{K,Q}(P)^*)^{\frac{p^{12}-1}{r}} \\ =& \texttt{0x17438C8FB457CC4BE0EEC5E08BAB5B23F94BFAC1A684346D401C9DA8D48F47BC} \cdot uv^2w \\ &+ \texttt{0xE2ACB6EE10120651DE1FC678F4FEB0588BD036E2BBBD983D7DC05034DF04E80} \cdot v^2w \\ &+ \texttt{0x1806AA72460E43D34783C3B397909CF959958F1BFF444900A73FB4081C01B31F} \cdot uvw \\ &+ \texttt{0x1CA01D91C7B3EE013D25E0AA50DE95AAD3ECB98DA3A56DA7F248D2FFA00881E2} \cdot vw \\ &+ \texttt{0x202B58077B1F88DAEF8D738CD99170D985387208ABA644FE68C7E1F795CE1EAE} \cdot uw \\ &+ \texttt{0x9C65013E38D3B086CCDD3B1AAF25DF913822E36C39D8E41C0EF7F65E2D5987E} \cdot w \\ &+ \texttt{0x17FC8B411E4BF48C80112A689BA7FA7899043B05D76AC1BF80D942F0E0F8AED3} \cdot uv^2 \\ &+ \texttt{0x19DF6DEEE1973946BBDA00B8637295605D445F997BCC385F45708C47F79C0AA5} \cdot v^2 \\ &+ \texttt{0x18A15D5B3753E5F62BA68BAEF044B1B137B29A7440ECE4DBE3F8662FF5BFE2CE} \cdot uv \\ &+ \texttt{0x4250703C0AD280871D255B211FF0ABB64B143D6EF39F67A8E995F94E2ED112} \cdot v \\ &+ \texttt{0x1EA051736A2CCA147E5D3F28B15E11320BD35FA2DBDAD9A444F6065E5F6471E5} \cdot u \\ &+ \texttt{0x110AB8913D3D6D188D2DFD52BDED929BFE2D3D3BDAC6C3772832F294507527AD}. \end{aligned}$$

Execution 5

In this execution, a double fault has been tried. First the two pulse generator configurations to create the two faults have been found independently. The results are shown on Table 6.1 and Table 6.2.

Table 6.1: Statistics for a fault on Miller only (total: 50 injections). AMP: −190 V, DELAY: 235.6 ns

Result	*Proportion*
Interrupt	36%
Undetected fault	18%
Correct execution	46%

Table 6.2: Statistics for a fault on the Final Exponetiation (fault on f_1, total: 50 injections). AMP: −150 V, DELAY: 329 ns

Result	*Proportion*
Interrupt	2%
Undetected fault	98%
Correct execution	0%

We can seen that better parameters have been found for the fault injection on the final exponentiation resulting in a better reproducibility. The first attempt at a double fault injection by combining the 2 configurations above was a failure. The resulting observations are shown on Table 6.3.

Table 6.3: Statistics for a double fault injection (total: 650 injections).

Result	*Proportion*
Interrupt	40.7%
Undetected fault (faulted Miller only)	12.8%
Undetected fault (faulted FE only)	46.5%
Undetected fault (double fault)	0%
Correct execution	0%

It appears that if a fault is successfully injected on the Miller algorithm, it prevents the fault on the final exponentiation. Our guess is that it creates a temporal shift in the program execution. First we searched a better parameter for the fault injection on the Miller algorithm and found that with an amplitude of −160 V and a delay of 69 ns we were able to inject undetected faults on the Miller algorithm with a 90% reproducibility. These parameters were kept for all subsequent double fault injections.

To try this hypothesis, we randomized the DELAY parameter of the pulse generator for the second fault injection. And finally double faults were successfully injected for a DELAY between 329.6 ns and 330 ns. The reason for this difference of ≈0.8 ns for the success of the fault injection is not obvious. Indeed if a fault on the Miller algorithm delays the course of the whole program it should also impact the trigger and finally no difference would be seen on the DELAYparameter of the second fault injection. Now, with the configurations for the double fault found, we first stop the program just after the second fault to read the faulty result and deduce the fault value. We compare it to the result with only a fault on the Miller algorithm, intermediate result obtained with a computation (because we know the points P and Q in our controlled experimental environment). We recall that it is not necessary to have this value (cf. Section 5.1), just convenient.

We find a differential value of

$$diff = 0x4ABB1E67F - 0x55763CB06 = -2880562311. \tag{6.1}$$

The error value is

$$\begin{aligned} e_1 &= diff/Res \\ &= \texttt{0X14544A8654A5D5EB4A33D643BE2F73E663916C939562565603175F018EFB8934}. \end{aligned} \tag{6.2}$$

Now we can observe the result of the algorithm with a double fault

$$\begin{aligned} f_3^{*(e_3,e_1)} =& \texttt{0x1DBF8332021E61C81F4EB07E0D9A8777758F8711CCAE6DFD30C669FFBB37F4C1} \cdot uv^2w \\ &+ \texttt{0x9309E0ADAB933DE4BB72A096BEA12F5F810F78CE521C8A0551AA6750B9435B6} \cdot v^2w \\ &+ \texttt{0x1051141E34CC27B8EC18AF8B1443C2F91781B8A2004AC30FADA29F684BD90522} \cdot uvw \\ &+ \texttt{0x1A9CD399A9189710015487CF2EC11CF416F88D08A237FC2D33A7ABE73FC088D2} \cdot vw \\ &+ \texttt{0x1CA00D25447AE666245905D4BB50BCC0083FF6EC3BD5B272F3CA09E783B9F261} \cdot uw \\ &+ \texttt{0x1097B1E69A811D87FAE447B3D5CBA2E79AAC5E69360E0D32B1E306AE8378AE1B} \cdot w \\ &+ \texttt{0x1850FC109E7833674F7C988ECEF8D013F837137A9A5EE09C348CE13784C290DC} \cdot uv^2 \\ &+ \texttt{0x14C06D775234BFFF657A6B49A4CA91F7973C3B1E066069CE6F0ED9075A822458} \cdot v^2 \\ &+ \texttt{0x14DEAFB6F303DF17868FF9B3EB03FD5861FE422F569EF57E5FC2D6AE0EF243DE} \cdot uv \\ &+ \texttt{0x9359EA65CCE222940ED82712000A861FFD78DCEA9FB7A39FD6DC104A35A85E8} \cdot v \\ &+ \texttt{0x141C366E1EA6ED81C6BE10FEAF794F3D7CBC5F3EF380F1A1CEF7C3E83032761B} \cdot u \\ &+ \texttt{0x27C6365AEFB541FB310F762B4D4E6CE02E69B259BE2EBB3EF7C8CEBE5CE48FF}. \end{aligned}$$

We used Sage for the same fault exploitation as in Section 5.1 and 5.3 to find $f_1^{*(e_3)}$ (in the canonical domain),

$$\begin{aligned} f_1^{*(e_3)} =& \texttt{0x9F171CF26B77CCD15C8311CAD3859E08EEEBB04AE28F4DDBE4C72B1BDE05F81} \cdot uv^2w \\ &+ \texttt{0x109DD0C2FDA4889A706D77A527D56628A07C19B3691578344AF6E165AA95C148} \cdot v^2w \\ &+ \texttt{0x1D281EEA8A167FEFFF4139CECF801BBD7973C6C547F0034FAE737D16C2A67EA6} \cdot uvw \\ &+ \texttt{0x7F9B169CEDEE6F82979DF6A8F28DB37DCC0E9E6E711F0DD6A31F9F5B75344A9} \cdot vw \\ &+ \texttt{0x1A4AB53B7BC8B5AC68E6C49F0C657BD04A18A2D1C1FBA0EE56B9DD71990E3B0F} \cdot uw \\ &+ \texttt{0x4EC908ED02E52E8A8D49BEF39DAC2EA370E4EEF3C0589570999FF755C1DF966} \cdot w \\ &+ \texttt{0x1510F589C6CF10E4605744E8261E86E6137832935C013EDAF217C2C9686C9FFE} \cdot uv^2 \\ &+ \texttt{0x21C002F52E712F6D754A0D428EC8F4ADFC9B29A513F9D8AFCE6AD42237873097} \cdot v^2 \\ &+ \texttt{0x10009CD8E53F8FC486518D0A92CC96295DD577C74DD7946F3D191B7621499637} \cdot uv \\ &+ \texttt{0xBA68A4C60A52C33F402B7188662FAAE5BE0FBF913959AC06304A920510FC085} \cdot v \\ &+ \texttt{0x16F7490593971BC9A37DBBA4DB03B7C372D04D9DC29D46227ABA221B39FC02F9} \cdot u \\ &+ \texttt{0x107093DF08FC9A9BD2F27BA6B915983CE70478BEEEA11B5441699C9691F80174}. \end{aligned}$$

Execution 6

The coefficient $K = \frac{g_1 - 1}{v \cdot h_1}$ can be computed,

$$\begin{aligned} K =& \texttt{0xE659F9958EA60E6BCFCA3762606EA533043F384AA9989DB4AAFA8406C566EA9} \cdot uv^2 \\ &+ \texttt{0x115D77641BB94297C4171B16D1E72A2403D58F308874BAEDDE62A76AAE336B18} \cdot v^2 \\ &+ \texttt{0x71D752138181D7C687C347EDC33D9CFD94A97D7095F53BD2735BF718930F252} \cdot uv \\ &+ \texttt{0xB6659CBDA898ACEA34E4EE919EC901DF686D4C21AA303D12571AED8C698F7} \cdot v \\ &+ \texttt{0x60B35886A51ED31A6DA511EF2B962D5923D7E7B68104ECCBE41FD855E89F583} \cdot u \\ &+ \texttt{0x68D6FC8D8B0D8174F8FDEC246E0072E3EB21C24666E35D7FD91B31B527C2096}. \end{aligned}$$

At this point the last pairing must be executed with a double fault. We used the exact same parameters as for the previous double fault since we target yet another modular addition for the second fault. As previously, we first stop the execution just after the second fault to read the fault value. The observed differential is

$$diff = 0x3CF92B90 - 0x037FCE17 = 964255097. \tag{6.3}$$

The error value is

$$\begin{aligned} e_1 &= diff/Res \\ &= \texttt{0X13BD91B58C9B355E3591C67891C016CDCD4AE0D9F6AAF9D10A23E7CB446C551C}. \end{aligned} \tag{6.4}$$

The result (in Montgomery domain) of the algorithm with the (e_3, e_2) double fault is

$$\begin{aligned} f_3^{*(e_3,e_2)} =& \texttt{0x1E7E263BE9EBAFA86F9BBECFC832782705AC53C929A4CC92BB5AFA1D249D94EB} \cdot uv^2w \\ &+ \texttt{0x5A8177EB5ED9B5E599EDC810F72443DB43BD0465A56ACB7B4578DF763A8A499} \cdot v^2w \\ &+ \texttt{0x1F41304BF4E7585CEF44F150D797B7843565830627F01EAD21B0F5692F76DD9F} \cdot uvw \\ &+ \texttt{0x19704E57CE1A1A3E69CEA7E41F2052C5975D69D735D9AA381DEE89AC7EFEBC50} \cdot vw \\ &+ \texttt{0x20862B8F267688611D2BCC7A05F7A4528FBE28CBB3A8967F18F340303542BB5B} \cdot uw \\ &+ \texttt{0x20EF0721E961FBC823FD11F964BD94CB651238BEDE0D560E1B23A0A22AA17009} \cdot w \\ &+ \texttt{0x109E8AAAB7EF73195E0108DA56C9512F60C7E824B250316C2C6A67DD334960E6} \cdot uv^2 \\ &+ \texttt{0x1ABF14B7304E862321258CF0AEE1682684A1C444E124809F79B7DF869BC3F7E9} \cdot v^2 \\ &+ \texttt{0x15B714314ECD13827BBE92E756328A882F5D230F84ABE15771629758D6073DA6} \cdot uv \\ &+ \texttt{0x5CDF0992A0BF5FCB1676C46C40E7595D50C1AC07DA49C83165C92619CC16AF0} \cdot v \\ &+ \texttt{0xA60786EE1470DE341ABA31327A11FE51E3379D8B3EAA3971C129B8ECDB02D23} \cdot u \\ &+ \texttt{0x223B83468B99E673A8E6BC71923114ADF35A9F39B56B6DA2644A6E96EC33CD5A}. \end{aligned}$$

As in Section 5.1.2, we find two candidates for g^{*e_3} among them the correct candidate is found

$$\begin{aligned} g^{*e_3} =& \texttt{0x1A785359B420156B6FC4171A8ECCA2B410A8319631AE28CB548B4E2678911401} \cdot uv^2 \\ &+ \texttt{0x91B2929F3B1EECB799548E497D5BF0E9F3918864AE2840DDFB029209EBBCF88} \cdot v^2 \\ &+ \texttt{0x15C7F6C642BAF378050C190B10C3E576C6353A911974E5E1DA6335A2699F2B4A} \cdot uv \\ &+ \texttt{0x150B1A6B05B9B0674E98B9C0100C4F083552637C3913C63417B7DB086C3FF3CD} \cdot v \\ &+ \texttt{0x15D2F3E145D661BA3522ADC4B3E9AE84D9B05B12DD83F47D635FB8FD48149FAE} \cdot u \\ &+ \texttt{0x197E4A6C69345FBDE33B81A91C3A1127446C53123E4BEFB3E3EB35241468831F}. \end{aligned}$$

And g^{*e_3} and K give h^{*e_3},

$$\begin{aligned}
h^{*e_3} =& \texttt{0x1D12F0CDC8811ECBA27D65F28AD95392F92A4F129C5028239FB1F059E536B6C6} \cdot uv^2 \\
&+ \texttt{0x22BFA9A2CD72D6B2D31F5F9125FC0A63C8A55441C38E8C730E2D378F692EE860} \cdot v^2 \\
&+ \texttt{0xC648A9ADE1263064E793941B1BF56ED1886D778FB937BB07E924D718812861} \cdot uv \\
&+ \texttt{0xC47ECB81ADCCEDAD5E1A210964EA3A3104AAC035314FD4D45D5F8617236F29A} \cdot v \\
&+ \texttt{0xA818D7C9F2BF8FEFD69158F91406CC8450B67CCAF49A4F4283B63E4846AFBC1} \cdot u \\
&+ \texttt{0x1E249C7E24E57B1BDFDCE50869447D22CA4FC58D0EB7ACF34979EE3E58661657}.
\end{aligned}$$

$f_{K,Q}(P)^*$ has been found.

To finish the attack, the same procedure as in Section 4.2 is used. $h_1(P)$ is computed (removing the factor $w^3/2$):

$$\begin{aligned}
h_1(P) =& \frac{f_{K,Q}(P)}{\left(f_{K,Q}(P)^*\right)^2} \\
=& \texttt{0} \cdot uv^2w \\
&+ \texttt{0} \cdot v^2w \\
&+ \texttt{0xA17D142DEECC8668A1C3BBFD12385544D2761AD4FCD0F85DCEF561A7F3297F4} \cdot uvw \\
&+ \texttt{0xAE2480DC8DA9E2154111EE78DF038649D73E44A92D9CED2229D99973D3D039A} \cdot vw \\
&+ \texttt{0} \cdot uw \\
&+ \texttt{0} \cdot w \\
&+ \texttt{0x13ED9D2A9F479B20BEE61C47CEFC680A6B7C1A72687FA1B279671A65D039359F} \cdot uv^2 \\
&+ \texttt{0x207B4F67D5556DF71847B3AA322216D07242A3EFE379FB9393EFDF4A2F9E6644} \cdot v^2 \\
&+ \texttt{0} \cdot uv \\
&+ \texttt{0} \cdot v \\
&+ \texttt{0x1A2EB33142FB757AFFC903D58FB8FD81FFEEDDFED3EEC780DACF371899A15F6E} \cdot u \\
&+ \texttt{0x229EB28B4DE041C0DEAC9E673D2E452E16C334B90E7836CF4FAB01365CDC4DA7}.
\end{aligned}$$

Since the same points than in Section 4.2 have been used, we can see that we have found the same $h_1(P)$ value, and we have already seen that this value leads to the secret point P.

6.3 Conclusion

A first practical fault attack on a complete pairing has been performed, allowing the discovery of the secret point. The use of a double fault is not out of reach with today's equipments and the instruction skip model. It is a demonstration that double faults are not hard to achieve and are a real threat to cryptographic algorithms. Other possibilities are possible to bypass the FE. A scan chain attack has been proposed in [EM09] and has been practically validated in [BGdSG+14]. In this latter paper, independently from our work, Blömer *et al.* make similar observations on the security of pairings with respect to fault attacks. They attack in practice a complete pairing (η pairing) with clock glitches. But depending on the particular implementation, it is not always possible to skip the FE. In this case, our proposal is better adapted.

Chapter 7

Conclusion and Perspectives

In this thesis, we have reviewed the fault attacks on pairings in order to evaluate the security of pairing implementations. We have demonstrated that fault attacks can be a threat to a secure pairing execution if not properly handled. For that purpose, we have chosen an implementation which is, in our mind, the most representative of what a modern pairing algorithm would look like (*i.e.* an Ate pairing on a BN curve with $k = 12$ in a large characteristic field). Yet most of the works detailed in this thesis may be adapted to other cases (other pairing algorithms, other fields, other curves, other coordinate systems...).

In details, we have proved the practical vulnerability of the Miller algorithm with respect to fault attacks. The countermeasures to protect this algorithm have been analysed and we have shown that some of them are inefficient. The theoretical vulnerability of a complex final exponentiation algorithm has been exposed (with 2 independent faults) and a practical demonstration has been proposed. By combining several fault attacks we have proved the vulnerability of the complete pairing algorithm both theoretically and practically. As our injection means for our fault attacks, we had to master an EM bench and we have, for the first time, demonstrated the feasibility of double faults attacks with it. The first fault injection forced us to modify the parameters for the second fault injection. A corollary (supported by [TK10, BGdSG+14]) is that double faults are a reality and should be taken into account for other cryptographic algorithms as well. Some PBC systems are already deployed (*e.g.* Voltage Security). Even if their parameter choices are different from ours (*e.g.* Voltage uses supersingular curves with large characteristic fields and $k = 2$), their implementations should now be examined under the light of the newly exposed vulnerabilities.

For our demonstration, we have performed EM fault injections on a modern microcontroller. We have shown that with these settings a high level fault model, the instruction skip, is relevant and achievable in practice. Yet, as all experimental fault attacks, our set-up is implementation dependent and should be modified for a different target. EM injection benches have the advantages to be low cost and fast to set-up (almost plug-and-play, no preparation of the board or the chip is required). Our EM bench is state-of-the-art and a lot of work has been invested in mastering it as the effects of the interaction of an EM pulse with a microcontroller are mostly terra incognita.

Our proposed fault attacks are a first approach to the problem and are not perfect. Our fault attack proposition in Section 5.1 against the FE reminds us that even if an algorithm is complex and is believed to be hard to invert, there are always ways to do it. We think in particular of the study of fault attacks against hash functions. It may be possible someday to devise tools able to automatically exploit the structure of an algorithm to propose fault attacks against it. We explored this idea which, even if not totally successful, led us to the fault attack in Section 5.2.

Other families of physical attacks should also be studied for PBC. Some effort has already been accomplished in this direction for SCA [WS06] but we believe that a lot of work has still to be done (notably the evaluation of countermeasures to protect the Miller algorithm against SCA). We expect that some novel fault attack schemes, more efficient, will be proposed in the future. Among them, combined attacks (both SCA and FA at the same time) should be particularly devastating. For example, measuring the EM radiation during the FE and injecting faults to invert the MA may yield interesting results.

Synopsis en français
French synopsis

Contents

Note: As required by the graduate school, an synopsis in French is provided below. The whole content is included and developed in the other chapters in English.

1 Introduction à la cryptographie

La cryptologie est la science des secrets. Elle se compose de la cryptographie, l'art d'écrire des secret, et de la cryptanalyse, qui tente de percer à jour le secret. La cryptographie est donc utilisée pour protéger des données, à la fois au repos ou lors d'une communication. Cette protection se fait avec au moins une des fonctionnalités suivantes.

- Confidentialité : l'accès aux données est limité aux destinataires voulus.
- Authenticité : s'assure de l'identité des entités qui communiquent.
- Intégrité : empêche les données d'être modifiées en dehors des entités autorisées.

Plus récemment, l'explosion des objets communicants ont fait apparaître de nouveaux besoins.

- Protection de la vie privée : l'identité des participants doit être protégée s'ils le désirent.
- Ergonomie : l'architecture du système doit permettre la sécurisation des données sans surcout pour l'utilisateur (ou il préférera s'en passer).
- Passage à l'échelle : les schémas cryptographiques doivent permettre de gérer un nombre très élevé d'entités (et en augmentation rapide).

La cryptanalyse au contraire essaie de contrer ces fonctionnalités et d'empêcher la protection des données. Cela peux se faire mathématiquement, en montrant la faiblesse d'un algorithme cryptographique ou du problème mathématique sur lequel il se fonde. Mais cela peux aussi se faire grâce aux attaques physiques qui exploitent les faiblesses de l'algorithme lors de son exécution.

1.1 Cryptographie symétrique et asymétrique

Pour que deux entités partagent des données de manière sûre, il leur est possible de se mettre d'accord sur un secret commun (appelé la clé), caché au reste du monde. Ce schéma est utilisé par la cryptographie dite symétrique. Les algorithmes symétriques offrent un bon compromis en terme de sécurité offerte et de temps de calcul. Des exemples d'algorithmes cryptographiques symétriques sont l'AES, le DES, Twofish, Serpent.

Le partage sécurisé d'une clé commune n'est pas toujours possible si les deux entités ne disposent pas d'un moyen sûr de le faire. Pour répondre à cette problématique, Diffie et Hellman ont proposé en 1976 [DH76] la cryptographie asymétrique (appelée aussi cryptographie à clé publique). Dans un algorithme à clé publique, chaque entité crée une paire de clé (une publique et une privée) et diffuse la clé privée à toute entité souhaitant converser avec elle. La clé publique est utilisée pour chiffrer mais seul le détenteur de la clé privée correspondante peut déchiffrer le message. Le premier algorithme asymétrique moderne est RSA, proposé en 1978 [RSA78], et encore utilisé aujourd'hui. Depuis, la cryptographie basée sur les courbes elliptiques (ECC) est considérée comme plus sûre et plus efficace. Enfin les couplages (ou pairings en anglais), s'appuyant sur les courbes elliptiques, permettent de nouveaux schémas comme le chiffrement basé sur l'identité (IBE) où la clé publique peut être une chaine de caractères arbitraire.

1.2 Introduction à la cryptanalyse

La cryptanalyse est le plus souvent utilisée pour récupérer la clé secrète lors d'une communication sécurisée. Les cryptanalyses possibles d'un algorithme peuvent se distinguer en fonction des capacités de l'adversaire. Soit F l'algorithme cryptographique (connu de l'adversaire), k la clé secrète, P le texte clair et $C = F(P, k)$ le texte chiffré. Une attaque peut alors être

- à textes chiffrés seulement : l'attaquant connait les C,
- à textes clairs connus : l'attaquant connait des couples (P, C),
- à textes clairs choisis : l'attaquant connait les textes chiffrés correspondant à des textes clairs choisis,
- à textes chiffrés choisis : l'attaquant connait les textes clairs correspondant à des textes chiffrés choisis,
- à clés apparentées : l'attaquant connait des couples (P, C) (choisis ou non) pour deux clés différentes liées par une relation connue.

Cryptanalyse classique

Des attaques sont toujours théoriquement possibles sur un algorithme, ne serait-ce que par force brute (l'attaquant connait un couple (P, C) et essaie tous les k jusqu'à trouver $C = F(P, k)$). Mais une attaque n'est possible en pratique que si le coût en calcul peut être atteint par l'attaquant. Les méthodes de cryptanalyse les plus courantes sont

- l'attaque par recherche exhaustive (ou par force brute) : l'attaquant essaie toutes les clés possibles jusqu'à trouver la bonne,
- les attaques statistiques : il est possible de retrouver des motifs statistiques dans les textes chiffrés qui permettent de remonter jusqu'aux textes clairs,
- la cryptanalyse différentielle : l'analyse de $C_2 - C_1 = F(P_2) - F(P_1)$ quand $P_2 - P_1$ est connue permet parfois de remonter au secret, cette technique est très utilisée dans le cadre des attaques en faute.

Pour s'assurer de la sécurité d'un algorithme, les cryptographes montrent l'équivalence entre casser l'algorithme et résoudre un problème mathématique considéré difficile. Des exemples de tels problèmes sont la factorisation, le problème du logarithme discret (DLP), le problème du logarithme discret sur courbes elliptiques (ECDLP)...

Attaques physiques

Même quand un algorithme est considéré sûr cryptographiquement, l'interaction de l'unité de calcul qui l'implémente avec son environnement peut permettre à l'attaquant d'avoir accès à des données intermédiaires du calcul ce qui contrevient au modèle de boite noire nécessaire à la sécurité de l'algorithme. Il s'agit du domaine des attaques physiques. On les sous-divise habituellement en plusieurs familles

- les attaques par canaux cachés (attaques non-invasives) : le circuit laisse fuir de l'information (analyse temporelle, consommation de courant, émission EM...) observée par l'attaquant,

- les attaques par injection de fautes (attaques semi-invasives) : une faute est crée lors du calcul (glitch d'horloge, pulse laser, pulse EM...) qui modifie le comportement de l'algorithme,
- les attaques invasives : la puce est modifiée de manière permanente (avec une Sonde Ionique Focalisée (Focused Ion Beam) par exemple) dans le but de lire ou d'écrire directement une valeur.

2 Les algorithmes calculant les couplages

Les couplages s'appuient sur de nombreuses notions mathématiques. Une résumé des notions nécessaires pour calculer un couplage est donné dans cette section.

2.1 Rappels d'algèbre

Definition 7.2.1 *Groupe*
Soit $\cdot$ une loi définit sur un ensemble S. On dit que S est un groupe si

- *Stabilité : $\forall a, b \in S$, $a \cdot b \in S$.*
- *Associativité : $\forall a, b, c \in S$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*
- *Élément neutre : $\exists e \in S$ tel que $\forall a \in S$ nous avons $e \cdot a = a \cdot e = a$.*
- *Inverse : $\forall a \in S, \exists b \in S$ tel que $a \cdot b = b \cdot a = e$. b est appelé l'inverse de a et est souvent noté a^{-1}.*

Le groupe $(S, \cdot)$ est dit commutatif (ou abélien) si en plus, $\forall a, b \in S, a \cdot b = b \cdot a$.

Definition 7.2.2 *Anneau*
Soit $(G, +)$ un groupe abélien. On appelle anneau $(G, +, \cdot)$ avec les deux lois $+$ et $\cdot$ s'il satisfait les propriétés suivantes:

- *$(G, +)$ est un groupe abélien.*
- *Stabilité par $\cdot$: $\forall a, b \in G$, $a \cdot b \in G$*
- *Associativité pour $\cdot$: $\forall a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$*
- *Élément neutre pour $\cdot$: $\exists 1 \in G$ tel que $\forall a \in G$ nous avons $1 \cdot a = a \cdot 1 = a$*
- *Distributivité : $\forall a, b, c \in G$, $a \cdot (b + c) = a \cdot b + a \cdot c$ et $(a + b) \cdot c = a \cdot c + b \cdot c$.*

De plus, on dit que l'anneau est commutatif s'il possède la propriété additionnelle : $\forall a, b \in G, a \cdot b = b \cdot a$.

Definition 7.2.3 *Corps*
Soit $(R, +, \cdot)$ un anneau commutatif. On dit que $(R, +, \cdot)$ est un corps s'il a la propriété additionnelle d'inverse multiplicative: $\forall a \in R \backslash \{0\}, \exists a^{-1} \in R$, tel que $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

2.2 Corps finis

Définitions

Definition 7.2.4 $\mathbb{Z}/p\mathbb{Z}$

Soit $\mathbb{Z}/p\mathbb{Z}$ l'ensemble formé par les éléments $\{0, 1, 2, \ldots, p-1\}$ (p premier) et doté des lois d'addition et multiplication modulaires ($\bmod p$). Alors $\mathbb{Z}/p\mathbb{Z}$ est un corps fini à exactement p éléments. Ce corps est souvent noté $\mathbb{F}_p$.

Definition 7.2.5 *Corps d'extension sur $\mathbb{F}_p$*

Un corps finis de cardinal q est noté $\mathbb{F}_q$. Nécessairement, pour que $\mathbb{F}_q$ soit un corps, il est nécessaire que q soit une puissance d'un nombre premier p. En effet, soit $f(x) \in \mathbb{F}_p[x]$ un polynomial irréductible de degré d avec des coefficients dans $\mathbb{F}_p$. Alors $\mathbb{F}_p/f(x)$ forme un corps fini à p^d éléments avec les lois d'addition et de multiplication $\bmod f(x)$. Le nouveau corps est un corps d'extension de $\mathbb{F}_p$ et un élément de cette extension peut être représenté à l'aide d'un vecteur à d coordonnées dans $\mathbb{F}_p$. Cette extension peut être notée $\mathbb{F}_{p^d}$ avec $f(x)$ implicite.

Calculs dans les corps finis

Puisque que nous nous intéressons aux attaques en faute, il faut regarder le détail de comment sont effectués les calculs, notamment l'implémentation des multiplications dans $\mathbb{F}_p$. En particulier sur les microcontrôleurs qui nous intéressent, les données doivent être manipulées par mots de 32 bits. On nomme B la base dans laquelle les données sont utilisables ($B = 2^{32}$ dans notre cas). Dans ce cas, le nombre de mots pour mémoriser une valeur dans $\mathbb{F}_p$ est $M = \lceil \log_2(p)/\log_2(B-1) \rceil$. Par exemple, pour mémoriser une valeur de 256 bits, nous avons besoin de 9 mots. Donc chaque élément $X \in \mathbb{F}_p$ pourra être représenté comme

$$X = \sum_{i=0}^{n-1} B^i x_i.$$

Dans la plupart des algorithmes, les données sont manipulées dans une représentation vectorielle où $(x_0, x_1, \ldots, x_{n-1})$ représente X. La coordonnée i de X peut être notée $X[i]$ ou x_i.

Les opérations élémentaires (addition et multiplication) dans les corps finis $\mathbb{F}_p$ constituent l'essentiel du temps de calcul d'un couplage. C'est pourquoi l'optimisation de ces opérations est de la première importance. En particulier, de nombreuses techniques ont été imaginées pour réaliser une multiplication modulaire. Une telle multiplication se fait en deux étapes, la multiplication binaire et la réduction. Ces deux étapes peuvent être successives ou bien entrelacées. Le multiplication binaire est réalisée avec l'un des algorithmes suivants.

- Méthode scolaire : elle consiste en des additions successives de produits partiels, souvent à l'aide d'algorithmes "double-and-add".
- La multiplication de Booth [Boo51] : il s'agit d'une optimisation de la technique de "double-and-add" où une addition est réalisée par groupe de '1' plutôt qu'une par '1'.
- La multiplication de Karatsuba: plutôt que de multiplier deux grands nombres de taille n, cette méthode permet de réaliser la multiplication à l'aide 3 multiplications de nombres de tailles $n/2$ accélérant ainsi le calcul.

Les premières techniques utilisaient ensuite une réduction tirée des méthodes suivantes:

- Méthode de Brickwell [Bri82].

- Méthode de Sedlak [Sed88].

Mais les techniques plus modernes préfèrent entrelacer les phases de multiplication de de réduction. On peut citer les méthodes de Barret [Bar87], de Blakely [Bla83] ou de Montgomery [Mon85]. Comme c'est cette dernière multiplication qui est utilisée, nous la détaillons ici.

La technique de Montgomery est basée sur l'observation suivante: soit un entier R tel que $pgcd(R, N) = 1$ et $N' = -\frac{1}{N} \mod R$ alors la congruence suivante est vérifiée:

$$UR^{-1} \mod N \equiv \frac{U + (UN' \mod R)N}{R}.$$

Ainsi, dans le domaine de Montgomery, avec R le résidu, la réduction se ramène à ajouter un multiple de N. La multiplication peut être entrelacée avec la réduction dans le domaine de Montgomery. Plusieurs algorithmes sont possibles et ont été analysé dans [KKAKJ96], la méthode CIOS a été décrétée la meilleure. C'est cette technique (cf. Algorithm 2) que nous utiliseront dans notre implémentation.

Calculs dans les corps d'extension

Pour le calcul des pairings, on se limite souvent aux extensions quadratiques et cubiques. C'est à dire aux extensions de la forme $\mathbb{F}_{p^n}$ avec $n = 2^i 3^j$. Ainsi tous les calculs dans les extensions sont des combinaisons des calculs dans les extensions quadratiques et cubiques aux formules simples (cf. Section 2.3).

2.3 Courbes elliptiques

Définitions

Definition 7.2.6 *Courbe elliptique*

Une courbe elliptique sur un corps K, de caractéristique différente de 2 et 3, est une courbe projective algébrique lisse de genre 1. Elle est composée des points satisfaisant l'équation de Weierstrass

$$E : y^2 = x^3 + ax + b, \tag{1}$$

auxquels on ajoute le point à l'infini 0_∞, neutre de la loi d'addition.

Le discriminant de cette courbe est

$$\Delta_E = -16(4a^3 + 27b^2), \tag{2}$$

qui doit être différent de 0. Lorsque K est fini (de cardinal p), le cardinal de $E(K)$ est fini et peut être encadré par le théorème de Hasse

$$|\text{card}\,(E(K)) - (p + 1)| \leq 2\sqrt{p}. \tag{3}$$

Opérations sur les courbes elliptiques

Les points sur la courbe elliptique, avec 0_∞, forment un groupe abélien. Plusieurs possibilités existent pour représenter un point sur une courbe elliptique. Ce sont les systèmes de coordonnées. Les plus utilisés sont les suivants.

- Coordonnées affines: (x, y) est un point sur la courbe s'il satisfait $y^2 = x^3 + ax + b$.

- Coordonnées projectives: $(x : y : z)$ peut être envoyé sur le point affine $(\frac{x}{z}, \frac{y}{z})$ si $z \neq 0$. Les points avec $z = 0$ représentent le point à l'infini 0_∞.
- Coordonnées jacobiennes: $(x : y : z)$ peut être envoyé sur le point affine $\left(\frac{x}{z^2}, \frac{y}{z^3}\right)$ si $z \neq 0$. Les points avec $z = 0$ représentent le point à l'infini 0_∞.

Les coordonnées projectives et jacobiennes rajoutent de la redondance dans la représentation d'un point mais permettent de représenter le point à l'infini 0_∞ comme un point comme les autres. Il est possible d'utiliser des opérations mixtes (*e.g.* un point en coordonnées affines et un point en coordonnées jacobiennes) en prenant $Z = 1$ dans un opération en coordonnées jacobiennes.

Les formules pour les opérations sur les courbes elliptiques sont détaillées dans la Section 2.4.2.

r-torsion

Soit $E(\mathbb{F}_q)$ une courbe elliptique, soit r un entier tel que $pgcd(r, q) = 1$ et $r|\text{card}\,(E(\mathbb{F}_q))$. Les points d'ordre un diviseur de r forment un groupe, $\{P|[r]P = 0_\infty\}$, noté $E(\mathbb{F}_q)[r]$. La r-torsion de E est le groupe $E(\overline{\mathbb{F}_q})[r]$ souvent noté $E[r]$. Le plus petit k tel que $E[r] \subset E(\mathbb{F}_{q^k})$ est appelé le degré de plongement de E par rapport à r. Il s'agit du plus petit entier positif qui satisfait $r|q^k - 1$. Si r est premier et $r \not| q - 1$ alors $E[r] \subset E(\mathbb{F}_{q^k}) \Leftrightarrow r|q^k - 1$ [BK98].

Twists de courbes elliptiques

Definition 7.2.7 *j-invariant*
Soit $E(\mathbb{F}_q) : y^2 = x^3 + ax + b$ une courbe elliptique. On note le j-invariant de E

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}. \tag{4}$$

Proposition 7.2.8 *Soit $E_1(\mathbb{F}_q)$ et $E_2(\mathbb{F}_q)$ deux courbes elliptiques, si $j(E_1) = j(E_2)$ alors il existe un isomorphisme entre $E_1(\overline{\mathbb{F}_q})$ et $E_2(\overline{\mathbb{F}_q})$.*

Cet isomorphisme ne tient pas forcement pour toutes les extensions $\mathbb{F}_{q^n}$. S'il existe un isomorphisme $\phi_2 : E_1(\mathbb{F}_q) \to E_2(\mathbb{F}_{q^2})$, alors on dit que E_1 est un twist de degré 2 de E_2. Il existe un twist de degré d selon la règle:

- $d = 2$ si $j(E) \notin \{0, 1728\}$,
- $d = 4$ si $j(E) = 1728$,
- $d = 6$ si $j(E) = 0$.

2.4 Couplages

Nous pouvons maintenant nous attaquer à la description des couplages. Cette description est principalement inspirée de [Mil04, Gal05, Eng13, Riv10].

Diviseurs

Dans cette section, $E(\mathbb{F}_q) : y^2 = x^3 + ax + b$ est une courbe elliptique définie par le polynôme $p_E(x, y) = x^3 + ax + b - y^2$ (*i.e.* $(x, y) \in E(\mathbb{F}_q) \Leftrightarrow p_E(x, y) = 0$).

Definition 7.2.9 *Corps des fractions rationnelles $K(E)$*

Le corps des fractions rationnelles $K(E)$ contient les fractions rationnelles $E(K) \to K$. Une fraction rationnelle (dans notre cas) est une fonction $f : E(K) \to K$ qui peut être écrite comme $f = \frac{f_1}{f_2}$ avec $f_1, f_2 \in K[x, y]$.

$f, g \in E(K)$ sont équivalentes ssi $f_1 g_2 - g_1 f_2 = h \cdot p_E$ avec $h \in K[x, y]$. En effet, $\forall P \in E(\mathbb{F}_q), f_1(P)g_2(P) - g_1(P)f_2(P) = h(P)p_E(P) = 0$ ce qui implique $f(P) = g(P)$.

Definition 7.2.10 *Uniformisante et ordre d'une fraction rationnelle*

L'uniformisante d'une courbe elliptique E au point $P \in E(K)$ est un générateur de l'idéal $\{f \in K(E) | f(P) = 0\}$. L'uniformisante est unique à une constante prêt dans $\overline{K}^*$.

Soit $f \neq 0 \in K(E)$ une fraction rationnelle non nulle, l'ordre de f au point P est l'unique entier n tel que $f = gu^n$ où u est l'uniformisante de E en P et $g(P) \in K^*$. Cet entier n est noté $ord_P(f)$. Si $ord_P(f) > 0$, alors $f(P) = g(P)u(P)^n = 0$ puisque $u(P) = 0$. f possède un zéro en P. Si $ord_P(f) \geq 0$ f est dite régulière, ou définie, en P. Si $ord_P(f) < 0$, f possède un pôle en P ($f(P)$ est indéfinie ou divisée par 0).

Definition 7.2.11 *Diviseur*

Un diviseur D sur E est la somme formelle

$$D = \sum_{P \in E} n_P(P), \tag{5}$$

où $n_P \in \mathbb{Z}$. Il y a un nombre fini (= card (E)) de n_P. Le support de D est $supp(D) = \{P | n_P \neq 0\}$. Le degré de D est

$$deg(D) = \sum_{P \in E} n_P. \tag{6}$$

Les diviseurs d'une courbe E forment un groupe $Div(E)$ avec la loi

$$\sum_{P \in E} n_P(P) + \sum_{P \in E} n'_P(P) = \sum_{P \in E} (n_P + n'_P)(P). \tag{7}$$

Les diviseurs de degré 0 forment un sous-groupe de $Div(E)$: $Div^0(E) = \{D \in Div(E) | deg(D) = 0\}$. Soit $f \in \overline{K}(E)^*$ une fraction rationnelle, nous pouvons associé le diviseur $div(f)$ à f de la manière suivante:

$$div(f) = \sum_{P \in E} ord_P(f)(P). \tag{8}$$

$div(fg) = div(f) + div(g)$ et $div(f/g) = div(f) - div(g)$ pour $f, g \in \overline{K}(E)^*$. Si $div(f) = 0$ alors $f \in \overline{K}$ est constante. Par conséquence, $div(f)$ détermine f à une constante près dans $\overline{K}$.

Definition 7.2.12 *Diviseur principal*

Un diviseur D est appelé principal si une fraction $f \in \overline{K}(E)$ existe tel que $D = div(f)$. Nous pouvons créer une relation d'équivalence avec $D_1 \sim D_2 \Leftrightarrow D_1 - D_2$ is principal. Les classes d'équivalence de diviseurs avec cette relation forment un groupe appelé le groupe de Picard.

Proposition 7.2.13 *Soit $f \in \overline{K}(E)^*$ une fraction rationnelle, alors [Sil09]*

$$deg(div(f)) = 0. \tag{9}$$

Proposition 7.2.14 *Soit $E(\mathbb{F}_q)$ une courbe elliptique. Soit $D = \sum_P n_P(P)$ un diviseur de degré 0 sur E. Alors*

$$\exists f \in \overline{\mathbb{F}_q}(E)^* | D = div(f) \Leftrightarrow \sum_{P \in E(\mathbb{F}_q)} [n_P]P = 0_\infty. \tag{10}$$

Definition 7.2.15 *Fraction rationnelle d'un diviseur*

Soit f une fraction rationnelle et $D = \sum_P n_P(P)$ avec $deg(D) = 0$ tel que $supp(div(f)) \cap supp(D) = \varnothing$. Alors nous pouvons définir

$$f(D) = \prod_P f(P)^{n_P}. \tag{11}$$

Si $g = cf$ pour un $c \in \overline{K}^*$ alors pour tous les diviseurs de degré 0, $f(D) = g(D)$. $f(D)$ dépend seulement de D et $div(f)$.

Proposition 7.2.16 *Loi de réciprocité de Weil*

Soit $f, g \in \overline{\mathbb{F}_q}(E)$ deux fractions rationnelles telles que $supp(div(f)) \cap supp(div(g)) = \varnothing$. Alors

$$f(div(g)) = g(div(f)). \tag{12}$$

Définitions

Definition 7.2.17 *Couplage*

Soit G_1 et G_2 deux groupes abéliens et soit G_T un groupe multiplicatif commutatif. Un couplage est une application $e : G_1 \times G_2 \to G_T$ munie des propriétés suivantes:

- *Non-dégénérescence: soit $P \in G_1$ et $Q \in G_2$, si $\forall P \in G_1, e(P, Q) = 1$ alors $P = 0$ et si $\forall Q \in G_2, e(P, Q) = 1$ alors $Q = 0$.*
- *Bilinéarité: soit $P, P_1, P_2 \in G_1$ et $Q, Q_1, Q_2 \in G_2$ alors*

$$e(P, Q_1 + Q_2) = e(P, Q_1)e(P, Q_2)$$
$$e(P_1 + P_2, Q) = e(P_1, Q)e(P_2, Q)$$

  Par conséquent, $\forall a, b \in \mathbb{Z}$, $e([a]P, [b]Q) = e(P, Q)^{ab}$.

- *Calcul efficace: $\forall P \in G_1$ et $\forall Q \in G_2$, le couplage $e(P, Q)$ est calculable efficacement.*

Definition 7.2.18 *Types de couplage*

Selon la relation entre G_1 et G_2, nous pouvons définir des types de couplage.

- *Type 1 (couplage symétrique): il y a un isomorphisme calculable efficacement $\phi_1 : G_1 \to G_2$ **et** $\phi_2 : G_2 \to G_1$ (avec la possibilité que $G_1 = G_2$).*
- *Type 2: il y a un isomorphisme calculable efficacement $\phi_1 : G_1 \to G_2$ **ou exclusif** $\phi_2 : G_2 \to G_1$.*
- *Type 3: il n'y a pas d'isomorphisme calculable efficacement entre G_1 et G_2.*

Les couplages de Type 2 et Type 3 sont dits asymétriques. Le type de couplage influence le coût des calculs et les protocoles qui peuvent être utilisés.

Couplage de Weil

Proposition 7.2.19 *Si $D \neq 0$ est un diviseur de degré 0 sur une courbe elliptique E, alors il y a un point unique P sur E tel que $D \sim (P) - (0_\infty)$.*

Soit $E[r]$ la r-torsion d'une courbe elliptique E et soit k le degré de plongement de $E(\mathbb{F}_q)$ par rapport à r. Soit D_1 et D_2 deux diviseurs de degré 0 sur E avec $supp(D_1) \cap supp(D_2) = \varnothing$, avec $rD_1 \sim 0$ et $rD_2 \sim 0$. Cela signifie que rD_1 et rD_2 sont des diviseurs principaux, *i.e.* $\exists f_1 | div(f_1) = rD_1$ et $\exists f_2 | div(f_2) = rD_2$ (une autre notation est $f_1 = f_{r,P_1}$ avec $D_1 \sim (P_1) - (0_\infty)$). Le couplage de Weil peut être défini (parmi plusieurs définitions équivalentes [Eng13]) comme l'application

$$\begin{aligned} e_W : E[r] \times E[r] &\to \mu_r \subset \mathbb{F}_{q^k} \\ (P_1, P_2) &\mapsto \frac{f_1(D_2)}{f_2(D_1)}, \end{aligned} \tag{13}$$

où $D_1 \sim (P_1) - (0_\infty)$ et $D_2 \sim (P_2) - (0_\infty)$. Premièrement, nous pouvons voir que $e_W(P_1, P_2) \in \mu_r$, selon la Definition 7.2.15 et la Proposition 7.2.16:

$$\frac{f_1(D_2)^r}{f_2(D_1)^r} = \frac{f_1(rD_2)}{f_2(rD_1)} = \frac{f_1(div(f_2))}{f_2(div(f_1))} = \frac{f_1(div(f_2))}{f_1(div(f_2))} = 1. \tag{14}$$

Si au lieu de $D_2 \sim (P_2) - (0_\infty)$, le diviseur $D_2' \sim (P_2) - (0_\infty)$ est utilisé, alors $\exists g \in \mathbb{F}_{q^k}^*(E) | D_2' = D_2 + div(g)$. Dans ce cas,

$$div(f_2') = rD_2' = rD_2 + r \cdot div(g) = div(f_2) + r \cdot div(g) \tag{15}$$

ce qui implique que $f_2' = f_2 g^r$. Finalement par la loi de réciprocité de Weil,

$$\frac{f_1(D_2')}{f_2'(D_1)} = \frac{f_1(D_2) f_1(div(g))}{f_2(D_1) g(D_1)^r} = \frac{f_1(D_2) f_1(div(g))}{f_2(D_1) g(div(f_1))} = \frac{f_1(D_2)}{f_2(D_1)}. \tag{16}$$

La bilinéarité sur l'opérande de gauche peut être montrée de la façon suivante (la bilinéarité à droite peut être montrée de manière similaire). Soit $P_3 = P_1 + P_2$ et soit $g \in \mathbb{F}_{q^k}^*(E)$ la fraction rationnelle telle que $D_3 \sim (P_3) - (0_\infty)$, $D_3 = D_1 + D_2 + div(g)$ (possible grâce à la Proposition 7.2.14). Par conséquent, si $div(f_1) = r(P_1) - r(0_\infty)$ et $div(f_2) = r(P_2) - r(0_\infty)$ alors

$$\begin{aligned} div(f_1 f_2 g^r) &= r(P_1) + r(P_2) - 2r(0_\infty) + r(P_3) - r(0_\infty) - r(P_1) - r(P_2) + 2r(0_\infty) \\ &= r(P_3) - r(0_\infty) \\ &= div(f_3). \end{aligned} \tag{17}$$

Soit $D_Q \sim (Q) - (0_\infty)$ un diviseur et $f_Q \in \mathbb{F}_{q^k} * (E)$ tel que $div(f_Q) = r(Q) - r(0_\infty)$. D_Q, D_1, D_2, D_3 ont tous des supports disjoints. Selon la Definition 7.2.15

$$f_Q(D_3) = f_Q(D_1) f_Q(D_2) f_Q(div(g)). \tag{18}$$

Alors

$$\begin{aligned} e_W(P_3,Q) &= \frac{f_3(D_Q)}{f_Q(D_3)} && (19)\\ &= \frac{f_1(D_Q)f_2(D_Q)g(D_Q)^r}{f_Q(D_1)f_Q(D_2)f_Q(div(g))} && (20)\\ &= \frac{f_1(D_Q)}{f_Q(D_1)}\frac{f_2(D_Q)}{f_Q(D_2)}\frac{g(D_Q)^r}{f_Q(div(g))} && (21)\\ &= \frac{f_1(D_Q)}{f_Q(D_1)}\frac{f_2(D_Q)}{f_Q(D_2)}\frac{g(D_Q)^r}{g(D_Q)^r} && (22)\\ &= e_W(P_1,Q)e_W(P_2,Q). && (23) \end{aligned}$$

Couplage de Tate

Le couplage de Tate est défini avec les même paramètres $E, \mathbb{F}_q, r, k$ que le couplage de Weil. Soit $\mathbb{F}_{q^k}$ l'extension minimale tel que $E[r] \subseteq E(\mathbb{F}_{q^k})$.

Le couplage de Tate e_T est défini comme

$$\begin{aligned} e'_T : E[r] \times E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k}) &\to \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^r \\ (P_1,P_2) &\mapsto f_1(D_2) \end{aligned} \tag{24}$$

avec $D_2 \sim (P_2) - (0_\infty), P_1 \notin supp(D_2)$ et $div(f_1) = r(P_1) - r(0_\infty)$.

La bilinéarité peut être montré de manière similaire que pour le couplage de Weil. Soit $P_1, P_2, P_3 = P_1 + P_2 \in E[r]$ et $Q \in E(\mathbb{F}_{q^k})$, alors $f_3(D_Q) = f_1(D_Q)f_2(D_Q)g(D_Q)^r$ où $D_3 \sim (P_3) - (0_\infty)$,$D_3 = D_1 + D_2 + div(g)$. Mais la valeur $g(D_Q)^r$ est dans la même classe que 1 dans $\mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^r$, donc $e'_T(P_3,Q) = e'_T(P_1,Q)e'_T(P_2,Q)$.

Pour des applications cryptographiques, il n'est pas pratique de manipuler des classes d'équivalence plutôt que des valeurs. Dans ce but, on définit la surjection:

$$\begin{aligned} \pi : \mathbb{F}_{q^k} * /(\mathbb{F}^*_{q^k})^r &\to \mu_r \subset \mathbb{F}_{q^k} \\ x &\mapsto x^{\frac{q^k-1}{r}}. \end{aligned} \tag{25}$$

Maintenant le couplage de Tate réduit définit par

$$\begin{aligned} e_T : E[r] \times E(\mathbb{F}_{q^k})/rE(\mathbb{F}_{q^k}) &\to \mu_r \subset \mathbb{F}_{q^k} \\ (P_1,P_2) &\mapsto \pi(f_1(D_2)) = f_1(D_2)^{\frac{q^k-1}{r}} \end{aligned} \tag{26}$$

envoie des éléments de la même classe d'équivalence vers la même valeur dans μ_r, le groupe des racines r-ième de l'unité ($\mu_r = \{x \in \mathbb{F}_{q^k} | x^r = 1\}$).

Couplage de Ate [HSV06]

Definition 7.2.20 *Courbe supersingulière*

Une courbe $E(\mathbb{F}_q)$ ($\mathbb{F}_q$ de caractéristique p ou $q = p^k$) est appelée supersingulière si

$$card\,(E(\mathbb{F}_q)) \equiv 1 \mod p. \tag{27}$$

Une courbe non supersingulière est appelée ordinaire.

Definition 7.2.21 *Trace de E*

Nous appelons trace de l'endomorphisme de Frobenius de E (ou trace de E) la valeur t telle que

$$t = q + 1 - card\,(E(\mathbb{F}_q))\,. \tag{28}$$

Soit $\pi_p((x,y)) = (x^p, y^p)$ l'endomorphisme de Frobenius sur E. Une courbe E est supersingulière ssi sa trace est un multiple de la caractéristique de $\mathbb{F}_q$.

Les courbes supersingulières sont possibles uniquement avec $k \leq 2$ for les corps à grande caractéristique, $k \leq 4$ pour les corps binaires et $k \leq 6$ pour les corps ternaires [CFA+05]. Lorsqu'un plus grand k est nécessaire, il faut alors se tourner vers les courbes ordinaires.

Le couplage de Ate est une amélioration du couplage de Tate qui a pour but une exécution plus rapide. Le couplage de Ate utilise le fait que l'endomorphisme de Frobenius π_p possède deux valeurs propres 1 et p dans $E\left(\mathbb{F}_{p^k}\right)[r]$. Soit P et Q les deux vecteurs propres respectifs $(\pi_p(P) = P, \pi_p(Q) = [p]Q)$, alors

$$E\left(\mathbb{F}_{p^k}\right)[r] =< P > \times < Q >$$

où $< X >$ est le groupe généré par X. De plus, si $k > 1$ alors $P \in E\left(\mathbb{F}_p\right)[r]$.

Le couplage de Ate e_A est défini comme suit. Soit $P \in G_1 = E(\mathbb{F}_q)[r] \cap \ker(\pi_p - [1])$, $Q \in G_2 = E(\mathbb{F}_q)[r] \cap \ker(\pi_p - [p])$ et $T = t - 1$ alors

$$e_A(Q,P) = f_{T,Q}(P)^{\frac{p^k-1}{r}}. \tag{29}$$

Dans cette équation, $f_{T,Q} \in \mathbb{F}^*_{p^k}(E)$ tel que $div(f_{T,Q}) = T(Q) - ([T]P) - (T-1)(0_\infty)$.

Soit $N = gcd(T^k - 1, p^k - 1)$ et $T^k - 1 = LN$, alors

$$e_T(Q,P)^L = f_{T,Q}(P)^{\frac{c(p^k-1)}{N}}, \tag{30}$$

où $c = \sum_{i=0}^{k-1} T^{k-1-i}p^i \equiv kp^{k-1} \mod r$ si $r \nmid L$. Cette équation permet d'établir un lien entre les couplages de Tate et de Ate, ce qui garantit les propriétés du couplage de Ate.

Couplage Ate-Optimal (OATE) [Ver10]

Le couplage Ate-Optimal (OATE) [Ver10] améliore le couplage de Ate en réduisant le nombre d'itérations dans l'algorithme de Miller utilisé pour calculer $f_{\lambda,Q}(P)$. Dans [Ver10], l'auteur montre qu'il y a une valeur optimale λ (de manière a obtenir un couplage non-dégénérescent) et comment le calculer. Dans le cas des courbes BN (cf. Section 7.2.4.0), le couplage OATE est définit par

$$e_O(Q,P) = \left(f_{\lambda,Q}(P) \cdot M\right)^{\frac{p^k-1}{r}}, \tag{31}$$

où $\lambda = 6x+2$ (x le paramètre de la courbe BN), $M = l_{Q_3,-Q_2}(P) \cdot l_{-Q_2+Q_3,Q_1}(P) \cdot l_{Q_1-Q_2+Q_3,[\lambda]Q}(P)$ et $Q_i = \pi_{p^i}(Q)$. Les $l_{A,B}(C)$ sont les équations de ligne et seront détaillées dans la Section 7.2.5.

Familles de courbes elliptiques pour les couplages

Les courbes elliptiques doivent avoir une certaine structure pour être à la fois calculables et sûres a utiliser pour des couplages. Une caractéristique nécessaire est d'avoir une degré de plongement modéré: un petit degré de plongement affaiblit la sécurité qu'offre la courbe mais accélère les

calcul. Au contraire, un grand degré de plongement améliore la sécurité mais rallonge le temps de calcul. Si E est une courbe aléatoire avec un sous-groupe d'ordre premier r, alors avec grande probabilité, $k \approx r$ alors que l'on veut $k \ll r$. Pour construire une courbe "pairing-friendly", on aimerait spécifier le degré de plongement k désiré, un premier p et un entier r pour pouvoir ensuite trouver une courbe $E(\mathbb{F}_{p^k})$ avec un sous-groupe d'ordre premier r. Certaines courbes possèdent des critères communs permettant de les catégoriser comme appartenant à la même famille de courbes. Si une méthode existe où $p = p(x)$ et $r = r(x)$ définissent une courbe elliptique quand $p(x)$ est premier, alors les courbes crées quand x varie forment une famille.

Le paramètre $\rho = \frac{\log p}{\log r}$ est introduit et compare les tailles du corps de base et celui du sous-groupe sur la courbe elliptique. Une courbe est meilleure si ρ est le plus proche possible de 1.

Nous détaillerons ici que les formules pour les courbes BN, celles que nous avons utiliser dans notre implémentation.

Les courbes BN sont particulièrement efficaces au niveau de sécurité 128-bit avec le degré de plongement $k = 12$. Elles sont efficaces parce que $\rho \approx 1$ et parce que pour $\log(p) \approx 256$, l'ECDLP dans $E(\mathbb{F}_p)$ et le DLP dans $\mathbb{F}_{p^k}$ ont un même niveau de sécurité 128-bit. Une courbe BN est définie par la valeur x, qui permet de calculer $t(x), r(x), p(x)$.

$$\begin{aligned} t(x) &= 6x^2 + 1, \\ r(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1, \\ p(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1. \end{aligned} \tag{32}$$

Les courbes sont définies pour des x où $p(x)$ et $r(x)$ sont des nombres premiers.

2.5 Miller algorithm

L'algorithme de Miller, proposé par Victor Miller dans [Mil86a], est l'algorithme principal pour calculer un couplage. Il utilise une relation de récurrence dans le but de trouver une fraction rationnelle $f_{n,P}$ telle que $div(f_{n,P}) = n(P) - ([n]P) - (n-1)(0_\infty)$.

Definition 7.2.22 *Fonction de ligne*

Soit l_{P_1,P_2} la fraction rationnelle telle que $l_{P_1,P_2}(x,y) = 0$ définisse la ligne passant par P_1 et P_2 ($P_1, P_2 \neq 0_\infty$, si $P_1 = P_2$ la tangente est utilisée). Sur une courbe elliptique cette fonction possède 3 zéros: P_1, P_2 et $-(P_1 + P_2)$. Par conséquent,

$$div(l_{P_1,P_2}) = (P_1) + (P_2) + (-(P_1 + P_2)) - 3(0_\infty). \tag{33}$$

Soit v_P la fraction rationnelle telle que $v_P(x,y) = 0$ définisse la ligne verticale passant par P. Sur une courbe elliptique cette fonction possède 2 zéros: P et $-P$. Par conséquent,

$$div(v_P) = (P) + (-P) - 2(0_\infty). \tag{34}$$

L'algorithme de Miller repose sur la relation suivante, $\forall n, m \in \mathbb{N}$

$$\begin{aligned} div(f_{n+m,P}) &= (n+m)(P) - ([n+m]P) - (n+m-1)(0_\infty), \\ &= (n)(P) - ([n]P) - (n-1)(0_\infty) \\ &+ (m)(P) - ([m]P) - (m-1)(0_\infty) \\ &+ ([n]P) + ([m]P) + (-[n+m]P) - 3(0_\infty) \\ &- ([n+m]P) - (-[n+m]P) + 2(0_\infty). \end{aligned} \tag{35}$$

Ou en utilisant les fonction de lignes et de verticales,

$$div(f_{n+m,P}) = div(f_{n,P}) + div(f_{m,P}) + div(l_{[n]P,[m]P}) - div(v_{[n+m]P}). \tag{36}$$

Cette relation implique une relation entre les fractions rationnelles :

$$f_{n+m,P} = f_{n,P} f_{m,P} \frac{l_{[n]P,[m]P}}{v[n+m]P}. \tag{37}$$

Cette relation de récurrence peut être complétée par le fait que $div(f_{0,P}) = div(f_{1,P}) = 0$ ce qui signifie que les deux fonctions sont constantes et peuvent être choisies égales à 1. A partir de la relation de récurrence, les cas particuliers suivant sont importants :

$$\begin{aligned} f_{i+1,P} &= f_{i,P} \cdot \frac{l_{[i]P,P}}{v_{[i+1]P}} \\ f_{2i,P} &= f_{i,P}^2 \cdot \frac{l_{[i]P,[i]P}}{v_{[2i]P}}. \end{aligned} \tag{38}$$

L'algorithme de Miller (Algorithm 6) permet de calculer $\forall n \in \mathbb{N}$ la valeur $f_{n,P}(Q)$ avec un algorithme inspiré de celui Square-and-Multiply (cf. Code A.11).

Algorithm 6: Algorithme de Miller pour le couplage de Tate

Data: $r = (r_n \dots r_0)_2$, $P \in \mathbb{G}_1$ et $Q \in \mathbb{G}_2$;
Result: $f_{r,P}(Q) \in \mathbb{G}_3$;
$T \leftarrow P$;
$f \leftarrow 1$;
for $i = n-1$ **to** 0 **do**
  $f \longleftarrow f^2 \times \frac{l_{T,T}(Q)}{v_{[2]T}(Q)}$;
  $T \leftarrow [2]T$;
  if $r_i = 1$ **then**
    $f \longleftarrow f \times \frac{l_{T,P}(Q)}{v_{T+P}(Q)}$;
    $T \leftarrow T + P$;
  end
end
return f

2.6 Exponentiation finale pour les couplages "Tate-like"

L'Exponentiation Finale (FE) pour un couplage "Tate-like" est l'exponentiation par le facteur $\frac{p^k-1}{r}$ dans le but d'envoyer les éléments dans $\mu_r \subset \mathbb{F}_{p^k}^*$. Il s'agit d'un grand exposant, lors d'un calcul dans $\mathbb{F}_{p^k}$, et par conséquent le calcul prends beaucoup de temps. Une méthode pour calculer efficacement cette exponentiation finale quand k est pair est proposée dans [SBC+09]. Nous écrivons $k = 2d$ et observons que

$$\frac{p^k - 1}{r} = \left(p^d - 1\right) \cdot \frac{p^d + 1}{\Phi_k(p)} \cdot \frac{\Phi_k(p)}{r}, \tag{39}$$

où $\Phi_k(X)$ est le $k^{ième}$ polynôme cyclotomique. Le but est d'exprimer l'exposant en utilisant au maximum les endomorphismes de frobenius.

Definition 7.2.23 *Endomorphisme de Frobenius*

L'endomorphisme de Frobenius est défini par l'application $x \in \mathbb{F}_q \mapsto x^{char(\mathbb{F}_q)}$. *Soit* $a, b \in \mathbb{F}_{p^k}$ *deux éléments dans un corps de caractéristique* p. *Alors*

$$(a+b)^p = a^p + b^p. \tag{40}$$

De manière plus générale, pour $n \in \mathbb{N}$,

$$(a+b)^{p^n} = ((a+b)^p)^{p^{n-1}} = (a^p + b^p)^{p^{n-1}} = \ldots = a^{p^n} + b^{p^n}. \tag{41}$$

Puisque tout élément $a \in \mathbb{F}_{p^k}$ *peut être écrit* $a = \sum_{i=0}^{k-1} a_i \cdot \omega^i$, *où* $a_i \in \mathbb{F}_p$ *et* $(1, \omega, \omega^2, \ldots, \omega^{k-1})$ *forme une base de* $\mathbb{F}_{p^k}$ *comme* k*-espace vectoriel sur* $\mathbb{F}_p$. *On peut voir que*

$$a^{p^n} = \sum_{i=0}^{k-1} a_i^{p^n} \cdot \omega^{i \cdot p^n} = \sum_{i=0}^{k-1} a_i \cdot \omega^{i \cdot p^n}. \tag{42}$$

Definition 7.2.24 *Polynôme cyclotomique*

Le $k^{ième}$ *polynôme cyclotomique* $\Phi_k(X)$ *est le polynôme minimal unitaire dont les racines sont également les* $k^{ième}$ *racines de l'unité. Par conséquent,* $\Phi_k(X) | X^k - 1$. *Les premiers polynômes cyclotomiques sont énoncés dans la Table 2.7.*

L'exponentation final peut être décomposée en trois exponentiations, deux faciles ($p^d - 1$ et $\frac{p^d+1}{\Phi_k(p)}$) puisqu'elles utilisent efficacement les endomorphismes de Frobenius et une difficile ($\frac{\Phi_k(p)}{r}$). De plus, un élément $a' = a^{p^d-1}$ est appelé unitaire [SB04] ce qui rend les inversions suivantes "gratuites" (équivalentes à une conjugaison).

L'exponentiation difficile est réalisée en exprimant l'exposant dans la base p : $h = \sum_{i=0}^{n-1} a_i p^i$. Ainsi, calculer f^h revient à calculer $f^{a_{n-1}p^{n-1}} \cdot \ldots \cdot f^{a_1 p} \cdot f^{a_0}$ ou de manière équivalente

$$f^h = \left(f^{p^{n-1}}\right)^{a_{n-1}} \cdot \ldots \cdot (f^p)^{a_1} \cdot f^{a_0}. \tag{43}$$

Les différents f^{p^i} sont d'abord calculés puis une multi-exponentiation est utilisée pour obtenir f^h. L'algorithme complet pour les courbes BN ($k = 12$) est représenté graphiquement sur la Figure 1, et le code peut être trouvé en appendice (Code A.13).

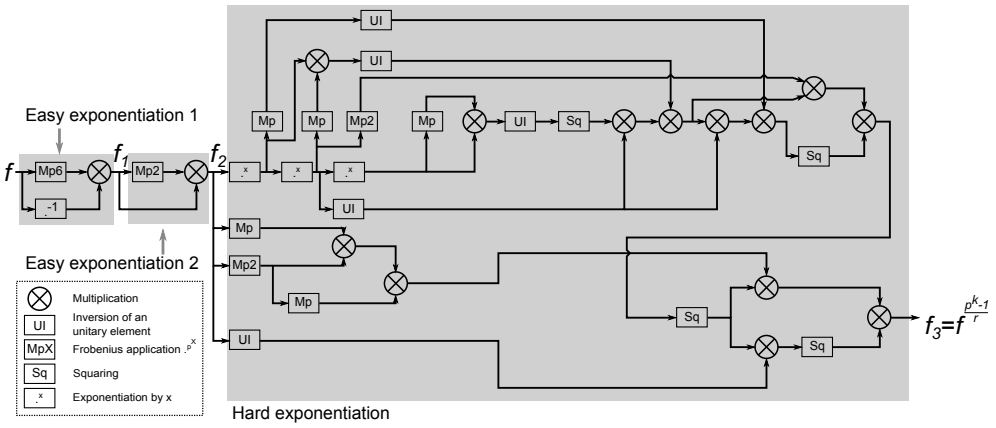

Figure 1: Algorithme pour l'exponentiation finale dans $\mathbb{F}_{p^{12}}$.

2.7 Protocoles pour la PBC

Un des principaux intérêts des couplages réside dans les nouveaux protocoles qu'ils autorisent. Nous présentons quelques uns de ces protocoles dans cette section.

Échange de clé tripartite en un tour

L'échange de clé tripartite en un tour tel que proposé par Joux [Jou00] fut la première utilisation constructive des couplages. Le but est que trois participant s'accordent sur une même clé secrète en un seul tour de communication.

En utilisant un couplage de Type 1, les utilisateurs s'accordent sur les paramètres publics e, G_1, G_2 où G_1, G_2 sont les deux groupes tels que $e : G_1 \times G_1 \to G_2$ est un couplage cryptographiquement sûr. Ils s'accordent également sur un générateur P tel que $< P >= G_1$.

Nommons les trois utilisateurs Alice, Bob et Charlie. Ils choisissent tous une clé secrète (respectivement $a, b, c \mod char(G_1)$) et ils publient leur clés publiques (respectivement $[a]P, [b]P, [c]P$) lors d'un seul tour de communication. Ils peuvent maintenant s'accorder une clé secrète commune $e(P, P)^{abc}$ de la manière suivante:

- Alice calcule $e([b]P, [c]P)^a = e(P, P)^{abc}$.
- Bob calcule $e([a]P, [c]P)^b = e(P, P)^{abc}$.
- Charlie calcule $e([a]P, [b]P)^c = e(P, P)^{abc}$.

Identity-Based Encryption (IBE) [BF01]

Un schéma d'IBE permet de simplifier un des plus grands problèmes de la cryptographie à clés publiques, la distribution des clés. Une PKI utilisant un schéma d'IBE est moins complexe et est plus facile à mettre à l'échelle lorsque le nombre d'utilisateurs grandit.

Dans un schéma d'IBE, la clé publique *est* l'identité d'une entité. Par conséquent, la clé privée associée ne peut pas être calculée par cette entité elle même mais doit être générée par le PKG. Le déchiffrement n'est possible que si cette clé privée est connue.

Une version simplifiée de l'IBE de Boneh-Franklin [BF01] est décrite en quatre algorithmes: Setup, Extract, Encrypt, Decrypt. Ce protocole est décrit ici avec un pairing de Type 1.

Setup: Le PKG génère les paramètres communs pour le calcul de couplage. Il choisit e, G_1, G_2 deux groupes d'ordre r tels que $e : G_1 \times G_1 \to G_2$ est un couplage. Il choisit $P \in G_1$, un générateur quelconque de G_1. Il choisit deux fonctions de hachage, $H_1 : \{0,1\}^* \to G_1^*$ et $H_2 : G_2 \to \{0,1\}^n$. Le PKG tire au hasard $s \in \mathbb{Z}_r$, sa clé privée (appelée master key), et il calcul $P_{PUB} = [s]P$ la clé publique globale du système.

Finallement, les paramètres publics sont $\{r, n, G_1, G_2, e, P, P_{pub}, H_1, H_2\}$.

Extract: L'algorithme extract fournit sa clé privée à un utilisateur. Soit l'utilisateur Alice d'identité $ID =$ "Alice" $\in \{0,1\}^*$, le PKG calcule le point identité publique $Q_A = H_1$("Alice") et sa clé privée associée $d_A = [s]Q_A$.

Encrypt: Si l'utilisateur Bob veux envoyer un message $M \in \{0,1\}^n$ à Alice, il utilise l'algorithme Encrypt.

- Il calcule $Q_A = H_1$("Alice").

- Il tire un nonce aléatoire k.
- Il calcule $g_A = e(Q_A, P_{PUB}) \in G_2^*$.
- Finalement il calcule le texte chiffré $C = \{[k]P, M \oplus H_2(g_A^k)\}$ et l'envoie à Alice.

Decrypt: Lorsque Alice veux déchiffrer le texte chiffré $C = \{U, V\}$ avec $U \in G_1, V \in \{0,1\}^n$, elle utilise l'algorithme Decrypt.

- Elle calcule $e(d_A, U) = e([s]Q_A, [k]P) = e(Q_A, P)^{sk} = e(Q_A, [s]P)^k = e(Q_A, P_{PUB})^k = g_A^k$.
- Elle obtient le message $M = V \oplus H_2(g_A^k)$.

Comme elle est la seule utilisatrice avec la connaissance de d_A, elle est la seule capable de déchiffrer le message.

Dans le schéma ci-dessus, toutes les clés privées sont révoquées si le PKG change la master key s (et donc P_{PUB}). Dans ce cas, tous les utilisateurs ont besoin d'une nouvelle clé privée.

2.8 Cryptanalysis of pairing based cryptography

Des progrès récents ont été obtenus concernant la cryptanalyse de certains algorithmes de couplage. Nous ne les décrirons pas en détails mais les résultats seront présentés avec quelques indications sur leurs fonctionnement. Avant cela, une description des problèmes cryptographiques couramment utilisés dans les protocoles pour la PBC est proposée.

Problèmes cryptographiques

Un problème cryptographique est un problème mathématique considéré difficile à résoudre et utilisé à des fins cryptographiques. Les plus importants pour la PBC sont listés ci dessous:

DLP: Le Discrete Logarithm Problem (DLP) affirme que pour $g, g^a \in \mathbb{F}_p$ connus, il est difficile (pas d'algorithme polynomial en la taille de g) de retrouver a.

ECDLP: L'Elliptic Curve Discrete Logarithm Problem (ECDLP) est une variante du DLP mais sur une courbe elliptique. Il affirme que pour $P, [a]P \in E(\mathbb{F}_p)$ connus, il est difficile de retrouver a. L'ECDLP peut être relié au DLP grâce aux couplages. Soit $P, [a]P \in E(\mathbb{F}_{p^k})[r]$ où k est le degré de plongement. Soit $X = e(P, P)$, alors $X^a = e(P, [a]P)$. a peut être retrouvé indépendamment en résolvant le DLP ou l'ECDLP [MOV93, FR94]. Par conséquent, k doit avoir une taille correcte pour équilibrer la sécurité offerte par les différents problème lorsqu'on utilise un couplage.

DDH: Le problème DDH affirme que pour un générateur $g \in \mathbb{F}_p$ et soit des valeurs aléatoires $a, b \in \mathbb{Z}_p$, la valeur g^{ab} n'est pas distinguable d'une valeur aléatoire g^c. Si le DDH est faux, alors l'attaquant est capable de décider si $X = g^c$ ou $X = g^{ab}$.

CDH: Le problème CDH affirme qu'étant donnés $g, g^a, g^b \in \mathbb{F}_p$, il est difficile de calculer g^{ab}.

BDH: Le problème BDH concerne les couplages symétriques. Soit $e : G_1 \times G_1 \to G_2$ un couplage de Type 1, le BDH affirme que connaissant $P, [a]P, [b]P, [c]P \in G_1$, il est difficile de calculer $e(P, P)^{abc}$.

co-BDH: Le problème co-Bilinear Diffie-Hellman est similaire au BDH mais pour les couplages asymétriques. Soit $e : G_1 \times G_2 \to G_T$, le co-BDH affirme que connaissant $P_1, [a]P_1, [b]P_1 \in G_1$ et $P_2, [a]P_2, [c]P_2 \in G_2$, il est difficile de calculer $e(P_1, P_2)^{abc}$.

Cryptanalyse et PBC

La sécurité du DLP pour les couplages La plupart des efforts de cryptanalyse contre la PBC ont été consacrés à résoudre le DLP (puisque k est petit avec les couplages). Soit $q = p^k$ et $n = \lceil \log_2(q) \rceil$. La discussion générale à propos des algorithms pour résoudre le DLP est inspirée de [Ste].

L'algorithme générique pour résoudre le DLP (qui fonctionne aussi pour l'ECDLP) est l'algorithme Pollar-ρ [Pol78] qui a un coût asymptotique en $\sqrt{\pi n/2}$. Pour l'ECDLP, l'algorithme Pollar-ρ est le meilleur connu et implique que la sécurité de l'ECDLP sur le groupe $E(\mathbb{F}_q)[r]$ est approximativement $\lceil \log_2(r)/2 \rceil$, ainsi r doit être au minimum de 256 bits au niveau de sécurité 128-bit.

Pour le DLP, de meilleurs algorithmes existent qui sont sous-exponentiels. L'"index calculus" consiste, en une explication simplifiée, à créer une base de facteurs de $< g > \subseteq \mathbb{F}_q$ de petits éléments irréductibles puis d'exprimer un élément g^a dans $< g >$ comme produit de ces facteur. Calculer le logarithme de g^a peut être fait en calculant le logarithme des facteurs. La méthode générique d'"index calculus" a un coût asymptotique $L_q[1/2, \sqrt{2} + o(1)]$ où $L_q[s, c] = \exp\left(c(\ln q)^s (\ln \ln q) 1 - s\right)$.

Plusieurs algorithmes ont été dérivées de cette méthode. Lorsque p est un grand nombre premier, le "Number Field Sieve" a un coût asymptotique $L_q\left[1/3, \sqrt[3]{64/9}\right]$. Lorsque $p \in \{2, 3\}$, jusqu'à récemment le "Function Field Sieve" de base avait un coût asymptotique $L_q\left[1/3, \sqrt[3]{32/9}\right]$ mais dans [Jou13], Joux a proposé un algorithme en $L_q\left[1/4 + \epsilon, c\right]$. Plus récemment encore, dans [BGJT13], un algorithme quasi polynomial a été proposé pour les corps de caractéristique 2 ou 3. A la lumière de ces dernier développement, les corps binaires et ternaires doivent être maintenant évités.

Temps de calcul versus sécurité A partir des complexités asymptotiques des problèmes cryptographiques, des hypothèses peuvent être établies sur la taille des corps nécessaire pour un niveau de sécurité donné. Avec le développement de nouveaux algorithmes pour résoudre le DLP, il est difficile d'avoir une complexité unique pour toutes les configurations. A la place, pour chaque corps sur lequel on désire résoudre le DLP, l'attaquant empruntera des étapes à tous ces nouveaux algorithmes. Aujourd'hui le meilleur algorithme pour résoudre le DLP, dans des corps d'intérêts, a pour complexité $L(1/4, c)$ où c est une constance avec $c \geq \left(\frac{\omega}{8}\right)^{1/4}$ où ω est la constante de la multiplication matricielle. Il existe un algorithme quasi polynomial (meilleure complexité asymptotique) mais les durées de calculs sont plus longues pour les corps qui nous intéressent.

L'estimation de la complexité à résoudre le DLP a été faite pour les corps ternaires $\mathbb{F}_{3^{6 \cdot 509}}$ dans [AMORH13b], $\mathbb{F}_{3^{6 \cdot 1429}}$ dans [AMORH13a] et pour les corps binaires $\mathbb{F}_{2^{4 \cdot 1223}}$ dans [GKZ14]. Ces corps assurent une sécurité respectivement de 2^{82}, 2^{96} et 2^{59}, valeurs à comparer avec les sécurités qu'ils étaient censés assurer: 2^{128}, 2^{192} et 2^{128} respectivement.

Mais il est connu que les corps binaires et ternaires sont plus rapides grâce a des opérations simplifiées sur les corps $\mathbb{F}_2$ et $\mathbb{F}_3$. Par conséquent, nous devons comparer la durée de calcul des couplages sur ces corps et les comparer avec la sécurité qu'ils fournissent comme montré sur la Table 2.8.

Les corps de petite caractéristique ont un niveau de sécurité plus bas pour un coût de calcul plus élevé, ils doivent être maintenant évités. Malheureusement, certains protocoles nécessitent des courbes supersingulières ce qui impose maintenant soit de réécrire le protocole avec des courbes ordinaires, soit d'utiliser des courbes supersingulières à grande caractéristique (et donc $k = 2$) sur lesquelles les calculs sont couteux.

3 Introduction aux attaques en faute sur la PBC

La sécurité des algorithmes cryptographiques a plusieurs dimensions. Elle peut être mathématique (quel est l'effort computationnel nécessaire à le "casser"). Mais elle dépends aussi des attaques physiques possibles sur l'algorithme. En particulier, dans notre cas, nous nous concentrerons sur les attaques en faute.

3.1 Attaques physiques

Une attaque physique consiste à manipuler physiquement (en observant ou modifiant) le comportement d'une puce pour en extraire l'information sensible.

Techniques pour les attaques physiques

Les attaques physiques se divisent en plusieurs familles, notamment les Side-Channels Analyses (SCAs) et les Fault Attacks (FAs). Les SCAs sont des attaques passives où l'attaquant observe la fuite d'information de la puce. Certains canaux fuitant de l'information sont listés ci dessous.

- Timing [KSWH98, Koc96]: la durée du calcul peut dépendre du secret.
- Consommation de courant [KJJ99]: la consommation du courant dépend des données manipulées à un instant donné.
- Rayonnement EM [AARR03]: le rayonnement EM dépend des données manipulées à un instant donné et est localisé spatialement au niveau des portes logiques manipulant ces mêmes données.
- Son [GST13]: le son crée par les régulateurs de tension est une image basse-fréquence de la consommation de courant.

Les attaques en faute (FAs) sont des attaques semi-invasives où l'attaquant modifie le cours normal de l'algorithme pour faire fuiter de l'information à la puce. Certaines techniques d'injection de fautes sont listées ci-dessous.

- Glitches d'horloge [ADN+10]: un cycle d'horloge particulier est raccourci pour créer des violations de temps de set-up dans les registres.
- Tension et Température [ZDC+12]: en dehors des valeurs nominales pour la tension et la température, des erreurs peuvent apparaître dans la puce.
- Injection de fautes par laser [SA03]: lorsque des portes logiques (ou les registres) sont illuminées par un laser, l'interaction est capable d'altérer les données.
- Injection de fautes EM [DDRT12]: un pulse EM est envoyé sur la puce pour modifier son comportement. Cette technique est détaillée plus tard car c'est celle que nous allons utiliser.

Une troisième famille existe, les attaques invasives, où la puce est modifiée. Mais cette classe d'attaque est encore difficile d'accès pour un attaquant standard.

Modèles de faute

Notre analyse se concentre sur les attaques en faute contre les couplages. Lors d'une attaque en faute, le modèle de faute doit être spécifié car il influe sur les capacités demandées à l'attaquant et l'analyse des fautes qu'il doit effectuer.

Une faute peut avoir pour effet de modifier une donnée dans une mémoire. Il est possible que les bits soit inversés, ou bien que des valeurs soient collées (collée à 1 si un bit est inversé si sa valeur est 0 uniquement par exemple). Un modèle souvent utilisé est la faute mono-bit (une seul bit est modifié) ou mono-mot (un seul mot de données est modifié).

Une faute peut avoir un effet plus haut-niveau, par exemple le saut d'instruction. Lors de l'injection d'une faute sur un microcontrôleur, une instruction n'est pas exécutée, elle a été sautée. Cette faute peut modifier des données (un mot de données) ou modifier le cours du calcul (en ciblant une instruction de "branch" par exemple). Ce modèle de faute est particulièrement utile contre les couplages et est utilisé par la suite.

3.2 Calibration du banc d'injection EM

Nous décrivons notre banc d'injection de fautes EM dans cette section ainsi que la cible utilisée. *La mise en place du banc a été faite en collaboration avec Amine Dehbaoui et Nicolas Moro.*

Device Under Test

La puce ciblée est un STM32F100RB, un microcontrôleur 32-bit en technologie CMOS $130nm$ qui embarque un cœur ARM Cortex-M3 et qui tourne à 56MHz (période d'horloge de $\approx 17.8ns$). Un calcul de couplage de Ate dure $\approx 16s$, $9s$ pour l'algorithme de Miller et $7s$ pour l'exponentiation finale. Par comparaison notre code inclut aussi un couplage de Ate-Optimal (qui utilise les mêmes routines) et qui tourne en $17ms$ sur un seul cœur d'un processeur Intel Core i5 2430M.

Le Cortex-M3 possède un pipeline à trois étages (FETCH, DECODE, EXECUTE). Nous suspectons que l'injection EM altère l'étape de FETCH en modifiant les données sur le bus (cette donnée peut être la valeur d'une variable mais aussi le code d'une instruction). Le résultat peut être soit assimilé à un saut d'instruction soit à une faute mono-mot sur une donnée.

La puce n'a pas été conçue en tant que puce sécurisée mais elle possèdes des capteurs pour mesurer le courant et détecter les glitches d'horloge, ce qui déclenche des interruptions. Ces capteurs sont activés lors de nos expériences et sont capables de détecter les pulses EM s'ils étaient trop forts. . La carte est sous-alimentée à $2.8V$ au lieu de $3.3V$ dans le but de la rendre plus sensible aux pulses EM.

La puce est reliée à un ordinateur à travers un ST-Link (similaire à un JTAG). Cela nous permet d'accéder à des états internes de la puce à des instants définis par des points d'arrêt choisis par l'utilisateur.

Programme cible

Dans le but de faire notre attaque en fautes, nous avons crée notre propre implémentation logicielle d'un calcul de couplage, inspirée de la librairie Miracl [Cer12]. Cela nous a permis un contrôle complet du programme cible pour pouvor observer l'effet d'éventuels changements. Cette librairie permet de calculer des couplages de Tate et de Ate avec les paramètres tirés

de [BGDM+10]. La courbe utilisée est une courbe BN au niveau de sécurité 128-bit. Le logiciel comprends $\approx$ 1800 lignes de code C. La demande en mémoire sur le microcontrôleur est de $7.2kB$ de RAM et de $13.6kB$ de ROM lorsque compilé sans optimisations. La durée de calcul est de $\approx 16s$. Un profilage du même code sur un PC nous donne le ratio des durées de calcul $Mult(\mathbb{F}_p)/Add(\mathbb{F}_p) = 9.7$.

Table 1: Nombre d'appels aux opérations primitives dans $\mathbb{F}_p$.

# d'appels	Additions $\mathbb{F}_p$	Soustractions $\mathbb{F}_p$	Multiplications $\mathbb{F}_p$	Inversions $\mathbb{F}_p$
Couplage de Ate ciblé	70538	41491	23020	1
Couplage de Ate-Optimal	50811	30023	15993	1

Protocole cible

Notre scénario d'attaque suppose qu'un couplage soit utilisé dans un protocole où un des deux points en entrée est le secret ciblé. Il a été affirmé dans [CKM14] qu'un tel protocole n'existe pas pour les couplages asymétriques. Mais dans le version complète de [BF01], le schéma d'IBE le plus connu, les auteurs précisent bien que le protocole fonctionne pour les couplages asymétriques (en remplaçant le problème BDH par le co-BDH dans la preuve de sécurité). Si un attaquant est capable de retrouver le point secret, il devient capable d'usurper l'identité de la cible.

Matériel

Plusieurs équipements ont été utilisés pour l'injection de fautes EM comme montré sur la Figure 2. La puce ciblée envoie un signal de déclenchement au moment désiré de l'injection de faute. Le signal de déclenchement est détecté par le générateur d'impulsion qui envoie une impulsion dans la sonde EM (une antenne à bobine positionnée à la surface de la puce cible), qui à son tour injecte une faute dans la puce. La durée des fronts montant et descendant du générateur d'impulsions est de $2ns$. Une faute est crée lorsque l'onde EM atteint le Power Ground Network de la puce. Des détails sur ces phénomènes physiques peuvent être trouvés dans [Deh11] (en français).

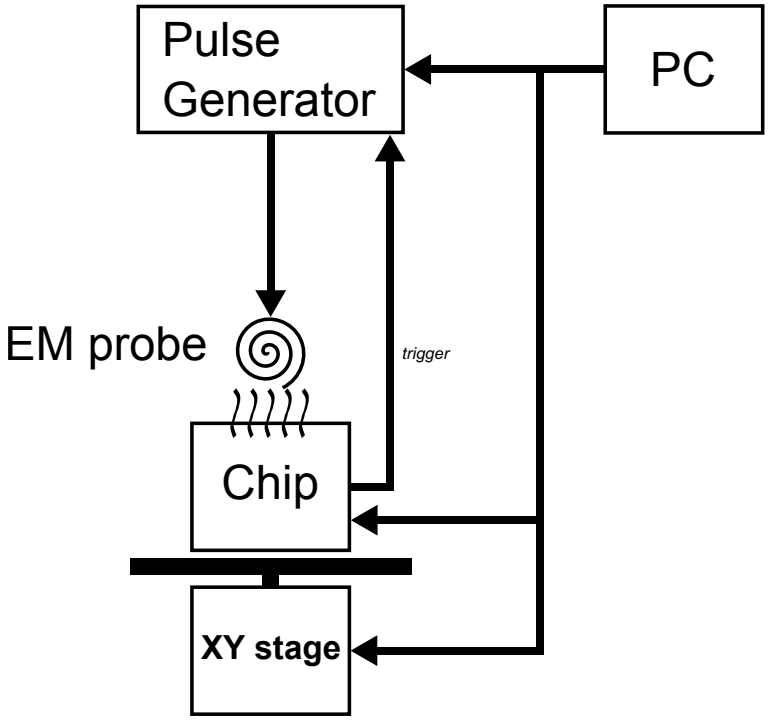

Figure 2: Schéma du banc d'injection defautes EM.

Un ordinateur est utilisé pour contrôler les opérations et les paramètres (notés en lettres majuscules) de l'expérience: les positions X et Y de la sonde EM par rapport à la puce, l'amplitude (AMP) de l'impulsion crée par le générateur d'impulsion, la largeur du pulse (WIDTH) et le délai (DELAY) entre le signal de déclenchement envoyé par la puce et l'injection de la faute.

Le diamètre de la sonde EM est de $1mm$ mais la table XY contrôlant la position relative entre la puce et la sonde possède une résolution de $1\mu m$. L'ordinateur nous permet d'automatiser l'exploration des effets des différents paramètres.

Expériences préliminaires

Le premier but de l'attaque est de trouver un ensemble de paramètres (XY location, AMP, WIDTH, DELAY) qui crée une faute sur la puce. Nous avons observé que faire varier la largeur de l'impulsion WIDTH dans l'intervalle $10ns$ à $100ns$ n'avait aucun effet sur les résultats. Ce paramètre a donc été fixé à $10ns$ au cours de toutes les expériences.

Notre première expérience était de localiser spatialement un point vulnérable de la puce, un emplacement où l'injection EM produit un effet observable qui n'est pas un crash. Un endroit où l'impulsion déclenche une interruption peut souvent être utilisé pour injecter une faute non-détectée, simplement en baissant l'amplitude de l'impulsion.

Dans le but de trouver la bonne configuration, un algorithme maquette (cf. Code 3.1) a été utilisé pour une exécution plus rapide et une interprétation plus aisée de l'effet des fautes.

Les expériences préparatoires nous ont permis de déterminer que pour notre puce, dans le but d'obtenir des fautes exploitables, les paramètres $X = 131200\mu m$, $Y = 191500\mu m$, WIDTH $= 10ns$ étaient les meilleurs, indépendamment du logiciel ciblé. De plus nous avons vu que les paramètres DELAY et AMP doivent être modifiés selon l'effet recherché sur le calcul. Une fois ces expériences préliminaires effectuées, il est possible d'utiliser le code cible.

4 Attaques en faute sur l'algorithme de Miller

Dans cette section, la sécurité de l'algorithme de Miller est évaluée indépendamment de l'exponentiation finale (on considère que le résultat de l'algorithme de Miller est accessible par l'attaquant). L'algorithme de Miller est d'abord étudié théoriquement puis nous testerons certaines attaques en pratique afin de les valider. Nous nous concentrerons sur le modèle de faute de saut d'instruction comme présenté dans la Section 7.3.1 (Modèles de faute).

Tout au long de ce document, une attaque sur l'algorithme de Miller signifie que l'un des deux points en entrée est le secret recherché alors que l'autre point est un paramètre public, connu de l'attaquant. Les deux cas possibles sont

- P est secret, Q est public,
- Q est secret, P est public.

Dans le but d'implémenter les attaques en faute en pratique, nous avons du choisir tous les paramètres nécessaires. Notre choix est un couplage de Ate twisté sur une courbe BN au niveau de sécurité 128-bit avec des paramètres tirés de [BGDM$^+$10]. Dans cette configuration, le point P est représenté avec des coordonnées affines et le point Q avec des coordonnées jacobiennes. Une courbe BN est définie par les valeurs paramétrées $t(x)$, $r(x)$ et $p(x)$ et donc x définit complètement la courbe, avec $r(x)$ et $p(x)$ de grands premiers. Dans notre cas $x =$ `0x3FC0100000000000` et $\mathbb{F}_{p^{12}}$ est construit avec la tour d'extensions:

$$\begin{aligned}\mathbb{F}_{p^2} &= \mathbb{F}_p[u]/(u^2-\beta),\\ \mathbb{F}_{p^6} &= \mathbb{F}_{p^2}[v]/(v^3-u),\\ \mathbb{F}_{p^{12}} &= \mathbb{F}_{p^6}[w]/(w^2-v),\end{aligned}$$

où $\beta = -5$ est un non-résidu quadratique dans $\mathbb{F}_p$, u est un non-résidu cubique dans $\mathbb{F}_{p^2}$, et v est un non-résidu quadratique dans $\mathbb{F}_{p^6}$.

La base pour $\mathbb{F}_{p^{12}}$ comme espace vectoriel sur $\mathbb{F}_{p^2}$ est donc $(1, w, w^2, w^3, w^4, w^5)$ et la base pour $\mathbb{F}_{p^{12}}$ sur $\mathbb{F}_p$ est $(1, w, w^2, w^3, w^4, w^5, w^6, w^7, w^8, w^9, w^{10}, w^{11})$. Ansi pour $R \in \mathbb{F}_{p^{12}}$, on note

$$R = \sum_{i=0}^{11} R_i \cdot w^i, \tag{44}$$

avec $w^2 = v \in \mathbb{F}_{p^6}, w^6 = u \in Fsq$ et $w^{12} = \beta \in \mathbb{F}_p$.

La courbe BN que nous utilisons est définie par l'équation $Y^2 = X^3 + 5$ et la courbe twistée E' est donnée par $Y^2 = X^3 - 5/u = X^3 - u$.

4.1 Attaques théoriques sur l'algorithme de Miller

Des attaques théoriques ont déjà été proposées contre l'algorithme de Miller dans la littérature. Un rappel de ces attaques est donnée ci-dessous ainsi que la méthode pour retrouver $h_1(P) = l_{T,T}(P)$ et $h_2(P) = l_{T,Q}(P)$. Finalement une analyse des contremesures sur cet algorithme est faite.

Attaques sur le flot de données

Une attaque sur le flot de données est une attaque dont les fautes injectées modifie la valeur de certaines données mais pas le cours de l'algorithme.

Attaque en faute de Whelan *et al.* [WS07] Une telle attaque a été proposée par Whelan *et al.* dans [WS07]. Les auteurs proposent une attaque en faute complète sur le couplage η (sur des courbes à petite caractéristique) mais leur proposition sur les courbes ordinaires sur $\mathbb{F}_{p^2}$ est plus pertinent maintenant que les petites caractéristiques ne sont plus sûres. Dans [WS07], les auteurs montrent que par une attaque par changement de signe (le signe d'une de coordonnées de l'évaluation de ligne est changé), ils sont capables d'inverser un couplage de Weil en observant le rapport d'un résultat fauté sur un résultat correct. Ce rapport fournit un système d'équation qui peut être résolu. Pourtant cette méthode ne fonctionne qu'avec un couplage de Weil avec une exponentiation finale simple (d'exposant $p-1$), exponentiation rajoutée pour utiliser l'optimisation par élimination du dénominateur. Ils montrent que leur méthode ne fonctionne pas avec l'exponentiation finale d'exposant $(p^2-1)/r$ car ils n'est pas possible d'inverser cette opération.

Une conclusion tirée par les auteurs est que le couplage de Tate (et ses dérivés) sont immunisés contre les attaques en faute grâce à l'exponentiation finale. Ils pensent que la seule façon de surmonter cette protection est d'utiliser une attaque en faute sur l'exponentiation finale en conjonction avec une attaque en faute sur l'algorithme de Miller, ce qui nécessite des fautes doubles.

Une adaptation de cette attaque pour les couplages sur des corps avec une grande caractéristique et $k = 2$ a été proposée dans [CKM14].

Variante avec le modèle de faute "controlled-add" Dans notre implémentation, les attaques par changement de signe sont impossible car notre représentation des éléments dans les corps finis sont des nombres toujours positifs (le changement de signe revient donc à faire $-x = p - x$). Il est pourtant possible de créer une faute sur les données qui permet d'inverser l'algorithme de Miller. Si la valeur de la faute est inconnue, on aimerais que cette valeur ait une entropie limitée (de là vient le terme "controlled"). Nous proposons une attaque qui permet d'inverser l'algorithme de Miller sur une courbe BN avec $k = 12$.

Dans cette attaque, le point secret est Q et P est connu de l'attaquant. On injecte une faute durant une addition dans $\mathbb{F}_p$. Cette opération nécessite un algorithme d'addition multi-mots sur la puce 32-bit et donc la valeur de la faute est limitée à ≈ 32 bits d'incertitude. Il est possible d'utiliser une faute sur une addition modulaire pour retrouver $h_1(P)$ si l'attaquant connaît le point P. Pour cela, il peut cibler l'évaluation d'une des coordonnées (R_0, R_3 or R_4, cf. équation (44)) de $R = h_1(P)$ durant la dernière itération de l'algorithme de Miller. Par exemple, la valeur R_0 est calculée avec un algorithme finissant avec ce pseudo code C ($t_0 \in \mathbb{F}_{p^2}$):

```
t0 = t0 + t0; //fast modular doubling
R0 = t0 * YP; //P = (XP : YP)
```

L'attaquant peut retrouver $h_1(P)$ en injectant une faute connue e sur l'addition modulaire donnant $t_0^* = t_0 + e$. Cette faute est propagée sur R_0:

$$R_0^* = t_0^* \cdot YP = (t_0 + e) \cdot YP = R_0 + e \cdot YP = R_0 + \Delta_{R_0}. \tag{45}$$

Puisque e et YP sont connues de l'attaquant, Δ_{R_0} l'est aussi. A la dernière itération de l'algorithme, nous avons:

$$f_{K,Q}(P) = f_1^2 \times h_1(P) \tag{46}$$

Si l'attaquant est capable d'injecter une faute connue Δ_{R_0} dans $h_1(P)$, il obtient

$$f_{K,Q}(P)^* = f_1^2 \times (h_1(P) + \Delta_{R_0}). \tag{47}$$

Comme il connaît $f_{K,Q}(P)^*$, $f_{K,Q}(P)$ et Δ_{R_0}, il peut trouver $h_1(P)$:

$$h_1(P) = \frac{f_{K,Q}(P) \times \Delta_{R_0}}{f_{K,Q}(P)^* - f_{K,Q}(P)}. \tag{48}$$

Si P est le secret et Q est connu, il est possible d'obtenir le même résultat avec une faute sur la dernière opération calculant R_3, qui est une soustraction modulaire.

Attaques sur le flot de contrôle

Un autre modèle de faute est réalisé lorsqu'une faute est injectée sur une instruction contrôlant le déroulement du programme (*e.g.* instruction de branch).

Attaque sur l'algorithme de Duursma-Lee [DL03] L'algorithme de Duursma-Lee [DL03] est une variante de l'algorithme de Miller optimisé pour les corps ternaires. Cette attaque a donc pour principal intérêt son principe de fonctionnement plutôt que son application. Cette attaque a été proposée dans [PV06] et fut la première contre un couplage. Les paramètres ci-dessous sont tirés du papier.

L'algorithme de Duursma-Lee (Algorithm 7) sert à calculer un couplage sur des courbes supersingulières sur $\mathbb{F}_q$ avec $q = 3^m$ et $k = 6$. Des courbes d'équation

$$E : y^2 = x^3 - x + b, \tag{49}$$

où $b = \pm 1$. La tour d'extensions utilisée est $\mathbb{F}_{q^3} = \mathbb{F}_q[\rho]/(\rho^3 - \rho - b)$ et $\mathbb{F}_{q^6} = \mathbb{F}_{q^3}[\sigma]/(\sigma^2 + 1)$. $G_1 = E(\mathbb{F}_q)[r]$ (même r que définit dans la Section 2.4.3), $G_2 = \mu_r \subset \mathbb{F}_{q^6}^*$.

Algorithm 7: L'algorithme de Duursma-Lee.

Data: $P = (x_P, y_P) \in \mathbb{G}_1$ et $Q = (x_Q, y_Q) \in \mathbb{G}_2$.
Result: $e(P, Q) \in \mathbb{G}_3$.

$f \leftarrow 1$;
for $i = 1$ **to** m **do**
  $x_P \leftarrow x_P^3$, $y_P \leftarrow y_P^3$;
  $\mu \leftarrow x_P + x_Q + b$;
  $\lambda \leftarrow -y_P y_Q \sigma - \mu^2$;
  $g \leftarrow \lambda - \mu\rho - \rho^2$;
  $f \leftarrow f \cdot g$;
  $x_Q \leftarrow x_Q^{1/3}$, $y_Q \leftarrow y_Q^{1/3}$;
end
return f^{q^3-1};

Dans [PV06], les auteurs proposent une attaque où deux exécutions d'un couplage sont effectuées, l'une fautée et l'une correcte. L'exécution fautée possède une itération additionnelle par rapport à la correcte, et le rapport du résultat fauté sur le résultat correct donne la valeur

$$g_{m+1} = -y_P^{3^{m+1}} \cdot y_Q \cdot \sigma - \mu_{m+1}^2 - \mu_{m+1} \cdot \rho - \rho^2, \tag{50}$$

où $\mu_i = x_P^{3^i} + x_Q^{3^{m-i+1}} + b$. Puisque $\forall z \in \mathbb{F}_{3^m}, z^{3^m} = z$ et en décomposant g_{m+1} dans $\mathbb{F}_q$, il est possible de trouver le secret P connaissant Q. Il est possible d'arriver au même résultat, du moment que l'attaquant est capable d'obtenir les résultats pour deux exécutions avec des nombres consécutifs d'itérations.

La phase suivante nécessite d'inverser l'exponentiation finale d'exposant $q^3 - 1$. Soit S le résultat de l'algorithme de Miller et R le résultat du couplage, *i.e.* nous avons $R = S^{q^3-1}$. R possède plusieurs antécédents par l'exponentiation finale mais S^*/S peut se distinguer par sa forme particulière:

$$\frac{R^*}{R} = \left(\frac{S^*}{S}\right)^{q^3-1} = g_{m+1}^{q^3-1}, \tag{51}$$

où R^* et S^* sont les résultats fautés. Inverser l'exponentiation finale peut être réaliser en deux étapes, premièrement en trouvant n'importe qu'elle racine correcte g de $R = g^{q^3-1}$, puis en dérivant la réponse correcte g_{m+1} depuis g. Il est facile de trouver les racines de $R = X^{q^3-1}$ en remarquant que cette équation est équivalente à

$$X^{q^3} - R \cdot X = 0, \tag{52}$$

qui est une équation linéaire en X. Puis les relations spéciales que g_{m+1} doit satisfaire sont utilisées (détails dans [PV06]) pour trouver le bon antécédent de l'exponentiation finale.

Une adaptation de cette attaque pour des couplages symétriques avec un corps ayant une grande caractéristique et $k = 2$ a été proposée dans [CKM14].

Attaque de El Mrabet [EM09] Dans [EM09], El Mrabet développe le travail de Page *et al.* dans [PV06] pour attaquer l'algorithme de Miller dans une configuration plus générale. L'auteur montre que le secret peut être retrouvé en obtenant deux résultats de l'algorithme de Miller avec des nombres consécutifs d'itérations. Les explications ci-dessous sont données pour le cas particulier de notre implémentation (couplage de Ate twisté, $k = 12$ sur une courbe BN, coordonnées mixtes). Nous comparons le résultat correct avec un résultat fauté où la dernière itération a été sautée.

Dans le cas du couplage de Ate, lors de la dernière itération, il y a seulement évaluation de la tangente:

$$f_{K,Q}(P) = f_1^2 \times h_1(P). \tag{53}$$

Donc si on saute la dernière itération, on obtient:

$$f_{K,Q}(P)^* = f_1. \tag{54}$$

Finalement, h_1 est simplement

$$h_1(P) = \frac{f_{K,Q}(P)}{(f_{K,Q}(P)^*)^2}. \tag{55}$$

Cette méthode est générique et peut être appliquée pour différentes courbes et différents systèmes de coordonnées.

Attaque avec sortie à la première itération Un cas particulier de l'attaque précédente est le cas où l'attaquant est capable d'obtenir un résultat fauté de l'algorithme de Miller où le programme est sorti de la boucle après la première itération. Dans ce cas,

$$f_{K,Q}(P)^* = h_1(P) \times h_2(P) \tag{56}$$

ou

$$f_{K,Q}(P)^* = h_1(P), \tag{57}$$

selon K. Aucune autre valeur n'est nécessaire pour retrouver le secret. De cette façon, une seule exécution de l'algorithme de Miller est nécessaire.

Attaque de Bae *et al.* [BMH13] Dans [BMH13], Bae *et al.* proposent une autre attaque sur le flot de contrôle en sautant l'étape d'addition dans la dernière itération d'un couplage de Tate. Pour cela, ils ciblent l'instruction *if* avec une attaque en faute. Dans le couplage de Tate, la dernière itération est

$$f_{K,Q}(P) = f_1^2 \times h_1(P) \times h_2(P). \tag{58}$$

En sautant l'étape d'addition, nous obtenons:

$$f_{K,Q}(P)^* = f_1^2 \times h_1(P). \tag{59}$$

Finalement, $h_2(P)$ peut être trouvé avec

$$h_2(P) = \frac{f_{K,Q}(P)}{f_{K,Q}(P)^*} \tag{60}$$

Le secret peut finalement être retrouvé avec $h_2(P)$ comme montré ci-dessous.

Comment retrouver le secret

Les équations proposées dans cette section sont dérivées des travaux précédents [PV06, EM09], avec des ajouts pour coller à notre cas particulier.

Retrouver le secret connaissant $h_1(P)$ (couplage de Ate) Maintenant que nous avons vu comment la valeur $h_1(P)$ est retrouvée avec des attaques en faute, nous allons montrer comment retrouver le secret à partir de cette valeur. Nous savons que $Q \in E(\mathbb{F}_{p^{12}})$ et $P \in E(\mathbb{F}_p)$. En pratique, le twist de degré 6 est utilisé pour représenter Q: nous simplifions les équations en écrivant Q comme le point $(x_q : y_q) \in E'(\mathbb{F}_{p^2})$ au lieu de $(x_q \cdot w^2 : y_q \cdot w^3) \in E(\mathbb{F}_{p^{12}})$. De plus, le point Q (et donc T) est représenté en coordonnées jacobiennes $(X_Q : Y_Q : Z_Q)$ ce qui est équivalent aux coordonnées affines $(X_Q/Z_Q^2 : Y_Q/Z_Q^3)$. L'attaquant connaît $h_1(P)$ avec:

$$\begin{aligned} h_1(P) =& (3X_T^3 - 2Y_T^2) \cdot w^6 + 2Y_T Z_T^3 y_p \cdot w^3 \\ & - 3X_T^2 Z_T^2 x_p \cdot w^4, \\ =& R_0 + R_3 \cdot w^3 + R_4 \cdot w^4, \end{aligned} \tag{61}$$

avec $R_0, R_3, R_4 \in \mathbb{F}_{p^2}$ (puisque $w^6 = u \in \mathbb{F}_{p^2}$) obtenus grâce à l'identification des termes et $T = [i]Q$ pour un i connu de l'attaquant. Pour le couplage de Ate R_0, R_3, R_4 donnent un système dans $\mathbb{F}_{p^2}$:

$$\begin{cases} R_0 &= (3X_T^3 - 2Y_T^2) \cdot u, \\ R_3 &= 2Y_T Z_T^3 y_p, \\ R_4 &= -3X_T^2 Z_T^2 x_p. \end{cases} \tag{62}$$

Premièrement, si P est le secret et Q est connu, P peut être obtenu trivialement avec ce système puisque $T = [i]Q$ est connu de l'attaquant et le système est linéaire en les coordonnées de P :

$$\begin{cases} x_p &= \frac{-R_4}{3X_T^2 Z_T^2}, \\ y_p &= \frac{R_3}{2Y_T Z_T^3}. \end{cases} \tag{63}$$

C'est pourquoi à partir de maintenant, nous nous consacrons au cas où Q est le secret et P est connu. La solution est à peine plus compliquée. Le système nous donne le polynôme en Z_T

$$\frac{R_0^2}{\beta} \cdot Z_T^{12} + \left(4\frac{R_0}{u}\lambda_2^2 - 9\lambda_3^3\right) \cdot Z_T^6 + 4\lambda_2^4 = 0 \tag{64}$$

avec $\lambda_2 = \frac{R_3}{2y_p}$ et $\lambda_3 = -\frac{R_4}{3x_p}$. Ce polynôme peut être résolu sur $\mathbb{F}_{p^2}$, donnant des candidats pour Z_T. Une fois que l'on connaît Z_T, on l'utilise dans le système initial pour obtenir X_T et Y_T. Les points candidats qui ne sont pas sur la courbe elliptique sont éliminés. Finalement, les possibilités pour $Q = [i^{-1}]T$ sont calculées.

Retrouver le secret connaissant $h_1(Q)$ (couplage de Tate) Dans cette section, les calculs sont faits pour un couplage de Tate twisté: $T = [i]P$ pour un i connu, $Q \in E'(\mathbb{F}_{p^2})$, $P \in E(\mathbb{F}_p)$. Les point P et Q sont échangés par rapport au couplage de Ate. Il est plus facile de trouver P connaissant $h_1(Q)$ que de trouver Q connaissant $h_1(P)$ car il y a moins d'inconnues dans le premier cas.

L'attaquant peut retrouver $h_1(Q)$ avec

$$h_1(Q) = \left(3X_T^3 - 2Y_T^2\right) - \left(3X_T^2 Z_T^2 x_q\right) \cdot w^2 + \left(2Y_T Z_T^3 y_q\right) \cdot w^3, \tag{65}$$

$$h_1(Q) = R_0 + R_2 w^2 + R_3 w^3, \tag{66}$$

avec $R_0, R_2, R_3 \in \mathbb{F}_{p^2}$ obtenus par identification et $T = [i]P$ pour un i connu de l'attaquant. Nous nous consacrons sur le cas où P est le secret et Q est connu. Le système d'équation suivant est obtenu :

$$\begin{cases} 3X_T^3 & = R_0 + 2Y_T^2 \\ X_T^2 & = -\frac{R_2}{3x_q} Z_T^{-2} = \lambda_2 Z_T^{-2} \\ Y_T & = \frac{R_3}{2y_q} Z_T^{-3} = \lambda_3 Z_T^{-3}. \end{cases} \tag{67}$$

A partir de ce système, nous obtenons l'équation

$$R_0^2 \cdot Z_T^{12} + \left(4R_0\lambda_3^2 - 9\lambda_2^3\right) \cdot Z_T^6 + 4\lambda_3^4 = 0. \tag{68}$$

Cette équation sur $\mathbb{F}_p$ peut être résolue, par exemple avec Sage [S+12]. Une fois Z_T obtenues, on l'utilise dans le système initial pour retrouver X_T et Y_T.

Retrouver le secret connaissant $h_2(Q)$ (couplage de Tate) Dans certains cas (notamment le saut de l'instruction *if* dans [BMH13]), l'attaquant obtient la valeur $h_2(Q)$. A partir de cette donnée, il peut retrouver le secret. Nous savons que

$$h_2(Q) = ((Y_T - Y_P Z_T^3)X_P - Y_P Z_R) + (Y_P Z_T^3 - Y_T)x_q \cdot w^2 + y_q Z_R \cdot w^3. \tag{69}$$

où $R = T + P$ et par conséquent $Z_R = Z_T(X_T - X_P Z_T^2)$. il y a 5 inconnues dans $\mathbb{F}_p$ et $h_2(Q)$ fournit 5 équations sur $\mathbb{F}_p$. Si nécessaire les équations de courbes pour T et P peuvent être ajoutées. Ce système peut être finalement résolu par l'utilisation de bases de Gröbner.

Un cas spécial apparaît quand la valeur $h_2(Q)$ connue est celle de la dernière itération comme dans [BMH13]. Puisque $[r]P = 0_\infty$, $h_2(Q)$ a une forme simplifiée dans la dernière itération

$$h_2(Q) = Z_P^2 x_Q - X_P, \tag{70}$$

où $P \in E(\mathbb{F}_p), Q \in E'(\mathbb{F}_{p^2})$. Si Q est le secret et P est connu, x_Q est obtenu directement et deux solutions sont possibles pour y_Q (avec l'équation de la courbe). Si P est secret et Q est public avec $x_Q = xq_1 \cdot u + xq_0$ connu, en écrivant $h_2(Q) = R_1 \cdot u + R_0$, le système

$$\begin{cases} R_1 = & Z_P^2 xq_1 \\ R_0 = & Z_P^2 xq_0 - X_P \end{cases} \tag{71}$$

est trouvé qui donne simplement Q.

Retrouver le secret connaissant $h_1(Q) \cdot h_2(Q)$ (couplage de Tate) Une autre possibilité est le cas où l'itération de la boucle de Miller évalue à la fois $h_1(Q)$ et $h_2(Q)$, si le bit correspondant dans r est égal à 1. Dans ce cas, l'attaquant obtient

$$h_1(Q) \cdot h_2(Q). \tag{72}$$

Nous savons que

$$h_1(Q) = \left(3X_T^3 - 2Y_T^2\right) - \left(3X_T^2 Z_T^2 x_q\right) \cdot w^2 + \left(2Y_T Z_T^3 y_q\right) \cdot w^3 \tag{73}$$

et

$$h_2(Q) = ((Y_D - Y_P Z_D^3)X_P - Y_P Z_R) + (Y_P Z_D^3 - Y_D)x_q \cdot w^2 + y_q Z_R \cdot w^3, \tag{74}$$

où $D = [2]T$, $R = D + P$ et par conséquent $Z_R = Z_D(X_D - X_P Z_D^2)$. A partir des équations pour doubler un point,

$$\begin{cases} X_D &= 9X_T^4 - 8X_T Y_T^2 \\ Y_D &= 12Y_T^2 X_T^3 - 8Y_T^4 - 3X_T^2(9X_T^4 - 8X_T Y_T^2) \\ Z_D &= 2Y_T Z_T. \end{cases} \tag{75}$$

Nous avons 5 inconnues $X_P, Y_P, X_T, Y_T, Z_T \in \mathbb{F}_p$, et la connaissance du produit $R = h_1(Q) \cdot h_2(Q)$ par l'attaquant permet de déduire le système

$$\begin{cases} P_i(X_P, Y_P, X_T, Y_T, Z_T) &= R_i, i \in \{0, 2, 3, 4, 5, 6, 8, 9, 10, 11\} \\ X_P^3 + 5 - Y_P^2 &= 0 \\ X_T^3 + 5 \cdot Z_T^6 - Y_T^2 &= 0, \end{cases} \tag{76}$$

avec les $R_i \in \mathbb{F}_p$ obtenus par identification sur $\mathbb{F}_p$. Ce système peut être résolu à l'aide d'une base de Gröbner [AL94] ce qui donne les valeurs X_P et Y_P.

4.2 Attaques pratiques sur l'algorithme de Miller

En utilisant le banc d'injection EM présenté dans la section Section 7.3.2, nous voulons valider que ces attaques, décrites uniquement de manière théorique jusqu'à maintenant, peuvent être implémentées en pratique.

Le but de cette attaque en faute est de sortir de la boucle de Miller à volonté avec une injection de fautes EM. Nous considérons le cas d'un algorithme de Miller fauté où la dernière itération a été sautée. L'implémentation cible est un couplage de Ate twisté dont l'exponentiation finale a été enlevée. Dans cette implémentations, les calculs sont faits dans le domaine de Montgomery et donc, les valeurs intermédiaires doivent être multipliée par $1/Res$, l'inverse du résidu de Montgomery, pour être converties vers le domaine canonique.

Comme détaillé dans la Section 4.2, cette méthode nous a permis de prouver la faisabilité d'une attaque en faute sur l'algorithme de Miller.

4.3 Les contremesures pour protéger l'algorithme de Miller

Puisque les attaques en faute sur l'algorithme de Miller ont déjà été étudiées, des contremesures ont déjà été proposées dans la littérature. Dans cette section, nous examinons ces propositions. *Ce travail est le résultat d'une collaboration avec Nadia El Mrabet et Marie Paindavoine.*

Les contremesures dans la littérature

Des contremesures ont été proposées pour contrer les attaques en faute montrées dans la Section 4.1. Nous présentons ici ces contremesures [EMPV12].

1. Une contremesure naïve consiste simplement à calculer deux fois le couplage. C'est à dire, nous pouvons calculer $e_1 = e(P, Q)$, $e_2 = e(P, Q)$ et vérifier si $e_1 = e_2$. De plus, la bilinéarité des couplages permet de calculer $e_1 = e(P, Q)$ et $e_2 = e(\alpha P, \beta Q)$, avec α, β des entiers aléatoires et de vérifier que $e_2 = e_1^{\alpha\beta}$.

2. Il est aussi possible de vérifier les valeurs intermédiaires. Par exemple, il est possible de vérifier que les points intermédiaires T sont bien des éléments de la courbe elliptique ou si la valeur f_1 dans l'algorithme de Miller est bien un élément de $\mathbb{F}_{p^k}$.

3. Dans [GS11], les auteurs proposent une contremesure particulière pour contrer leur modèle de faute qui vérifie si l'algorithme de Miller a fait plus de 2 itérations.

4. Des codes robustes pour la détection de fautes ont été proposés dans [OGS07] pour les corps de caractéristique 3. La nécessité d'un compteur de boucle robuste est également examinée.

Des contremesures plus populaires et plus élégantes (puisque leur impact sur les performances est plus faible) utilisent des schémas de masquage dans l'algorithme de Miller.

5. **Le masquage des coordonnées** est proposé dans [KTH+06] où les entrées du couplage sont masquées. En coordonnées jacobiennes, le point $P = (X_P : Y_P : Z_P)$ est équivalent au point $(\lambda^2 X_P : \lambda^3 Y_P : \lambda Z_P)$, pour un λ non nul. La contremesure consiste a modifier les coordonnées de P avant le calcul du couplage. Par conséquent, les valeurs dans les équations de l'algorithme de Miller sont changées.

6. **Masquage de la variable de Miller**: Une autre contremesure proposée dans [Sco05] consiste à masquer les valeurs intermédiaires de la fonction de Miller f. A chaque itération de Miller, la fonction f est multipliée par une valeur aléatoire λ dans un sous-corps strict de $\mathbb{F}_{p^{12}}$. Cette contremesure n'influence pas le résultat du couplage puisque l'exponentiation finale élimine ce facteur en l'envoyant vers 1.

7. **Masquage additif**: En utilisant la bilinéarité, comme proposé par Page *et al.* dans [PV06], il est possible de masquer un point en entrée avec un masquage additif. Pour un point aléatoire $M \in E(\mathbb{F}_{p^k})$, nous avons $e(P, Q) = \frac{e(P,Q+M)}{e(P,M)}$ ou de manière équivalente $e(P, Q) = e(P, Q + M) \cdot e(P, -M)$.

8. **Masquage multiplicatif**: Comme décrit dans [PV06], le masquage multiplicatif utilise le fait que les points P et Q vérifient $e(\alpha P, \beta Q) = e(P, Q)^{\alpha\beta}$. En choisissant α, β tels que $\alpha \cdot \beta = 1 \pmod r$, alors $e(\alpha P, \beta Q) = e(P, Q)$.

9. Shirase *et al.* [STO08] ont proposé une contremesure spécifique aux corps de caractéristique 3 et non applicable à notre cas où ils ajoutent une valeur de masquage lors de l'évaluation de la ligne.

Il faut noter que les deux contremesures par masquage (5 et 6) ont d'abord été proposées dans le contexte de la SCA et non contre les attaques en faute. Néanmoins il a été suggéré qu'elles étaient également efficaces dans ce cas dans [EMPV12].

5 Attaques en faute sur l'Exponentiation Finale

Maintenant que nous avons étudié les attaques en faute sur le MA, que nous avons démontré qu'elles étaient réalisables en pratique et avons étudié les contremesures associées, nous présentons des attaques qui ont pour but d'inverser l'exponentiation finale.

5.1 Une attaque en faute pour inverser l'exponentiation finale en 3 fautes indépendantes

Comme mentionné dans [WS07], la FE dans les couplages "Tate-like" est un calcul long et complexe. Nous montrons maintenant comment des fautes précisément choisies peuvent aider à trouver les valeurs intermédiaires critiques qui permettent d'inverser l'exponentiation finale. Pour simplifier l'exploitation de la faute, nous considérons que sa valeur est comprise entre 0 et $2^l - 1$.

Notre travail est basé sur les algorithmes proposés par Scott *et al.* dans [SBC+09]. Nous nous concentrons sur la FE dans les corps avec un degré de plongement pair. On pose donc $d = k/2$. L'optimisation décrite dans [SBC+09], toujours utilisée dans les implémentations de couplage, se base sur la décomposition de la FE en trois étapes. Comme $\frac{p^k-1}{r}$ peut se réécrire $\frac{p^k-1}{r} = \left(p^d - 1\right) \cdot \frac{p^d+1}{\Phi_k(p)} \cdot \frac{\Phi_k(p)}{r}$ où $\Phi_k(p)$ est le k-ième polynôme cyclotomique (cf. Definition 7.2.24). La FE peut être réalisée comme une succession de trois exponentiations. Deux sont "faciles" (celles d'exposants $p^d - 1$ et $\frac{p^d+1}{\Phi_k(p)}$) puisqu'elles reposent sur des exponentiations à la puissance p^n pour un n fixé et peuvent donc être calculées avec l'aide de l'endomorphisme de Frobenius (cf. Definition 7.2.23), qui est peu couteux à calculer. La dernière exponentiation est dite "difficile" (elle ne fait pas appel au frobenius de manière simple) et est d'exposant $\frac{\Phi_k(p)}{r}$. Par exemple pour $k = 12$, nous avons

$$\frac{p^{12} - 1}{r} = \left(p^6 - 1\right) \cdot \left(p^2 + 1\right) \cdot \frac{p^4 - p^2 + 1}{r}. \tag{77}$$

Soit f une valeur aléatoire dans $\mathbb{F}^*_{p^k}$, symbolisant le résultat de l'algorithme de Miller.Nous notons les résultats des exponentiations intermédiaires

$$f_1 = f^{p^d-1} \; ; \; f_2 = f_1^{\frac{p^d+1}{\Phi_k(p)}} \text{ and } f_3 = f_2^{\frac{\Phi_k(p)}{r}}. \tag{78}$$

L'attaquant connaît le résultat f_3 et veut retrouver f. Il faut noter que f_1, f_2 et f_3 appartiennent à des sous-groupes différents de $\mathbb{F}^*_{p^k}$. Puisque $f \in \mathbb{F}^*_{p^k}$, l'équation suivante est satisfaite

$$f^{p^k-1} = 1 \; ; \; f_1^{p^d+1} = 1 \; ; \; f_2^{\Phi_k(p)} = 1 \text{ and } f_3^r = 1. \tag{79}$$

Ainsi $f_1 \in \mu_{p^d+1}$, $f_2 \in \mu_{\Phi_k(p)}$ et $f_3 \in \mu_r$. Ces sous-groupes ont pour tailles $p^d + 1$, $\Phi_k(p)$ et r respectivement. Par exemple pour $k = 12$, f_1 contient ≈ 1536 bits d'entropie, f_2 contient ≈ 1024 bits d'entropie et f_3 contient ≈ 256 bits d'entropie.

Récupérer f_1

Dans cette section, nous montrons comment une faute sur la valeur intermédiaire f_1 permet de retrouver cette valeur.

Découverte d'un candidat

Lemma 7.5.1 *Soit $\mathbb{F}_{p^k} = \mathbb{F}_{p^d}[w]/(w^2 - v)$ la règle de construction pour l'extension $\mathbb{F}_{p^k}$. v est un non-résidu quadratique dans $\mathbb{F}_{p^d}$ et est un paramètre public. Soit $x \in \mathbb{F}_{p^k}$ tel que $x = g + h \cdot w$ avec $g, h \in \mathbb{F}_{p^d}$. Ainsi $x^{p^d+1} = g^2 - v \cdot h^2 \in \mathbb{F}_{p^d}$.*

Preuve Nous avons $x^{p^d} = g - h \cdot w$ puisque $x^{p^d} = (g + h \cdot w)^{p^d} = g^{p^d} + h^{p^d} \cdot w^{p^d} = g + h \cdot (-w)$. Nous obtenons $x^{p^d+1} = x^{p^d} \cdot x = (g - h \cdot w) \cdot (g + h \cdot w) = g^2 - w^2 \cdot h^2 = g^2 - v \cdot h^2$ puisque $w^2 = v$. ∎

Soit $f_1 = g_1 + h_1 \cdot w$ avec $g_1, h_1 \in \mathbb{F}_{p^d}$. Nous avons

$$f_1^{p^d+1} = f_3^r = 1. \tag{80}$$

Ainsi par le Lemme 7.5.1

$$g_1^2 - v \cdot h_1^2 = 1. \tag{81}$$

Mais l'équation (5.4) est satisfaite uniquement parce que $f_1 \in \mu_{p^d+1}$. Soit $e \in \mathbb{F}_{p^d}$ une faute injectée sur f_1 (par exemple durant la multiplication produisant f_1 ou durant le chargement de f_1 pour la seconde exponentiation "facile", cf. Figure 3) telle que la valeur fautée f_1^* vaille

$$f_1^* = f_1 + e \notin \mu_{p^d+1}. \tag{82}$$

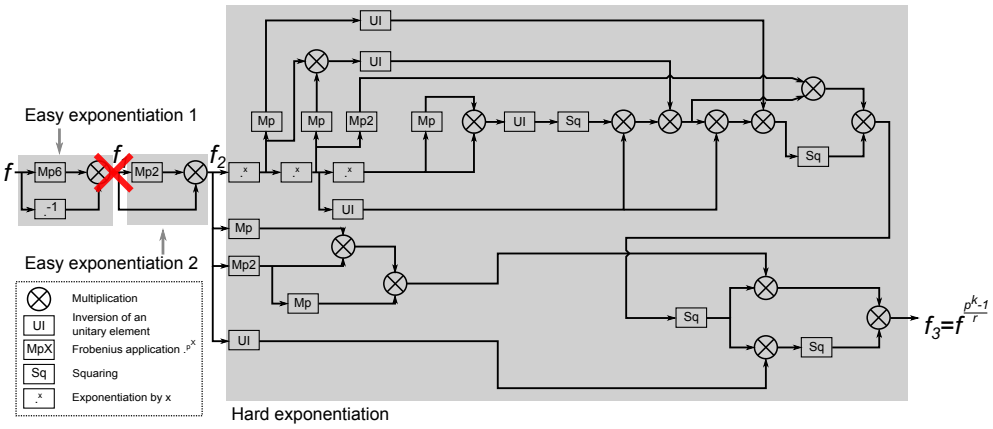

Figure 3: Emplacement de la première faute lors de la FE.

Nous considérons que la faute e n'a lieu que sur la coordonnée g_1[2] (ce qui est compatible avec notre modèle de faute si $2^l < p^6$), *i.e.*

$$f_1^* = (g_1 + e) + h_1 \cdot w. \tag{83}$$

[2]Si sur h_1, le même raisonnement tient toujours.

$(f_1^*)^{p^d+1}$ peut être calculé par l'attaquant en utilisant le résultat fauté obtenu f_3^* puisque r est un paramètre public.

$$(f_1^*)^{p^d+1} = (f_3^*)^r \in \mathbb{F}_{p^d}. \tag{84}$$

En utilisant le Lemme 7.5.1, l'équation (81) et l'équation (83) nous avons

$$\begin{aligned}(f_1^*)^{p^d+1} &= (g_1+e)^2 - v \cdot h_1^2 \\ &= g_1^2 - v \cdot h_1^2 + 2 \cdot e \cdot g_1 + e^2 \\ &= 1 + 2 \cdot e \cdot g_1 + e^2.\end{aligned}$$

Finalement, g_1 peut s'écrire:

$$g_1 = \frac{(f_1^*)^{p^d+1} - 1 - e^2}{2 \cdot e}. \tag{85}$$

Deux valeurs sont possibles pour h_1, calculées avec l'équation (81):

$$h_1^+ = \sqrt{\frac{g_1^2 - 1}{v}} \; ; \; h_1^- = -\sqrt{\frac{g_1^2 - 1}{v}}. \tag{86}$$

Vérification des candidats Les deux candidats $f_1^+ = g_1 + h_1^+ \cdot w$ et $f_1^- = g_1 + h_1^- \cdot w$ peuvent ainsi être vérifiés en s'assurant si $(f_1^+)^{\frac{p^d+1}{r}} = f_3$ ou $(f_1^-)^{\frac{p^d+1}{r}} = f_3$. Si la valeur e est inconnue, l'attaquant doit deviner la valeur. Pour chaque hypothèse, deux candidats sont trouvés et vérifiés. Un candidat n'est égal à la valeur correcte f_1 que quand la valeur correcte est devinée pour e.

Dans notre modèle de faute, $0 < e < 2^l$ ainsi $2^l - 1$ essais devront être fait pour trouver f_1 avec certitude. A ce stade, on est en droit de se demander quelle est la probabilité que l'attaquant trouve un candidat pour f_1 (et la valeur d'erreur associée) valide par rapport à toutes les observations mais qui n'est pas égal à f_1 (*i.e.* un faux positif). Le candidat pour f_1 est noté f_{1c} et la valeur d'erreur associée est notée e_c.

$$f_{1c}^{p^d+1} = 1 \tag{87}$$

$$(f_{1c} + e_c)^{p^d+1} = (f_3^*)^r. \tag{88}$$

Mais l'attaquant observe $f_3 = f_1^{\frac{p^d+1}{r}}$ et $f_3^* = (f_1+e)^{\frac{p^d+1}{r}}$. La question est quelle est la probabilité que $f_{1c} \neq f_1$ mais que

$$f_3 = f_{1c}^{\frac{p^d+1}{r}} \tag{89}$$

$$f_3^* = (f_{1c} + e_c)^{\frac{p^d+1}{r}}. \tag{90}$$

En utilisant l'équation (87), la probabilité que l'équation (90) soit vérifiée peut être évaluée comme étant égale à $1/r$ pour une valeur f_{1c} aléatoire dans μ_{p^d+1}. En effet, nous savons déjà que $f_{1c}^{\frac{p^d+1}{r}}$ est dans μ_r et $1/r$ est la probabilité qu'un élément aléatoire dans μ_{p^d+1} soit envoyé vers la valeur fixée f_3 dans μ_r. De manière similaire, à partir de l'équation (88), nous pouvons

déduire que la probabilité pour que l'équation (89) soit vérifiée est égale à $1/r$ pour une valeur f_{1c} aléatoire dans $\mathbb{F}_{p^k}^*$ puisque $(f_3^*)^r = (f_{1c} + e_c)^{p^d+1} \in \mu_{p^d-1}$. Ainsi $f_3^* \in \mu_{r\cdot(p^d-1)}$ et $(f_3^*)^r$ possède r antécédents dans $\mu_{r\cdot(p^d-1)}$. Par conséquent, la probabilité d'obtenir l'antécédent correct est $1/r$.

Nous pouvons combiner ces probabilités et évaluer la probabilité d'avoir un candidat incorrect pour f_1 qui corresponde à toutes les observations de l'attaquant. La probabilité qu'un candidat aléatoire satisfasse l'équation (87) et l'équation (88), satisfasse aussi l'équation (89) et l'équation (90), correspondant aux observations de l'attaquant, est égale à $1/r^2$. Dans le cas $k = 12$, nous avons usuellement $r \approx 2^{256}$, la probabilité de trouver un candidat valide qui n'est pas égal à f_1 est de $1/2^{512}$.

Finalement nous avons montré comment une faute injectée sur f_1 peut être utilisée pour retrouver cette valeur, avec une forte probabilité, en utilisant la sortie correcte f_3 et la sortie fautée f_3^* de la FE.

Récupérer f

Connaissant la valeur de f_1, nous pouvons maintenant voir comment récupérer f.

Découverte d'un candidat La stratégie consiste à utiliser des équations similaires à celles utilisées précédemment et d'inclure la nouvelle information à propos de f_1 obtenue par l'attaquant.

Lemma 7.5.2 *Soit $f = g + h \cdot w$, $f^{-1} = g' + h' \cdot w$ et $f_1 = g_1 + h_1 \cdot w$.*
Alors $\frac{g_1-1}{v\cdot h_1} = \frac{h'}{g'} = -\frac{h}{g} \Leftrightarrow f_1 = f^{p^d-1}$.

1: $f_1 = f^{p^d-1} \Rightarrow \frac{g_1-1}{v\cdot h_1} = \frac{h'}{g'} = -\frac{h}{g}$. **Preuve**

$$f_1 = \bar{f} \cdot f^{-1} = (f - 2 \cdot h \cdot w) \cdot f^{-1} = f \cdot f^{-1} - 2 \cdot h \cdot w \cdot f^{-1} = 1 - 2 \cdot h \cdot w \cdot (g' + h' \cdot w.)$$

Ainsi

$$g_1 = 1 - 2 \cdot h \cdot h' \cdot w^2 = 1 - 2 \cdot h \cdot h' \cdot v$$
$$h_1 = -2 \cdot h \cdot g'.$$

Finalement

$$\frac{g_1 - 1}{v \cdot h_1} = \frac{-2 \cdot h \cdot h' \cdot v}{-2 \cdot h \cdot v \cdot g'} = \frac{h'}{g'}.$$

De plus

$$g' = \frac{g}{g^2 - v \cdot h^2}$$
$$h' = \frac{-h}{g^2 - v \cdot h^2}.$$

Donc

$$\frac{g_1 - 1}{v \cdot h_1} = -\frac{h}{g}.$$

∎

2: $\frac{g_1-1}{v\cdot h_1} = \frac{h'}{g'} = -\frac{h}{g} \Rightarrow f_1 = f^{p^d-1}$.

Preuve Nous écrivons

$$\bar{f} \cdot f^{-1} = (g - h \cdot w) \cdot (g' + h' \cdot w) \tag{91}$$
$$= g \cdot g' - v \cdot h \cdot h' + (g \cdot h' + h \cdot g') \cdot w. \tag{92}$$

avec

$$g' = \frac{g}{g^2 - v \cdot h^2} = \frac{1}{g\,(1 - v \cdot K^2)} \text{ and } h' = \frac{-h}{g^2 - v \cdot h^2} = \frac{1}{h\,(v - 1/K^2)}. \tag{93}$$

Par conséquent :

$$g \cdot g' - v \cdot h \cdot h' = \frac{1 + v \cdot K^2}{1 - v \cdot K^2} = \frac{v \cdot h_1^2 + g_1^2 - 2 \cdot g_1}{v \cdot h_1^2 - g_1^2 + 2 \cdot g_1 - 1} = \frac{2 \cdot g_1 \cdot (g_1 - 1)}{2 \cdot (g_1 - 1)} = g_1.$$

Et

$$g \cdot h' + h \cdot g' = \frac{K}{1 - v \cdot K^2} - \frac{1}{K \cdot (v - 1/K)} = \frac{2 \cdot K}{1 - v \cdot K^2}$$
$$= \frac{2 \cdot (g_1 - 1) \cdot h_1}{v \cdot h_1^2 - g_1^2 + 2 \cdot g_1 - 1} = \frac{2 \cdot (g_1 - 1) \cdot h_1}{2 \cdot (g_1 - 1)} = h_1.$$

∎

Dans la suite, soit K la valeur connue (parce que nous connaissons g_1 et h_1 par f_1, trouvée précédemment) $K = \frac{g_1 - 1}{v \cdot h_1} = -\frac{h}{g}$.
Par conséquent, la connaissance de f_1 permet de trouver des antécédents aléatoires en prenant un $g \in \mathbb{F}_{p^d}$ aléatoire et en choisissant $h = -K \cdot g$.

Pour trouver f, l'attaquant crée une nouvelle faute $e_2 \in \mathbb{F}_{p^d}$ durant l'inversion de la première exponentiation facile (cf. Figure 4).

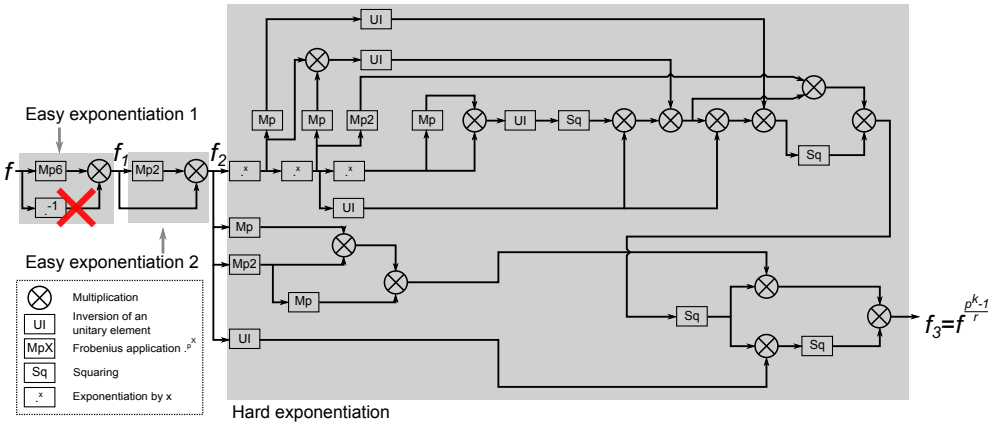

Figure 4: Emplacement de la seconde faute dans la FE.

Puis

$$f_1 = f^{p^d - 1} = \bar{f} \cdot f^{-1} \text{ and } f_1^* = \bar{f} \cdot (f^{-1} + e_2).$$

Soit Δ_{f_1} la différence : $\Delta_{f_1} = f_1^* - f_1 = \bar{f} \cdot e_2$. Puisque $e_2 \in \mathbb{F}_{p^d}$, nous pouvons écrire $\Delta_{f_1} = \Delta_{g_1} + \Delta_{h_1} \cdot w$ avec

$$\Delta_{g_1} = e_2 \cdot g \text{ and } \Delta_{h_1} = -e_2 \cdot h.$$

Puisque f_1^* n'est pas dans μ_{p^d+1} avec une grande probabilité égale à $(1 - \frac{1}{p^d-1})$, l'attaquant peut calculer $(f_1^*)^{p^d+1} = (f_3^*)^r \in \mathbb{F}_{p^d}$. Dans ce cas

$$\begin{aligned}(f_1^*)^{p^d+1} &= (g_1 + \Delta_{g_1})^2 - v \cdot (h_1 + \Delta_{h_1})^2 \\ &= (g_1 + e_2 \cdot g)^2 - v \cdot (h_1 - e_2 \cdot h)^2.\end{aligned}$$

ce qui donne l'équation de degré 2 suivante (en utilisant la relation $h = -g \cdot K$)

$$g^2 \cdot e_2^2 \cdot (1 - v \cdot K^2) + g \cdot 2 \cdot e_2 \cdot (g_1 - v \cdot K \cdot h_1) + 1 - (f_1^*)^{p^d+1} = 0. \tag{94}$$

En résolvant cette équation, nous trouvons deux candidats pour g:

$$\begin{aligned}g^+ &= \frac{v \cdot K \cdot h_1 - g_1 + \sqrt{(g_1 - v \cdot K \cdot h_1)^2 - (1 - v \cdot K^2) \cdot \left(1 - (f_1^*)^{p^d+1}\right)}}{e_2 \cdot (1 - v \cdot K^2)} \\ g^- &= \frac{v \cdot K \cdot h_1 - g_1 - \sqrt{(g_1 - v \cdot K \cdot h_1)^2 - (1 - v \cdot K^2) \cdot \left(1 - (f_1^*)^{p^d+1}\right)}}{e_2 \cdot (1 - v \cdot K^2)}.\end{aligned}$$

h peut être calculé à l'aide de g et K: $h = -g \cdot K$. Nous avons donc deux candidats potentiels pour f.

Vérification des candidats Même si e_2 est inconnue, cette procédure donne deux candidats en émettant une hypothèse sur e_2. Maintenant, que cette hypothèse soit correcte ou fausse, chaque candidat f_c possède la propriété suivante: $f_c^{p^d-1} = f_1$ et ainsi $f_c^{\frac{p^k-1}{r}} = f_3$. L'attaquant a ainsi trouvé plusieurs antécédents valides de f_3 et doit décider celui qui est correct.

En vérifiant si $(\bar{f}_c \cdot (f_c^{-1} + e_2))^{\frac{p^d+1}{r}}$ est égal au résultat fauté f_3^* permet d'éliminer un de ces deux candidats pour cette hypothèse de e_2. Nous obtenons finalement un candidat pour chaque hypothèse de e_2 et ce candidat correspond à toutes les observations faites par l'attaquant. L'ensemble des candidats possible à la même taille que l'ensemble des valeurs possibles d'erreur.

L'attaquant doit donc générer une troisième faute e_3, différente de e_2, au même endroit que celle ci et calculer l'intersection des deux ensembles de candidats en résultant pour trouver le candidat correct. Malheureusement cette intersection ne contient pas nécessairement un seul élément. Mais nous pouvons évaluer la taille de cette ensemble intersection. Premièrement, nous négligeons la probabilité qu'un élément aléatoire de $\mathbb{F}_{p^k}^*$ soit envoyer vers la valeur f_1 (la probabilité est $1/(p^d+1)$). L'équation (94) fournit un candidat pour f, f_{c1}, en supposant $e_2 = 1$. Ainsi l'ensemble de candidats est $\{f_{c1}, f_{c2}, \ldots, f_{c(2^l-1)}\}$ avec f_{ci} correspondant à l'hypothèse $e_2 = i$. Si nous remplaçons le produit $g \cdot e_2$ par $\frac{g}{i} \cdot (i \cdot e_2)$ dans l'équation (94), nous pouvons voir que l'ensemble précédent peut être réécrit comme $\{f_{c1}, \frac{f_{c1}}{2}, \ldots, \frac{f_{c1}}{2^l-1}\}$.

De manière similaire pour e_3, l'équation (94) fournit un candidat pour f, f'_{c1}, en supposant $e_3 = 1$ et avec $f'_{ci} = f'_{c1}/i$. Le deuxième ensemble de candidats $\{f'_{c1}, \frac{f'_{c1}}{2}, \ldots, \frac{f'_{c1}}{2^l-1}\}$.

Soit e_{2t} et e_{3t} les deux fautes vraiment injectées. Puisque la valeur correcte de f se trouve dans les deux ensembles de candidats, d'abord égale à f_{c1}/e_{2t} puis égale à f'_{c1}/e_{3t}, nous avons

$$f = \frac{f'_{c1}}{e_{3t}} = \frac{f_{c1}}{e_{2t}}. \tag{95}$$

En écrivant $a = \frac{e_{2t}}{e_{3t}}$, l'équation (95) peut se transformer en $f'_{c1} = f_{c1}/a$. Et le second ensemble de candidats en $\{\frac{f_{c1}}{a}, \frac{f_{c1}}{2a}, \ldots, \frac{f_{c1}}{(2^l-1)a}\}$. Ainsi un même candidat se trouve dans les deux ensembles à chaque fois que l'équation

$$a \cdot i = j \tag{96}$$

est satisfaite avec $i, j \in [\![1, 2^l - 1]\!]$. Dans notre modèle de faute, nous pouvons prendre les éléments e_{2t} et e_{3t} comme des éléments de $\mathbb{N}$ et le nombre de solutions à cette équation devient $\left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor$.

Preuve Soit e_{2t} et e_{3t} dans notre modèle de faute : $0 < e_{2t}, e_{3t} < 2^l - 1$ et $p \gg 2^l$. Soit $a = \frac{e_{2t}}{e_{3t}} \in \mathbb{F}_p$, nous voulons trouver le nombre de couples (i, j) solutions de l'équation (96): $a \cdot i = j$ avec $i, j \in [\![1, 2^l - 1]\!]$. Nous pouvons écrire

$$\frac{e_{2t}}{e_{3t}} = \frac{j}{i}.$$

Cette fraction peut se réécrire $\frac{u}{v}$, et en la simplifiant :

$$u = \frac{e_{2t}}{gcd(e_{2t}, e_{3t})}$$
$$v = \frac{e_{3t}}{gcd(e_{2t}, e_{3t})}.$$

Toutes les paires de solutions de l'équation (96) peuvent s'écrire comme $(k \cdot u, k \cdot v)$, $k \in \mathbb{N}^+$. Les conditions $i, j \in [\![1, 2^l - 1]\!]$ sont équivalentes à $k \leq \frac{2^l-1}{u}$ et $k \leq \frac{2^l-1}{v}$ ce qui, combinées, donne $k \leq \frac{2^l-1}{max(u,v)}$.

A partir des définitions de u et v, nous avons : $max(u, v) = \frac{max(e_{2t}, e_{3t})}{gcd(e_{2t}, e_{3t})}$.

Finalement, nous obtenons une solution pour chaque entier k dans l'intervalle

$$[\![1, (2^l - 1) \cdot \frac{gcd(e_{2t}, e_{3t})}{max(e_{2t}, e_{3t})}]\!]$$

La borne supérieure nous donne le nombre de solutions pour notre équation (5.20). ∎

Finalement, la taille de l'intersection, qui contient aussi le candidat correct, est

$$\#intersection = \left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor, \tag{97}$$

et le nombre de mauvais candidats est

$$\#intersection - 1 = \left\lfloor (2^l - 1) \cdot \frac{\gcd(e_{2t}, e_{3t})}{\max(e_{2t}, e_{3t})} \right\rfloor - 1. \tag{98}$$

L'intersection de ces ensembles de candidats obtenue avec e_2 et e_3 contient au moins un élément si les deux valeurs sont correctement supposées au moins une fois.

Le coût en calcul pour retrouver f est bas puisque l'attaquant doit utiliser la procédure pour retrouver un andidat avec l'équation (94) une seule fois par faute injectée en prenant pour hypothèses $e_2 = 1$ et $e_3 = 1$.

Il suffit ensuite de mémoriser les candidats correspondant et de calculer le rapport $a = f_{c1}/f'_{c1}$. Finalement, il faut résoudre l'équation (96), en essayant tous les $i \in [\![1, 2^l - 1]\!]$ et en vérifiant si $a \cdot i \in [\![1, 2^l - 1]\!]$, ce qui fournit e_{2t} et e_{3t} (les seules solutions s'il n'y a pas de mauvais candidats). Avec e_{2t}, il faut calculer $f = f_{c1}/e_{2t}$.

Il n'est pas possible d'éviter l'apparition de mauvais candidats. Pour conclure l'attaque, il faut obtenir un unique candidat qui satisfasse toutes les observations.

Si plus d'un candidat est obtenu avec e_2 et e_3, l'attaquant doit alors générer d'autre fautes, de valeurs différentes, au même emplacement jusqu'à ce qu'un seul candidat soit valide.

Résumé de l'attaque sur l'exponentiation finale

Au moins quatre exécutions du même couplage (avec les mêmes entrées) sont nécessaires pour réaliser notre attaque.

1. Le premier calcul de couplage est réalisé sans fautes. L'attaquant mémorise f_3, le résultat correct.

2. Une première faute est crée sur f_1 comme dans la Section 7.5.1 (Retrouver f_1). L'attaquant mémorise le résultat fauté f_3^*. f_1 est récupérée grâce à l'équation (85) et à l'équation (81).

3. Une seconde faute e_2 est injectée durant l'inversion dans la première exponentiation facile comme dans la Section 7.5.1 (Retrouver f). L'attaquant mémorise le résultat fauté f_3^*, et extrait un candidat f_{c1} pour f en supposant $e_2 = 1$ avec l'équation (94) et le Lemme 7.5.2.

4. De manière similaire, une troisième faute $e_3 \neq e_2$ est injectée. Avec le résultat fauté f_3^*, l'attaquant extrait un nouveau candidat f'_{c1} pour f en supposant $e_3 = 1$. La valeur $a = f_{c1}/f'_{c1}$ est ensuite calculée. Un couple (i, j) solution de l'équation $a * i = j$ avec $i, j \in [\![1, 2^l - 1]\!]$ lui permet de calculer $f = f_{c1}/j$.

Si plusieurs couples solutions (i, j) sont trouvés, d'autres fautes peuvent être nécessaires pour s'assurer de l'unicité du candidat pour f. Cette attaque en faute ne nécessite pas d'injecter plusieurs fautes au cours d'une même exécution, ce qui la rend expérimentalement plus facile à réaliser.

Contremesures

Il existe des contremesures naturelles contre cette attaque en faute. La première vient du fait que pour accélérer le calcul des inversions des éléments unitaires ($\in \mu_{p^d+1}$), il est possible de remplacer l'inversion par une conjugaison. Ainsi dans la FE toutes les inversions à part la première peuvent être remplacées par des conjugaisons. Dans ce cas, notre attaque ne fonctionne pas car l'attaquant n'est plus capable d'obtenir la valeur $(f_1^*)^{\frac{p^d+1}{r}}$. f_1^* n'étant pas unitaire, le résultat obtenu par l'attaquant n'est plus la valeur attendue. De manière plus générale, l'utilisation d'une représentation compressée [NBS08, AKL+11] des valeurs dans la FE empêche l'attaque en faute. Enfin il est possible de contrer l'attaquant en vérifiant l'appartenance des différentes valeurs intermédiaires à leurs sous-groupes respectifs.

5.2 Une attaque en faute pratique pour inverser l'exponentiation finale

L'attaque en faute précédente sur l'exponentiation finale (Section 7.5.1) a été réalisée en pratique pour la valider expérimentalement. Pour cela nous avons utiliser le banc d'injection EM présenté

dans la Section 7.3.2. Nous avons du modifier le code de l'exponentiation finale pour forcer l'utilisation d'une inversion et non une conjugaison pour les éléments unitaires.

Le description de cette attaque est présente dans la Section 5.3.

6 Attaques en fautes sur un couplage complet

Précédemment, nous avons présenté des attaques en faute sur l'algorithme de Miller et sur l'Exponentiation Finale indépendamment, et nous les avons testées séparément. Aucunes de ces attaques seules n'est une menace contre les implémentations de couplage. Il faut combiner ces attaques dans le but d'inverser un couplage complet. Dans cette section, nous proposons des stratégies pour attaquer un couplage complet.

Dans ces nouvelles attaques en faute, nous considérons que l'attaquant est capable de sauter des instructions durant le calcul d'un couplage. Nous devons alléger nos contraintes et autoriser les fautes doubles, deux fautes injectées lors de la même exécution (ce qui a déjà été fait [TK10, BGdSG+14]). Expérimentalement, il peut être difficile d'injecter plusieurs fautes lors d'une même exécution. Cette difficulté dépends de la durée entre les injections et des équipements utilisés. C'est pourquoi nous préférons tout de même des schémas d'attaques avec un nombre limité de fautes dans la même exécution, même si cela implique de réaliser plus d'exécutions. *La recherche de nouveaux schémas d'attaques contre des couplages complets a été faite en collaboration avec Hélène Le bouder.*

Le principe derrière nos attaques est de récupérer des valeurs intermédiaires dans l'algorithme. Puisque l'attaquant observe la sortie, il est plus facile de récupérer des valeurs intermédiaires proches de la sortie, puis de remonter progressivement l'algorithme.

La première étape est de récupérer le résultat correct de l'algorithme de Miller en utilisant l'attaque en faute présentée dans la Section 7.5.1. Une fois que l'attaquant connaît le résultat correct du couplage $e(P, Q)$, la faute e_1 est utilisée pour retrouver f_1 puis une seconde fault e_2 est utilisée pour retrouver f, le résultat de l'algorithme de Miller.

Maintenant, dans le but de trouver P, le point secret (ou Q), l'attaquant doit fauter à la fois l'algorithme de Miller et l'Exponentiation Finale, durant la même exécution.

Dans le modèle de faute de saut d'instruction, l'attaquant peut sauter l'itération de la boucle de Miller qu'il veut, il peut donc sortir de la boucle à l'itération désirée. Dans ce cas, la faute est parfaitement reproductible et aura toujours le même effet sur le calcul.

6.1 Faute e_3 sur la dernière itération

Si le saut de boucle e_3 est effectué dans la dernière itération, une attaque complète serait la suivante.

1. Pas de fautes: obtient $f_3 = e(P, Q)$.
2. e_1: obtient f_1.
3. e_2: obtient f le résultat correct de l'algorithme de Miller.
4. e_3: obtient f_3^* un résultat de Miller fauté après une Exponentiation Finale correcte.
5. (e_3, e_1): obtient f_1^*.
6. (e_3, e_2): obtient f^* le résultat de l'algorithme de Miller fauté.

Finalement, avec f et f^*, les attaquants peuvent retrouver le secret comme montré dans Section 4.2. Cette attaque en faute nécessite 6 exécutions avec 3 exécutions à fautes simples et 2 exécutions à fautes doubles. Pour que cela fonctionne, il faut que la faute e_3 soit répétée dans plusieurs exécutions différentes.

Les exécutions 1,2,3 sont nécessaires pour retrouver le résultat correct de l'algorithme de Miller en suivant le schéma d'attaque exposé dans la Section 7.5.1. La faute e_3 correspond au saut de boucle, il manque la dernière itération à l'algorithme de Miller fauté. Les exécutions 4,5,6 combinent le saut de boucle avec le schéma d'attaque sur la FE présenté dans la Section 7.5.1. Cela fonctionne parce que la faute e_3, le saut de la dernière itération, donne exactement la même valeur $f_{K,Q}(P)^*$. Ces 6 exécutions donnent à la fois $f_{K,Q}(P)$ et $f_{K,Q}(P)^*$ ce qui permet d'utiliser l'attaque en faute présentée dans la Section 7.4.1 (Attaques sur le flot de données).

Ce schéma d'attaque complet a été implémenté en pratique et est présenté dans la Section 6.2.

6.2 Faute e_3 sur la première itération

Si l'attaquant est capable de sortir de la boucle après la première itération, il peut réduire le nombre total d'exécutions nécessaires à 3.

1. e_3: obtient f_3^* un résultat de Miller fauté après une FE correcte.
2. (e_3, e_1): obtient f_1^*.
3. (e_3, e_2): obtient f^* le résultat de Miller fauté.

L'attaque nécessite 3 exécutions avec 1 exécution avec faute simple et 2 exécutions avec des fautes doubles. Dans ce schéma, e_3 correspond au cas où l'attaquant est capable de sortir de la boucle après la première itération. e_3 doit être répétée sur de multiples exécutions : 1,2,3. Ces 3 exécutions permettent d'inverser l'exponentiation finale comme présenté dans la Section 7.5.1 ce qui fournit la valeur $f_{K,Q}(P)^*$. Cette seule valeur permet d'inverser le MA comme montré dans la Section 7.4.1 (Attaque avec sortie à la première itération).

7 Conclusion

Dans cette thèse, nous avons passé en revue les attaques en faute sur les couplages dans le but dévaluer la sécurité des implémentations de couplage. Nous avons vu que les attaques en faute menacent l'exécution sûre d'un couplage si elle n'est pas correctement menée. Dans ce cadre, nous avons choisis une implémentation qui est, à nos yeux, la plus représentative de ce à quoi un couplage moderne ressemblerait (un couplage de Ate sur une courbe BN avec $k = 12$ sur un corps de grande caractéristique). Pour autant, la plupart des travaux détaillés dans cette thèse peuvent facilement être adaptés à d'autres cas (d'autres algorithmes de couplage, d'autres corps, d'autres courbes, d'autres systèmes de coordonnées...).

De manière détaillée, nous avons prouvé la vulnérabilité de l'algorithme de Miller, en pratique, vis à vis des attaques en faute. Les contremesures pour protéger cet algorithme ont été analysées et nous avons montré que certaines d'entre elles sont inefficaces. La vulnérabilité théorique de l'algorithme d'exponentiation finale complexe a été exposée (avec 2 fautes indépendantes au minimum) et elle a ensuite été démontrée en pratique. En combinant plusieurs attaques en faute, nous avons prouvé la vulnérabilité de l'algorithme complet de couplage. Comme moyen d'injection pour nos attaques, un banc d'injection EM a du être maitrisé. Pour la première fois, des fautes doubles ont été crées avec un tel banc. La première injection de faute nous a forcé à

modifier les paramètres de la seconde injection. Un corollaire (supporté par [TK10, BGdSG+14]) est que les fautes doubles sont maintenant une réalité et doivent être prises en compte pour tous les algorithmes cryptographiques. Il existe des systèmes reposant sur la PBC déjà déployés (*e.g.* Voltage Security). Même si leurs choix de paramètres sont différents (*e.g.* Voltage utilise des courbes supersingulières sur des corps à grande caractéristique et $k = 2$), leurs implémentations doivent maintenant être examinées à la lumière des ces nouvelles vulnérabilités.

Pour nos démonstrations, nous avons effectué des injections de faute EM sur un microcontrôleur moderne. Nous avons montré que dans ce contexte, un modèle de faute haut niveau, le saut d'instruction, est efficace et réalisable en pratique. Pourtant, comme toutes les expériences d'attaques en faute, notre configuration dépend de l'implémentation et doit être modifiée si nous changeons de cible. Les bancs d'injection EM ont l'avantage d'être peu couteux et faciles à mettre en œuvre (pas de préparation de la puce nécessaire). Notre banc d'injection est à l'état-de-l'art et beaucoup de travail a été investi dans le but de le maitriser puisque les effets de l'interaction entre une impulsion EM et un microcontrôleur nous sont encore très peu connus.

Nos attaques en faute proposées sont une première approche du problème et sont loin d'être parfaites. Notre attaque en faute proposée dans la Section 7.5.1 contre la FE nous rappelle que même si un algorithme est complexe et qu'on le croit difficile à inverser, il existe toujours un moyen de contourner le problème. Nous pensons en particulier aux fonctions de hachage. Peut être qu'un jour il sera possible de construire des outils permettant d'utiliser la structure d'un algorithme pour proposer des attaques en faute contre lui. Nous avons exploré brièvement cette idée, qui si elle n'as pas été fructueuse, nous a conduit à l'attaque présentée dans la Section 5.2.

Les autres familles d'attaques doivent également être étudiées. Des efforts ont déjà été fait dans ce sens [WS06] pour la SCA mais nous croyons qu'il reste beaucoup de travail dans ce domaine (notamment l'évaluation des contremesures pour protéger l'algorithme de Miller contre la SCA). Nous prédisons que des schémas plus efficaces seront proposés dans le futur. Parmi eux, les attaques combinées (à la fois SCA et FA) devraient être particulièrement efficaces.

Bibliography

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In Burton S. Kaliski, cetin K. Koc, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin Heidelberg, 2003.

[ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 174, 1998.

[ABLR13] Diego F. Aranha, Paulo S. L. M. Barreto, Patrick Longa, and Jefferson E. Ricardini. The realm of the pairings. Cryptology ePrint Archive, Report 2013/722, 2013. `http://eprint.iacr.org/`.

[ADN+10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin Heidelberg, 2010.

[AKL+11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In KennethG. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 48–68. Springer Berlin Heidelberg, 2011.

[AL94] William W Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3. American Mathematical Society Providence, 1994.

[ALH10] Diego F. Aranha, Julio López, and Darrel Hankerson. High-speed parallel software implementation of the ηt pairing. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 89–105. Springer Berlin Heidelberg, 2010.

[AMORH13a] Gora Adj, Alfred Menezes, Thomaz Oliveira, and Francisco Rodriguez-Henriquez. Weakness of $F_{3^{6*1429}}$ and $F_{2^{4*3041}}$ for discrete logarithm cryptography. Cryptology ePrint Archive, Report 2013/737, 2013. `http://eprint.iacr.org/`.

[AMORH13b] Gora Adj, Alfred Menezes, Thomaz Oliveira, and Francisco Rodríguez-Henríquez. Weakness of $F_{3^{6*509}}$ for discrete logarithm cryptography. Cryptology ePrint Archive, Report 2013/446, 2013. `http://eprint.iacr.org/`.

[ARM13] ARM. Keil μvision, 2013. `http://www.keil.com/uvision/default.asp`.

[Bar87] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proc. of CRYPTO '86*, pages 311–323. Springer-Verlag, 1987.

[BBB+06] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management-part 1: General (revised. In *NIST special publication*. Citeseer, 2006.

[BBG05] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 440–456. Springer Berlin Heidelberg, 2005.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of eliminating errors in cryptographic computations. In Lecture Notes in Computer Science, editor, *EUROCRYPT'97*, pages 37–51. Springer, 1997.

[BF01] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer Berlin Heidelberg, 2001.

[BGDM+10] Jean-Luc Beuchat, JorgeE. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto-naehrig curves. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing-Based Cryptography - Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin Heidelberg, 2010.

[BGdSG+14] J. Blömer, R. Gomes da Silva, P. Günther, J. Krämer, and J.-P. Seifert. A practical second-order fault attack against a real-world pairing implementation. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, Sept 2014.

[BGJT13] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. *CoRR*, abs/1306.4244, 2013.

[BGV94] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO' 93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer Berlin Heidelberg, 1994.

[BIP05] Jean-Claude Bajard, Laurent Imbert, and Thomas Plantard. Modular number systems: Beyond the mersenne family. In Helena Handschuh and M.Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 159–169. Springer Berlin Heidelberg, 2005.

[BJ02] Olivier Billet and Marc Joye. The jacobi model of an elliptic curve and side-channel analysis. Cryptology ePrint Archive, Report 2002/125, 2002. `http://eprint.iacr.org/`.

[BK98] R. Balasubramanian and Neal Koblitz. The improbability that an elliptic curve has subexponential discrete log problem under the menezes-okamoto-vanstone algorithm. *Journal of Cryptology*, 11(2):141–145, 1998.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer Berlin Heidelberg, 2007.

[Bla83] G. R. Blakely. A computer algorithm for calculating the product ab modulo m. *IEEE Trans. Comput.*, 32(5):497–500, May 1983.

[BLS03] Paulo S.L.M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, volume 2576 of *Lecture Notes in Computer Science*, pages 257–267. Springer Berlin Heidelberg, 2003.

[BLTMR+09] Jean-Luc Beuchat, Emmanuel López-Trejo, Luis Martínez-Ramos, Shigeo Mitsunari, and Francisco Rodríguez-Henríquez. Multi-core implementation of the tate pairing over supersingular elliptic curves. In JuanA. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 413–432. Springer Berlin Heidelberg, 2009.

[BMH13] Kiseok Bae, Sangjae Moon, and Jaecheol Ha. Instruction fault attack on the miller algorithm in a pairing-based cryptosystem. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*, pages 167–174, July 2013.

[BN06] Paulo S.L.M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer Berlin Heidelberg, 2006.

[Boo51] A.D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[Bri82] Ernest F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In *Proc. of CRYPTO'82*, pages 51–60. Plenum, 1982.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Jr. Kaliski, BurtonS., editor, *Advances in Cryptology - CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer Berlin Heidelberg, 1997.

[BSS99] Ian F Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic curves in cryptography*, volume 265. Cambridge university press, 1999.

[BT13] Karim Bigou and Arnaud Tisserand. Improving modular inversion in rns using the plus-minus method. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 233–249. Springer Berlin Heidelberg, 2013.

[BW05] Friederike Brezing and Annegret Weng. Elliptic curves suitable for pairing based cryptography. *Designs, Codes and Cryptography*, 37(1):133–141, 2005.

[CDF+11] Ray C.C. Cheung, Sylvain Duquesne, Junfeng Fan, Nicolas Guillermin, Ingrid Verbauwhede, and Gavin Xiaoxu Yao. Fpga implementation of pairings using residue number system and lazy reduction. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 421–441. Springer Berlin Heidelberg, 2011.

[Cer12] Certivox. Miracl library (v 5.6.1), 2012. `https://certivox.com/solutions/miracl-crypto-sdk/`.

[CFA+05] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[CKM14] Sanjit Chatterjee, Koray Karabina, and Alfred Menezes. Fault attacks on pairing-based protocols revisited. Cryptology ePrint Archive, Report 2014/492, 2014. `http://eprint.iacr.org/`.

[D+98] Jean-François Dhem et al. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Universit Catholique de Louvain-Facultdes Sciences Appliqus-Laboratoire de microectronique, Louvain-la-Neuve, 1998.

[DDRT12] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 7–15, Sept 2012.

[Deh11] Amine Dehbaoui. *Analyse Sécuritaire des Émanations Électromagnétiques des Circuits Intégrés*. PhD thesis, 2011. Thèse de doctorat dirigée par Maurine, Philippe Génie électrique, électronique, photonique et systemes Montpellier 2 2011.

[DEM05] Régis Dupont, Andreas Enge, and François Morain. Building curves with arbitrary small mov degree over finite prime fields. *Journal of Cryptology*, 18(2):79–89, 2005.

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

[DL03] Iwan Duursma and Hyang-Sook Lee. Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 111–123. Springer Berlin Heidelberg, 2003.

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, volume 7864 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2013.

[EM09] Nadia El Mrabet. What about vulnerability to a fault attack of the miller's algorithm during an identity based protocol? In JongHyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-hoon Kim, and Sang-Soo Yeo, editors, *Advances in Information Security and Assurance*, volume 5576 of *Lecture Notes in Computer Science*, pages 122–134. Springer Berlin Heidelberg, 2009.

[EMFG+14] N. El Mrabet, J. Fournier, L. Goubin, R. Lashermes, and M. Paindavoine. Practical validation of several fault attacks against the miller algorithm. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, 2014.

[EMPV12] Nadia El Mrabet, Dan Page, and Frederik Vercauteren. Fault attacks on pairing-based cryptography. In Marc Joye and Michael Tunstall, editors, *Fault Analysis in Cryptography*, Information Security and Cryptography, pages 221–236. Springer Berlin Heidelberg, 2012.

[Eng13] Andreas Enge. Bilinear pairings on elliptic curves, 2013.

[FIP77] FIPS. Data encryption standard (des): Fips 46, 1977.

[FIP01] NIST FIPS. Advanced encryption standard (aes): Fips 197, 2001.

[FR94] Gerhard Frey and Hans-Georg Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of computation*, 62(206):865–874, 1994.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[Gal05] Steven Galbraith. Pairings. In Ian F Blake, Gadiel Seroussi, and Nigel P Smart, editors, *Advances in elliptic curve cryptography*, volume 317. Cambridge University Press, 2005.

[GE13] Glenn Greenwald and MacAskill Ewen. Nsa prism program taps in to user data of apple, google and others, June 2013. `http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data`.

[GKZ14] Robert Granger, Thorsten Kleinjung, and Jens Zumbrägel. Breaking '128-bit secure' supersingular binary curves (or how to solve discrete logarithms in $F_{2^{4\cdot 1223}}$ and $F_{2^{12\cdot 367}}$). Cryptology ePrint Archive, Report 2014/119, 2014. `http://eprint.iacr.org/`.

[GL09] Conrado Porto Lopes Gouvêa and Julio López. Software implementation of pairing-based cryptography on sensor networks using the msp430 microcontroller. In Bimal Roy and Nicolas Sendrier, editors, *Progress in Cryptology - INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 248–262. Springer Berlin Heidelberg, 2009.

[GMV07] S.D. Galbraith, J.F. McKee, and P.C. Valença. Ordinary abelian varieties having small embedding degree. *Finite Fields and Their Applications*, 13(4):800 – 814, 2007.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[GS11] Dipanwita Roy Chowdhury Ghosh Santosh, Mukhopadhyay Debdeep. Fault attack and countermeasures on pairing based cryptography. *International Journal of Network Security*, 12(1):21–28, 2011.

[GST13] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. `http://eprint.iacr.org/`.

[Hab65] D.H. Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *Nuclear Science, IEEE Transactions on*, 12(5):91–100, Oct 1965.

[Hes08] Florian Hess. Pairing lattices. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography - Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 18–38. Springer Berlin Heidelberg, 2008.

[HNT+13] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. Breaking and entering through the silicon. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &; Communications Security*, CCS '13, pages 733–744, New York, NY, USA, 2013. ACM.

[HPS02] K. Harrison, D. Page, and NP Smart. Software implementation of finite fields of characteristic three, for use in pairing-based cryptosystems. *LMS Journal of Computation and Mathematics*, 5(1):181–193, 2002.

[HSV06] F. Hess, N.P. Smart, and F. Vercauteren. The eta pairing revisited. *Information Theory, IEEE Transactions on*, 52(10):4595–4602, Oct 2006.

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

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference, 1995., Proceedings of Tenth Annual IEEE*, pages 134–147, Jun 1995.

[JOP] Antoine Joux, Andrew Odlyzko, and Cécile Pierrot. The past, evolving present and future of discrete logarithm. `http://www.dtc.umn.edu/~odlyzko/doc/discretelogs2014.pdf`.

[Jou00] Antoine Joux. A one round protocol for tripartite diffie-hellman. In Wieb Bosma, editor, *Algorithmic Number Theory*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–393. Springer Berlin Heidelberg, 2000.

[Jou13] Antoine Joux. A new index calculus algorithm with complexity $l(1/4 + o(1))$ in very small characteristic. Cryptology ePrint Archive, Report 2013/095, 2013. `http://eprint.iacr.org/`.

[JQ01] Marc Joye and Jean-Jacques Quisquater. Hessian elliptic curves and side-channel attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 402–410. Springer Berlin Heidelberg, 2001.

[Ker83] Auguste Kerckhoff. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, 161–191, 1883.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999.

[KKAKJ96] C. Kaya Koc, T. Acar, and B.S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, 1996.

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

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO 96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 1996.

[KSS08] Ezekiel J. Kachisa, Edward F. Schaefer, and Michael Scott. Constructing brezing-weng pairing-friendly elliptic curves using elements in the cyclotomic field. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography - Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 126–135. Springer Berlin Heidelberg, 2008.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer Berlin Heidelberg, 1998.

[KTH+06] TaeHyun Kim, Tsuyoshi Takagi, Dong-Guk Han, HoWon Kim, and Jongin Lim. Side channel attacks and countermeasures on pairing based cryptosystems over binary fields. In David Pointcheval, Yi Mu, and Kefei Chen, editors, *Cryptology and Network Security*, volume 4301 of *Lecture Notes in Computer Science*, pages 168–181. Springer Berlin Heidelberg, 2006.

[LFG13] Ronan Lashermes, Jacques Fournier, and Louis Goubin. Inverting the final exponentiation of tate pairings on ordinary elliptic curves using faults. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 365–382. Springer Berlin Heidelberg, 2013.

[LRD+12] R. Lashermes, G. Reymond, J. Dutertre, J. Fournier, B. Robisson, and A. Tria. A dfa on aes based on the entropy of error distributions. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 34–43, Sept 2012.

[LW10] Allison Lewko and Brent Waters. New techniques for dual system encryption and fully secure hibe with short ciphertexts. In Daniele Micciancio, editor, *Theory of Cryptography*, volume 5978 of *Lecture Notes in Computer Science*, pages 455–479. Springer Berlin Heidelberg, 2010.

[MDH+13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88, Aug 2013.

[Mil86a] Victor S Miller. Short programs for functions on curves, 1986.

[Mil86b] VictorS. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO' 85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Berlin Heidelberg, 1986.

[Mil04] Victor S. Miller. The weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, 2004.

[MNT01] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. New explicit conditions of elliptic curve traces for fr-reduction. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 84(5):1234–1243, 2001.

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

[MOV93] A.J. Menezes, T. Okamoto, and S.A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *Information Theory, IEEE Transactions on*, 39(5):1639–1646, Sep 1993.

[MVOV97] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of applied cryptography*. CRC, 1997.

[NBS08] Michael Naehrig, Paulo S.L.M. Barreto, and Peter Schwabe. On compressible pairings and their computation. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 371–388. Springer Berlin Heidelberg, 2008.

[OGS07] Erdinc Ozturk, Gunnar Gaubatz, and Berk Sunar. Tate pairing with strong fault resiliency. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '07, pages 103–111, Washington, DC, USA, 2007. IEEE Computer Society.

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

[PV06] D. Page and F. Vercauteren. A fault attack on pairing-based cryptography. *Computers, IEEE Transactions on*, 55(9):1075–1080, Sept 2006.

[Riv10] Matthieu Rivain. Promising algorithm for pairing computations. Technical report, CryptoExperts, 2010.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[S+12] W. A. Stein et al. *Sage Mathematics Software (Version 5.5)*. The Sage Development Team, 2012. `http://www.sagemath.org`.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, cetin K. Koc, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer Berlin Heidelberg, 2003.

[SB04] Michael Scott and Paulo S.L.M. Barreto. Compressed pairings. In Matt Franklin, editor, *Advances in Cryptology - CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 140–156. Springer Berlin Heidelberg, 2004.

[SBC+09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In Hovav Shacham and Brent Waters, editors, *Pairing-Based Cryptography - Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin Heidelberg, 2009.

[Sco05] Michael Scott. Computing the tate pairing. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 293–304. Springer Berlin Heidelberg, 2005.

[Sco07] M. Scott. Implementing cryptographic pairings. *Lecture Notes in Computer Science*, 4575:177, 2007.

[Sco11] Michael Scott. On the efficient implementation of pairing-based protocols. Cryptology ePrint Archive, Report 2011/334, 2011. `http://eprint.iacr.org/`.

[Sed88] Holger Sedlak. The rsa cryptography processor. In *Proc. of EUROCRYPT'87*, pages 95–105, Berlin, Heidelberg, 1988. Springer-Verlag.

[Sil09] Joseph H Silverman. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.

[SKW+98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. *NIST AES Proposal*, 15, 1998.

[Ste] Jacques Stern. Evaluation report on the discrete logarithm problem over finite fields.

[STM] STM. Stm32vldiscovery. `http://www.st.com/web/en/catalog/tools/FM116/SC959/SS1532/PF250863?sc=stm32-discovery`.

[STO08] Masaaki Shirase, Tsuyoshi Takagi, and Eiji Okamoto. An efficient countermeasure against side channel attacks for pairing computation. In Liqun Chen, Yi Mu, and Willy Susilo, editors, *Information Security Practice and Experience*, volume 4991 of *Lecture Notes in Computer Science*, pages 290–303. Springer Berlin Heidelberg, 2008.

[TK10] E. Trichina and R. Korkikyan. Multi fault laser attacks on protected crt-rsa. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 75–86, Aug 2010.

[Ver10] F. Vercauteren. Optimal pairings. *Information Theory, IEEE Transactions on*, 56(1):455–461, Jan 2010.

[Wat11] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography - PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 53–70. Springer Berlin Heidelberg, 2011.

[WS06] Claire Whelan and Mike Scott. Side channel analysis of practical pairing implementations: Which path is more secure? In PhongQ. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 2006*, volume 4341 of *Lecture Notes in Computer Science*, pages 99–114. Springer Berlin Heidelberg, 2006.

[WS07] Claire Whelan and Michael Scott. The importance of the final exponentiation in pairings when considering fault attacks. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing-Based Cryptography - Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 225–246. Springer Berlin Heidelberg, 2007.

[YFCV11] G.X. Yao, J. Fan, R.C.C. Cheung, and I. Verbauwhede. A high speed pairing coprocessor using rns and lazy reduction. Cryptology ePrint Archive, Report 2011/258, 2011.

[Yu11] Kewei Yu. Optimal pairings on bn curves. Technical report, University of Waterloo, 2011.

[ZDC+12] Loic Zussa, J.M. Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, et al. Investigation of timing constraints violation as a fault injection means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, 2012.

Appendices

Appendix A

Pairing algorithms

In this appendix, some details about the C implementation of a pairing computation library are described. This library has been kept simple in its implementation and does not represent the state-of-the-art in terms of speed, code size or any other metrics. Yet it is a good representation of what a non-optimized pairing library computing Tate-like pairings would look like. This library has been designed to be highly tweakable, which provoked some unnecessary redefinitions or features (such as a software stack for the *big*s). The algorithm in its most simple functional form may be given rather then the C implementation.

A.1 Preliminaries

The C structures to store the elements in the fields $\mathbb{F}_p, \mathbb{F}_{p^2}, \mathbb{F}_{p^6}, \mathbb{F}_{p^{12}}$ are shown in Code A.1. It is a vector-based representation with 32-bits coordinates.

Code A.1: Field data structures

```
//a big stores a number in Fp
typedef struct
{
    word w[BIGDATA_SIZE_IN_WORDS];
} bigtype;

#define BIGSTRUCT_SIZE_IN_BYTES sizeof(bigtype)
typedef bigtype *big;

//number in Fp2
typedef struct
{
    big a;
    big b;
} zzn2;

//number in Fp6
typedef struct
{
        zzn2 a;
        zzn2 b;
        zzn2 c;
} zzn6;

//number in Fp12
typedef struct
{
    zzn6 a;
    zzn6 b;
```

```
} zzn12;
```

A.2 Operations in $\mathbb{F}_p$

The main operations in $\mathbb{F}_p$ are modular addition, subtraction, multiplication and inversion (cf. Section 2.2.2). To these operations must be added the computation of the Montgomery residue, and the constant *nprime0* used in the modular multiplication. These latter algorithms as well as the inversion algorithms come from [HVM04].

Code A.2: Modular addition

```
//Add two bigs to get a third (c = a+b)
void big_add(big a, big b, big c)
{
    word sum, carry = 0;
    word i;
    for(i = 0; i < BIGDATA_SIZE_IN_WORDS; i++)
    {
        sum = a->w[i] + b->w[i] + carry;
        if(sum > a->w[i]) carry = 0;
        else if(sum < a->w[i]) carry = 1;
        c->w[i] = sum;
    }
}
```

Code A.3: Modular subtraction

```
//Substract the first big by the second to get a third (c= a-b)
void big_sub(big a, big b, big c)
{
    word diff, borrow = 0;
    word i;
    for(i = 0; i < BIGDATA_SIZE_IN_WORDS; i++)
    {
        diff = a->w[i] - b->w[i] - borrow;
        if(diff < a->w[i]) borrow = 0;
        else if(diff > a->w[i]) borrow = 1;
        c->w[i] = diff;
    }
}
```

Code A.4: Modular multiplication

```
//structure for double word packing and unpacking
typedef union
{
    dword d;
    word w[2];
} doubleword;

//(carry, prod) = a*b
void mul2(word a, word b, word* carry, word* prod)
{
    doubleword dble;
    dble.d=(dword)a*b;
    *prod=dble.w[BOTTOM2];
    *carry=dble.w[TOP2];
}


//(carry, accumulator) = a*b + accumulator
```

```
void mul2_acc(word a, word b, word* carry, word* accumulator)
{
    doubleword dble;
    dble.d=(dword)a*b+*accumulator;
    *accumulator=dble.w[BOTTOM2];
    *carry=dble.w[TOP2];
}


//(carry, accumulator) = a*b + carry + accumulator
void mul2_acc_carry(word a, word b, word* carry, word* accumulator)
{
    doubleword dble_ac;
    dble_ac.d=(dword)a*b+*carry+*accumulator;
    *accumulator=dble_ac.w[BOTTOM2];
    *carry=dble_ac.w[TOP2];
}


//(carry, sum) = carry + sum
void add2_self(word* carry, word* sum)
{
    doubleword dble;
    dble.d=(dword)*carry+*sum;
    *sum=dble.w[BOTTOM2];
    *carry=dble.w[TOP2];
}

/*Montgomery multiplication (CIOS) of two bigs to get a third.
c = a*b mod pmngr->modulus in montgomery domain.
pmngr->zzn_mul_helper is a big used as a temporary variable.
pmngr->nprime0 is a constant computed with a dedicated algorithm.
*/
void zzn_mul(big a, big b, big c)
{
    word resS, resSplus1, prod, carry, M, i, j;

    big_zero(pmngr->zzn_mul_helper);

    //fast zero test
    if(a->w[0] == 0 || b->w[0] == 0)
    {
        if((big_is_zero(a) == TRUE) || (big_is_zero(b) == TRUE))
        {
            big_zero(c);
            return;
        }
    }

    resS = 0;
    resSplus1 = 0;
    carry = 0;
    prod = 0;
    M = 0;

    for(i = 0; i < pmngr->modulus_word_count; i++)
    {
        carry = 0;

        for(j = 0; j < pmngr->modulus_word_count; j++)
        {
            mul2_acc_carry(a->w[j], b->w[i], &carry, &(pmngr->zzn_mul_helper->w[j]));
        }

        add2_self(&carry, &resS);
        resSplus1 = carry;

        mul2(pmngr->zzn_mul_helper->w[0], pmngr->nprime0, &carry, &M); //carry not used
```

```
        prod = pmngr->zzn_mul_helper->w[0];
        mul2_acc(M, pmngr->modulus->w[0], &carry, &prod);

        for(j = 1; j < pmngr->modulus_word_count; j++)
        {
            prod = pmngr->zzn_mul_helper->w[j];
            mul2_acc_carry(M, pmngr->modulus->w[j], &carry, &prod);
            pmngr->zzn_mul_helper->w[j-1] = prod;
        }

        add2_self(&carry, &resS);
        pmngr->zzn_mul_helper->w[pmngr->modulus_word_count-1] = resS;
        resS = resSplus1 + carry;
    }

    if(big_compare(pmngr->zzn_mul_helper, pmngr->modulus) >= 0)
    {
        big_sub(pmngr->zzn_mul_helper, pmngr->modulus, pmngr->zzn_mul_helper);
    }

    big_copy(pmngr->zzn_mul_helper, c);
}
```

Code A.5: Nprime0 constant computation

```
//Compute the montgomery multiplication constant nprime0 = Modulus[0]^{-1} mod 2^{32}
word precompute_nprime0()
{
    word sum;
    word y1 = 1, y2 = 0, i, mask, temp;

    for(i = 2; i < WORD_SIZE_IN_BITS; i++)
    {
        sum = (pmngr->modulus->w[0])*y1;
        mask = (1 << i) - 1;
        temp = 1 << (i-1);
        if((sum & mask) < temp)
        {
            y2 = y1;
        }
        else
        {
            y2 = y1 + temp;
        }
        y1 = y2;
    }
    return -y1;
}
```

Let R be the Montgomery residue, in order to convert an element of $\mathbb{F}_p$ from the canonical domain to the Montgomery domain, we must perform a Montgomery multiplication with R^2. The constant $r\_square = R^2$ is computed as follows

Algorithm 8: r_square constant computation.

Result: r_square
$temp \leftarrow 1 \cdot 1 \cdot 1$; `// Montgomery multiplications`
$r\_square \leftarrow temp^{-1}$; `// Binary inversion in the canonical domain`
return r_square

The binary inversion is done with the binary division algorithm shown in Code A.6. The conversion back from the Montgomery domain to the canonical domain is done with a Montgomery multiplication by 1.

Code A.6: Binary division

```
//c = a/b mod modulus (canonical domain)
void zzn_division_classic_domain(big a, big b, big c)
{
    big u, v, x1, x2;

    big_copy(b, u);
    big_copy(pmngr->modulus, v);
    big_copy(a, x1);
    big_zero(x2);

    if(big_is_zero(b) == TRUE)
    {
        pmngr->error = DIVISON_BY_0;
        return;
    }

    while((big_is_one(u) == FALSE) && (big_is_one(v) == FALSE))
    {
        while((u->w[0] & 1) == 0)//u even
        {
            big_self_shift_right(u,1);
            if((x1->w[0] & 1) == 0)
                big_self_shift_right(x1,1);
            else
            {
                big_add(x1, pmngr->modulus, x1);
                big_self_shift_right(x1,1);
            }
        }
        while((v->w[0] & 1) == 0)//u even
        {
            big_self_shift_right(v,1);
            if((x2->w[0] & 1) == 0)
            {
                big_self_shift_right(x2,1);
            }
            else
            {
                big_add(x2, pmngr->modulus, x2);
                big_self_shift_right(x2,1);
            }
        }

        if(big_compare(u, v) >= 0) //u >= v
        {
            big_sub(u,v,u);

            if(big_compare(x1,x2) < 0) // x1 < x2
            {
                big_add(x1, pmngr->modulus, x1);
            }

            big_sub(x1, x2, x1);
        }
        else
        {
            big_sub(v, u, v);

            if(big_compare(x2,x1) < 0) // x2 < x1
            {
                big_add(x2, pmngr->modulus, x2);
            }

            big_sub(x2, x1, x2);
        }
    }
```

```
    if(big_is_one(u) == TRUE)
        big_copy(x1, c);
    else
        big_copy(x2, c);
}
```

Code A.7: Montgomery inversion

```
//Partial montgomery inversion
void zzn_partial_inv(big b, word* k)
{
    big u, v, x1, x2;
    *k = 0;

    big_copy(pmngr->modulus, v);
    big_copy(b, u);

    //1
    big_zero(x1);
    x1->w[0] = 1;

    big_zero(x2);

    while(big_is_zero(v) == FALSE)
    {
        if((v->w[0]&1) == 0)
        {
            big_self_shift_right(v, 1);
            big_self_shift_left(x1, 1);
        }
        else if((u->w[0]&1) == 0)
        {
            big_self_shift_right(u, 1);
            big_self_shift_left(x2, 1);
        }
        else if(big_compare(v, u) >= 0)
        {
            big_sub(v,u,v);
            big_self_shift_right(v, 1);
            big_add(x2,x1,x2);
            big_self_shift_left(x1,1);
        }
        else
        {
            big_sub(u,v,u);
            big_self_shift_right(u, 1);
            big_add(x1,x2,x1);
            big_self_shift_left(x2,1);
        }
        (*k)++;
    }

    if(big_is_one(u) == FALSE) pmngr->error = NO_INVERSE;

    if(big_compare(x1, pmngr->modulus) > 0) big_sub(x1, pmngr->modulus, x1);

    big_copy(x1, b);
}

//Montgomery inversion
void zzn_inv(big b)
{
    word k;
    big temp;

    if(big_is_zero(b) == TRUE)
```

```
    {
        pmngr->error = DIVISON_BY_0;
        return;
    }

    zzn_partial_inv(b, &k);

    if(k == pmngr->rounded_modulus_bit_count)
    {
        zzn_mul(b, pmngr->r_square, b);
    }
    else
    {
        if(k < pmngr->rounded_modulus_bit_count)
        {
            zzn_mul(b, pmngr->r_square, b);
            k += pmngr->rounded_modulus_bit_count;
        }

        zzn_mul(b, pmngr->r_square, b);

        big_zero(temp);
        big_set_bit(temp, 2*(pmngr->rounded_modulus_bit_count) - k, 1);
        zzn_mul(temp, b, b);
    }
}
```

A.3 Operations in Extension Fields

The operations on the extension field L are expressed as operations on the base field K.

A.3.1 Quadratic fields

Algorithm 9: Addition in the quadratic extension L.

Data: $x = x_0 + x_1 \cdot u \in L, y = y_0 + y_1 \cdot u \in L$
Result: $z = x + y \in L$
$z_0 \leftarrow x_0 + y_0$;
$z_1 \leftarrow x_1 + y_1$;
return $z = z_0 + z_1 \cdot u$

Algorithm 10: Multiplication in the quadratic extension L.

Data: $x = x_0 + x_1 \cdot u \in L, y = y_0 + y_1 \cdot u \in L$
Result: $z = x \cdot y \in L$
$t_0 \leftarrow x_0 \cdot y_0$;
$t_1 \leftarrow x_1 \cdot y_1$;
$t_2 \leftarrow x_0 + x_1$;
$t_3 \leftarrow y_0 + y_1$;
$t_4 \leftarrow t_2 \cdot t_3$;
$t_5 \leftarrow t_4 - t_0$;
$z_1 \leftarrow t_5 - t_1$;
$t_6 \leftarrow t_1 \cdot \beta$;
$z_0 \leftarrow t_0 + t_6$;
return $z = z_0 + z_1 \cdot u$

Algorithm 11: Squaring in the quadratic extension L.

Data: $x = x_0 + x_1 \cdot u \in L$
Result: $z = x^2 \in L$
$t_0 \leftarrow x_0^2$;
$t_1 \leftarrow x_1^2$;
$t_2 \leftarrow x_0 + x_1$;
$t_3 \leftarrow t_2^2$;
$t_4 \leftarrow t_3 - t_0$;
$z_1 \leftarrow t_4 - t_1$;
$t_5 \leftarrow t_1 \cdot \beta$;
$z_0 \leftarrow t_0 + t_5$;
return $z = z_0 + z_1 \cdot u$

Algorithm 12: Inverse in the quadratic extension L.

Data: $x = x_0 + x_1 \cdot u \in L$
Result: $y = x^{-1} \in L$
$t_0 \leftarrow x_0^2$;
$t_1 \leftarrow x_1^2$;
$t_2 \leftarrow \beta \cdot t_1$;
$t_3 \leftarrow t_0 - t_2$;
$t_4 \leftarrow t_3^{-1}$;
$y_0 \leftarrow x_0 \cdot t_3$;
$y_1 \leftarrow -x_1 \cdot t_3$;
return $y = y_0 + y_1 \cdot u$

As an example, the corresponding C code is seen on Code A.8.

Code A.8: $\mathbb{F}_{p^2}$ inversion

```
void zzn2_inv(zzn2 *w)
{
    big w1, w2, w6;

    zzn_sqr(w->a,w1);
    zzn_sqr(w->b,w2);

    //beta = 5
    //w1 <- w1 + 5*w2, use w6 as temp variable
    zzn_add(w2, w2, w6);//2*w2
    zzn_add(w6, w6, w6);//4*w2
    zzn_add(w6, w2, w6);//5*w2
    zzn_add(w1, w6, w1);//w1+5*w2

    zzn_inv(w1);

    zzn_mul(w->a,w1,w->a);
    zzn_negate(w1,w1);
    zzn_mul(w->b,w1,w->b);
}
```

A.3.2 Cubic fields

Algorithm 13: Addition in the cubic extension L.

Data: $x = x_0 + x_1 \cdot v + x_2 \cdot v^2 \in L, y = y_0 + y_1 \cdot v + y_2 \cdot v^2 \in L$
Result: $z = x + y \in L$
$z_0 \leftarrow x_0 + y_0$;
$z_1 \leftarrow x_1 + y_1$;
$z_2 \leftarrow x_2 + y_2$;
return $z = z_0 + z_1 \cdot v + z_2 \cdot v^2$

Algorithm 14: Multiplication in the cubic extension L.

Data: $x = x_0 + x_1 \cdot v + x_2 \cdot v^2 \in L, y = y_0 + y_1 \cdot v + y_2 \cdot v^2 \in L$
Result: $z = x \cdot y \in L$
$t_0 \leftarrow x_0 \cdot y_0$;
$t_1 \leftarrow x_1 \cdot y_1$;
$t_2 \leftarrow x_2 \cdot y_2$;
$t_3 \leftarrow x_0 + x_1$;
$t_4 \leftarrow x_0 + x_2$;
$t_5 \leftarrow x_1 + x_2$;
$t_6 \leftarrow y_0 + y_1$;
$t_7 \leftarrow y_0 + y_2$;
$t_8 \leftarrow y_1 + y_2$;
$t_9 \leftarrow t_5 \cdot t_8$;
$t_{10} \leftarrow t_3 \cdot t_6$;
$t_{11} \leftarrow t_4 \cdot t_7$;
$t_{12} \leftarrow t_9 - t_1$;
$t_{13} \leftarrow t_{12} - t_2$;
$t_{14} \leftarrow t_{10} - t_0$;
$t_{15} \leftarrow t_{14} - t_1$;
$t_{16} \leftarrow t_{11} - t_0$;
$t_{17} \leftarrow t_{16} - t_2$;
$z_2 \leftarrow t_{17} + t_1$;
$t_{18} \leftarrow t_{13} \cdot \xi$;
$z_0 \leftarrow t_{18} + t_0$;
$t_{19} \leftarrow t_2 \cdot \xi$;
$z_1 \leftarrow t_{15} + t_{19}$;
return $z = z_0 + z_1 \cdot v + z_2 \cdot v^2$

Algorithm 15: Squaring in the cubic extension L.

Data: $x = x_0 + x_1 \cdot v + x_2 \cdot v^2 \in L$
Result: $z = x^2 \in L$
$t_0 \leftarrow x_0^2$;
$t_1 \leftarrow x_1^2$;
$t_2 \leftarrow x_2^2$;
$t_3 \leftarrow x_0 \cdot x_1$;
$t_4 \leftarrow x_0 \cdot x_2$;
$t_5 \leftarrow x_1 \cdot x_2$;
$t_6 \leftarrow t_2 \cdot \xi$;
$t_7 \leftarrow t_3 + t_3$;
$t_8 \leftarrow t_4 + t_4$;
$t_9 \leftarrow t_5 \cdot \xi$;
$t_{10} \leftarrow t_9 + t_9$;
$z_0 \leftarrow t_0 + t_{10}$;
$z_1 \leftarrow t_6 + t_7$;
$z_2 \leftarrow t_1 + t_8$;
return $z = z_0 + z_1 \cdot v + z_2 \cdot v^2$

Algorithm 16: Inverse in the cubic extension L.

Data: $x = x_0 + x_1 \cdot v + x_2 v^2 \in L$
Result: $y = x^{-1} \in L$
$t_0 \leftarrow x_0^2$;
$t_1 \leftarrow x_1^2$;
$t_2 \leftarrow x_2^2$;
$t_3 \leftarrow x_0 \cdot x_1$;
$t_4 \leftarrow x_0 \cdot x_2$;
$t_5 \leftarrow x_1 \cdot x_2$;
$t_6 \leftarrow \xi \cdot t_5$;
$t_7 \leftarrow t_0 - t_6$;
$t_8 \leftarrow \xi \cdot t_2$;
$t_9 \leftarrow t_8 - t_3$;
$t_{10} \leftarrow t_1 - t_4$;
$t_{11} \leftarrow x_0 \cdot t_7$;
$t_{12} \leftarrow x_2 \cdot t_9$;
$t_{13} \leftarrow \xi \cdot t_{12}$;
$t_{14} \leftarrow x_1 \cdot t_{10}$;
$t_{15} \leftarrow \xi \cdot t_{14}$;
$t_{16} \leftarrow t_{11} + t_{13}$;
$t_{17} \leftarrow t_{16} + t_{15}$;
$t_{18} \leftarrow t_{17}^{-1}$;
$y_0 \leftarrow t_7 \cdot t_{18}$;
$y_1 \leftarrow t_9 \cdot t_{18}$;
$y_2 \leftarrow t_{10} \cdot t_{18}$;
return $y = y_0 + y_1 \cdot v + y_2 \cdot v^2$

A.4 Line evaluations and point operations

The line evaluations and the point operations are combined: the tangent evaluation and the point doubling on one hand, the line evaluation and point addition on the other hand.

Code A.9: Tangent evaluation and point doubling

```
//return T = [2]R and h1(P)
void bnpair_dbl_leval(zzn2 *XR, zzn2 *YR, zzn2 *ZR, big XP, big YP, zzn2 *XT, zzn2 *YT, zzn2 *ZT, zzn2
     *l00, zzn2 *l10, zzn2 *l11)
{
    zzn2 t0,t1, t2, t3, t4, t5, t6, zr2;

    //... allocate zzn2s

    zzn2_sqr(XR, &t0); //t0 = XR*XR
    zzn2_sqr(YR, &t1); //t1 = YR*YR

    zzn2_sqr(&t1, &t2); //t2 = t1*t1

    zzn2_add(&t1, XR, &t3); //t3 = t1+XR
    zzn2_sqr(&t3, &t3); //t3 = t3*t3
    zzn2_sub(&t3, &t0, &t3); //t3 = t3-t0
    zzn2_sub(&t3, &t2, &t3); //t3 = t3-t2
    zzn2_add(&t3, &t3, &t3); //t3 = 2*t3

    zzn2_add(&t0, &t0, &t4); //t4 = 2*t0
    zzn2_add(&t4, &t0, &t4); //t4 = 3*t0

    zzn2_add(XR, &t4, &t6); //t6 = t4 + XR

    zzn2_sqr(&t4, &t5);     //t5 = t4*t4

    zzn2_add(&t3, &t3, &zr2); //zr2 = 2*t3
    zzn2_sub(&t5, &zr2, XT); //XT = t5-zr2

    zzn2_sqr(ZR, &zr2); //zr2 = ZR*ZR
    zzn2_add(YR, ZR, ZT); //ZT = YR+ZR
    zzn2_sqr(ZT,ZT);  //ZT = ZT*ZT
    zzn2_sub(ZT, &t1, ZT); //ZT = ZT-t1
    zzn2_sub(ZT, &zr2, ZT); //ZT = ZT-zr2

    zzn2_add(&t2, &t2, &t2); //t2 = 2*t2X
    zzn2_add(&t2, &t2, &t2); //t2 = 4*t2X
    zzn2_add(&t2, &t2, &t2); //t2 = 8*t2X
    zzn2_sub(&t3, XT, YT); //YT = t3 - XT
    zzn2_mul(YT, &t4, YT); //YT = YT*t4
    zzn2_sub(YT, &t2, YT); //YT = YT-t2

    zzn2_negate(&t4,&t4); //t4 = -t4
    zzn2_mul(&t4, &zr2, &t3); //t3 = t4*zr2
    zzn2_add(&t3, &t3, &t3); //t3 = 2*t3

    zzn2_smul(&t3, XP, l10); //l10 = t3*xp

    zzn2_add(&t1, &t1, &t1); //t1 = 2*t1X
    zzn2_add(&t1, &t1, &t1); //t1 = 4*t1X
    zzn2_sqr(&t6, &t6); //t6 = t6*t6
    zzn2_sub(&t6, &t1, &t6); //t6 = t6-t1
    zzn2_sub(&t6, &t0, &t6); //t6 = t6-t0
    zzn2_sub(&t6, &t5, l11); //l11 = t6-t5

    zzn2_mul(ZT, &zr2, &t0); //t0 = ZT*zr2
    zzn2_add(&t0, &t0, &t0); //t0 = 2*t0

    zzn2_smul(&t0, YP, l00); //l00 = t0*yp
```

```
    //... free zzn2s
}
```

Code A.10: Line evaluation and point addition

```
//return T = R + Q and h2(P)
void bnpair_add_leval(zzn2 *XQ, zzn2 *YQ, zzn2 *XR, zzn2 *YR, zzn2 *ZR, big XP, big YP, zzn2 *XT, zzn2
     *YT, zzn2 *ZT, zzn2 *l00, zzn2 *l10, zzn2 *l11)
{
    zzn2 t0,t1,t3, t4, t5, t6, t9, zr2;

    //... allocate zzn2s

    zzn2_sqr(ZR, &zr2); // zr2 = ZR*ZR
    zzn2_mul(XQ, &zr2, &t0); //t0 = zr2*XQ = XQ*ZR*ZR

    zzn2_add(ZR, YQ, &t1); //t1 = ZR+YQ
    zzn2_sqr(&t1, &t1); //t1 = (ZR+YQ)^2
    zzn2_sqr(YQ, &t3); //t3 = YQ*YQ
    zzn2_sub(&t1, &t3, &t1); //t1 = t1-t3
    zzn2_sub(&t1, &zr2, &t1); //t1 = t1-zr2
    zzn2_mul(&t1, &zr2, &t1); //t1 = t1*zr2

    zzn2_sub(&t0, XR, &t0); //t0 = t0 - XR
    zzn2_sqr(&t0, &t3); //t3 = t0^2

    zzn2_add(&t3, &t3, &t4); //t4 = 2*t3
    zzn2_add(&t4, &t4, &t4); //t4 = 4*t3

    zzn2_mul(&t4, &t0, &t5); //t5 = t4*t0

    zzn2_sub(&t1, YR, &t6); //t6 = t1 - YR
    zzn2_sub(&t6, YR, &t6); //t6 = t1 - 2*YR

    zzn2_mul(&t6, XQ, &t9); //t9 = t6*XQ

    zzn2_mul(&t4, XR, &t1); //t1 = t4*XR

    zzn2_sqr(&t6, XT);     //XT = t6^2
    zzn2_sub(XT, &t5, XT); //XT = XT - t5
    zzn2_sub(XT, &t1, XT); //XT = XT - t1
    zzn2_sub(XT, &t1, XT); //XT = XT - t1

    zzn2_add(ZR, &t0, ZT); //ZT = ZR + t0
    zzn2_sqr(ZT, ZT);     //ZT = ZT^2
    zzn2_sub(ZT, &zr2, ZT); //ZT = ZT-zr2
    zzn2_sub(ZT, &t3, ZT); //ZT = ZT - t3

    zzn2_add(YQ, ZT, &t3); //t3 = YQ+ZT

    zzn2_sub(&t1, XT, &t4); //t4 = t1 - XT
    zzn2_mul(&t4, &t6, &t4); //t4 = t4*t6

    zzn2_mul(YR, &t5, &t0); //t0 = t5*YR
    zzn2_add(&t0, &t0, &t0); //t0 = 2*t0

    zzn2_sub(&t4, &t0, YT); //YT = t4 - t0

    //[this (below) have to be optimised (YQ2 computed for the second time!)
    zzn2_sqr(&t3, &t0); //t0 = t3^2
    zzn2_sqr(YQ, &t3); //t3 = YQ^2
    zzn2_sub(&t0, &t3, &t3); //t3 = t0 - t3
    zzn2_sqr(ZT, &t0); //t0 = ZT^2
    zzn2_sub(&t3, &t0, &t3); //t3 = t3 - t0
    //]

    zzn2_add(&t9, &t9, &t9); //t9 = 2*t9
```

```
    zzn2_sub(&t9, &t3, l11); //l11 = t9 - t3

    zzn2_smul(ZT, YP, &t3); //t3 = ZT*YP (special mul)
    zzn2_add(&t3, &t3, l00); //l00 = 2*t3

    zzn2_negate(&t6, &t6);  //t6 = -t6

    zzn2_smul(&t6, XP, &t1); //t1 = t6*XP (special mul)
    zzn2_add(&t1, &t1, l10); //l10 = 2*t1

    //... free zzn2s
}
```

A.5 Miller algorithm

Code A.11: Miller algorithm (signed)

```
void bnpair_miller(big xp, big yp, zzn2 *XQ, zzn2 *YQ, zzn12 *f, zzn2 *XT, zzn2* YT, zzn2 *ZT)//Ate pairing
{
        sword i;
    zzn2 l00,l10, l11, YQneg;

    //T <- Q
    zzn2_copy(XQ, XT);
    zzn2_copy(YQ, YT);
    //ZT = 1
    zzn2_zero(ZT);
    big_copy(pmngr->one, ZT->a);

    zzn2_negate(YQ, &YQneg);

    //f <- 1
    zzn12_zero(f);
    big_copy(pmngr->one, f->a.a.a);

    for(i = pmngr->T_ate.bit_len - 2; i > -1; i--)
    {

        //f <- f^2*lTT(P); T <- 2T
        bnpair_dbl_leval(XT,YT,ZT,xp, yp, XT, YT, ZT, &l00, &l10, &l11);
        zzn12_sqr(f, f);  //f = f^2
        zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lTT(P)

        if(bnpair_get_sen_bit(&(pmngr->T_ate), i, TRUE))
        {
            bnpair_add_leval(XQ, YQ, XT, YT, ZT, xp, yp, XT, YT, ZT, &l00, &l10, &l11);
            zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lTQ(P)
        }
        else if(bnpair_get_sen_bit(&(pmngr->T_ate), i, FALSE))
        {
            bnpair_add_leval(XQ, &YQneg, XT, YT, ZT, xp, yp, XT, YT, ZT, &l00, &l10, &l11);
            zzn12_specialmul(f, &l00, &l10, &l11, f); //f = f*lT(-Q)(P)
        }
    }
}
```

A.6 Final Exponentiation

Code A.12: Signed fast exponentiation

```
/*
Signed fast exponentiation (square and multiply).
Works only for unitary elements!! (we use conjugation)
*/
void bnpair_signed_fast_exponentiation(zzn12* f, zzn12* result, signed_exponent *sen)
{
    zzn12 fconj;
    sword i;

    zzn12_copy(f, result); //result = f
    zzn12_conj(f, &fconj);

    for(i=sen->bit_len-2; i > -1; i--)
    {

        zzn12_sqr(result, result);
        if(bnpair_get_sen_bit(sen, i, TRUE))
        {
            zzn12_mul(f, result, result);
        }
        else if(bnpair_get_sen_bit(sen, i, FALSE))
        {
            zzn12_mul(&fconj, result, result);
        }
    }
}
```

Code A.13: Final Exponentiation

```
void bnpair_final_exponentiation(zzn12* f)
{
    zzn12 y0, y1, y2, y3;

    zzn12_conj(f, &y1);
    zzn12_copy(f, &y2);
    zzn12_inv(&y2);
    zzn12_mul(&y1, &y2, f); //f = \f * f^-1

    zzn12_copy(f, &y1);
    bnpair_frob_p2(&y1);
    zzn12_mul(&y1, f, f);  //f = f**(p**2) * f

    //y2 = ft
    //y1 = ft3
    bnpair_signed_fast_exponentiation(f, &y2, &(pmngr->t)); //y2 = f**t
    bnpair_signed_fast_exponentiation(&y2, &y3, &(pmngr->t)); //y3 = y2**t = f**(t**2)
    bnpair_signed_fast_exponentiation(&y3, &y1, &(pmngr->t)); // y1 = y3**t = f**(t**3)
    zzn12_copy(&y1, &y0); //y0 <- y1
    bnpair_frob_p(&y0); //y0 = y1**p
    zzn12_mul(&y0, &y1, &y0); //y0 = y0*y1
    zzn12_conj(&y0, &y0); //y0 = conj(y0)
    zzn12_sqr(&y0, &y0); //y0 = y0^2

    //now we can forget f**(t**3)
    zzn12_conj(&y3, &y1); //y1 = conj(y3)
    zzn12_mul(&y0, &y1, &y0); //y0 = y0*y1
    zzn12_copy(&y3, &y1);
    bnpair_frob_p(&y1); //y1 = (f**(t**2))**p

    zzn12_mul(&y1, &y2, &y1); //y1 = y1*y2
    zzn12_conj(&y1, &y1); //y1 = conj(y1)
    zzn12_mul(&y0, &y1, &y0); //y0 = y1*y0
    bnpair_frob_p(&y2); //y2 = y2**p
    zzn12_conj(&y2, &y2); //y2 = conj(y2)
    zzn12_conj(&y3, &y1); //y1 = conj(y3) !!! redundant !!!
    zzn12_mul(&y2, &y0, &y2); //y2 = y0*y2
```

```
    zzn12_mul(&y2, &y1, &y2); //y2 = y2*y1 ->t1
    zzn12_sqr(&y2, &y2); //y2 = y2^2

    bnpair_frob_p2(&y3); //y3 = y3**(p**2)
    zzn12_mul(&y0, &y3, &y0); //y0 = y3*y0
    zzn12_mul(&y0, &y2, &y0); //y0 = y2*y0
    zzn12_sqr(&y0, &y0); //y0 = y0^2

    zzn12_copy(f, &y1);
    zzn12_copy(f, &y2);

    bnpair_frob_p(&y1); //y1 = f**p
    bnpair_frob_p2(&y2); //y2 = f**(p**2)
    zzn12_mul(&y1, &y2, &y1); //y1 = y1*y2
    bnpair_frob_p(&y2); //y2 = f**(p**3)
    zzn12_mul(&y1, &y2, &y1); //y1 = y1*y2
    zzn12_conj(f, &y2); //y2 = conj(f)
    zzn12_mul(&y0, &y1, &y1); //y1 = y1*y0
    zzn12_mul(&y0, &y2, &y2); //y2 = y2*y0
    zzn12_sqr(&y2, &y2); //y2 = y2^2
    zzn12_mul(&y1, &y2, f); //f = y1*y2
}
```

Appendix B

Attack example against the "Coordinates randomization" countermeasure for the Miller algorithm

We compute two random elements a and b in $\mathbb{F}_p$:

$$a = \texttt{0x1765DA9894FB9BD4DEE6662BE31F412412C2E5D9415A82019A1F90475F2A4048},$$
$$b = \texttt{0x1901BA49B75D6B8CEDBDFF1ED3DF4BFF4A5D25AAEC5AFC192F3C28AF8530B302}.$$

We performed the first Miller loop blinding P with a, stopped it at the 30^{th} iteration, and the second blinding P with b and stopped it at the 31^{th} iteration. In order to use the Sage `groebner_basis()` method, we rewrite the identification system in the $\mathbb{F}_{p^{12}}$ basis over $\mathbb{F}_p$.

$$R_0 = \texttt{0x3F30ACF9C5FA43BEC20353D3F982AD9C04A4B78E7E54E2DAA442D317E05CE7A} \ ,$$
$$R_2 = \texttt{0x221B66B52F80A7CCDA7E52AC83A5843A41876D9DE2ABD578A7D508E3B5C01DC8} \ ,$$
$$R_3 = \texttt{0x1BF6E3ADF55B7763A3A53D037786B1E1C2D8A9DC5D445859B0C43A7D0302EA2C} \ ,$$
$$R_8 = \texttt{0xD1620667CAC4A44FFB1F2ADD74C210822FBA3990712DF11BE9C7ACB63F79B31} \ ,$$
$$R_9 = \texttt{0x193588D85075891D21EA3334CFFF04EF375FC35F720E8A36F8DB330D70AA265C} \ .$$

We write the Q coordinates as $X_Q = x_0 + ux_1$ and $Y_Q = y_0 + uy_1$. And finally we have the following system:

$$\begin{cases} f = L(3X_T^4 - 2Y_T^2) - R_0 \\ g = L(-3X_T^2 Z_T^2 x_0) - R_2 \\ h = L(2Y_T Z_T^3 y_0) - R_3 \\ j = L(-3X_T^2 Z_T^2 x_1) - R_8 \\ i = L(2Y_T Z_T^3 y_1) - R_9 \\ k = Y_T^2 - X_T^3 - 5Z_T^6. \end{cases} \tag{B.1}$$

The last polynomial in the Gröbner basis for the lexicographic order ($L < X_T < Y_T < Z_T$) is:

$$Z_T^8 + \texttt{0x480241471C60937DFA39D973151CA499BA4B7D27F3D96ECEE2B0CD3E12028AF} \cdot Z_T^6 +$$
$$\texttt{0x155A67415381E63FC72CF9541218DE8D124185B69986BB9F52AED9FCBFD21221} \cdot Z_T^4 +$$
$$\texttt{0x1BC16E6A2588E7ECA258CA8ED6DC3A075296C372990FAD26194EA5B120C2BFD} \cdot Z_T^2 +$$
$$\texttt{0x4C92AA7C9981349AEAEE1DA5E0F4070811E2564FEBD52F73F617456A571394E}.$$

We used the `factor()` method to obtain the following possibilities for Z_T:
z_{t1} = `0x13DA3A501A33FC911B8B88206435FAD8415C2B9B8A3C127058D0F8C28782FBA5`,
z_{t2} = `0xF96C0B4830D132D32EA927822AF07693BA613A48DC3ED9125AF673D787D045C`.

The first before last and second before last polynomials in the Gröbner basis allow us to obtain the possible Y-coordinates and X-coordinates as they are in the form: $Y_T + \mathrm{P}(Z_T)$ and $X_T + \mathrm{Q}(Z_T)$ where P and Q are polynomial in Z_T only. Thus, we have two possible points for T. At the 30^{th} iteration, $T = [\texttt{1189213705}]P$.

We finally compute the inverse of `1189213705` (mod p) and that gives us the two possibilities for P. We compute the Miller algorithm for both, and obtain the secret point P.