Università degli Studi di Napoli "Federico II"

Corso di Dottorato in Scienze Computazionali ed Informatiche

XXV Ciclo

New experiments for cryptanalysis on elliptic curves

Alberto Pizzirani

Università degli Studi di Napoli "Federico II"

Corso di Dottorato in Scienze Computazionali ed Informatiche

XXV Ciclo

Alberto Pizzirani

New experiments for cryptanalysis on elliptic curves

Advisor

Prof.re Maurizio Laporta

Tutor

Prof.re Massimo Benerecetti

Author's address:

Alberto Pizzirani

e-mail: a.pizzirani@gmail.com

To Gabriella and my family.

Contents

Introduction

The public key cryptosystems play an important role in digital communications. Several public key cryptosystems were proposed in the past, among them the RSA (acronym of the surnames of the authors) by Ron Rivest, Adi Shamir and Leonard Adleman in 1977 and the ECC (Elliptic Curve Cryptography) proposed independently by Victor Miller and Neal Koblitz around 1985. These last two cryptosystems are actually the most popular, even if over the years ECC increased its popularity. Some factors that made the ECC more attractive than RSA are:

- existence of subexponential algorithm (some of them based on elliptic curves) to solve the *integer factorization problem* on which rely the RSA
- more flexibility in the choice of the parameters for the ECC
- a computationally efficient arithmetic on elliptic curves that allows implementations also for devices with low computational power.

This thesis continue and expands the work presented previously in the master thesis [25] and in the articles [22] and [23]. It is focused on improving some tools commonly used for cryptanalytical applications on elliptic curves, and some of them can be applied also when performing modular arithmetic in a more general context than the cryptological one.

In chapter 1 it is given an introduction to the NVidia CUDA programming model and described some problems that can appear while writing code that must run on graphic processing units.

In chapter 2 it is described a full implementation for *single-instruction multiple-data* architectures of a fast modular arithmetic library, with emphasis on the modular inversion. It is presented a variant of the Stein's algorithm that reduces divergence among thread and allows to consider it as a good alternative (for sufficiently big prime fields) to the branch-free algorithm based on Euler-Fermat theorem.

In chapter 3 it is discussed the Rho-Pollard implementation for single-instruction multiple-data architectures that uses the *negation maps*. It is presented also a variant of the classical iterating function of the Rho-Pollard algorithm to reduce the overhead to check for fruitless cycles.

Chapter 4 contains the description of an experimental work performed on SAGE and aimed to apply the Smart's attack on anomalous elliptic curves (defined on prime fields) to a curve defined over a ring $\mathbb{Z}n_1n_2$ with n_1n_2 points. To realize these experiments, the author, implemented into SAGE a complete system of addition laws for elliptic curves over rings, and the functions to perform arithmetic on polyadic numbers.

Acknowledgments

I would like to express my deep gratitude to Professor M. Laporta, my research supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this research work.

I wish to acknowledge the help provided by Professor M. Benerecetti with his valuable and constructive suggestions during the writing of this thesis.

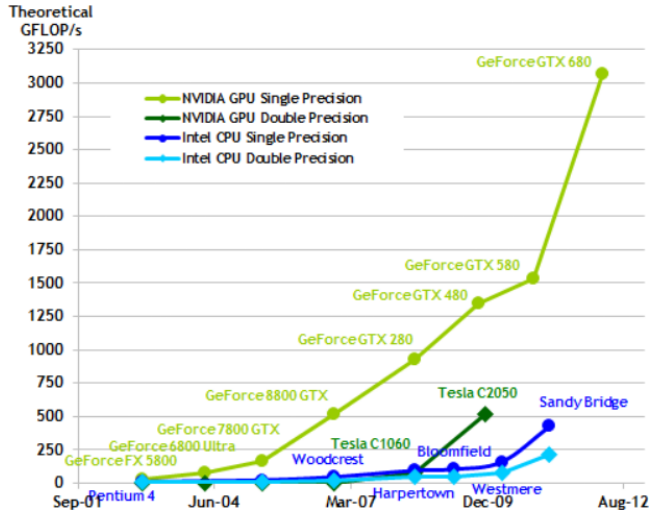
I would also like to offer my special thanks to Professor J.H.Silverman and Professor H.W.Lenstra for their kind and detailed answers to my questions.

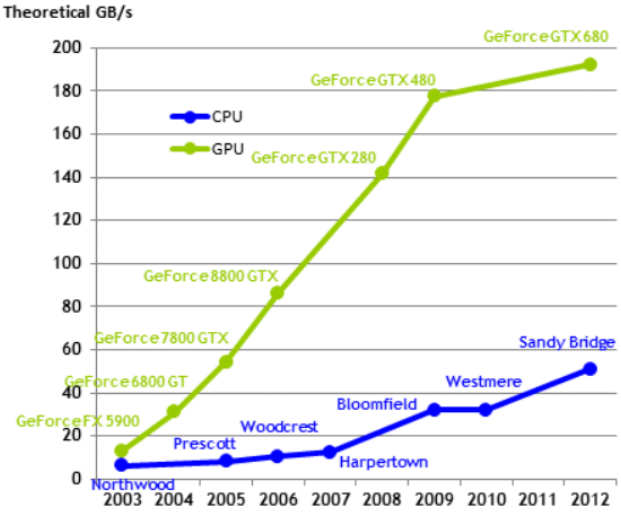
Finally, I wish to thank Gabriella and my parents for their support and encouragement throughout my study.

Chapter 1

High performance computing on graphic processing units

Due to the increasing request from the market of realtime, high-definition 3D graphics, in recent years *Graphic Processor Units* (**GPU**s) have evolved into higly parallel, multithreaded, multicore processors with an extremely high computational power and very high bandwidth as can be clearly seen from the next figures.





Typical problems where the GPUs are well suited are the ones that can be expressed as data-parallel (where the same program is executed on many data elements in parallel), as, for example, in graphic rendering, where the computation of the whole scene can be performed applying the same algorithm to smaller portions of the scene itself.

Anyway, data parallel problems are not restricted to image rendering field, in fact there are many algorithms that take advantage from the data-parallel processing from very different fields: signal processing, physics simulation, computational finance and computational biology.

1.1 CUDA

CUDA is the acronym of **Compute Unified Device Architecture**. It is the programming model created by **NVIDIA** and implemented on their GPUs. This programming model allows developers to access GPUs and the memory on video cards for generic computations like common CPUs. This approach to solve generic problems on GPUs is called **General-Purpose computing on Graphic Processing Units** (**GPGPU**). CUDA platforms can

be programmed through CUDA accelerated libraries available as extensions for industry-standard languages like C/C++ and Fortran or through third-party wrappers for Python, Perl, Java, Ruby, Matlab and Mathematica.

1.1.1 The CUDA data parallel threading model

The core of the parallel programming model of CUDA are three main abstractions:

- a hierarchy of thread groups
- shared memories
- barrier syncronization.

These abstractions are available to the programmer through a minimal set of language extensions and provide an efficient way to finely handle:

- problems that can be split into subproblems that can be solved **indipendently** in parallel by blocks of threads
- subproblems that can be furtherly split into smaller pieces that can be solved **cooperately** by all threads within a block.

This hierarchy of threads allows automatic scalability on different GPUs: each block of threads can be scheduled on any of the **Streaming Multiprocessors** (**SM**s) available on a GPU in any order (sequentially or concurrently) so a CUDA program can execute on GPUs with any number of SM and only the runtime system needs to know the physical number of multiprocessors available.

In CUDA C the programmer defines C functions called *kernels* that, when called, are executed N times in parallel by N different CUDA *threads*.

A kernel is defined with the **__global__** declaration specifier and the number of CUDA threads that execute that kernel is specified within the brackets <<<...>>> between the name of the kernel and the list of arguments in a kernel call.

Each thread has a unique *thread ID* that is accessible within the thread itself through a **threadIdx** variable.

As example, in the following code (taken from [24]):

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
int main() {
        ...
        // Kernel invocation with N threads
        VecAdd<<<1, N>>>(A, B, C);
        ...
}
```

are instantiated N threads, each of them executing VecAdd().

1.1.2 Thread hierarchy and syncronization

Threads are grouped into **blocks**.

Since all the theads into a block are expected to reside on the same processor core and share the limited memory resources of that core, there is a maximum number of threads per block that actually is 1024. Anyway a kernel can be executed by multiple blocks of threads, grouped into a logical structure called **grid**.

Threads (resp. blocks) can be organized into a one, two or three dimensional block (resp. grid).

Blocks of threads must run independently, in this way they can be scheduled on the available SMs in parallel or in series.

Threads are executed in groups of 32 for each streaming multiprocessor. A group of threads is called **Warp**.

Each SM loads a single instruction at time and execute it on all threads running on it, this means that inside a warp each thread do the same thing at the same time.

Threads within a block on the other hand, can cooperate and share some informations through some *shared memory* and synchronize their execution to coordinate memory accesses. The function **__syncthreads()** is used (as the name suggests) to syncronize threads within a block, and acts as a barrier that all thread into a warp must reach before any of them is allowed to continue.

1.1.3 Memory hierarchy

CUDA offers multiple memory spaces to threads, each of them with their own charateristics:

- each thread has access to a **private local memory**,
- each thread into a block has access to a **shared memory** with the same lifetime of the block,
- all threads have access to the same **global memory**,
- two additional read-only memory spaces are available to all threads: **the constant** and **the texture memory spaces**.

The global, constant and texture memory spaces are persistent across kernel launches by the same application.

1.1.4 Memory access

The access to data in the global memory space is a critical factor for the performances of a CUDA application.

Global memory is implemented with dynamical random access memories (DRAM) and the read of it is a very slow process.

Modern DRAMs use a parallel process and each time a memory location is requested, many consecutive locations are read at the same time including the requested one.

Since all thread into a warp execute the same instruction, maximum performances are achieved when all threads into a warp have to read the same location or a group of consecutive locations. When this conditions are met the hardware try to *coalesce* multiple reading requests into a single one. NVidia GPUs can *coalesce* accesses to 8-bit words into a request for 32Byte, 16-bit words into 64Byte, and for 32-bit and 64-bit words 128Byte, if needed it can be issued a smaller request to avoid the waste of bandwidth due to unused words.

Figure 1.1 shows two different ways to store the elements of n vectors of length m that lead to an uncoalesced memory access (a) and a coalesced memory access (b). Element i of vector j is denoted by v_j^i. Each thread in the GPU kernel is assigned to one m-length vector. Threads in CUDA are grouped in an array of blocks and every thread in GPU has a unique id which can be defined as $indx = bd \times bx + tx$, where bd represents block dimension, bx denotes the block index and tx is the thread index in each block. Let's suppose that

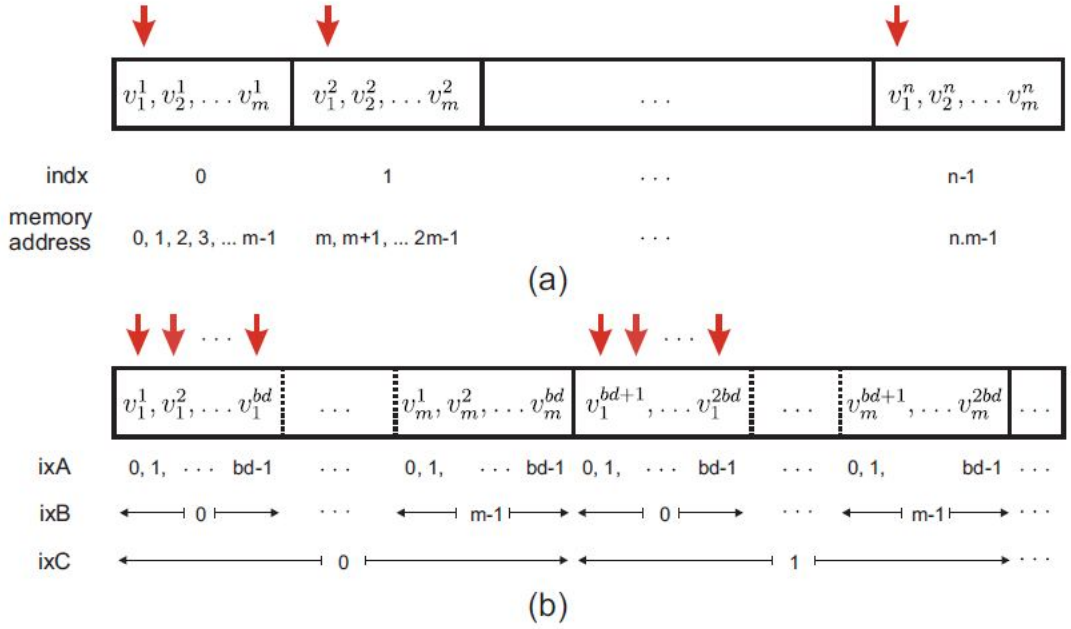


Figure 1.1: Uncoalesced memory access (a) Vs Coalesced memory access (b)

each thread, in parallel, accesses to the first element of its own vector. In the first case (a) with a linear memorization of the vectors, we need to access to addresses $0, m, 2m, \ldots$ of the memory (pointed by arrows). These locations are not consecutive, so the reading requests will read also locations contiguous to the needed ones leading to a waste of bandwidth due to the loading of unused words. If, on the other hand, these elements (first elements of each vector) are packed together like in (b), all of them can be loaded with fewer reading requests and the wasted bandwidth will be much lower (hopefully will be zero). Since the allowed size of residing threads per GPU block is limited to bd, a coalesced data arrangement can be nicely achieved storing the first elements of the first bd vectors in consecutive order, followed by the second elements of the first bd vectors and so on, and starting again with the second db vectors that will be stored in a similar way like shown in figure 1.1 (b). In the linear data storage (a), the component i $(0 \leq i < m)$ of vector $indx$ $(0 \leq indx < n)$ is addressed by $m \times indx + i$; the same component in the coalesced storage pattern (b) is addressed as

$(m \times bd)ixC + bd \times ixB + ixA = bx$, where $ixC = floor[(m \times indx + i)/(m \times bd)]$, $ixB = i$ and $ixA = mod(indx, bd) = tx$. In the coalesced memory storage the access to the elements of the vectors is a little tricky, but leads to a significant higher memory bandwidth of GPU global memory.

1.1.5 Thread divergence

Since each SM loads a single instruction and execute it on all threads running on it, it means that inside a warp each thread do the same thing at the same time.

This lead us to the main cause of performance loss when executing code on GPUs: the presence of the so-called "divergent threads".

The divergence of a thread (or a subset of threads) happens when it needs to execute an instruction different from other threads of the warp.

Let's suppose that we want to execute a simple piece of code like this:

```
...
if(x<0.0)
{
        z=x-2.0;
}
else
{
        z=sqrt(x);
}
...
```

if the x variable is different in various threads, the compiler can't decide if the whole if-else statement can be optimized discarding one branch or the other one, so, through the so called "predicated instructions" mechanism, it is associated a predicate to the condition of the if statement and each instruction of the statement is executed.

```
...
p = (x<0.0);
if (p) z = x-2.0;
if (!p) z = sqrt(x);
...
```

It will be chosen then the correct value of z according the predicate that is true or false.

Chapter 2

Modular arithmetic for multiword integers on GPU

One of the key factors for an efficient implementation of cryptographic primitives or cryptanalytical attacks is an efficient modular arithmetic. On GPUs one can take advantage of the huge processing power available, unfortunately some factors must be taken into account to not waste such a huge power. For the memory accesses, one can adopt some memorization patterns that allow coalesced accesses. For the divergence of threads all algorithms must be evalued carefully to reduce such divergence to the minimum, or completely avoid if possible. In this chapter we describe our implementation of basic modular multiword arithmetic and show a modified algorithm for modular inversion, derived by the Stein's algorithm, that performs better than the Euler-Fermat modular exponentiation (widely adopted since it can be implemented without divergenace) on prime fields that have a characteristic with a binary representation greater than $\tilde{5}00$ bits.

2.1 Addition, subtraction and product

For our implementation we used a radix-32 representation of unsigned integers[1] stored into C arrays following patterns that grant coalesced memory accesses for reading and writing into global memory by all threads.

These patterns will not be considered in the description of the algorithms, so in our pseudocode an integer A in radix-32 representation that needs $Maxlength = \lceil \frac{\log A}{32} \rceil$ words, is stored into an array $A[Maxlength] = a[0], a[1], ..., a[Maxlength - 1]$ where $a[0]$ contains the less significant bits.

In the description of our algorithms P will always be the modulus, A and B will always be the operands, $Maxlength$ is the size in number of words of P.

2.1.1 Addition and subtraction

Sum and difference modulo P are performed in a similar trivial way.

Both use two temporary arrays named $Result_0$ and $Result_1$ of size $Maxlength$ to store intermediate computations and two integers *carry* and *borrow* initially set to 0.

For addition, given A,$B < P$, we perform a cycle of length $Maxlength$ where for each couple of words $a[i]$ and $b[i]$ we store[2] $(carry, result_1[i]) = a[i] + b[i] + carry$ and in $(borrow, result_0[i]) = result_1[i] - p[i] - borrow$.

In a similar way it is performed subtraction modulo P.

In both algorithms, once the *For* cycle ends we return the array $result_{borrow}[]$ of $Maxlength$ words that contains the correct result.

[1] we worked on a NVidia's gpu of the FERMI family, where single word 32-bit multiplication is natively supported and is more efficient than 24-bit multiplication that was natively supported on previous architectures by NVidia.

[2] Carry and borrow generated by single word operands can be computed using a temporary 64-bit word.

Algorithm: Addition(P, A, B)
input : A prime P and integers $A, B \in \mathbb{F}_P$
output: $A + B \mod P$

```
1 carry ← 0;
2 borrow ← 0;
3 result_0[Maxlength] ← 0;
4 result_1[Maxlength] ← 0;
5 for i ← 0 to Maxlength − 1 do
6 |   (carry, result_1[i]) ← a[i] + b[i] + carry;
7 |   (borrow, result_0[i]) ← result_0[i] − p[i] − borrow;
8 end
9 return result_borrow[];
```

Algorithm 1: Addition

Algorithm: Subtraction(P, A, B)
input : A prime P and integers $A, B \in \mathbb{F}_P$
output: $A - B \mod P$

```
1 carry ← 0;
2 borrow ← 0;
3 result_0[Maxlength] ← 0;
4 result_1[Maxlength] ← 0;
5 for i ← 0 to Maxlength − 1 do
6 |   (borrow, result_0[i]) ← a[i] − b[i] − borrow;
7 |   (carry, result_1[i]) ← result_0[i] + p[i] + carry;
8 end
9 return result_borrow[];
```

Algorithm 2: Subtraction

2.1.2 Product

We considered the standard Montgomery product.

Following values (that can be precomputed) are needed:

- $R = 2^{Maxlength*32} (\mathrm{mod}\, P)$
- R^{-1} and P' are the integers for which $RR^{-1} - PP' = 1$ (they can be easily computed through the Extended Euclidean Algorithm)

Into the arrays $R[], R^{-1}[]$ and $P'[]$ (with size $Maxlength$) we store their representation

radix-32 mod P

We call $\overline{A}$ the Montgomery representation of the integer A defined as $\overline{A} = AR(\text{mod } P)$ while Montgomery product is defined as

$\overline{AB}R^{-1}(\text{mod}\, P) = (AR)(BR)R^{-1}(\text{mod}\, P) = ABR(\text{mod}\, P) = \overline{AB}(\text{mod } P)$.

As pointed out in [10] the most efficient implementation of this product is the one that uses the (so called) Coarsely Integrated Operand Scanning approach.

Algorithm: MonPro($P, A, B, p'[0]$)
input : A prime P, integers $A, B \in \mathbb{F}_P$ and value $p'[0]$
output: The Montgomery product of A and B

```
1  carry ← 0, sum ← 0;
2  result[Maxlength + 2];
3  for i ← 0 to Maxlength − 1 do
4  |   carry ← 0;
5  |   for j ← 0 to Maxlength − 1 do
6  |   |   (carry, sum) ← result[j] + a[j] ∗ b[i] + carry;
7  |   |   result[j] ← sum;
8  |   end
9  |   (carry, sum) ← result[Maxlength] + carry;
10 |   result[Maxlength] ← sum;
11 |   result[Maxlength + 1] ← carry;
12 |   carry ← 0;
13 |   m ← result[0] ∗ p′[0] mod 2^32;
14 |   (carry, sum) ← result[0] + m ∗ p[0];
15 |   for j ← 1 to Maxlength − 1 do
16 |   |   (carry, sum) ← result[j] + m ∗ p[j] + carry;
17 |   |   result[j − 1] ← sum;
18 |   |   (carry, sum) ← result[Maxlength] + carry;
19 |   |   result[Maxlength − 1] ← sum;
20 |   |   result[Maxlength] ← result[Maxlength + 1] + carry;
21 |   |   result[] ← Normalize(result[],p[]);
22 |   end
23 end
24 return result[];
```

Algorithm 3: Montgomery product (CIOS)

From the relation $RR^{-1} - PP' = 1$ we have $PP' = -1\ (Mod\ R) = -1\ (Mod\ 2^{32*Maxlength})$ and $p[0]p'[0] = -1\ (Mod\ 2^{32})$. This relation is used into lines from 15 to 22 to compute

$result := result + result[0] * p'[0] * p[]$ equivalent to the partial product modulo P (potentially bigger than R) but with last 32 bit all set to zero (so it can be right shifted without any information loss). A minor improvement can be obtained in CUDA substituting line 12 (where is performed a full 32-bit product and then an addition with carry propagation) with this line:

```
1 carry ← __umulhi(m, p[0]) + (result[0]?1 : 0);
```

where we just compute higher half of the product $m * p[0]$ with the function $__umulhi()$ since we know that the lower half will be 0 when we will add $result[0]$, and this last addition will generate a carry every time $result[0]$ will be different from 0 (from which $result[0]?1 : 0$).

Into line 21 it is called function $Normalize$ that takes in input $result[]$ of size $Maxlength+1$ containing the product $0 \leq \overline{AB} < 2P$ and (similarly to the subtraction) computes $\overline{AB}$ $(Mod\ P)$.

Algorithm: Normalize(P, A)
input : A prime P and an integer $0 \leq A < 2P$
output: $A \bmod P$

```
1  carry ← 0;
2  borrow ← 0;
3  temp ← 0;
4  result_0[Maxlength] ← 0;
5  result_1[Maxlength] ← 0;
6  for i ← 0 to Maxlength − 1 do
7  |  (borrow, result_0[i]) ← a[i] − p[i];
8  |  result_1[i] ← a[i];
9  end
10 (borrow, temp) ← a[Maxlength] − borrow;
11 return result_borrow[];
```

Algorithm 4: Auxiliary function for the Montgomery Product

To convert an integer A into its Montgomery representation, we can use $MonPro$ with R^2 in input as second operand, infact $AR^2R^{-1}(ModP) = AR(ModP) = \overline{A}$. While to convert

$\overline{A}$ back into its "standard" representation we can call $MonPro$ with 1 as second operand.

2.2 Modular inversion

2.2.1 Euler-Fermat

Modular inverse can be computed applying the Euler-Fermat theorem. Working in the $\mathbb{F}_P$ field we have[3] $\forall a \in \mathbb{F}_P$, $a^{\varphi(P)} \equiv a^{P-1} \equiv 1(Mod\ P)$ from which $a^{P-2} \equiv a^{-1}(Mod\ P)$.

This exponentiation can be computed with $O(\log P)$ Montgomery multiplications through a square-and-multiply approach.

Algorithm: EulerFermatInversion(P, A)
input : A prime P and an integer $A \in \mathbb{F}_P$
output: $A^{-1} \bmod P$

```
1  Ptemp[] ← P[] − 2;
2  Ā[] ← MonPro(A, R², P, Maxlength, P′[0]);
3  t[] ← Ā[];
4  mask ← 2^⌊log Ptemp⌋;
5  for i ← 0 to ⌊log Ptemp⌋ do
6  |  currentbit ← Ptemp[] & mask;
7  |  t[] ← MonPro(t, t, P, Maxlength, P′[0]);
8  |  if (currentbit == 1) then
9  |  |  t[] ← MonPro(t, Ā, P, Maxlength, P′[0]);
10 |  end
11 |  mask ← ShiftRight(mask, 1);
12 end
13 t[] ← MonPro(t, 1, P, Maxlength, P′[0]);
14 return t[];
```

Algorithm 5: Inversion through the Euler-Fermat theorem

It is the most used method when computing the inverse on SIMD architectures, indeed, once the modulus is fixed, the algorithm follow the same control flow whatever is the integer $A \in \mathbb{F}_P$ in input.

[3] $\varphi(n)$ is the *Euler's totient function* that gives the number of integers less than n that are coprime with n

Since each Montgomery multiplication has a complexity of $O(\log^2 P)$ and the total number of multiplication is proportional to $\log P$ the algorithm has a cubic complexity.

The worst case is when the prime P is in the form $2^k + 1$, since the binary representation of $P - 2$ is a sequence of all ones.

2.2.2 "Sloppy reduction"

Recently, Bos et al. [6] have proposed a really fast reduction algorithm that exploits the redundant representation of the modulus.

In the resolution of the ECDLP problem over the prime field of 112 bits it appeared that the modulus $P = \frac{2^{128}-3}{11 \cdot 6949}$. They decided to use the reduction modulo $\tilde{P} = 2^{128} - 3$ to perform all the computations and only when needed to check for distinguishing property or partitioning, they switched back to the modulo P representation.

Performing reductions modulo $\tilde{P}$ is extremely fast since $x \cdot 2^{128} (\mathrm{mod}\, \tilde{P}) \equiv 3x (\mathrm{mod}\, \tilde{P})$.

The algorithm is called "sloppy" since it has a negligible probability to produce an incorrect result.

For this reason, as stated in the paper, the algorithm is not suitable for cryptographical application, while it is for cryptanalytical ones.

2.3 Stein's algorithm

The binary gcd (or Stein's algorithm) is a variation of the classical Euclidean algorithm for the greatest common divisor computation.

It is based on four simple observations:

- if a, b are both even, then the $\gcd(a, b) = 2\gcd(a/2, b/2)$
- if a is even and b is odd, then $\gcd(a, b) = \gcd(a/2, b)$
- $\gcd(a, b) = \gcd(a - b, b)$
- if a, b are both odd, then $|a - b|$ is even and less than $\max\{a, b\}$

These observations lead to an extremely simple algorithm that uses only multiplications and divisions by 2 (performed through left and right shifts) and subtractions.

Algorithm: BinaryGCD(A, B)
input : Integers $A, B \in \mathbb{Z}$
output: The greatest common divisor of A and B

```
1  u ← A;
2  v ← B;
3  while u ≡ 0(mod 2) and v ≡ 0(mod 2) do
4  |   u ← u/2;
5  |   v ← v/2;
6  |   e ← 2e;
7  end
8  while u ≠ 0 do
9  |   while u ≡ 0(mod 2) do
10 |   |   u ← u/2;
11 |   end
12 |   while v ≡ 0(mod 2) do
13 |   |   v ← v/2;
14 |   end
15 |   if u ≥ v then
16 |   |   u ← u − v;
17 |   else
18 |   |   v ← v − u;
19 |   end
20 end
21 return v · e;
```

Algorithm 6: Binary GCD

The algorithm has been extensively studied by [17][9][1], its average complexity is $O(\log^2(ab))$ with a constant $c \simeq 0.8$ while in the worst case scenario [17, (66) p.353, (39) p.653] the al-

gorithm performs $\max\{\lfloor \log a \rfloor, \lfloor \log b \rfloor\} + 1$ subtractions and $\log(ab)$ shifts as happens with inputs $a = 2^{n+1} - 2$ and $b = 2^{n+1} - 1$.

Algorithm: ExtBinAlgForInv(P, A)
input : A prime P and an integer $A \in \mathbb{F}_P$
output: $A^{-1} \bmod P$

```
1  u ← A;
2  v ← P;
3  x1 ← 1;
4  x2 ← 0;
5  while u ≠ 1 and v ≠ 1 do
6  |  while u ≡ 0(mod 2) do
7  |  |  u := u/2;
8  |  |  if x1 ≡ 0(mod 2) then
9  |  |  |  x1 ← x1/2;
10 |  |  else
11 |  |  |  x1 ← (x1 + P)/2;
12 |  |  end
13 |  end
14 |  while v ≡ 0(mod 2) do
15 |  |  v ← v/2;
16 |  |  if x2 ≡ 0(mod 2) then
17 |  |  |  x2 ← x2/2;
18 |  |  else
19 |  |  |  x2 ← (x2 + P)/2;
20 |  |  end
21 |  end
22 |  if u ≥ v then
23 |  |  u ← u − v;
24 |  |  x1 ← x1 − x2;
25 |  else
26 |  |  v ← v − u;
27 |  |  x2 ← x2 − x1;
28 |  end
29 end
30 if u == 1 then
31 |  return (x1 mod P);
32 else
33 |  return (x2 mod P);
34 end
```

Algorithm 7: Modular inversion through the Extended Binary GCD

Like the "classical" Euclidean algorithm, also the binary algorithm has an "extended"

version that computes also the so called Bézout coefficient x and y of the equation $ax+by = \gcd(a,b)$.

Working in a field $\mathbb{F}_P$ we know that $\forall a \in \mathbb{F}_P, \gcd(a,P) = 1(\mathrm{mod}\, P)$, we can use this information to compute $a^{-1}(\mathrm{mod}\, P)$. In fact from the Bézout's equation $ax + Py = \gcd(a,P) = 1(\mathrm{mod}\, P)$ from which $ax \equiv 1 - yP(\mathrm{mod}\, P) \equiv 1(\mathrm{mod}\, P)$, and $x = a^{-1}(\mathrm{mod}\, P)$.

This version of the algorithm sadly is not suitable for SIMD architectures since, once fixed the modulus, it follows a different control flow according the binary representation of the given integer to invert.

A trivial observation is that we can obtain an algorithm that follows a unique control flow regardless of the integer to invert, if we always perform the maximum number of loops of the main *while* cycle and for each loop a subtraction.

Of course the subtraction is not needed if we have u or v even, in that case the result of the subtraction is discarded, if a subtraction is needed when the shift function is called, it is called with 0 as number of bits that the operand must be shifted.

The Algorithm 8 is an implementation on SIMD architectures for multiword integers:

Algorithm: SIMDExtBinAlgForInv(P, A)
input : A prime P and an integer $A \in \mathbb{F}_P$
output: $A^{-1} \bmod P$

1 $A_0[] \leftarrow P$;
2 $A_1[] \leftarrow A$;
3 $B_0[] \leftarrow 0$;
4 $B_1[] \leftarrow 1$;
5 $temp[] \leftarrow 0$;
6 $t \leftarrow 0$;
7 $greater \leftarrow 1$;
8 $newgreater \leftarrow 0$;
9 $parity_A \leftarrow 0$;
10 $parity_B \leftarrow 0$;
11 $*targetdifference, *result$;
12 $targetdifference \leftarrow temp$;
13 $result \leftarrow temp$;
14 **for** $j \leftarrow 0$ **to** $2 * \lceil \log P \rceil + 1$ **do**
15 $\quad targetdifference[] \leftarrow temp[]$;
16 $\quad result[] \leftarrow (B_{greater}[] - B_{(1-greater)}[]) \bmod P$;
17 $\quad t \leftarrow A_{greater}[0]\&1$;
18 $\quad A_{greater}[] \leftarrow \texttt{ShiftRight}(A_{greater}[], 1-t)$;
19 $\quad parity_B \leftarrow B_{greater}[0]\&1$;
20 $\quad parity_B \leftarrow t?0 : parity_B$;;
21 $\quad \texttt{AddModulus}(B_{greater}, B_{greater}, P, parity_B)$;
22 $\quad B_{greater}[] \leftarrow \texttt{ShiftRight}(B_{greater}[], (1-t))$;
23 $\quad parity_A \leftarrow A_{greater}[0]\&1$;
24 $\quad newgreater \leftarrow \texttt{GreaterThan}(temp, A_0, A_1, P)$;
25 $\quad greater \leftarrow parity_A?newgreater : greater$;
26 $\quad targetdifference \leftarrow parity_A?A_{greater} : temp$;
27 $\quad result \leftarrow parity_A?B_{greater} : temp$;
28 $\quad result \leftarrow (newgreater \neq 2)?result : temp$;
29 $\quad targetdifference \leftarrow (newgreater \neq 2)?targetdifference : A_1$;
30 $\quad greater \leftarrow (newgreater \neq 2)?greater : 1$;
31 **end**
32 **return** $B_1[]$;

Algorithm 8: Modular inversion through the Extended Binary GCD on SIMD architecture

This algorithm uses some additional functions:

- function *GreaterThan* takes 4 integers as input $Result, A, B, P$ (respectively: the

destination of the result, the two integers that must be compared and the modulus), writes into $Result = |A - B| \bmod P$ and returns $0, 1$ o 2 if $A > B, A < B$ o $A = B$

- function $AddModulus$ takes in input integers $Result, A$ and P and a boolean value $parity$, and returns $Result = A$ if parity is 0 otherwise $Result = A + P$ if parity is 1.

- function[4] $ShiftRight$ takes in input two integers A(in multiword representation or not) and t, and performs a right shift of the array of bits that represents A of t positions to the right.

It is important to note that each branch do not cause divergence since it is translated by the compiler into a single "compare" instruction.

2.3.1 A "reduced divergence" version of the Stein's algorithm

When one of the operands in Stein's algorithm is even, one bit is removed performing a right shift of the operand itself for each iteration of the inner *while* cycle. Since for multiword integer the shift operation is costly and takes the same time if we perform a shift of one or more bits (since it must be shifted the whole array where the operand is stored), it would be nice to remove as many zeroes as possible in the same iteraction of the main *while* cycle.

Instead of performing a loop to detect the number of "trailing zeros" of an operand, a well known trick can be used.

Obtaining the number of zeroes in the least significant position of a non zero integer A is clearly equivalent to obtain the position of the first bit set to 1 starting from the least significant positions of its binary representation.

[4]The function $ShiftLeft$ that will be used in the following algorithms operates in a similar way performing a left shift

Let's consider a one word integer A. First we need to isolate the least significant bit of the integer A. We can do that easily in two's complement representation computing the value $X = A\&(-A)$ in this way we obtain a binary string X that contains a one in the position of the least signficant bit set (if there's one) of the word A.

After that we can obtain the number of trailing zeroes of X (and of course of A) with the help of *De Bruijn's sequences* [21].

A De Bruijn's sequence of length $n = 2^k$ is a cyclic sequence of 0 and 1 where it's possible to find exactly once, as substring, every sequence of length $\log n = k$. An example of length 8 is 00011101. In this string can be recognized as substrings (from left to right) $000, 001, 011, 111, 110, 101$ and, if we wrap around the string, 010, 100.

These sequences have the amazing property that multiplied with a string Y of the same length with just one bit set, they return a new string where the most significant $k = \log n$ bits contains a different value for each string of length n that contains just one bit set. In this way the most significant k bits can be used as a hash value to obtain the number of trailing bits of Y.

So if we have a 32 bits integer A, at first we compute $B = A\&-A$, then we multiply by the corresponding De Bruijn sequence of 32 bit that is the hexadecimal string $077CB531$ (or in binary 00000111011111001011010100110001), then we do a right shift of 27 bits $(32-\log 32)$, now we have a value in the interval $[0, 31]$ that can be used as index into an array of 32 elements to obtain the position of the first bit set to 1 in the string B (that is the same position of the first non zero bit of the string A).

Into line 3 it is returned value 32 if the unsinged integer in input is zero (infact $A = 0$ and $A \equiv 1 \bmod 2$ generate same output after the multiplication by the De Bruijn sequence),

Algorithm: LeastSignificantBit(A)
input : An unsigned (one word) integer A
output: The number of bits set to zero before the first set to one starting from the least significant bit of the integer A. The function return 32 if $A = 0$

1 $MultiplyDeBruijnBitPosition[32] \leftarrow \{0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8, 31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9\}$;
2 $r \leftarrow MultiplyDeBruijnBitPosition[\texttt{ShiftRight}(((A\&-A)*0x077CB531U), 27)]$;
3 **return** $A?r : 32$;

Algorithm 9: Function LeastSignificantBit

in this way we can give in input to the function $LeastSignificantBit$ every 32bits value, without any ambiguity.

With this function we can implement a "new" version of the Stein's algorithm for multi-word integers (Algorithm 10) that removes up to 32bits from one of the two operands (if it is even) within a single iteraction of the *while* loop.

The function $PowerOfTwoDiv$ performs a division of the (multiword) integer taken in input by a power of 2 up to 2^{32}. In this function the value $p'[0]$ is used in a similar way to what has been done for the Montgomery multiplication algorithm for intermediate products.

This version of the Stein's algorithm performs the same operations on all threads until the condition of the *while* loop isn't false. When the condition of the *while* loop is false it calls the function __syncthreads() that puts the thread into a "waiting state" until all other threads in the same block have reached their __syncthreads() call. The worst case of the $DeBruijnBinaryInversion$ algorithm is (of course) still linked to the worst case of the Stein's algorithm and happens (for example) when the modulus is in the form of $2^{n+1}-1$ and the integer to invert is in the form 2^n . In that case the algorithm performs $n + Maxlength$ iterations of the *while* loop (one for each shift+subtraction and one for each shift followed by another shift).

Algorithm: DeBruijnBinaryInversion(P, A)
input : A prime P and an integer $A \in \mathbb{F}_P$
output: $A^{-1} \bmod P$

1 $A_0[] \leftarrow P$;
2 $A_1[] \leftarrow A$;
3 $B_0[] \leftarrow 0$;
4 $B_1[] \leftarrow 1$;
5 $temp[] \leftarrow 0$;
6 $t \leftarrow 0$;
7 $greater \leftarrow 1$;
8 $newgreater \leftarrow 0$;
9 $parity \leftarrow 0$;
10 $*targetdifference$, $*result$;
11 $targetdifference \leftarrow temp$;
12 $result \leftarrow temp$;
13 **while** $greater < 2$ **do**
14 $\quad targetdifference[] \leftarrow temp[]$;
15 $\quad result[] \leftarrow (B_{greater}[] - B_{(1-greater)}[]) \bmod P$;
16 $\quad t \leftarrow$ `LeastSignificantBit`$(A_{greater}[0])$;
17 $\quad A_{greater}[] \leftarrow$ `ShiftRight`$(A_{greater}[], t)$;
18 $\quad$ `PowerOfTwoDiv`$(B_{greater}, t, P, P'[0])$;
19 $\quad parity \leftarrow A_{greater}[0]\&1$;
20 $\quad newgreater \leftarrow$ `GreaterThan`$(temp, A_0, A_1, P)$;
21 $\quad greater \leftarrow parity?newgreater : greater$;
22 $\quad targetdifference \leftarrow parity?A_{greater} : temp$;
23 $\quad result \leftarrow parity?B_{greater} : temp$;
24 **end**
25 __syncthreads();
26 **return** B_1;

Algorithm 10: Modular inversion through the Extended Binary GCD (using De Bruijn sequences and Montgomery arithmetic) on SIMD architecture

The algorithm can be easily modified to operate without synchronization, performing always the maximum number of cycles of the main loop, with a little overhead to save the correct result. This can be useful for cryptographic purposes as countermeasure to side channel attacks.

Algorithm: PowerOfTwoDiv
input : A prime P and an integer $A \in \mathbb{F}_P$
output: $A/2^t \bmod P$

```
1  carry ← 0;
2  m ← (A[0] * P'[0])&(2^t − 1);
3  (MulHi, MulLow) ← m * P[0];
4  (carry, sum) ← MulLow + A[0];
5  A[0] ← sum;
6  carry ← MulHi + carry;
7  for i ← 1 to Maxlength do
8  |  (MulHi, MulLow) ← m * P[i];
9  |  (carry, sum) ← carry + A[i] + MulLow;
10 |  A[i] ← sum;
11 |  carry ← carry + MulHi;
12 end
13 A[] ← ShiftRight(A[], t);
14 A[Maxlength − 1] ← A[Maxlength − 1] + (ShiftLeft(carry, 32 − t));
15 A[] ← (A[] − P[]) mod P;
16 return A[];
```

Algorithm 11: Division modulo P of an integer A by 2

2.3.2 Test of Modular inversion algorithms on CUDA

In this section we examine performances of the various modular inversion algorithms presented (except "sloppy reduction").

Each test was performed on a prime field loading a batch of integers belonging to it. To test the algorithms, we choose the prime fields where are defined some of the problems of the CERTICOM ECC Challenge[11], with bit sizes from 79 bits to 359 bits, and, together with them, we tested the algorithms also on two prime fields with characteristic $2^{521} - 1$ and $2^{607} - 1$ (the 13th and 14th Mersenne prime) to have the worst case of the algorithms based on Binary GCD.

For each batch of integers we perform a modular inversion with each of the algorithms presented earlier. The total amount of integers loaded for each batch is 336 to have one integer to invert for each shader of the GeForce GTX 460 card used in the tests.

Each batch is composed of 335 integers randomly choosen (through the mpz_urandomm function of The GNU Multiple Precision Arithmetic Library) in the interval $[0, P-1]$. Together with them it is loaded also the integer $2^{\lfloor \log P \rfloor}$, in this way the number of subtractions performed by the binary gcds will never be lower than $\mathcal{H}(P)-1$ (where $\mathcal{H}(P)$ is the Hamming weight of P). Table 2.1 shows the timings in milliseconds of each kernel launch:

Field	$\mathcal{H}(P)$	Algorithm 5	Algorithm 8	Algorithm 10	Algorithm 10*
$\mathbb{F}_{p_{79}}$:	39	0.252898	1.089207	0.643887	0.734762
$\mathbb{F}_{p_{97}}$:	46	0.420742	1.868598	1.101820	1.292625
$\mathbb{F}_{p_{109}}$:	61	0.494477	2.126993	1.207894	1.440418
$\mathbb{F}_{p_{131}}$:	61	1.058161	3.389219	1.874419	2.300335
$\mathbb{F}_{p_{163}}$:	84	1.762847	5.194430	2.932256	3.570646
$\mathbb{F}_{p_{191}}$:	101	2.086245	6.109649	3.417678	4.134653
$\mathbb{F}_{p_{239}}$:	132	4.271451	10.159895	5.807271	7.128032
$\mathbb{F}_{p_{359}}$:	165	12.477047	23.607784	13.480877	16.418306
$\mathbb{F}_{p_{521}}$:	521	64.032623	96.310081	68.006874	72.007965
$\mathbb{F}_{p_{607}}$:	607	93.859047	128.207870	88.764221	93.967064

Table 2.1:

- Algorithm 5 = Euler-Fermat.
- Algorithm 8 = Inversion through the Binary GCD.
- Algorithm 10 = Binary GCD with De Bruijn sequences and synchronization.
- Algorithm 10* = Binary GCD with De Bruijn sequences without synchronization.

We underline that while the Euler-Fermat method and the Binary GCD as implemented in the Algorithm 10* (performing the maximum number of iteractions of the main loop) always take the same time, once fixed the modulus regardless the integer to invert, the Binary GCD as implemented in the Algorithm 8 and Algorithm 10 can be faster than the timings shown in the table since performances have been tested in one of the worst cases (the worst for prime fields defined on Mersenne primes).

Some considerations

In this chapter we presented a generic method that do not exploit any optimization based on the particular nature of the prime that defines the field.

The method proposed (unlike the traditional Stein's algorithm) uses also multiplications, anyway the number of single word multiplications performed is $Maxlength + 1$ for each loop of the algorithm (one into the $LeastSignificantBit()$ function and $Maxlength$ into the $PowerOfTwoDiv()$ function), leading again to a quadratic complexity.

Our method is faster than the Euler-Fermat method for sufficiently big prime fields. Of course if we renounce to the generality of the algorithm there are other optimizations that exploit the particular structure of the prime to obtain better performances (like the "sloppy reduction" or methods based on sliding windows).

If modular inversions can be delayed (like in a ρ/λ-Pollard implementation), it is still a good practice to use the method described by Montgomery in [12, p.209], in this way the impact on performance of the inversion operations is ammortized.

Chapter 3

A new iterating function in the rho-Pollard method

The ρ-Polard method is the most efficient algorithm to solve generic ECDLP instances. Further speed up can be obtained parallelizing the algorithm on multiple processor and using automorphisms in the group structure. Unfortunately, while iterating, the algorithm could close into loops that don't provide any information for the ECDLP resolution. These loops are commonly called *fruitless cycles*. In this chapter it is presented a new iterating function for the ρ-Pollard algorithm. This function has the advantage that considerably reduce the overhead to check for fruitless cycles when the algorithm is implemented taking advantage by automorphism of order two also called *negation maps*.

3.1 The discrete logarithm problem

Definition 3.1.1. *Let G be a group with an operation written multiplicatively. The discrete logarithm problem (DLP) to the base $g \in G$ is:*

$$\textit{given } y \in G, \textit{ find an integer } x \textit{ such that } g^x = y. \tag{3.1}$$

Note that such an integer x exists if and only if y belongs to the subgroup of G generated by g, usually denoted by $\langle g \rangle$. For example, if G is the multiplicative group of the positive real numbers, then it is well known that the equation in (3.1) is solvable for any given y with respect to any fixed base $g \neq 1$, but the solution $x = \log_g y$ is not necessarily an integer. On the other side, the word *discrete* indicates that the computational problem under consideration here is not the classical *continuous* case. Namely, (3.1) concerns with a finite cyclic group $\langle g \rangle$, where, by assuming that the order of $\langle g \rangle$ does not exceed n, one may restrict the search for the DL to the interval $0 \leq x \leq n-1$. In particular, given integers a, b that are nonzero modulo a prime p, the classical DLP is to find an integer k such that[1] $a^k \equiv b \bmod p$. Since k is a solution if and only if so is $k+(p-1)$, any DL should be regarded as being defined mod $p-1$, or modulo a divisor d of $p-1$ whenever $a^d \equiv 1 \bmod p$. Analogous considerations hold for the multiplicative group $\mathbb{F}_q^*$ of a finite field. For example, by taking $G = \mathbb{F}_{19}^* := (\mathbb{Z}/19\mathbb{Z})^*$ and $g = 2$ it is readily seen that (3.1) is solvable for $y = 7$ with $x = 6$.

Finally, observe that, if the operation of G is assigned additively, then the equation in (3.1) becomes $xg = y$. In particular, such an additive formulation applies to discrete logarithm problem on the the elliptic curves (ECDLP):

for $P, Q \in E(\mathbb{F}_q)$ find an integer x such that $Q = xP$.

[1] Sometimes we write shortly $a^k \equiv_p b$.

3.1.1 The Pollard rho method for discrete logarithms

There are several cryptographic applications of the DLP, where it turns out that the security of the cryptosystems relies on the difficulty of solving such a problem in a suitable finite group. Trying all possible values of x to solve (3.1) becomes impractical when the solution is an integer of several hundred digits, which is a typical size used in cryptography. Therefore, some less trivial techniques have been introduced to attack DLP of cryptographic interests. Among these one finds the index calculus in the multiplicative group of a finite field [39], that, however, does not work for DLP in a general finite abelian group of a rather large order. In this regard, we remark that discrete logarithms in a finite abelian group[2] G with composite order can be efficiently reduced to multiple instances of discrete logarithms into subgroups of G through the method of Pohlig-Hellman [26]. Also we recall the baby step-giant step method [32] that applies to any G and has both complexity and space requirements $O(\sqrt{|G|})$.

In 1975, John M. Pollard [27] introduced a so called Monte Carlo method to factor a composite integer. Soon it turned out that very similar ideas can be adapted for discrete logarithm computations. Indeed, in [28] there is a first description of the so called *rho algorithm* for the discrete logarithm in the group $\mathbb{F}_p^*$. However, it was immediately clear that the same method is applicable to any finite cyclic group G for which one can assign numerical labels to the group elements and for which it is possible to perform efficiently the group operations and to check whether $y = w$ for any $y, w \in G$. Moreover, Pollard's setting requires that G can be partitioned into r disjoint sets of roughly equal size for a sufficiently small r (the original choice is $r = 3$), so that it can be efficiently determined which of these

[2]Hereafter, when it is not explicitly mentioned, G denotes a finite abelian group.

sets contains any given $y \in G$. It has been observed that in general the algorithms based on the rho method have roughly the same run time of those based on the baby step-giant step method. However, the space requirements of the rho method that in the original settings is $O(\sqrt{|G|})$, can be reduced up to $O(1)$ through some tricks at the expense of few calculations. Moreover, an important feature of the rho method is that DLP computations can be easily distributed to many processors.

The meaning of rho. At the core of the rho algorithms design there is a suitable choice of an iterating function $f : G \to G$ that generates a sequence

$$x_{i+1} = f(x_i)$$

from an arbitrarily chosen $x_0 \in G$. Since G is finite, it is plain that there exist $i_0 < j_0$ such that $x_{i_0} = x_{j_0}$. This immediately yields

$$x_{i_0+1} = f(x_{i_0}) = f(x_{j_0}) = x_{j_0+1}$$

and more in general one gets the *collisions* (or *matches*) $x_{i_0+s} = x_{j_0+s}$ for all integers $s \geq 0$. Thus, the resulting sequence $\{x_i\}_{i\geq 0}$ is cyclic having a divisor of $j_0 - i_0$ (proper or not) as period. One might diagram such a sequence with the Greek letter ρ, whose tail indicates the precyclic part, while the cyclic part is represented by the oval of the letter.

One of the main aspects of such a periodic sequence is that the collisions yield relations between different elements of G. Let us show how in the original rho method introduced by Pollard one could exploit such matches in order to solve instances of the DLP in $\mathbb{F}_p^*$.

The original setting of the rho method. Let us suppose that, given $g, h \in \mathbb{F}_p^* = \{1, 2, \ldots, p-1\}$ for any fixed prime $p > 3$, our goal is to find an integer k such that

$$g^k \equiv h \bmod p.$$

Assuming further that g is a generator of $\mathbb{F}_p^*$ (i.e. $\langle g \rangle = \mathbb{F}_p^*$) and recalling that $g^{p-1} = 1$,

we might think of k as the least nonnegative value within a residue class modulo $p-1$ that we are searching for, and simply write $g^k = h$ in $\mathbb{F}_p^*$. Then, the original iterating function $f_{\mathcal{P}} : \mathbb{F}_p^* \to \mathbb{F}_p^*$ is defined as follows:

$$f_{\mathcal{P}}(x) = \begin{cases} gx & \text{if} \quad x \in T_1 \\ x^2 & \text{if} \quad x \in T_2 \\ hx & \text{if} \quad x \in T_3 \end{cases}$$

where $T_j := [(j-1)p/3, jp/3] \cap \mathbb{F}_p^*$, $j = 1, 2, 3$. In particular, the sequence $\{x_i\}_{i \geq 0}$ generated by $f_{\mathcal{P}}$, with a starting point $x_0 = 1$, is given by

$$x_i = g^{a_i} h^{b_i}$$

where $a_0 = b_0 = 0$ and for any $i \geq 0$ one has

$$\begin{cases} a_{i+1} \equiv_{p-1} a_i + 1 \ , & b_{i+1} = b_i & \text{if} \quad x_i \in T_1 \\ a_{i+1} \equiv_{p-1} 2a_i \ , & b_{i+1} \equiv_{p-1} 2b_i & \text{if} \quad x_i \in T_2 \\ a_{i+1} = a_i, & b_{i+1} \equiv_{p-1} b_i + 1 & \text{if} \quad x_i \in T_3 \end{cases}$$

Since which set T_j an element is in has seemingly nothing to do with the group $\mathbb{F}_p^*$, one may think of the sequence $\{x_i\}_i$ as *random*. Whenever a collision $x_{i_0} = g^{a_{i_0}} h^{b_{i_0}} = x_{j_0} = g^{a_{j_0}} h^{b_{j_0}}$ is found, then by using $g^k = h$ we can write $g^{a_{i_0} + k b_{i_0}} = g^{a_{j_0} + k b_{j_0}}$, that infers

$$a_{i_0} - a_{j_0} \equiv k(b_{j_0} - b_{i_0}) \mod p - 1.$$

Now, it is well known that the latter congruence has the solution

$$k \equiv (b_{j_0} - b_{i_0})^{-1}(a_{i_0} - a_{j_0}) \mod (p-1)/d \, ,$$

where $d := \text{g.c.d.}(p-1, b_{j_0} - b_{i_0})$ and $(b_{j_0} - b_{i_0})^{-1}(b_{j_0} - b_{i_0}) \equiv 1 \bmod (p-1)/d$.

In particular, if d is sufficiently small, then it is possible to verify all the admissible d choices for k:

$$k = (b_{j_0} - b_{i_0})^{-1}(a_{i_0} - a_{j_0}) + m(p-1)/d \quad \text{for some } m = 0, \ldots, d-1 \ .$$

Note that the case $d = 1$ yields immediately the desired solution $k = (b_{j_0} - b_{i_0})^{-1}(a_{i_0} - a_{j_0})$ for the DLP, indicating that we ran across a *golden* collision. On the other side, when $d = p - 1$ the above congruence equation does not allow to determine the discrete logarithm k, and then one has to start over the algorithm to look for another collision. However, it is worthwhile to remark that the case $d < p-1$ occurs with the large probability $(p-2)/(p-1)$. Later we will see that it may be assumed that there is at least a collision before $O(\sqrt{p})$ iterations.

The general setting of the rho method into an additive group. For the sake of clarity, in view of next discussion regarding the ECDLP, let us generalize the Pollard setting of the rho method to a finite abelian group G written additively. So the DLP to the base $P \in G$ is formulated as:

$$\textit{given } Q \in G, \ \textit{find an integer } k \textit{ such that } kP = Q. \tag{3.2}$$

Divide G into three disjoint subsets $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ of approximately the same size $|G|/3$ and such that none of the $\mathcal{T}_i$ is a subgroup of G. Assuming that at random we can choose four integers $a_i, b_i \in \mathbb{Z}/n\mathbb{Z}$, $i = 1, 2$, where $n := |G|$, we define the iterating function $f : G \to G$ as

$$f_{\mathcal{P}}(X) = \begin{cases} X + a_1 P + b_1 Q & \text{if} \quad X \in \mathcal{T}_1 \\ 2X & \text{if} \quad X \in \mathcal{T}_2 \\ X + a_2 P + b_2 Q & \text{if} \quad X \in \mathcal{T}_3 \end{cases}$$

Clearly, from a starting point $a_0 P + b_0 Q \in G$ the sequence generated by $f_{\mathcal{P}}$ is of the type[3]

$$\{u_j P + v_j Q\}_{j \geq 0} \text{ with } u_j, v_j \in \mathbb{Z}/n\mathbb{Z}.$$

Therefore, the recurrence formula becomes

$$u_{j+1} P + v_{j+1} Q = u_j P + v_j Q + a_i P + b_i Q$$

$$\Updownarrow$$

$$(u_{j+1}, v_{j+1}) = (u_j, v_j) + (a_i, b_i) = (u_j + a_i, v_j + b_i) \mod n.$$

Thus, a collision $u_{j_0} P + v_{j_0} Q = u_{i_0} P + v_{i_0} Q$ in G is equivalent to the congruence $k \equiv (v_{j_0} - v_{i_0})^{-1}(u_{i_0} - u_{j_0}) \mod n/d$, where $d := \text{g.c.d.}(n, v_{j_0} - v_{i_0})$.

Of course, at this point we can recover all the remarks previously drafted for the original setting of the rho method. Further, note that here the case n prime is particularly sensitive, because the possible choices for d are solely 1 and n, which correspond to a golden collision and a starting over the algorithm, respectively.

3.1.2 Alternative iterating functions

The randomness of the iterating function. Any iterating sequence of the type described before is a simulation of a *walk* in the group G. The finiteness of G ensures that at some point such a walk becomes a loop. However, the first collision occurrence strictly depends on the iterating function defined on G. For example, given $f : \mathbb{F}_p^* \to \mathbb{F}_p^*$, the largest integer

[3]Note that the straightforward analogous of Pollard's original function is obtained by taken $a_0 = b_0 = 0$, $a_1 = b_2 = 1$ and $a_2 = b_1 = 0$.

i such that for some $x_0 \in \mathbb{F}_p^*$ the elements

$$x_0,\ x_1 = f(x_0),\ x_2 = f(x_1),\ x_3 = f(x_2),\ ...,\ x_{i-1} = f(x_{i-2})$$

are all distinct, is the so-called *epact* of the prime p with respect to f.

In particular, if f is a linear function, i.e. $f(x) = ax + b$, then it is easy to see that the epact of p is p when $p|a-1$ and $p\not|b$, while it is the order of a mod p if $p\ \not|a-1$, and such an order is usually a large divisor of $p-1$. Concerning quadratic functions of the form x^2+c, while it is easily proved that $c = 0, -2$ produce bad epacts, it is still conjectured that for $c = 1$ the epact of p is $O(\sqrt{p \log p})$. Nevertheless, according to next proposition (see [18], Proposition V.2.1) one could give an estimate of how many iterations are required before we encounter the first match.

Proposizione 3.1.1. *Let G^G denote the set of all the maps from a finite set G to itself and for a fixed integer $m \geq 1$ let $G^{G,m}$ be the set of pairs $(x_0, f) \in G \times G^G$ such that the following elements of G are distinct:*

$$x_0,\ f(x_0),\ f(f(x_0)),\ \ldots,\ (\underbrace{f \circ f \ldots \circ f}_{m \text{ times}})(x_0).$$

If $m = 1 + \left[\sqrt{2\alpha|G|}\right]$ for a fixed real number $\alpha > 0$, then

$$\frac{|G^{G,m}|}{|G \times G^G|} = \prod_{j=1}^{m} \left(1 - \frac{j}{|G|}\right) < \frac{1}{e^\alpha}.$$

In particular, note that the latter upper bound does not depend on the cardinality of G, which is generally large in the application.

Somehow, a first motivation of the partitioning $\mathbb{F}_p^*$ into the *rules* T_i in the Pollard original function is an attempt to reduce the possibility of long epacts. A further important reason is that the basic assumption in the analysis of the expected run time of the rho method is

the *randomness* of the walks. Indeed, it appears that the iterating function f and the initial element x_0 must be as much *random* as they can be, for otherwise either the algorithm would fail most of the times or it would succeed for an instance of the DLP just because that instance is not a problem at all. But what does *random* really mean here? According to Teske [37][38], a function is *random* "in the sense that each of the $|G|^{|G|}$ functions $f : G \to G$ is equally probable". Thus, by starting with a point randomly chosen in G according to the uniform probability theory, one generates a so-called[38] *random random walk*. However, since "the very notion of randomness is suspect in the world of Turing machine and serial programs" [13], here we just say that, assuming that somehow the initial element x_0 might be chosen at random in the sense of probability theory, roughly speaking, an *admissible* iterating function should always generate an unpredictable sequence $\{x_i\}_{i\geq 0}$ of elements in G. In other words, *a priori* the length of both the tail and the oval of the resulting ρ diagram should be unknown, and the sole predictable fact should remain the certainty of collisions at some points. Thus, what we always know is that there exist two uniquely determined smallest integers, the *preperiod* $\mu \geq 0$ and the *period* $\lambda \geq 1$, such that $x_i = x_{i+\lambda}$ for every $i \geq \mu$.

Adding walks and mixed walks. The expected value for $\mu + \lambda$ of the idealized version of Pollard's rho method is proved to be close to[4] $\sqrt{\pi|G|/2}$. Thus, the approximate value $1.2533\sqrt{|G|}$ might be assumed to be the average number of generated items needed to have more than 50% chances to generate a collision in an ideal situation. However, with the support of experimental results, Teske [38] observed that the average performance of the

[4]Somehow this is related to the argument of the so called *birthday problem* (known also as *birthday paradox* since the solution is a value incredibly small). The birthday problem is stated as follows: *how many people must be into a room to have a probability higher than 50% that two of them have the same birthday? (considering only day and month of birth, and disregarding leap years).* The answer is $[\sqrt{\pi 365/2}] = 23$.

Pollard classical function, i.e. $f_{\mathcal{P}} : \mathbb{F}_p^* \to \mathbb{F}_p^*$, is worse than expected for a random mapping, namely collisions occur after an average number of $1.37\sqrt{p-1}$ iterations. Even worst is the scenario regarding the generalization of the Pollard function for arbitrary groups (of prime order), namely an experimental average of almost $1.56\sqrt{|G|}$ iterations. In this sense, the walks generated by the Pollard function are in general not *random enough*, from whence the need of alternative iterating functions. Arguing heuristically, Teske showed that remarkable improvements are allowed by increasing the number of the partition sets of the group, namely up to $r = 100$ *rules* instead of $r = 3$ as in the Pollard case. Indeed, with an appropriate choice of the parameters (for example a fixed $r \geq 16$) Teske's new functions yield an average performance that is hardly distinguishable from the performance of a random mapping. For example, in the case of an arbitrary group of rather large order experiments indicate that $r = 20$ leads to a speed-up by about 20% compared to the Pollard function. A more general heuristic due to Brent and Pollard implies that nonrandomness slows down this type of walk by a factor $\sqrt{1 - r^{-1}}$.

Teske classifies the iterating walks as r-*adding* walks and $r + q$-*mixed* walks, where r and $r + q$ denote the number of *rules* for the respective iterating functions. However, while at each step of the first walks it is performed one of the r prescribed additions, the mixed walks include also the possibility of doublings as in the Pollard original setting. We describe here the two type of walks for a finite abelian group G written additively.

Given positive integers r, q, let $M_1, \ldots, M_r \in G$ be randomly chosen and let us take two hash functions,

$$\mathcal{H} : G \to \{1, ..., r\} \qquad \text{and} \qquad \mathcal{K} : G \to \{1, ..., r + q\} \ .$$

The functions $f_{\mathcal{H}} : G \to G$ and $f_{\mathcal{K}} : G \to G$ are said to be respectively *r-adding* and *r + q-mixed* if they are defined as

$$f_{\mathcal{H}}(X) := X + M_{\mathcal{H}(X)}, \qquad f_{\mathcal{K}}(X) := \begin{cases} X + M_{\mathcal{K}(X)} & \text{if} \quad 1 \leq \mathcal{K}(X) \leq r \\ 2X & \text{if} \quad \mathcal{K}(X) \geq r+1 \end{cases} .$$

By setting for every $i \in \{1, ..., r\}, j \in \{1, ..., r+q\}$ the *rules*

$$H_i := \{Y \in G : \ \mathcal{H}(Y) = i\} \qquad \text{and} \qquad K_j := \{Y \in G : \ \mathcal{K}(Y) = j\} \ ,$$

we obtain the more explicit assignments

$$f_{\mathcal{H}}(X) = \begin{cases} X + M_1 & \text{if} \quad X \in H_1 \\ \vdots & \vdots & \vdots \\ X + M_i & \text{if} \quad X \in H_i \\ \vdots & \vdots & \vdots \\ X + M_r & \text{if} \quad X \in H_r \end{cases} \qquad f_{\mathcal{K}}(X) = \begin{cases} X + M_1 & \text{if} \quad X \in K_1 \\ \vdots & \vdots & \vdots \\ X + M_r & \text{if} \quad X \in K_r \\ 2X & \text{if} \quad X \in K_{r+1} \\ \vdots & \vdots & \vdots \\ 2X & \text{if} \quad X \in K_{r+q} \end{cases}$$

Note that the Pollard original function $f_{\mathcal{P}}$ is $2 + 1$-mixed. Further, bearing in mind the DLP (3.2), the points $M_i := a_i P + b_i Q$ are precomputed with random generated $a_i, b_i \in \mathbb{Z}/n\mathbb{Z}$, where $n := |G|$. However, r-adding walks can be computed even if the group order is not known. Indeed, since the increase of the terms in the above additions is linear in the number of iterations, it is not necessary to perform the reduction modulo $|G|$.

On the other side, while in [38] we read that

"In experiments with cyclic elliptic curve (sub)groups, collisions in 20-adding walks occur after an average of $1.26\sqrt{n}$. This is approximately the performance of a random walk."

in [5] it is asserted that

"Additive walks have disadvantages. The walks are noticeably nonrandom and need more iterations than the generic rho method to find a collision. This effect disappears as r grows, but if r is large then the precomputed table M_i does not fit into fast memory. Additive walks also have trouble with automorphisms..."

The last sentence is important for our next discussion on the rho method on equivalence classes. But first let us devote the final part of the present to a short excursion on the problem of the collisions detection and the memory requirements in a rho algorithm implementation.

The collisions detection and the memory requirements. The trivial detection of the collisions consists in storing any element generated while running the algorithm until the next occurrence of a collision (hopefully a *golden* one that helps to solve the DLP). As already mentioned, an algorithm based on the rho method runs in approximately the same time as a baby step-giant step algorithm. However, a trivial detection of the collisions would take around $\sqrt{|G|}$ storage, which is similar to baby step-giant step. Nevertheless, at the cost of a little more computation, a strong improvement concerning the space complexity for the rho method with respect to the naive collisions detection can be obtained through an expedient, the so-called *Floyd cycle-finding* method, also known as *the tortoise and the hare* algorithm, that we describe as follows.

Beyond the usual sequence generated by the iterating function f applied once at each step, it is required to generate another sequence by a double application of f. That is

$$X_{j+1} = f(X_j) \qquad (\textit{the tortoise walk})$$

$$X_{2(j+1)} = f(f(X_{2j})) \ (\textit{the hare walk})$$

Clearly, the hare sequence is a subsequence of the tortoise one. The key observation is that once there is a collision for two indices differing by the period of the tortoise sequence, all subsequent indices differing by such a period will yield collisions. More precisely, if $\mu \geq 0$ and $\lambda \geq 1$ are respectively the preperiod and the period of the first sequence, then one has

$$j \geq \mu \wedge \lambda | j \Longrightarrow X_{2j} = X_j.$$

Hence, the Floyd method requires the computation of the pairs (X_j, X_{2j}) for $j = 0, 1, \ldots$, but only the storage of the current pair at each step. With such a simple expedient the space requirements are lowered to $O(1)$ at the cost of the extra computation of the hare sequence.

Another method of finding a match consists in storing only points that satisfy a certain property (call them *distinguished points*), that should be easily testable to avoid an high overhead on computations. A typical distinguishing requirement is the last w bits of the binary representation of the x-coordinate to be 0. Thus, on the average, one out of every 2w points X_i is stored. If X_i is not one of the distinguished points, but it is involved in a collision like $X_i = X_j$, then one might expect X_{i+s} to be a distinguished point for some s such that $1 \leq s \leq 2^w$, approximately. Consequently, one gets the collision $X_{i+s} = X_{j+s}$ between distinguished points with only a little more computation. Note that on one hand a small value of w gives more chances to detect a collision at an earlier step, on the other it increases the number of stored points. Clearly, for $w = 0$ the space complexity does not differ from the trivial detection.

3.2 The rho method via the negation map

3.2.1 The rho method on equivalence classes

An alternative use of the rho method on a finite abelian group G is to iterate a walk trough the equivalence classes introduced by a given equivalence relation on G. In other words, we can iterate a function on the quotient group of G with respect to such a relation. Since the cardinality of the quotient group, i.e. the number of the equivalence classes, is of the type $|G|/m$, a collision between equivalence classes should be found in an ideal time $\sqrt{\pi|G|/(2m)}$, which improves on $\sqrt{\pi|G|/2}$ plainly when $m > 1$.

Equivalence classes by automorphisms. A way to introduce equivalence relations on G is by using automorphisms of G. Assuming that G is written additively, recall that an automorphism of G is a bijective map

$$\sigma : G \to G \text{ such that } \sigma(X + Y) = \sigma(X) + \sigma(Y) \text{ for all } X, Y \in G.$$

The set of the automorphisms of G is denoted by $Aut(G)$ and the map composition makes $Aut(G)$ a group of order $|G|$. Samples of automorphisms are the *identity* map and the *negation* map, respectively

$$id : X \in G \to id(X) = X \in G,$$

$$-id : X \in G \to -id(X) = -X \in G.$$

By definition $\sigma^0 = id$ for every $\sigma \in Aut(G)$. The order of $\sigma \in Aut(G)$ is the least nonnegative integer m such that

$$\sigma^m := \underbrace{\sigma \circ \sigma \ldots \circ \sigma}_{m \text{ times}} = id.$$

In particular, the negation map has order 2 being plain that $(-id)^2 = id$. Any $\sigma \in Aut(G)$ induces an equivalence relation in G:

$$X \sim Y \iff Y = \sigma^i(X) \text{ for some nonnegative integer } i,$$

and an equivalence class is denoted as $\overline{X} = \{Y \in G : \ Y \sim X\}$.

If the order of σ is m, then the quotient group of G with respect to the equivalence relation induced by σ has cardinality $|G|/m$. As said, in this case the rho algorithm applied to the equivalence classes would have an expected speed-up of $\sqrt{m}$. However, there are basic requirements for the method to work efficiently like the relatively easiness of computing equivalence classes. In this regard, the negation map is with no doubt the most simple one.

Iterating functions on equivalence classes. We could choose an iterating function f that operates as usual trough the group G, while we could search for the matches on the equivalence classes only: if the points P_i, P_j of an iterating sequence generated by f are in the same class, i.e. $\overline{P_i} = \overline{P_j}$, then we try to find a collision between elements of the class. However, in order to avoid non-deterministic path when one has to parallelize the algorithm (see [16]), it is advisable to look for iterating functions well defined on the equivalence classes, that is $\overline{f(X)} = \overline{f(Y)}$ whenever $\overline{X} = \overline{Y}$, and to iterate as $P_{i+1} \in \overline{f(P_i)}$.

3.2.2 The inverse-point strategy for the ECDLP

It is well known that in the case of an elliptic group $E(\mathbb{K})$, the order of its automorphism group is a divisor of 24, depending on the characteristic char($\mathbb{K}$) of the field $\mathbb{K}$ and the so called *j-invariant* $j(E)$ of $E(\mathbb{K})$ (see [33], Ch.3, Theorem 10.1). More precisely, for

$j_0 := 3 \cdot 24^2 = 1728$ one has

$$|Aut(E(\mathbb{K}))| = \begin{cases} 2 & \text{if} \quad j(E) \neq 0, j_0 \\ 4 & \text{if} \quad j(E) \neq j_0, \ \text{char}(\mathbb{K}) \neq 2, 3 \\ 6 & \text{if} \quad j(E) \neq 0, \ \text{char}(\mathbb{K}) \neq 2, 3 \\ 12 & \text{if} \quad j(E) = 0 = j_0, \ \text{char}(\mathbb{K}) = 3 \\ 24 & \text{if} \quad j(E) = 0 = j_0, \ \text{char}(\mathbb{K}) = 2 \end{cases}$$

For the *anomalous curve* (or *Koblitz curves*) over binary extensions fields $\mathbb{F}_{2^m}$, Wiener and Zuccherato [40] exploit the Frobenius automorphism of degree m in the rho algorithm application to obtain a speed-up by a factor of up to $\sqrt{2m}$. In the same paper, an *inverse-point strategy* is introduced in order to exploit the simplicity of the negation map of an elliptic group over a finite field $\mathbb{K} = \mathbb{F}_{p^m}$. Since such an automorphism has order 2, the strategy allows a speed-up by a factor of up to $\sqrt{2}$ in the application of the rho algorithm to the ECDLP. As already observed, through the equivalence relation induced by the negation map each group element P is paired with its inverse $-P$ in order to halve the search space of the algorithm.

Within the implementation of the Pollard rho algorithm for the ECDLP via the negation map, the equivalence class $\overline{P} = \{P, -P\}$ is usually represented by the point with y-coordinate of least absolute value.

3.2.3 A technicality of the strategy: fruitless cycles

Dealing with equivalence classes, it is not surprising that the most obvious usage of the automorphisms in the context of the Pollard rho method leads to useless cycles trapping the

random walks, that is the iterating process falls into a loop that keeps on generating always the same equivalence class. Commonly called *fruitless cycles*, such loops do not provide any useful information for the ECDLP resolution right after their first occurrence.

The occurrence of 2*-cycles.* If the rho algorithm is implemented via the negation map, i.e. the inverse-point strategy, it is easy to explain why fruitless cycles of length 2 might appear. In this regard, let us consider a r-adding function $f_{\mathcal{H}}$ defined on an elliptic group $E(\mathbb{F}_{p^m})$. Further, let us assume that by the negation map each $P = (x, y) \in E(\mathbb{F}_{p^m})$ is identified with $-P$, but $f_{\mathcal{H}}$ is always applied to the representative of the class $\overline{P}$ with the least y-coordinate. Thus, it might happen that two consecutive points, R_i and R_{i+1} say, of an iterating sequence generated by $f_{\mathcal{H}}$ satisfy both the following properties:

1) $\mathcal{H}(R_i) = \mathcal{H}(R_{i+1}) = j$, i.e. $R_i, R_{i+1} \in H_j$, that yields

$$R_{i+1} \in \overline{R_i + M_j} \quad \text{and} \quad R_{i+2} \in \overline{R_{i+1} + M_j}$$

2) $R_{i+1} = -(R_i + M_j)$ is the representative of the class $\overline{R_i + M_j}$ that has to be chosen in order to calculate $R_{i+2} = f_{\mathcal{H}}(R_{i+1})$

Consequently, it turns out that $R_{i+2} = f_{\mathcal{H}}(-R_i - M_j) = -R_i - M_j + M_j = -R_i$, revealing the occurrence of the 2-cycle $\overline{R_{i+2}} = \overline{R_i}$.

The probability of the fruitless cycles. More in general, a rho algorithm implemented with a r-adding function $f_{\mathcal{H}}$ on the equivalence class defined by a $\sigma \in Aut(E(\mathbb{F}_{p^m}))$ might be affected by fruitless cycles of length t like

$$R_1,\ R_2,\ ...,\ R_t,\ R_{t+1} = R_1,$$

that can be analyzed as follows. Let us set $r_i := \mathcal{H}(R_i)$ for $i = 1, 2, \ldots, t$ and, if R_{i+1} is picked as the representative of the class $\overline{f_{\mathcal{H}}(R_i)} = \overline{R_i + M_{r_i}}$ via the automorphism $\varepsilon_i := \sigma^{m_i}$, then we write

$$[\varepsilon_i](R_i + M_{r_i}) := R_{i+1}$$

Thus, it is easy to see that

$$[\varepsilon_t^{-1}]R_{t+1} = [\varepsilon_{t-1}...\varepsilon_1]R_1 + T$$

where $T := [\varepsilon_{t-1}...\varepsilon_1]M_{r_1} + [\varepsilon_{t-1}...\varepsilon_2]M_{r_2} + \ ... \ + [\varepsilon_{t-1}]M_{r_{t-1}} + M_{r_t}$.

Thus, this is a fruitless cycles of length t if $T = O$, where O is the infinity point of the curve. As already said, we may assume that the points M_j are randomly chosen, so that there is no trivial relation between them involving powers of σ. If this is the case, then a necessary condition for having $T = O$ is that for all $i \in \{1, 2, ..., t\}$ exists a $j \neq i$ such that $\mathcal{H}(R_i) = \mathcal{H}(R_j)$.

Clearly, the speedup of the rho method via the automorphisms depends on the expected number of t-cycles. In this regard, let us quote Proposition 31 from [16].

Proposizione 3.2.1. *If $\mathcal{P}(t)$ denotes the probability that a useless t-cycle occurs when a r-adding function is iterated on the equivalence classes defined by $\sigma \in Aut(G)$ of order m, then*

$$\mathcal{P}(t) \leq \sum_{k=1}^{\min(r,t/2)} \frac{1}{m^k} \min\left(1, \frac{r!k^t}{(r-k)!r^t}\right) .$$

More precisely, for $t \in \{2, 3, 4\}$ one has

t	pattern	$\mathcal{P}(t)$
2	$r_1 = r_2$	$= (mr)^{-1}$
3		$= 0$ *if* $3 \not\mid m$, $\leq 2(mr)^{-2}$ *if* $3 \mid m$
4	$r_1 = r_2 = r_3 = r_4$ $r_1 = r_1 = r_3, r_2 = r_4$	$\leq (1 - 1/m)^2 m^{-1} r^{-2}$ $= (1 - 1/r)(mr)^{-2}$

Note that the expected number of t-cycles is $\mathcal{P}(t)t|G|/m$, and in particular for $t = 2$ it is $|G|/2r$. Further, observe that the probability to find a t-cycle reduces as r and t increase. On the other side, large values of r are impractical, while the most promising medium values are not always compatible with all environments (see [6]). So the detection of the cycles occurrence and recurrence and their treatment are key topics in the development of the rho algorithms when dealing with negation maps or other automorphisms.

Eliminating the fruitless cycles. In the literature there are many concrete proposals on the use of the negation map in the Pollard rho method. Mostly, they are a combination of several ideas for avoiding, detecting, and escaping fruitless cycles. A good review and comparison of such proposals is the Appendix of [5], where the authors remark that none of the proposed algorithms perform efficiently on SIMD architectures because of frequent conditional operations.

In the same paper [5] an alternative method to detect and escape fruitless cycles is deeply discussed. Basically it requires checking from time to time fruitless 2-cycles, while longer cycles are checked even more rarely, because 2-cycles appear with the larger probability $(2r)^{-1}$ according to the previous Proposition. Further, 2-cycles persist after they appear, wasting subsequent iterations (in the sense that new points and new collision opportunities do

not occur), until we check for them. A planned check every w iterations yields approximately the probability $w(2r)^{-1}$ that a 2-cycle appears, and for it we expect to waste approximately $w/2$ iterations on average if it does appear. However, a sharply small value for w does not become necessary here. Indeed, if a cycle has not appeared, then checking for it wastes an iteration. Thus, the total of wasted iterations amounts to one iteration for the check plus the average number of wasted iterations multiplied by the probability that a 2-cycle occurs:

$$1 + (w/2)w/(2r) = 1 + w^2/(4r) \text{ out of } w.$$

An optimization of the quotient $(1 + w^2/(4r))w^{-1} = 1/w + w/(4r)$ requires w to be very close to $2\sqrt{r}$.

More generally, fruitless cycles of relatively small length t appear with probability approximately proportional to $(\sqrt{r})^{-t}$, so the optimal checking frequency is approximately proportional to $(\sqrt{r})^{-t/2}$, which rapidly goes to zero as t increases.

In [5] the authors also claim that their PlayStation 3 software takes $r = 2048$ checks for 2-cycles every 48 iterations, and checks for larger cycles with even lower frequency. To simplify the software they unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations, whereas the choice $r = 512$ requires checking for 2-cycles every 24 iterations.

The occasional check for 2-cycles works as follows. One computes R_{i+2} and checks if it is equal to R_i. Then, the definition of R_{i+3} changes according to the two possible cases (but the first one is the most probable):

$$R_{i+3} := \begin{cases} R_{i+2} & \text{if} \quad R_{i+2} \neq R_i \\ \|2\min(R_{i+2}, R_{i+1})\| & \text{if} \quad R_{i+2} = R_i \end{cases}$$

where $\min(R_{i+2}, R_{i+1})$ is taken in the lexicographic sense and the symbol $\|R\|$ means that one has to take the point in the class $\overline{R}$ according to a prefixed canonical selection (such as the least y-coordinate or the parity of the y-coordinate). Similarly, one proceeds in case of fruitless cycles of lengths $6, 8$, etc., up to the smallest even length that exceeds $(\log n)/(\log r)$, where n is the order of the elliptic group. This escaping technique was used earlier by Escott ([14]).

Now observe that, from a point on the cycle, $R = aP + bQ$ say, the subsequent points are generated by adding one of the $M_i = a_iP + b_iQ$ or by doubling, and negating if needed. If $c \geq 1$ is the number of doublings in the cycle, then we get a relation of the form

$$R = \pm 2^c R + \sum_{i=0}^{r-1} c_i M_i = \pm 2^c R + \sum_{i=0}^{r-1} c_i a_i P + \sum_{i=0}^{r-1} c_i b_i Q$$

with $c_i \in \mathbb{Z}$, that yields

$$\Big((1 \mp 2^c)a - \sum_{i=0}^{r-1} c_i a_i\Big)P + \Big((1 \mp 2^c)b - \sum_{i=0}^{r-1} c_i b_i\Big)Q = O \ .$$

Since $1 \mp 2^c \neq 0$, the expression$\big((1 \mp 2^c)a - \sum_{i=0}^{r-1} c_i a_i\big)$ is most likely not divisible by the group order. This is the argument that Bos, Kleinjung and Lenstra ([7]) followed to get the following heuristic: *a cycle with at least one doubling is most likely not fruitless.*

3.3 A new iterating function

Anything that breaks linearity would be convenient.

This is what Duursma, Gaudry and Morain claim while they tell about *Taking care of useless cycles* in [16]. So we try to *break linearity* in a linear way by proposing the following

iterating function for the Pollard rho algorithm. It is a sort of $r+1$-mixed function, but has two variables: the first one for an element of the group and the other for the step number of the iteration.

More precisely, for a fixed integer $r \geq 3$ let $M_1, \ldots, M_r$ be preassigned points of an elliptic group $E(\mathbb{F}_{p^m})$. Then, we set

$$F_{\mathcal{H}}(X,i) := \begin{cases} X + M_{\mathcal{H}(X)+i} & \text{if} \quad i \not\equiv 0 \bmod r \\ 2X & \text{if} \quad i \equiv 0 \bmod r \end{cases}$$

where the hash value $\mathcal{H}(X)$ is the least positive residue mod r of the $x-$coordinate of X and $\mathcal{H}(M_k) \equiv_r k$ (hereafter, the index k of M_k is always reduced mod r).

From a starting pair $(X_0, 1)$ with a randomly chosen $X_0 \in E(\mathbb{F}_{p^m})$, the initial $r+1$ steps of the iterating sequence $X_{i+1} := F_{\mathcal{H}}(X_i, i+1)$ are

$$X_0 \overset{X_0+M_{x_0+1}}{\longrightarrow} X_1 \overset{X_1+M_{x_1+2}}{\longrightarrow} X_2 \longrightarrow \ldots \longrightarrow X_i \overset{X_i+M_{x_i+i+1}}{\longrightarrow} X_{i+1} \longrightarrow \ldots$$

$$\ldots \longrightarrow X_{r-2} \overset{X_{r-2}+M_{x_{r-2}+r-1}}{\longrightarrow} X_{r-1} \overset{2X_{r-1}}{\longrightarrow} X_r \overset{X_r+M_{x_r+1}}{\longrightarrow} X_{r+1} \ldots$$

where for brevity we have set $\mathcal{H}(X_j) := x_j$. More in general, for any index $i \geq 0$ and any integer $s \geq 1$ one has

$$X_{i+s} = X_i + \sum_{j=i}^{i+s-1} Y_j \ , \tag{3.3}$$

where

$$Y_j := \begin{cases} M_{x_j+j+1} & \text{if} \quad j+1 \not\equiv 0 \bmod r \\ X_j & \text{if} \quad j+1 \equiv 0 \bmod r \end{cases} .$$

The period of the iterating sequence. From (3.3) it follows immediately that a collision

$X_{i+s} = X_i$ occurs if and only if

$$\Delta_s(X_i) := \sum_{j=i}^{i+s-1} Y_j = O \ .$$

Moreover, for every $i \geq 0$ and every $s \geq 1$ it turns out that

$$\Delta_s(X_{i+1}) = \Delta_s(X_i) + Y_{i+s} - Y_i \ .$$

Thus, if we assume that $X_{i+s} = X_i$ for some $s \equiv 0 \bmod r$, it is easily seen that $Y_{i+s} = Y_i$, which in turn yields $\Delta_s(X_{i+1}) = O$. Therefore,

$$X_{i+s} = X_i \text{ for some } s \equiv_r 0 \Longrightarrow X_{i+s+1} = X_{i+1} \ .$$

More in general, by induction on k we have the following implication between collisions of points at distance $s \equiv 0 \bmod r$:

$$X_{i+s} = X_i \Longrightarrow X_{i+s+k} = X_{i+k} \text{ for every } k \geq 0 \ .$$

The above discussion and next proposition show that the period of the sequence X_j has to be necessarily a multiple of r.

Proposizione 3.3.1. *Let $s \not\equiv 0 \bmod r$. Then*

$$X_{i+s} = X_i \Longrightarrow X_{i+s+1} \neq X_{i+1} \ . \tag{3.4}$$

Proof. Let us assume that (3.4) is not true for some i, i.e. one has the simultaneous

occurrence of the matches

$$X_{i+s} = X_i \text{ and } X_{i+s+1} = X_{i+1} \ . \tag{3.5}$$

This is equivalent to $\Delta_s(X_i) = \Delta_s(X_{i+1}) = O$, that implies $Y_{i+s} = Y_i$.

Now, since $s \not\equiv 0 \bmod r$, the only possible cases are:

1) $i + 1 \not\equiv_r 0$ and $i + s + 1 \not\equiv_r 0$

2) $i + 1 \not\equiv_r 0$ and $i + s + 1 \equiv_r 0$

3) $i + 1 \equiv_r 0$

Let us show that (3.5) leads to a contradiction in all the cases above.

1) Note that

$$i + 1 \not\equiv_r 0 \Longrightarrow Y_i = M_{x_i+i+1},$$

$$i + s + 1 \not\equiv_r 0 \Longrightarrow Y_{i+s} = M_{x_{i+s}+i+s+1} = M_{x_i+i+s+1},$$

where the last equality holds because $X_{i+s} = X_i$. Hence, from (3.5) and the previous implications we infer that $M_{x_i+i+1} = Y_i = Y_{i+s} = M_{x_i+i+s+1}$ in contradiction with the hypothesis $s \not\equiv 0 \bmod r$.

2) In this case one has

$$i + 1 \not\equiv_r 0 \Longrightarrow Y_i = M_{x_i+i+1},$$

$$i + s + 1 \equiv_r 0 \Longrightarrow Y_{i+s} = X_{i+s} \ .$$

Hence, from (3.5) we see that $M_{x_i+i+1} = Y_i = Y_{i+s} = X_{i+s} = X_i$, which yields $\mathcal{H}(X_i) = \mathcal{H}(M_{x_i+i+1}) \equiv_r x_i + i + 1 \equiv_r \mathcal{H}(X_i) + i + 1$ against $i + 1 \not\equiv_r 0$.

3) Since $s \not\equiv 0 \bmod r$, then from $i+1 \equiv_r 0$ it follows $i+s+1 \not\equiv_r 0$. Therefore, in this case (3.5) implies

$$X_i = Y_i = Y_{i+s} = M_{x_i+i+s+1} = M_{x_i+s} \ ,$$

again a contradiction with $s \not\equiv 0 \bmod r$.

This conclude the proof of the proposition.

Formula (3.3) suggests that any subsequence $\{X_j\}_{j \bmod r}$ can be seen as generated by a classical Pollard rho function whose *rules* are given by $\Delta_r(X_j)$, that prescribes a succession of r addition (one of which is a doubling). Seen in this way, the new function has the same behaviour of a classical iterating function of one variable. Thus, r valid points are generated on the curve, and hopefully a couple of distinguished points to solve the ECDLP.

The occurence of kr-cycles. For what we have seen before, the smallest cycle that can appear within such an iterating function[5] is after r steps of the iterating function. By taking $s = kr$ in (3.3) for some $k \geq 1$ we have

$$X_{i+kr} = X_i + \sum_{j=i}^{i+kr-1} Y_j = X_i + \sum_{t=0}^{kr-1} Y_{t+i} \ .$$

Now, let us consider $i_r \in \{0, \ldots, r-1\}$ such that $i_r + i + 1 \equiv_r 0$ and write

$$X_{i+kr} = X_i + \sum_{t=0}^{i_r-1} M_{t,i} + (\text{etpt}) + \sum_{t=i_r+1}^{i_r+r-1} M_{t,i} + (\text{etpt}) +$$

$$\sum_{t=i_r+r+1}^{i_r+2r-1} M_{t,i} + (\text{etpt}) + \ldots + \sum_{t=i_r+(k-2)r+1}^{i_r+(k-1)r-1} M_{t,i} + (\text{etpt}) + \sum_{t=i_r+(k-1)r+1}^{kr-1} M_{t,i} \ ,$$

where we mean all the sums of the $M_{t,i} := M_{x_{t+i}+t+i+1}$ to be zero when $i_r = 0$, and at any

[5] As usual, we also assume that the points M_i are randomly generated and there are in no trivial relations.

(etpt) we mean that one has to add "everything that precedes this" (etpt). More precisely, for

$$P_0 := X_i + \sum_{t=0}^{i_r-1} M_{t,i} = X_{i+i_r} \ , \qquad \sum_{(j)} := \sum_{t=i_r+(j-1)r+1}^{i_r+jr-1} M_{t,i}, \quad (j = 1, \ldots, k-1),$$

since

$$\sum_{t=i_r+(k-1)r+1}^{kr-1} M_{t,i} = X_{i+kr} - X_{i+i_r+(k-1)r} \ .$$

it is easy to see that

$$X_{i+kr} = 2^k P_0 + 2^{k-1}\sum_{(1)} + 2^{k-2}\sum_{(2)} + \ldots + 2\sum_{(k-1)} + \sum_{t=i_r+(k-1)r+1}^{kr-1} M_{t,i} =$$

$$2^k P_0 + \sum_{j=1}^{k-1} 2^{k-j}\sum_{(j)} + X_{i+kr} - X_{i+i_r+(k-1)r} \ .$$

Hence, this reduces to

$$X_{i+i_r+(k-1)r} = 2^k P_0 + \sum_{j=1}^{k-1} 2^{k-j}\sum_{(j)} \ .$$

In case of a collision $X_{i+kr} = X_i$, one has $X_{i+i_r+(k-1)r} = X_{i+i_r} = P_0$ and we conclude that a cycle is possible only if

$$P_0 = 2^k P_0 + \sum_{j=1}^{k-1} 2^{k-j}\sum_{(j)} \ ,$$

where

$$\sum_{(j)} := \sum_{t=i_r+(j-1)r+1}^{i_r+jr-1} M_{t,i}$$

has $r-1$ summands picked with possible repetitions from the set of the r precomputed points M_i. So that they are $\binom{2r-1}{r-1}$.

The previous expression is similar to the one we quoted previously from [7], where with a heuristic argument the authors assert that only small cycles have to be taken into account when using a function that generates mixed random walks. However, with our new iterating function there is no need to check for cycles smaller than r.

Chapter 4

Smart's attack and elliptic curves over rings

The purpose of the work described in this chapter is to adapt to elliptic curves over a ring $\mathbb{Z}/n\mathbb{Z}$ with n points the attack to the ECDLP over anomalous curves (over finite fields) as described by Smart in [35]. This experimental project required the implementation on SAGE of the functions to work on elliptic curves defined over rings and a basic arithmetic to work on *polyadic numbers*. The research led to some results concerning *amicable pairs* like lemma 4.4.1 and showed that, starting with a curve $E'(\mathbb{F}_p)$ with $\sharp E'(\mathbb{F}_p) = q$, sometimes is really easy to build a curve $E''(\mathbb{F}_q)$ with $\sharp E''(\mathbb{F}_q) = p$. Right now this is a work that is still in progress and it's being the object of further investigations.

4.1 The anomalous elliptic curves over a field $\mathbb{F}_p$

An elliptic curve E defined over a field $\mathbb{F}_p$ is said to be *anomalous* if $\sharp E(\mathbb{F}_p) = p$ or equivalently if the *trace of Frobenius* $t = p + 1 - \sharp E(\mathbb{F}_p) = 1$. These curves are particularly

weak from a cryptographic point of view. The first attacks described on these curves were proposed by Semaev[31] and Satoh-Araki[30]. In the following we will describe the attack by Smart [35], but, before talking of the attack, we need some tools.

4.1.1 Lifts and Hensel's Lemma

Given a polynomial $f(X) \in \mathbb{Z}[X]$ and a x such that $f(x) \equiv 0 \bmod p$, if we want to find x' such that $f(x') \equiv 0 \bmod p^2$ and $x' \equiv x \bmod p$, we can use the *Hensel's Lemma*:

Theorem 4.1.1. *For $f(X) \in \mathbb{Z}[X]$ let x a root modulo p^s and let $f'(x)$ be invertible modulo p. If we call u the inverse of $f(x)$ modulo p, then*

$x' = x - uf'(x)$

is such that $x' \equiv x \bmod p^s$ and $f(x') \equiv 0 \bmod p^{s+1}$.

The value x' is called *lift* of x modulo p^{s+1}. The Hensel's Lemma, allow us to lift a solution of a polynomial, from a base field to any extension of it and also to *p-adic numbers*.

4.1.2 p-adic numbers

Given a prime p and a rational number a, then a can be expressed as $a = p^r \frac{m}{n}$ where $r \in \mathbb{N}$ and $m, n \in \mathbb{Z}$ are not divisible by p. We then define

$$\mathrm{ord_p}(a) = r \quad \textit{and} \quad |a|_p = \begin{cases} p^{-r}, & \text{if} \quad a \neq 0 \\ \\ 0, & \text{if} \quad a = 0 \end{cases}.$$

The function $|.|_p : \mathbb{Q} \to [0, \infty)$ is a norm on $\mathbb{Q}$, in fact:

1. $|a|_p = 0 \Leftrightarrow a = 0$

2. $|ab|_p = |a|_p|b|_p$

3. $|a+b|_p \leq |a|_p + |b|_p$

This norm satisfy a stronger condition than the common triangular inequality(3):

$$|a+b|_p \leq \max\{|a|_p, |b|_p\}$$

and induce a metric $d_p(.,.)$ on $\mathbb{Q}$ defined as:

$$d_p(a,b) = |a-b|_p$$

The field $\mathbb{Q}_p$ of p-adic numbers is defined as the *completion* of $\mathbb{Q}$ for the metric d_p. The elements of this field are the equivalence classes of the *Cauchy sequences*, where two sequences are equivalent if their difference converges to zero according the metric defined above.

Every element x of $\mathbb{Q}_p$ can be written in a unique way as an infinite series:

$$\sum_{i=k}^{\infty} a_i p^i$$

where k is some integer for which $a_k \neq 0$ and each $a_i \in 0, \ldots, p-1$.

This series, according to the metric d_p, converges to x.

An element $a \in \mathbb{Q}_p$ is called p-adic integer, if $\mathrm{ord}_p(a) \geq 0$ and the set of p-adic integers is denoted as $\mathbb{Z}_p$.

4.1.3 Expansion around O of an elliptic curve

Let's consider an elliptic curve E defined over a field $\mathbb{K}$ by a Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{4.1}$$

with the following change of variables

$$u = -\frac{x}{y} \text{ and } w = -\frac{1}{y}$$

we obtain a new representation of the elliptic curve in uw coordinates, where the equation that describe the curve become:

$$w = u^3 + a_1uw + a_2u^2w + a_3w^2 + a_4uw^2 + a_6w^3 \tag{4.2}$$

If we substituite the equation recursively into itself, we obtain w as a power series in u:

$$\begin{aligned}
w &= u^3 + (a_1u + a_2u^2)w + (a_3 + a_4u)w^2 + a_6w^3 = && (4.3)\\
&= u^3 + (a_1u + a_2u^2)(u^3 + (a_1u + a_2u^2)w + (a_3 + a_4u)w^2 + a_6w^3) + && (4.4)\\
&+ (a_3 + a_4u)(u^3 + (a_1u + a_2u^2)w + (a_3 + a_4u)w^2 + a_6w^3)^2 + && (4.5)\\
&+ a_6(u^3 + (a_1u + a_2u^2)w + (a_3 + a_4u)w^2 + a_6w^3)^3 && (4.6)\\
&= \ldots = u^3 + a_1u^4 + (a_1^2 + a_2)u^5 + (a_1^3 + 2a_1a_2 + a_3)u^6 + && (4.7)\\
&+ (a_1^4 + 3a_1^2a_2 + 3a_1a_3 + a_2^2 + a_4)u^7 + \ldots && (4.8)
\end{aligned}$$

This procedure can be shown to converge [33, Chapter IV, Proposition 1.1] to a power series $w(u) \in \mathbb{Z}[a_1, a_2, a_3, a_4, a_6][[u]]$.

From the power series $w(u)$ obtained, we can derive the *Laurent series* for x and y:

$$\begin{aligned}
x(u) &= \frac{u}{w(u)} = \frac{1}{u^2} - \frac{a_1}{u} - a_2 - a_3u - (a_4 + a_1a_3)u^2 - \ldots && (4.9)\\
y(u) &= -\frac{1}{w(u)} = -\frac{1}{u^3} + \frac{a_1}{u^2} + \frac{a_2}{u} + a_3 + (a_4 + a_1a_3)u + \ldots && (4.10)
\end{aligned}$$

The couple $(x(u), y(u))$ is a *formal solution* of the equation 4.1, meaning that substituiting the power series for $x(u)$ and $y(u)$ we obtain the same formal power series on each side. To obtain the points of an elliptic curve $E(\mathbb{K})$ through the u-coordinate, it is needed that the series $x(u)$ and $y(u)$ converge in the field $\mathbb{K}$.

If $\mathbb{K} = \mathbb{Q}_p$ the convergence is assured[33] if $\mathrm{ord}_p(u) \geq 1$ (or equivalently u is in the maximal ideal of $\mathbb{Q}_p$) and if coefficients a_1, a_2, a_3, a_4, a_6 are in $\mathbb{Z}_p$.

4.1.4 Reduction $\mathrm{mod}\, p$

If we have an elliptic curve E defined over $\mathbb{Q}_p$ we can obtain the curve $\tilde{E}(\mathbb{F}_p)$ reducing the coefficients of E modulo p. Similarly, if $A = (x_1, y_1, z_1)$ is a point on $E(\mathbb{Q}_p)$ we obtain the reduced point $\tilde{A}$ on $\tilde{E}(\mathbb{F}_p)$ reducing the coordinates of A modulo p. In this way it is defined a reduction map:

$$E(\mathbb{Q}_p) \rightarrow \tilde{E}(\mathbb{F}_p) \tag{4.11}$$

$$A \rightarrow \tilde{A} \tag{4.12}$$

4.1.5 Formal group associated to an elliptic curve

From the formal power series $x(u)$ and $y(u)$ it can be derived another formal power series that express the addition law for $E(\mathbb{K})$. Given two points (u_1, w_1) and (u_2, w_2) of an elliptic curve $E(\mathbb{K})$, the coordinate u_3 of the sum of these points is given by the following power series:

$$u_3 = F(u_1, u_2) = u_1+u_2-u_1u_2-a_2(u_1^2u_2+2a_3u_1u_2^3)-(2a_3u_1^3u_2-a_1a_2-3a_3)u_1^2u_2^2+2a_3u_1u_2^3)+\dots \tag{4.13}$$

From the corresponding properties for E, we deduce that also the power series $F(u_1, u_2)$ satisfy commutativity, associativity and existence of inverse. This power series $F(u_1, u_2)$ is called *formal group associated to the elliptic curve* $E(\mathbb{K})$ and can be seen as "a group law without any group elements" [33].

If E is an elliptic curve defined over $\mathbb{Q}_p$, the group $\hat{E}(p\mathbb{Z}_p)$ is the set $p\mathbb{Z}_p$ with the addition law $x \oplus y := F(x, y)$, for all $x, y \in p\mathbb{Z}_p$.

It can be built a group isomorphism $\log_F : E(p\mathbb{Z}_p) \to p\mathbb{Z}_p$, where $p\mathbb{Z}_p$ is equipped with the usual addition law. Since it is an isomorphism, we must have:

$$\log_F F(u_1, u_2) = \log_F(u_1) + \log_F(u_2) \text{ with } u_1, u_2 \in p\mathbb{Z}_p \tag{4.14}$$

To obtain this isomorphism the first thing needed is a power series P, such that

$$P(F(T, S))F_X(T, S) = P(T) \tag{4.15}$$

where F_X is the partial derivative of F with respect to the first variable. Setting $T = 0$ we have

$$P(S)F_X(0, S) = P(0). \tag{4.16}$$

So, every power series P that satisfy 4.15, has the form:

$$P(T) = aF_X(0,T)^{-1} \text{ with } a \in \mathbb{Q}_p. \tag{4.17}$$

If we choose $a = 1$, P has the form

$$P(T) = 1 + d_1T + d_2T^2 + d_3T^3 + \dots \tag{4.18}$$

The *formal logarithm* map is the power series

$$\log_F(T) = \int P(T)dT = T + \frac{d_1}{2}T^2 + \frac{d_2}{3}T^3 + \dots \tag{4.19}$$

To show that it is an homomorphism we integrate 4.15 with respect to T obtaining

$$\log_F F(T,S) = \log_F(T) + C(S) \tag{4.20}$$

where $C(S)$ is a constant depending on S. If we set $T = 0$, we obtain

$$C(S) = \log_F(S) \tag{4.21}$$

To show that $\log_F$ induces an isomorphism from $E(p\mathbb{Z}_p)$ to $p\mathbb{Z}_p$ it is needed an inverse power series that converges on $p\mathbb{Z}_p$. This power series is called $\exp_F$ and its existence is guaranteed by a classical result on the formal power series [33, ch.IV, lemma 2.4].

4.1.6 The subgroups $E_n(\mathbb{Q}_p)$

In the description of the Smart's attack on anomalous elliptic curves it is useful to define the subgroup

$$E_n(\mathbb{Q}_p) = P \in E(\mathbb{Q}_p) | \operatorname{ord}_p(x(P))) \leq -2n \cup O_\infty \tag{4.22}$$

where $x(P)$ is the x-coordinate of the point P. Among all groups $E_n(\mathbb{Q}_p)$, two of them play an important role:

- $E_0(\mathbb{Q}_p) = \tilde{E}(\mathbb{F}_p)$
- $E_1(\mathbb{Q}_p) = \{P \in E(\mathbb{Q}_p) : \tilde{P} = \tilde{O}_\infty\}$

where $\tilde{}$ denotes the reduction mod p. In general each of the groups $E_n(\mathbb{Q}_p)$ is isomorphic to $\hat{E}(p^n\mathbb{Z}_p)$.

4.1.7 Smart's attack

Given an elliptic curve $\tilde{E}(\mathbb{F}_p)$ with $\sharp\tilde{E}(\mathbb{F}_p) = p$, as first step to compute the value m such that $\tilde{Q} = [m]\tilde{P}$ with $\tilde{Q}, \tilde{P} \in \tilde{E}(\mathbb{F}_p)$ we compute the lifts $P, Q \in E(\mathbb{Q}_p)$ of $\tilde{P}, \tilde{Q}$ through the Hensel's lifting. Since the reduction modulo p is an homomorphism, we have

$$Q - [m]P = R \in E_1(\mathbb{Q}_p) \tag{4.23}$$

We have that $\tilde{E}(\mathbb{F}_p) \simeq E(\mathbb{Q}_p)/E_1(\mathbb{Q}_p)$ and [33, Chapter VII, ex. 7.4] $E_1(\mathbb{Q}_p)/E_2(\mathbb{Q}_p) \simeq \mathbb{F}_p^+$.

From the relations above and since $\sharp\tilde{E}(\mathbb{F}_p) = p$, we have that the multiplication by $[p]$ maps the elements of $E(\mathbb{Q}_p)$ to $E_1(\mathbb{Q}_p)$ and elements of $E_1(\mathbb{Q}_p)$ to $E_2(\mathbb{Q}_p)$.

If we multiply both sides of equation 4.23 by p, we obtain:

$$[p]Q - [m]([p]P) = [p]R \in E_2(Q_p) \tag{4.24}$$

Since both $[p]P$ and $[p]Q$ are in $E_1(\mathbb{Q}_p)$, we can apply the isomorphism $\log_F$ to obtain

$$\log_F([p]Q) - m\log_F([p]P) \in p^2\mathbb{Z}_p. \tag{4.25}$$

and from this equation we can derive m simply computing $\frac{\log_F [p]Q}{\log_F [p]P} \bmod p$.

4.2 Elliptic curves over rings

Elliptic curves can be also defined on commutative rings. The most important application of elliptic curves over rings is in the factorization method proposed by Lenstra [15] and known as ECM (Elliptic Curves Method). When an elliptic curve is defined over a ring the addition and doubling formulas defined for elliptic curves over fields may fail, anyway, a full theory of elliptic curves on commutative rings can be defined through a different set of addition formulas. Let R a ring (commutative and with 1), a tuple of elements $(x_1, x_2, \ldots)$ of R is said to be *primitive* if there exist elements $r_1, r_2, \ldots \in R$ such that $r_1x_1 + r_2x_2 + \ldots = 1$. Two primitive triples (x, y, z) and (x', y', z') are said to be equivalent if there exists a unit $u \in R^\times$ such that $(x', y', z') = (ux, uy, uz)$. A two dimensional projective space on R is defined as:

$$P^2(R) = \{(x, y, z) \in R^3 | (x, y, z) \text{ is primitive}\} \bmod \text{equivalence} \tag{4.26}$$

The equivalence class of (x, y, z) is denoted by $(x : y : z)$.

Two additional conditions are needed to define elliptic curves over rings:

1. $2 \in R^{\times}$

2. If (a_{ij}) is an $m \times n$ matrix such that $(a_{11}, a_{12}, \ldots, a_{mn})$ is primitive and such that all 2×2 subdeterminants vanish, then some R-linear combination of the rows is a primitive n-tuple.

If we have a ring R satisfying conditions 1 and 2 described above, an elliptic curve E on R is given by an homogeneous equation

$$y^2z = x^3 + Axz^2 + Bz^3 \tag{4.27}$$

with $A, B \in R$ such that $4A^3 + 27B^2 \in R^{\times}$. Define

$$E(R) = \{(x : y : z) \in P^2(R) | y^2z = x^3 + Axz^2 + Bz^3\}. \tag{4.28}$$

On this set the addition laws are much more complicated that the ones for elliptic curves over rings. Given $(x_i, y_i, z_i) \in E(R)$ for $i = 1, 2$, we consider the following three sets of equations:

1.

$$\begin{aligned}
x'_3 &= (x_1y_2 - x_2y_1)(y_1z_2 + y_2z_1) + (x_1z_2 - x_2z_1)y_1y_2 \\
&- A(x_1z_2 + x_2z_1)(x_1z_2 - x_2z_1) - 3B(x_1z_2 - x_2z_1)z_1z_2 \\
y'_3 &= -3x_1x_2(x_1y_2x_2y_1) - y_1y_2(y_1z_2 - y_2z_1) - A(x_1y_2 - x_2y_1)z_1z_2 \\
&+ A(x_1z_2 - x_2z_1)(y_1z_2 - y_2z_1) + 3B(y_1z_2 - y_2z_1)z_1z_2 \\
z'_3 &= 3x_1x_2(x_1z_2 - x_2z_1) - (y_1z_2 + y_2z_1)(y_1z_2 - y_2z_1) \\
&+ A(x_1z_2 - x_2z_1)z_1z_2
\end{aligned}$$

2.

$$\begin{aligned}
x''_3 &= y_1y_2(x_1y_2 + x_2y_1) - Ax_1x_2(y_1z_2 + y_2z_1) - A(x_1y_2 + x_2y_1)(x_1z_2 + x_2z_1) \\
&- 3B(x_1y_2 + x_2y_1)z_1z_2 - 3B(x_1z_2 + x_2z_1)(y_1z_2 + y_2z_1) + A^2(y_1z_2 + y_2z_1)z_1z_2 \\
y''_3 &= y_1^2y_2^2 + 3Ax_1^2x_2^2 + 9Bx_1x_2(x_1z_2 + x_2z_1) - A^2x_1z_2(x_1z_2 + 2x_2z_1) \\
&- A^2x_2z_1(2x_1z_2 + x_2z_1) - 3ABz_1z_2(x_1z_2 + x_2z_1) - (A^3 + 9B^2)z_1^2z_2^2 \\
z''_3 &= 3x_1x_2(x_1y_, + x_2y_1) + y_1y_2(y_1z_2 + y_2z_1) + A(x_1y_2 + x_2y_1)z_1z_2 \\
&+ A(x_1z_2 + x_2z_1)(y_1z_2 + y_2z_1) + 3B(y_1z_2 + y_2z_1)z_1z_2
\end{aligned}$$

3.

$$
\begin{aligned}
x_3''' &= (x_1y_2 + x_2y_1)(x_1y_2 - x_2y_1) + Ax_1x_2(x_1z_2 - x_2z_1) \\
&+ 3B(x_1z_2 + x_2z_1)(x_1z_2 - x_2z_1) - A^2(x_1z_2 - x_2z_1)z_1z_2 \\
y_3''' &= (x_1y_2 + x_2y_1)y_1y_2 - 3Ax_1x_2(y_1z_2 - y_2z_1) + A(x_1y_2 + x_2y_1)(x_1z_2 - x_2z_1) \\
&+ 3B(x_1y_2 - x_2y_1)z_1z_2 - 3B(x_1z_2 + x_2z_1)(y_1z_2 - y_2z_1) + A^2(y_1z_2 - y_2z_1)z_1z_2 \\
z_3''' &= -(x_1y_2 + x_2y_1)(y_1z_2 - y_2z_1) - (x_1z_2 - x_2z_1)y_1y_2 \\
&- A(x_1z_2 + x_2z_1)(x_1z_2 - x_2z_1) - 3B(x_1z_2 - x_2z_1)z_1z_2
\end{aligned}
$$

A *complete system of addition laws* for $E(R)$ is a collection of addition laws with the property that for any pair of points on $E(R)$, at least one of the addition laws can be used to add the points. It can be shown [8][19][20] that the minimal cardinality of a complete system of addition law for elliptic curves over rings is two. If we refer to the equation given above, we have a complete system if we consider the addition laws:

- x_3', y_3', z_3' and x_3'', y_3'', z_3''
- x_3'', y_3'', z_3'' and x_3''', y_3''', z_3'''

while the set consisting of addition laws x_3', y_3', z_3' and x_3''', y_3''', z_3''' do not form a complete system of addition laws. The justification to the last sentence can be easily found if we consider the so called *exceptional set* of points for a given addition law.

A couple of points is exceptional for a given addition law if that addition law cannot be used to add those points.

A pair of points P_1, P_2 is exceptional[8]

- for x_3', y_3', z_3' if and only if $P_1 = P_2$
- for x_3'', y_3'', z_3'' if and only if the difference of the y-coordinate of $P_1 - P_2$ is zero
- for x_3''', y_3''', z_3''' if and only if the difference of the x-coordinate of $P_1 - P_2$ is zero.

These set of equations, when R is a field, give the usual group law and the output is a point in $P^2(R)$.

Theorem 4.2.1. *Let n_1, n_2 be odd integers such that $\gcd(n_1, n_2) = 1$ and let E be an elliptic curve defined over $\mathbb{Z}/n_1n_2\mathbb{Z}$. Then there is a group isomorphism[39]*

$$E(\mathbb{Z}/n_1n_2\mathbb{Z}) \simeq E(\mathbb{Z}/n_1\mathbb{Z}) \oplus E(\mathbb{Z}/n_2\mathbb{Z}) \tag{4.29}$$

Proof. Suppose that E is given by an equation like 4.27 with $A, B, \in \mathbb{Z}/n_1n_2\mathbb{Z}$ and $4A^3 + 27B^2 \in (\mathbb{Z}/n_1n_2\mathbb{Z})^\times$. We can also consider A and B as elements of $\mathbb{Z}/n_i\mathbb{Z}$ and $4A^3 + 27B^2 \in (\mathbb{Z}/n_i\mathbb{Z})^\times$. From the Chinese Remainder Theorem, we know that there is an isomorphism of rings

$$\mathbb{Z}/n_1n_2\mathbb{Z} \simeq \mathbb{Z}/n_1\mathbb{Z} \oplus \mathbb{Z}/n_2\mathbb{Z} \tag{4.30}$$

given by

$$x \bmod n_1n_2 \leftrightarrow (x \bmod n_1, x \bmod n_2)). \tag{4.31}$$

In this way we have a bijection between triples (this holds also for primitive triples) in $\mathbb{Z}/n_1n_2\mathbb{Z}$ and pairs of triples, one in $\mathbb{Z}/n_1\mathbb{Z}$ and the other in $\mathbb{Z}/n_2\mathbb{Z}$.

Moreover

$$y^2z = x^3 + Axz^2 + Bz^3 (\mathrm{mod}\, n_1 n_2) \Leftrightarrow \begin{cases} y^2z = x^3 + Axz^2 + Bz^3 (\mathrm{mod}\, n_1) \\ y^2z = x^3 + Axz^2 + Bz^3 (\mathrm{mod}\, n_2) \end{cases}$$

The bijection $\psi : E(\mathbb{Z}/n_1n_2\mathbb{Z}) \to E(\mathbb{Z}/n_1\mathbb{Z}) \oplus E(\mathbb{Z}/n_2\mathbb{Z})$ can be easily shown to be an homomorphism. Let $P_1, P_2 \in E(\mathbb{Z}/n_1n_2\mathbb{Z})$ and $P_3 = P_1 + P_2$. If all the computation are reduced $\mathrm{mod}\, n_i$ (for $i = 1, 2$) we have the same result $\mathrm{mod}\, n_i$: the point $P_3 \,\mathrm{mod}\, n_i = P_1 \,\mathrm{mod}\, n_i + P_2 \,\mathrm{mod}\, n_i$. □

The previous theorem, implicitly, gives also a method to construct an elliptic curve E over a ring $\mathbb{Z}/n_1n_2\mathbb{Z}$ starting from two curves $E'(\mathbb{Z}/n_1\mathbb{Z})$ and $E''(\mathbb{Z}_{n_2})$ through the Chinese Remainder Theorem.

4.3 Building elliptic curves with a given number of points

Suppose we want to build an elliptic curve over a field $\mathbb{F}_p$ with a given number of points $N = p+1-t$. The **Hasse Theorem** states that the integer N belong to the so called *Hasse interval*

$$\mathcal{H}_p = [p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]. \tag{4.32}$$

If the order of the curve is not $N = p + 1$ (in that case any supersingular elliptic curve over $\mathbb{F}_p$ is good) or if no curve on $\mathbb{F}_p$ with j-invariant 0 or 1728 has order N, we can simply try to build random curves starting from their j-invariant and then check if a point on them has order $p + 1 \pm t$. An elliptic curve E over a field $\mathbb{F}_p$ with j-invariant $j(E)$ can be built

from equation

$$E : y^2 = x^3 + \frac{3j(E)}{1728 - j(E)}x + \frac{2j(E)}{1728 - j(E)} \tag{4.33}$$

If the point on a curve E has order $p + 1 - t$ then we've found the curve, if the point has order $p + 1 + t$ then the quadratic twist of the curve E will have order $p + 1 - t$.

This method, even if is quite simple, is not really efficient. A more efficient method, due to Atkin and Morain [2], rely on the theory of *complex multiplication*. The theory of complex multiplication to build elliptic curves with a given number of points involves deep arguments of analytic number theory. In the following we will describe only the steps that must be performed, referring the reader to [2],[33],[13] and [39] for the underlying theory. If we are working on a prime field $\mathbb{F}_p$ and want a curve with $N = p + 1 - t$ points the first step is to write $4p = t^2 - Dv^2$. Since we know both p and t we can compute the the value $\Delta = -Dv^2$ and from Δ we can build the associated **Hilbert Polynomial** $\mathbb{H}_\Delta(x)$. An alternative approach that can save some computation when Δ has a large square factor, consists into considering the *fundamental discriminant of the field* $\mathbb{Q}(\sqrt{-D})$ where $-D$ is the square free factor of Δ. The root of this Hilbert Polynomial is the j-invariant of a curve $E(\mathbb{F}_p)$. Since a curve and its quadratic twist have the same j-invariant we have only to check (as said before) if the order is $p + 1 - t$ or $p + 1 + t$ and eventally perform a quadratic twist of the curve.

4.4 Amicable pairs and aliquot cycles

The classical definition of *aliquot cycle* in number theory is a list of integers $(n_1, n_2, ..., n_l)$ such that

$$\tilde{\sigma}(n_1) = n_2, \quad \tilde{\sigma}(n_2) = n_3, \quad \ldots, \quad \tilde{\sigma}(n_{l-1}) = n_l, \quad \tilde{\sigma}(n_l) = n_1 \tag{4.34}$$

where $\tilde{\sigma}(n) = \sigma(n) - n$ is the sum of the proper divisors of n. An aliquot cycle of lenght one is called *perfect number*, and an aliquot cycle of lenght two is called *amicable pair*. Similarly for elliptic curves it can be defined the notion of aliquot cycle:

$$\sharp\tilde{E}_{p_1}(\mathbb{F}_{p_1}) = p_2, \quad \sharp\tilde{E}_{p_2}(\mathbb{F}_{p_2}) = p_3, \quad \ldots, \quad \sharp\tilde{E}_{p_{l-1}}(\mathbb{F}_{p_{l-1}}) = p_l, \quad \sharp\tilde{E}_{p_l}(\mathbb{F}_{p_l}) = p_1 \tag{4.35}$$

where p_k are primes and $\sharp\tilde{E}_{p_k}(\mathbb{F}_{p_k})$ is the cardinality of the group defined by the elliptic curve E reduced modulo p_k.

For elliptic curves, an aliquot cycle of lenght 2 is still called amicable pair like in the classical case while for aliquot cycles of lenght one is used the name of *anomalous prime* (and anomalous elliptic curve, like already defined previously).

In [36] it is shown that exist elliptic curves with arbitrary long aliquot cycles. Here we are interested particularly into amicable pairs. Since the group orders of an elliptic curve over a prime field $\mathbb{F}_p$ are restricted to the Hasse interval $\mathcal{H}_p = [p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$, to have an amicable pair, we need two primes such that one belongs to the Hasse interval of the other and vice versa.

Lemma 4.4.1. *Let p and q primes different from 2 and 3 and $p \neq q$. If q belongs to the Hasse interval of p, then p belongs to the Hasse interval of q.*

Proof. If q is the order of $\tilde{E}_p(\mathbb{F}_p)$ then it belongs to the *Hasse interval* $\mathcal{H}_p$:

$$p + 1 - 2\sqrt{p} \leq q = p + 1 - t \leq p + 1 + 2\sqrt{p} \tag{4.36}$$

If p ($p = q - 1 + t$ from the previous equation) is in the Hasse interval of q, then:

$$\begin{aligned}
q + 1 - 2\sqrt{q} \leq \;& p = q - 1 + t & \leq q + 1 + 2\sqrt{q} \\
1 - 2\sqrt{q} \leq \;& t - 1 & \leq 1 + 2\sqrt{q} \\
-2\sqrt{q} \leq \;& t - 2 & \leq 2\sqrt{q} \\
& |t - 2| & \leq 2\sqrt{q} \\
& |t - 2| & \leq 2\sqrt{p + 1 - t}
\end{aligned}$$

Since $q = p + 1 - t$ is supposed to be a prime different from p we have that $t \neq 0, 1, 2$. Squaring the inequality lead to:

$$\begin{aligned}
(t - 2)^2 &\leq (2\sqrt{p + 1 - t})^2 \\
t^2 + 4 - 4t &\leq 4p + 4 - 4t \\
t^2 &\leq 4p
\end{aligned}$$

that is always true since, from the equation 4.36, we have that $|t| \leq 2\sqrt{p}$. □

The existence of an elliptic curves with order t for every t inside the Hasse interval is guaranteed by the *Deuring's theorem*. So if we have an elliptic curve $E(\mathbb{F}_p)$ with order q prime, we are sure that (p, q) is an amicable pair.

It can be shown also that $a_p^2 - 4p = a_q^2 - 4q$. Infact, if (p, q) is an amicable pair for elliptic curves we have

$$q = p + 1 - a_p \qquad \text{and} \qquad p = q + 1 - a_q \tag{4.37}$$

where a_p(resp. a_q) is the trace of Frobenius of the curve defined over $\mathbb{F}_p$ (resp. $\mathbb{F}_q$). Subtracting the second equation to the first in 4.37 we have:

$$\begin{aligned} q - p &= p + 1 - a_p - q - 1 + a_q \\ a_p - 2p &= a_q - 2q \\ (a_p - 2p)^2 &= (a_q - 2q)^2 \\ a_p^2 - 4p(a_p - p) &= a_q^2 - 4q(a_q - q) \end{aligned}$$

if we substitute $a_p - p = 1 - q$ and $a_q - q = 1 - p$ (from equation 4.37)

$$\begin{aligned} a_p^2 - 4p(1 - q) &= a_q^2 - 4q(1 - p) \\ a_p^2 - 4p + 4pq &= a_q^2 - 4q + 4pq \\ a_p^2 - 4p &= a_q^2 - 4q. \end{aligned}$$

4.5 Our work

In cryptography, anomalous elliptic curves over fields are weak since the Smart's attack can solve an instance of ECDLP over them in linear time. Good candidates are the elliptic curves over a field $\mathbb{F}_p$ that have a number of points q prime and with $p \neq q$ because the most efficient algorithm to solve an ECDLP instance over them requires an exponential time. We asked ourselves if it could be performed a "lifting" of an elliptic curve $E'(\mathbb{F}_p)$ with $\sharp E'(\mathbb{F}_p) = q$ to a curve $E(\mathbb{Z}/pq\mathbb{Z})$ such that $\sharp E(\mathbb{Z}/pq\mathbb{Z}) = pq$, and once performed this lift if it could be applied the Smart's algorithm to solve ECDLP. As we've seen in the previous paragraphs, if we have an elliptic curve $E'(\mathbb{F}_p)$ with cardinality q with p and q primes greater than 3 and $p \neq q$ it is always possible to build another elliptic curve $E''(\mathbb{F}_q)$ with cardinality q. This operation, if we apply the "complex multiplication" algorithm requires the resolution of an Hilbert class polynomial. In [3] is given a tighter bound to the complexity of the analytic algorithm for the computation of the Hilbert class polynomial $\mathbb{H}_D$ (with discriminant D) that is $O(|D|(\log|D|)^3(\log\log|D|)^3)$. We underline here that the discriminant D could be really small compared to the field size. As example we considered the curve *secp192k1* of the document *SEC 2: Recommended Elliptic Curve Domain Parameters* [29] over the finite field $\mathbb{F}_p$ with (values are in hexadecimal format):

$$
\begin{aligned}
p &= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFEE37 \\
&= 2^{192} - 2^{32} - 2^{12} - 2^8 - 2^7 - 2^6 - 2^3 - 1
\end{aligned}
$$

described by the equation $E : y^2 = x^3 + ax + b$ with:

$$\begin{aligned} a &= 0000000000000000000000000000000000000000000000000000 \\ b &= 0000000000000000000000000000000000000000000000000003 \end{aligned}$$

and with order:

$$q = FFFFFFFFFFFFFFFFFFFFFFFFE26F2FC170F69466A74DEFD8D$$

In few seconds with SAGE we are able to compute the curve over $\mathbb{F}_q$ with p points. This curve must have:

$$\begin{aligned} a_q &= q + 1 - p = \\ &= -146402144145231529258894028969 \end{aligned}$$

let's compute the discriminant of the Hilbert class polynomial:

$$\begin{aligned} D &= (a_q)^2 - 4q = \\ &= -367481913122557251444932678164864559652605022300197084534 7 \end{aligned}$$

and its factorization:

$$factor(D) = -1 * 3 * 31^4 * 4931^2 * 54059^2 * 1366252466671812972^2$$

Since D has a huge square factor, the Hilbert class polynomial can be solved really fast, leading to a j-invariant for the curve:

$$j = 57837080126160879996434911420134774866196024963175661695 61$$

Now we build the curve over $\mathbb{F}_q$ with equation $y^2 = x^3 + \frac{3j}{1728-j}x + \frac{2j}{1728-j}$. It can be easily checked (with SAGE) that this curve doesn't have p points, but performing a quadratic twist leads to the desired curve with order p and with:

$$a = 60683579107935107896055811419299890707039871091732723237 18$$

$$b = 51638013375564409012746785903611298652737276163871218408 29$$

This example shows that in special conditions it could be really easy to build an amicable pair (p, q). Building an elliptic curve $E(\mathbb{Z}/pq\mathbb{Z})$ with $\sharp E(\mathbb{Z}/pq\mathbb{Z}) = pq$ starting with an amicable pair (p, q) such that $\sharp E'(\mathbb{F}_p) = q$ and $\sharp E''(\mathbb{F}_q) = p$ is trivial through the Chinese Remainder Theorem. The curve $E(\mathbb{Z}/pq\mathbb{Z})$ have some interesting properties as group structure[34]:

$$E(\mathbb{Z}/pq\mathbb{Z}) \simeq \mathbb{Z}_p \oplus \mathbb{Z}_q \simeq E'(\mathbb{Z}/p^2\mathbb{Z}) \simeq E''(\mathbb{Z}/q^2\mathbb{Z}). \quad (4.38)$$

An instance of ECDLP $Q' = k'P'$ from $E'(\mathbb{F}_p)$ can be esily moved to $E(\mathbb{Z}/pq\mathbb{Z})$ to points Q, P in a way that preserve the relation $Q = k'P$ if we use the Chinese Remainder Theorem to build[1] $Q = C.R.T(Q', O''_\infty)$ and $P = C.R.T(P', O''_\infty)$ or we can obtain $Q = kP$ with $k \equiv k' \bmod p$ if we build $Q = C.R.T(Q', P'')$ and $P = C.R.T(P', P'')$, where P'' is any point different from the point to infinity over $E''(\mathbb{F}_q)$. While it is trivial to map an instance of ECDLP from $E'(\mathbb{F}_p)$ to $E(\mathbb{Z}/pq\mathbb{Z})$, is it much more complicate to operate over the curve $E(\mathbb{Z}/pq\mathbb{Z})$ and even harder to try to apply the attack by Smart. To work on elliptic curves modulo pq we implemented a complete set of addition laws in SAGE like described in section 4.2. The attack by Smart(since we're not working anymore over a field with a prime p, but we're on a ring $\mathbb{Z}/pq\mathbb{Z}$) required the implementation of the modular arithmetic of *Polyadic Numbers* as described by Bennett in [4].

[1] O''_∞ is the point to infinity of the curve $E''(\mathbb{F}_q)$

Bibliography

[1] Ali Akhavi and Brigitte Vallée. Average bit-complexity of euclidean algorithms. In *Proceedings ICALPŠ00, Lecture Notes Comp. Science 1853*, pages 373–387. Springer, 2000.

[2] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp*, 61:29–68, 1993.

[3] J. Belding, R. Bröker, A. Enge, and K. Lauter. Computing Hilbert Class Polynomials. *ArXiv e-prints*, February 2008.

[4] Albert A. Bennett. The modular theory of polyadic numbers. *Annals of Mathematics*, 23(1):83–90, September 1921.

[5] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the correct use of the negation map in the pollard rho method. *IACR Cryptology ePrint Archive*, 2011:3, 2011.

[6] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *IJACT*, 2(3):212–228, 2012.

[7] Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra. On the use of the negation map in the pollard rho method. In Guillaume Hanrot, François Morain, and Emmanuel Thomé, editors, *ANTS*, volume 6197 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 2010.

[8] W. Bosma. Complete systems of two addition laws for elliptic curves. *Journal of Number Theory*, 53:229–240, 1995.

[9] Richard P. Brent. Analysis of the binary euclidean algorithm. In *Directions and Recent Results in Algorithms and Complexity*, pages 321–355. Academic Press, 1976.

[10] T. Acar C. K. Koç and B. S. Kalinski Jr. Analyzing and Comparing Montgomery multiplication Algorithms. *IEEE Micro*, 16:26–33, 1996.

[11] CERTICOM. Certicom ECC challenge. 1997.

[12] H. Cohen and G. Frey, editors. *Handbook of elliptic and hyperelliptic curve cryptography.* CRC Press, 2005.

[13] R. Crandall and C. Pomerance. *Prime numbers. A computational perspective.* Springer-Verlag, 2001.

[14] A. Escott. Implementing a parallel Pollard rho attack on ECC. 1998.

[15] H.W.Lenstra. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126(3):649–673, November 1987.

[16] Pierrick Gaudry Iwan M. Duursma and François Morain. Speeding up the discrete log computation on curves with automorphisms. In Kwok-Yan Lam, Eiji Okamoto,

and Chaoping Xing, editors, *ASIACRYPT*, volume 1716 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 1999.

[17] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997.

[18] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer, 1988.

[19] H. Lange and W. Ruppert. Complete system of addition laws on abelian varieties. *Inventiones mathematicae*, 79:603–610, 1985.

[20] H. Lange and W. Ruppert. Addition laws on elliptic curves in arbitrary charateristics. *Journal of Algebra*, 107:106–116, 1987.

[21] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de bruijn sequences to index a 1 in a computer word, 1998.

[22] M. Laporta S. Migliori M.Chinnici, S. Cuomo and A. Pizzirani. Cuda based implementation of parallelized pollard's rho algorithm for ecdlp. In *Proceedings of the FINAL WORKSHOP OF PROJECTS FUNDED BY "PON RICERCA 2000-2006, AVVISO 1575"*, pages 97–101, 2010.

[23] M. Laporta S. Migliori M.Chinnici, S. Cuomo and A. Pizzirani. Pollard's rho algorithm for ecdlp on graphic cards. In J. Vigo-Aguiar, editor, *Proceedins of the 2010 International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 363–372, 2010.

[24] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, May 2012.

[25] A. Pizzirani. *Il problema del logaritmo discreto sulle curve ellittiche e relazioni con la crittografia.* Tesi di Laurea presso Università degli studi di Napoli "Federico II", 2009.

[26] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). *IEEE Trans. Inf. Theor.*, 24(1):106–110, September 2006.

[27] J. M. Pollard. A Monte Carlo method for factorization. 15(3):331–334, September 1975.

[28] J. M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[29] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. In *Standards for Efficient Cryptography*, 2000.

[30] T. Satoh and K. Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Math. Univ. St. Pauli*, 1998.

[31] I. A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Mathematics of Computation*, 1998.

[32] D. Shanks. Class number, a theory of factorization, and genera. In *Number Theory Institute, 1969*, volume 20 of *Proceedings of Symposia in Pure Mathematics*, pages 415–440. 1969.

[33] J. H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 1986.

[34] J. H. Silverman. private e-mail communication, 2013.

[35] N. P. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of Cryptology*, 12:193–196, 1999.

[36] K. E. Stange and J. H. Silverman. Amicable pairs and aliquot cycles for elliptic curves. *ArXiv e-prints*, December 2009.

[37] E. Teske. Speeding up Pollard's rho method for computing discrete logarithms. *Lecture Notes in Computer Science*, 1423:541–554, 1998.

[38] E. Teske. On random walks for Pollard's rho method. *Math. Comput.*, 70(234):809–825, 2001.

[39] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2003.

[40] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography, LNCS 1556*, pages 190–200. Springer-Verlag, 1998.