Bachelor thesis
Computer Science

Radboud University

Performing an online template attack on two different implementations of Four$\mathbb{Q}$

Author:
Ischa Stork
s4483111

First supervisor/assessor:
prof. dr. Lejla Batina
lejla@cs.ru.nl

Second supervisor/assessor:
MSc Niels Samwel
nsamwel@cs.ru.nl

August 17, 2018

Abstract

Online template attack is a recently developed side channel attack that can be performed on a widespread number of crypto schemes. In this thesis two different implementations of Four$\mathbb{Q}$ are analysed and an attempt is made to perform an online template attack on said implementations on an ARM Cortex-M4 microcontroller. The attack was successfully modified to work on the windowed scalar multiplication version of Four$\mathbb{Q}$, however I was unsuccessful in modifying the attack so that it would work on the efficient version of Four$\mathbb{Q}$. Ultimately, I conclude that it is possible to tailor the attack so that it works on windowed scalar multiplications schemes but that it is hard in specific cases, e.g. when such schemes use scalar decomposition.

Contents

Chapter 1

Introduction

Traditional assymetric cryptosystems rely on hard problems such as the integer factorisation problem or the discrete logarithm problem for their security. A cryptosystem like RSA has remain secure for decades using the integer factorisation problem, but ever increasing computing power in combination with more complex factorisation algorithms force people to pick larger security parameters (e.g. prime factors), which in turn results into larger computations. This is bad in the modern day where even the smallest devices use cryptography to ensure their security.

This is where elliptic curves come into play, they allow us to pick relatively small security parameters even though providing equal security strength. One of these elliptic curves is Four$\mathbb{Q}$, this curve combines state-of-the-art elliptic curve principles to acquire a very efficient curve. However, even if we assume that all the curve parameters were carefully chosen (meaning there is no trivial way to break the scheme) that does not necessarily mean that the curve is inherently secure. There are other techniques to attack the curve, one of which is side channel analysis.

Side channel analysis is different way to attack encryption schemes. Instead of targeting the (mathematical) foundation of the scheme, the main focus lies on attacking the implementation of said scheme. This technique exploits the fact that one can measure certain physical aspects like power usage or electromagnetic radiation during computations. An Online Template Attack (OTA) is a sophisticated attack that is specifically useful for attacking schemes that make use of modular exponentiation or methods that are analogous to modular exponentiation. The attack can be used on a wide range of elliptic curves and additionally can be easily adapted to fit specific curves.

In this case, an online template attack is performed on two different Four$\mathbb{Q}$ implementations. The first version uses a basic windowed scalar multiplica-

tion, while the second version of the algorithm implements scalar decomposition for more efficiency. The standard online template attack is adapted so that it can be applied on both implementations.

The structure of this paper is as follows. The first chapter contains some general background information about elliptic curves and side channel analysis. Additionally, both Four$\mathbb{Q}$ and online template attacks are discussed in detail. As for Four$\mathbb{Q}$, both implementations are discussed separately to maintain a clear boundary between them.
In the second chapter, it is described how the attack is performed on the windowed scalar version. More specifically, there is some information about the setup, analysis of the scalar multiplication algorithm and adaption of the attack in order to work on said algorithm. Lastly, we discuss the results and make some suggestions on how to improve these results.

In the third chapter the focus shifts towards the efficient implementation of Four$\mathbb{Q}$, it is argued that in order to generate templates for the attack it is necessary to reverse the recoding and decomposition phase. Both reversion schemes are defined and an attempt is made to use them in order to perform the attack.

Chapter 2

Preliminaries

2.1 Elliptic curves

2.1.1 Fundamentals

An elliptic curve defines a set of points by using the formula:

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

Furthermore, an elliptic curve is defined over a field, which means that a specific field is used to define a new field structure. A point P on the curve is then an element in this new field. Following the group axioms, there must be an identity element and an inverse element for every point. The identity element of an elliptic curve is called the point at infinity ($\mathcal{O}$), which is an imaginary element. The inverse of a point $P = (x, y)$ can just be written as $-P = (x, -y)$.

Additionally, there are two operations that can be performed on points on the curve: addition and doubling. These operations have specific formulas based on which form the curve formula has (the aforementioned formula is in Weierstrass form).

Given two points P and Q, the point addition $R = P + Q$ can be computed as follows. First, the slope of the line goes through both P and Q is computed:

$$\lambda = \frac{y_p - y_q}{x_p - x_q}$$

Next, the coordinates of a point R can be computed:

$$x_r = \lambda^2 - x_p - x_q$$
$$y_r = y_p + \lambda(x_r - x_p)$$

Doubling is very similar to addition, the difference is the way the slope is computed (because there is only one point, the difference between the points cannot be used). Instead, the slope can be computed as follows:

$$\lambda = \frac{3x_p^2+a}{2y_1}$$

where a is the parameter from the curve equation. Using this slope, the value of a point $R = 2P$ can be computed be using the aforementioned formulas. Figure 2.1 and 2.2 show point addition and doubling on Weistrass curves from a geometrical point of view. For doubling, the slope is necessary because $2P$ can be found by finding another point P' on the tangent line of P. Mirroring this point on the $x - axis$ yields $R = 2P$.

For cryptography purposes, the elliptic curve is often defined over a prime

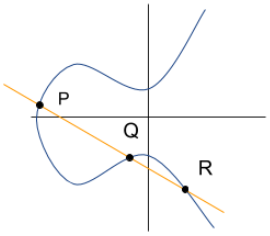

Figure 2.1: Point addition

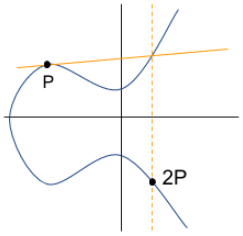

Figure 2.2: Point doubling

field $\mathbb{F}_p$. The points on the curve are those points (x, y) with $x, y < p$ for which the equation $y^2 = x^3 + ax + b$ holds. Counting the number of points on the curve can be hard, but several methods have been devised that make it possible.

The elliptic curve cryptography principle is analogous to the discrete logarithm problem (DLP). DLP relies on the fact that given a, x and $a^b = x$ it is difficult to find b whereas with elliptic curves we are given two points P and Q and the problem is finding a k for which $Q = kP$. There are various ways to compute kP, some of which are discusses in one of the next chapters.

2.1.2 (Twisted) Edwards curves

Edwards curves [4] are a special kind of elliptic curves, they are interesting from a cryptography perspective because they allow more efficient point arithmetic than Weistrass curves. Edwards curves are defined using a different equation which introduces a parameter d:

$$x^2 + y^2 = 1 + dx^2y^2$$

Edwards curves make it possible to use another coordinate system, i.e. the extended twisted Edwards coordinates. An affine point $P = (x, y)$ can be represented as extended twisted Edwards coordinates [7] $(X : Y : Z : T)$ for $x = X/Z$, $y = Y/Z$ and $x \cdot y = T/Z$. These coordinates can be used by different addition and doubling formulas. The definition of these formulas is not relevant in this case. These formulas are also harder to visualize geometrically speaking, whereas the Weistrass formulas are rather intuitive.

2.1.3 Elliptic curve point multiplication

Double-and-add algorithm

The most basic point multiplication algorithm is double-and-add. Given a scalar k, we take its binary representation $k_0 + 2k_1 + 2^2k_2 + ... + 2^nk_n$ where $n+1$ is the number of bits in the binary representation. This binary representation is then used in one of the two versions of this algorithm: the left-to-right algorithm or the right-to-left algorithm. The left-to-right algorithm starts with the most significant bits while the right-to-left algorithm begins with the least significant bits. Both version work by processing one bit at a time, if a bit is zero then the intermediate point is only doubled. Alternatively, if this bit is 1 then an additional point addition is performed. This is done for all bits of the scalar.

Algorithm 2.1: Left-to-Right

```
1  input: k = k_0 + 2^k_1 + ...2^n k_n, P
2  output: Q = kP
3  begin
4    R ← P
5    Q ← 0
6    for i from 0 to n
7      if k_i = 1 then
8        Q ← Q + R
9      R ← 2 · R
10   end
11   return Q
12 end
```

Algorithm 2.2: Right-to-left

```
1  input: k = k_0 + 2^k_1 + ...2^n k_n, P
2  output: Q = kP
3  begin
4    Q ← 0
5    for i from n to 0
6      Q ← 2 · P
7      if k_i = 1 then
8        Q ← Q + P
9    end
10   return Q
11 end
```

Windowed scalar multiplication

Another approach is the windowed scalar multiplication. Instead of processing the scalar bit by bit, the scalar is divided into chunks of a given size, these chunks are called windows. The window size w can be chosen arbitrarily, but it is often 4. Thus, the scalar is written as $k = k_0 + 2^wk_1 + 2^{2w}k_2 + ...2^{nw}k_n$ where n is the number of windows. The upside of this algorithm is that it uses fewer point additions. Point additions are typically more computa-

tionally expensive than point doublings, which is why many efficient scalar multiplication routines try to minimize the number of additions.

The algorithm works by processing one window at a time, starting with the window that corresponds to the most significant bits. For every iteration (window) the input point is doubled w times where w is the window size. Subsequently a point is added, but only if the value of the current window, i.e. k_i is not equal to zero. The points that are added are retrieved from a precomputed table, this table consists of the points $P, 2P, ..., 2^{w-1}P$. Using such a table further lowers the number of additions as the points in this table only have to computed once.

Algorithm 2.3: Windowed scalar multiplication

```
1   input: k = k_0 + 2^w k_1 + 2^{2w} k_2 + ...2^{nw} k_n, P, w
2   output: Q = k * P
3   begin
4     table ← precompute_table(k, P)
5
6     Q ← 0
7     for i from n to 0
8       Q ← 2^w · Q
9       if k_i > 0 then
10        Q ← Q + table[k_i]
11    end
12  end
```

GLV Decomposition

Another approach on scalar multiplication is Gallant-Lambert-Vanstone (GLV) decomposition [6]. Similar to windowed scalar multiplication, the goal is to minimize the number of point additions and/or doublings. This is achieved by replacing these operations by an evaluation of an endomorphism. An endomorphism is a map $\phi : E \to E$ for which $\phi(\mathcal{O}) = \mathcal{O}$. Thus, there is a map which maps every point on some curve E to another point in said curve. An endomorphism that is often used for scalar decomposition is the Frobenius endomorphism $\phi : E \to E$ defined as

$$(x, y) \to (x^q, y^q)$$

This endomorphism map is simply exponentiation of the point coordinates, which can be done in linear time. Suppose that we want to compute kP where k is some scalar and P is some point on a curve. Using GLV, it is possible to compute $kP = k_1P + k_2\phi(P)$ for some k_1 and k_2. Finding these two sub scalars k_1 and k_2 is the decomposition part of the algorithm.

There are multiple ways to perform the decomposition, but one of them is explained in the original GLV paper. The goal is to generate two small

integers $k_1 + k_2\lambda$ (mod n) where λ is the root of the characteristic polynomial of ϕ. They have to be small because otherwise there is not going to be a significant speedup in comparison with other multiplication schemes. It is also possible to see k_1, k_2 as two vectors $(k_1, k_2) \in \mathbb{Z} \times \mathbb{Z}$, who again have to be small. Furthermore, lets say there is a homomorphism $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_n$ defined by $(i, j) \to (i, \lambda j)$ mod n. A homomorphism is a map for which the group operations are preserved, but unlike an endomorphism the domain and codomain are not necessarily equivalent. Now, the goal is to find a vector $u \in \mathbb{Z} \times \mathbb{Z}$ for which $f(u) = k$. Finding such an u for which the this holds is trivial, i.e. if $u = (k, 0)$ then $f(u) = k$, but this is not a good solution as one of the components is not small.

The suggested approach for finding this small vectors first computes two linearly independent vectors v_1, v_2 such that $f(v_1) = f(v_2) = 0$. It is then possible to find a v that is close to $(k, 0)$ in the integer lattice generated by v_1 and v_2. Consequently, $u = (k, 0) - v$ with $f(u) = f((k, 0)) - f(v) = k$. Finding linearly independent vectors and subsequently finding a vector close to the integer lattice can be done by using lattice basis reduction algorithms like LLL[10].

2.2 Four$\mathbb{Q}$

2.2.1 Fundamentals

Four$\mathbb{Q}$[3] is an extremely efficient curve because it combines state-of-the-art elliptic curve cryptography techniques. The result is a curve that is a potential successor of current standard curves as defined by NIST, which makes this curve an interesting research topic.

More formally, Four$\mathbb{Q}$ is defined as a complete twisted Edwards curve $\mathcal{E}(\mathbb{F}_{p^2})$: $-x^2 + y^2 = 1 + dx^2y^2$ where the quadratic extension field $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ for $i^2 = -1$. Furthermore, p is the conveniently chosen Mersenne prime $2^{127} - 1$. All operations are performed on the subgroup $\mathcal{E}(\mathbb{F}_{p^2})[N]$ where N is a 246-bit prime number. The neutral element on the field is defined as $\mathcal{O} = (0, 1)$ and the inverse of a point $P = (x, y)$ is defined as $-P = (-x, y)$.

The curve has a Weierstrass form $\mathcal{E}_w$ and two Twisted Edwards forms $\hat{\mathcal{E}}$ and $\mathcal{E}$. The first form makes it possible to use affine coordinates while the last form enables fast addition formulas using the Extended Twisted coordinates. However, the $\mathcal{E}_w$ is not isomorphic with $\mathcal{E}$, which is why an additional Twisted Edwards form $\hat{\mathcal{E}}$ has to be introduced. The are two mappings $\delta : \mathcal{E}_w \to \hat{\mathcal{E}}$ and $\tau : \hat{\mathcal{E}} \to \mathcal{E}$: and its respective duals δ^{-1} and τ^{-1} available to switch between these forms.

2.2.2 Implementation

Several Four$\mathbb{Q}$ implementations can be found on Github [12], including an ARM implementation with countermeasures. Initially, the idea was to use this implementation, disable the countermeasures and try to perform the attack. However, this implementation (which was originally written for a ARM Discovery board) in combination with the wrapper code used to communicate with the microcontroller resulted in unwanted behavior, e.g. a stack overflow. Attempts to fix this behavior were unsuccessful.

Fortunately, a 32-bit implementation was also available, so this version of the code was used for the attack. Additionally, it also contains a scalar multiplication where the endomorphisms are not used to speed up the multiplication. This scalar multiplication is more straight forward and therefore it is the first target of the attack.

2.2.3 Scalar multiplication without endomorphisms

The scalar multiplication of this implementation uses the following steps:

1. Reduce the scalar modulo the curve order;
2. Convert the scalar to an odd scalar;
3. Precompute the lookup table;
4. Recode the (possible reduced) scalar into pairs of digits and masks;
5. Perform the scalar multiplication using the digits and masks.

Modular reduction

The scalar is reduced modulo the curve order for efficiency. Instead of using standard modulo operation, Montgomery arithmetic [14] is used because it is more efficient. Additionally, this makes the operation resistant against timing attacks and alike because the arithmetic is performed in constant time.

Conversion to odd

After the modular reduction it is checked whether the scalar is an odd number. The scalar has to be an odd number because the digits and masks that are generated during the recoding phase can only represent odd numbers. Converting the scalar is done by adding the curve order, ultimately this means every scalar k is in the set $\{1, 3, 5, ..., 2\#E(\mathbb{F}_p) - 1\}$.

Precomputing the lookup table

The lookup table consists of all points $P, 3P, 5P, ...15P$, these points are simply computed by doubling the P and adding this 2P to the initial P.

This will generate the aforementioned sequence. The negative points do not have to be explicitly computed as these can just be computed by taking the negative value of the x-coordinate. This means only 8 points have to be computed in total.

Recoding the scalar

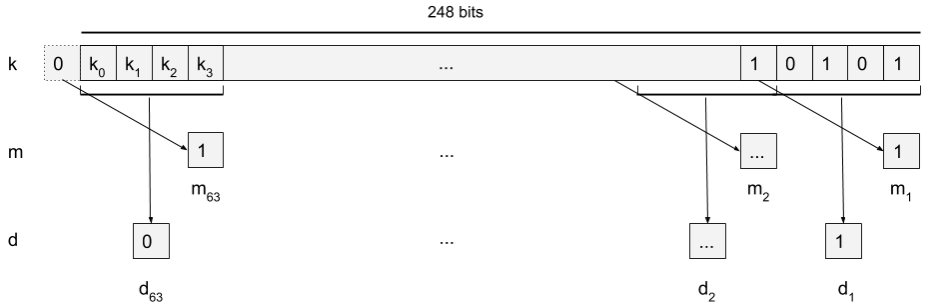

Figure 2.3: Recoding the scalar k into digit/mask pairs

The recoding phase is necessary to facilitate the main loop of the scalar multiplication. The scalar is recoded into a set of digits $\{d_1, ..., d_{63}\}$ and masks $\{m_1, ..., m_{63}\}$ where $d_i \in \{0, ..., 7\}$ and $m_i \in \{0, 1\}$. The recode phase is performed by consuming 4 bits of the scalar at a time, until no bits are left. The recode phase begins with the least significant 4 bits. The 5th least significant is equal to the mask bit. The 4 bits are converted into a digit based on this 5^{th} (mask) bit, namely if this bit is equal to 1 then the digit becomes $\frac{k_i - 1}{2}$, where k_i are the 4 bits being processed. Similarly, if the mask is 0 then the digit is equal to $\frac{(15 - k_i) - 1)}{2}$.

Main loop of the scalar multiplication

This implementation uses a windowed version of the scalar multiplication with window size w = 4. Before the main loop, the last digit and mask are used to initialise our point P. The main loop, consisting of 62 iterations, starts by processing the most significant bits first which corresponds to the "highest (indexed)" digits and masks. For every iteration, P is doubled 4 times and afterwards a intermediate point R from the precomputed table is either added or subtracted. Point R is defined as $R = 2 \cdot d_i + 1$ where d_i is the digit value of the current iteration i. This point is added if the mask m_i is 1 and subtracted if the mask m_i is 0.

As a simple example, suppose we have scalar $k = 1$. Trivially, this means that if the input is point P then the resulting point is also P. The scalar

1 corresponds to 62 digit/mask pairs of $(7, 0)$. This digit/mask pair is a subtraction (mask is 0) of point $R = 2 \cdot 7 + 1 = 15$. Ultimately, this means for each of the 62 iterations P is first multiplied with 16 and subsequently 15P is subtracted. The sequence $((16 \cdot P - 15P) \cdot 16) - 15P)$ is generated which ultimately evaluates to P, which was expected.

2.2.4 Scalar multiplication with endomorphisms

The Four$\mathbb{Q}$ scalar multiplication routine consists of the following steps:

1. Computing the endomorphisms $\phi(P)$, $\psi(P)$, $\psi(\phi(P))$ using explicit formulas;
2. Generate the lookup table using the precomputed endomorphisms;
3. Decomposing the scalar k into a multiscalar (k_1, k_2, k_3, k_4)
4. Recoding the multiscalar (k_1, k_2, k_3, k_4) into pairs of digits and masks.
5. Performing the windowed scalar multiplication using the acquired pairs of digits and masks and the lookup table.

Computing the endomorphisms

The first step is to compute $\phi(P)$, $\psi(P)$, $\psi(\phi(P))$. This is done by evaluating the explicit formulas that can be found in the Four$\mathbb{Q}$ paper. Even though $\psi(\phi(P)) = \phi(\psi(P))$ the former is chosen as evaluating ψ is much faster than ϕ. Ultimately this means ϕ is only evaluated once and ψ twice (per scalar multiplication).

Generating the lookup table

Secondly, all the points in the lookup table are generated. These are the points:

$$T[i] = P + [i_0]\phi(P) + [i_1]\psi(P) + [i_2]\psi(\phi(P)) \text{ for } i = (i_2, i_1, i_0)_2 \text{ in } 0 \leq 1 < 7$$

Since the basic elements $\phi(P), \psi(P)$ and $\psi(\phi(P))$ were already computed in the previous steps, it is simply a matter of adding these points in a smart order so that there are no redundant computations.

Decomposing the scalar

After generating the lookup table, the scalar $k \in [0, 2^{256})$ is decomposed into a 4-dimensional multiscalar $(a_1, a_2, a_3, a_4) \in \mathbb{Z}^4$ such that $k = a_1 + a_2\lambda_\phi + a_3\lambda_\psi + a_4\lambda_\phi\lambda_\psi \pmod{N}$ where $0 \leq a_i \leq 2^{64} - 1$ for $i = 1, 2, 3, 4$ and such that a_1 is odd. These λ_ϕ and λ_ψ are the eigenvalues of the endomorphisms ϕ and ψ which can be precomputed using the curve's parameters.

This 4-dimensional GLV decomposition[13] is a more sophisticated version

of the standard GLV decomposition as discussed in 2.1.3. Instead of only using the Frobenius endomorphism ϕ another endomorphism ψ is used to further reduce the scalar k.

The use of another endomorphism makes the decomposition process significantly more difficult. That being said, one of the big upsides of this particular decomposition is the fact that many values are independent of the scalar. Therefore these values can be precomputed.

In essence, the decomposition is similar to the standard GLV decomposition. There is a zero decomposition lattice

$$\mathcal{L} = \langle (z_1, z_2, z_3, z_4) \in \mathbb{Z}^4 | z_1 + z_2\lambda_\phi + z_3\lambda_\psi + z_4\lambda_\phi\lambda_\psi = 0 \pmod{\mathrm{N}} \rangle$$

and there is a trivial vector $k = (k, 0, 0, 0)$. In combination, they form the lattice coset $(k, 0, 0, 0) + \mathcal{L}$. Given a basis $B = (b_1, b_2, b_3, b_4)$ of $\mathcal{L}$ and a scalar k the Babai rounding technique makes it possible to compute $(\alpha_1, \alpha_2, \alpha_3, \alpha_4) \in \mathbb{Q}^4$ for which $(k, 0, 0, 0) = \sum_{i=1}^{4} \alpha_i b_i$. Subsequently it computes the required multiscalar

$$(a_1, a_2, a_3, a_4) = (k, 0, 0, 0) - \textstyle\sum_{i=1}^{4} \lfloor \alpha_i \rceil \cdot b_i$$

The basis of $\mathcal{L}$ is important as it is used for every scalar decomposition and it impacts the speed of the decomposition. Luckily, this basis can be precomputed using the curve's parameters. Furthermore $\alpha_i = \hat{\alpha}_i \cdot k/N$ where $\hat{\alpha}_i$ are predefined constants.

For every scalar decomposition four roundings $\lfloor \alpha_i \rceil = \lfloor \hat{\alpha}_i \cdot k/N \rceil$ have to be computed, it is more efficient to precompute $\ell_i = \left\lfloor \frac{\hat{\alpha}_i}{N} \cdot \mu \right\rceil$ where μ is a chosen power of 2. These constants can then be used at run time to compute $\left\lfloor \frac{\ell_i \cdot k}{\mu} \right\rceil = \left\lfloor \frac{\left\lfloor \frac{\hat{\alpha}_i}{N} \cdot \mu \right\rceil \cdot k}{\mu} \right\rceil = \lfloor \hat{\alpha}_i \cdot k/N \rceil$. The division of μ can be done by a simply bit shift.

On the flip side, this approach sometimes yields the wrong result, i.e. the resulting may be off by 1. Supposedly, choosing a high μ decreases the possibility of this error, but this is not enough as the occurence of such an error could leak information about the key. Thus the error has to be fully eliminated.

The approximation will either be the correct value $\left\lfloor \frac{\hat{\alpha}_i}{N} \cdot k \right\rceil$ or the incorrect value $\left\lfloor \frac{\hat{\alpha}_i}{N} \cdot k \right\rceil - 1$. In order to remove this roundoff error, the approximation $\lfloor \alpha_i \rceil$ is substituted by $\tilde{\alpha}_i = \lfloor \alpha_i \rceil - \epsilon_i$ where $\epsilon_i \in 0, 1$ for i $= 1, 2, 3, 4$. By using this substitution all possible errors are accounted for.

The last step of the decomposition is to make sure that the multiscalar is in fact odd (i.e. a_1 is odd) as this is required by the recoding phase. In order to accomplish this, two vectors $c = 5b_2 - 3b_3 + 2b_4$ and $c' = 5b_2 - 3b_3 + 3b_4 \in \mathcal{L}$ can be precomputed such that either $(a_1, a_2, a_3, a_4) + c$ or $(a_1, a_2, a_3, a_4) + c'$ will have an odd a_1. Choosing which vector is added can be implemented by using a bit mask.

Ultimately, the multiscalar (a_1, a_2, a_3, a_4) is defined as

$$\begin{aligned} a_1 &= k - \tilde{a_1} \cdot b_1[1] - \tilde{a_2} \cdot b_2[1] - \tilde{a_3} \cdot b_3[1] - \tilde{a_4} \cdot b_4[1] \\ a_2 &= \quad - \tilde{a_1} \cdot b_1[2] - \tilde{a_2} \cdot b_2[2] - \tilde{a_3} \cdot b_3[2] - \tilde{a_4} \cdot b_4[2] \\ a_3 &= \quad - \tilde{a_1} \cdot b_1[3] - \tilde{a_2} \cdot b_2[3] - \tilde{a_3} \cdot b_3[3] - \tilde{a_4} \cdot b_4[3] \\ a_4 &= \quad - \tilde{a_1} \cdot b_1[4] - \tilde{a_2} \cdot b_2[4] - \tilde{a_3} \cdot b_3[4] - \tilde{a_4} \cdot b_4[4] \end{aligned}$$

where k is the original scalar, $\tilde{a_1} = \left\lfloor \frac{\ell_i \cdot m}{\mu} \right\rfloor$ and b_i are the Babai-optimal basis.

Recoding the multiscalar

After decomposing, there is a recoding phase similar to the recoding phase described earlier. The multiscalar (a_1, a_2, a_3, a_4) is converted into pairs of digits $(d_{64}, ..., d_0)$ with $0 \leq d_i \leq 7$ and masks $(m_{64}, ..., m_0)$ with every $m_i \in \{-1, 0\}$.

These digits and masks are used in the implementer version of the GLV-SAC representation[5]. In this representation one of the scalars $k_j \in k$ becomes the "sign-aligner", i.e. the column which determines the sign of the other columns. In case of Four$\mathbb{Q}$, which has 4 scalars (a_1, a_2, a_3, a_4), this "sign-aligner" is the first sub scalar a_1. The recoding converts this scalar to a set of mask values. The other scalars a_2, a_3, a_4 are used to compute the digits.

The column that determines the sign has one restriction: it has to be odd. The decomposition phase conveniently takes care of this. This restriction is necessary because it enables conversion to a full signed nonzero representation. Since there are no "0" values all additions or subtractions will be non-trivial, which guarantees a constant time routine. Formally, the resulting sign column is denoted as $b^J = b_0^J, ..., b_{64}^J$. This conversion is done by taking every bit string 00...1 of a certain length in the column and replacing it by a bit string $1\bar{1}...\bar{1}$ of the same length. One can see easily that $\mathbf{1} \cdot 2^n \mathbf{-1} \cdot 2^{n-1} ... \mathbf{-1} \cdot 2^0 = \mathbf{0} \cdot 2^n + \mathbf{0} \cdot 2^{n-1} + ... + \mathbf{1} \cdot 2^0$ for some arbitrary n.

The value (either -1 or 1) of the sign column at a specific position i also

directly impacts the possible values for the other columns at that same position, denoted as b_i^j. In particular, the only values b_i^j are in $\{0, B_i^j\}$. Thus if the sign column is negative at position i then the other columns are also negative (or zero) at that same position i. This makes sure that a digit can not be represented in multiple ways. For example, the digit 1 can only be represented as $b = (1, 0, 0, 1) = 0 \cdot 2^2 + 0 \cdot 2 + 1 \cdot 1 = 1$ and not by $b = (1, 0, 1, -1) = 0 \cdot 2^2 + 1 \cdot 2 - 1 \cdot 1 = 1$.

After that the "sign-aligner" has been computed using the aforementioned method, the other columns are converted. this is done by consuming all columns b^j bit by bit. Every bit b_i^j of the output column is computed by multiplying the first bit of input scalar with the corresponding sign mask: $b_i^j = k_0^j \cdot b_i^J$. Subsequently the scalar k_i is reduced: $k_i = \lfloor k_i/2 \rfloor - \left\lfloor b_i^j/2 \right\rfloor$.

A nice example of the entire conversion can be found in [[5], §3.1].

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \bar{1} & 1 & \bar{1} & 1 \\ 1 & \bar{1} & 0 & \bar{1} & 0 \\ 1 & 0 & 0 & \bar{1} & 0 \\ 0 & 0 & 1 & \bar{1} & 1 \end{bmatrix}$$

This example contains a multiscalar (k_1, k_2, k_3, k_4), whose components have less bits (5) than the Four$\mathbb{Q}$ multiscalar (64) but this is irrelevant as the conversion itself is identical. k_1 is the sign-aligner and conversion of $0 + 0 + ... + 1$ with $1 - 1 - 1... - 1$ is clearly visible. The digits $(d_4, ...d_0)$ can be read from the columns in the final matrix. $d_4 = (1, 1, 0) = 6, d_3 = (-1, 0, 0) = 4$, etc. Similarly, the masks $(m_4, ..., m_0)$ are just the values in the first row, $m_4 = k_4^1 = 1, m_3 = k_3^1 = -1$, etc.

Performing the scalar multiplication

For the actual scalar multiplication the initial result point is first set using the last digit and mask. The last last mask is always -1, which means the initial point is always positive. After that, the other digits are processed digit by digit, starting with the highest digit. For every digit, the point is doubled once and one point from the lookup table is added. A mask of -1 corresponds to adding a point while a mask of 0 means subtraction of that point. The point that is added is determined by the digit, for example $d_i = 5 = (\mathbf{1}, \mathbf{0}, \mathbf{1})_2$ means point $P + \mathbf{1} \cdot \phi(P) + \mathbf{0} \cdot \psi(P) + \mathbf{1} \cdot \psi(\phi(P)) = P + \phi(P) + \psi(\phi(P))$ is either added or subtracted.

2.3 Side channel analysis

2.3.1 Fundamentals

Side channel analysis (SCA) is based on the fact that certain physical aspects can be measured during computations, these measurements can then be used to distinguish specific operations of the computation. The most basic form of SCA is a timing attack [8], which means it is measured how long the computations take. In the context of double-and-add, an addition is only performed if the bit being processed is a '1'. This means that keys which have a lot of ones will have a longer computation time than a key with a small number of ones. Ultimately this means you can extract information about the key.

Another interesting physical aspect that can be exploited is the power usage of a computation. The power measurement of a computation is called a power trace. Such a trace consists of samples, which are basically values that indicate how much power was used at a specific time during the computation. Fundamentally, there are two different approach for power based side channel analysis: vertical and horizontal. The former involves taking multiple power traces in an attempt to find a (statistical) pattern. In the latter approach, only one trace is analysed and the goal is to find a pattern by looking at the trace horizontally. Below one horizontal (SPA) and one vertical attack (DPA) is discussed more in depth.

2.3.2 Simple power analysis (SPA)

Simple power analysis [9] is the most basic form of power analysis. It involves visually interpreting the traces in order to recognise certain operations. For example, if we consider an RSA implementation which uses exponentiation by squaring, we could distinguish the square and multiply operations. These operations directly relate to the bits of the RSA key. Ultimately, this means that secret keys used can be directly retrieved without any difficult (statistical) analysis. There can be situations when even one trace is sufficient to retrieve the keys.

2.3.3 Differential power analysis (DPA)

SPA can be easily prevented by implementing the exponentiation without branching. When targeting such an implementation, Differential power analysis becomes a more suitable approach. DPA is a statistical approach which makes it possible to detect smaller variations of the data but unlike SPA, DPA always requires many traces to retrieve the key.

For the attack, first an intermediate variable is chosen that can be ultimately

used to distinguish a correct and an incorrect hypotheses. Subsequently, a set of all possible values for this intermediate variable is generated. The value of the intermediate variable is computed for all of these inputs and the corresponding traces are grouped into two subsets. For both subsets the average curve is computed, these averages curves can be used to get the differential curve. This differential curve will only show a peak if the right key was guessed. If the guessed key was wrong then the differential curve will be flat.

2.3.4 Online template attacks

Description

An online template attack [1] is a more sophisticated technique, it makes use of the fact that one can guess which point is being doubled during an iteration of the scalar multiplication. The attack begins by obtaining a power trace of the encryption operation when the target scalar is used, this trace is the so called *target trace*.

The goal is to determine the bit(s) used in every iteration of the scalar multiplication of the target trace, similar to SPA. Suppose a right-to-left version (Algorithm 2.2) of the double-and-add scalar multiplication is being used. In every iteration a point is being doubled regardless of the bit value, the exact point that is being doubled at a certain iteration is unknown because we do not know the scalar. However, it is possible to make an educated guess.

More specifically, in the first iteration of this algorithm P is doubled. After that, if $k_n = 1$, P is added to the result of the doubling, this means that in the second iteration the intermediate result is either $\mathcal{O}$ or P. The doubling in the second iteration allows you to distinguish these points. In particular, one could send two scalars $k = 0$ and $k = 1$ and retrieve traces of the computations with these scalars as input. The next step is to correlate both of these *template traces* with the target trace. Only the doubling in the second iteration of each template trace is correlated with the doubling in the second iteration of the target trace, as this is the part of the trace that is dependent of the (first) bit value.

This process can be repeated, that is, the doubling in the third iteration allows you to determine the second bit of the scalar, the fourth iteration the third bit and so fort. Following this principle, all bits of the scalar can be retrieved.

Correlation

The Pearson correlation coefficient is the most widely used correlation coefficient. The coefficient gives a value between -1 and 1 based on how well two variables are linearly correlated. The coefficient can be computed as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})}}$$

The covariance of the variables x and y is divided by the product of their respective standard deviations. The covariance measures the similarity between x and y. Dividing this by the product of the standard deviations results in a weighted coefficient between -1 and 1 disregarding the (possibly small) variety of the input variables.

Chapter 3

Attack on FourQ without endomorphisms

3.1 Setup

The target of the attack is an ARM Cortex M4 (Blue) with most of the capacitors removed in order to reduce noise. The board is connected in series to an 2GHz LeCroy oscilloscope (Orange) and a separate power source (Red). The oscilloscope is configured at a sampling rate of 250Ms/s, this means the traces will consist of roughly 15 million samples. Every single sample has a value between 0 and 256. A trigger (green) makes it possible to starting measuring on demand and stop when the computation has finished. The probe (purple) actually measures the power and is directly connected to the oscilloscope.

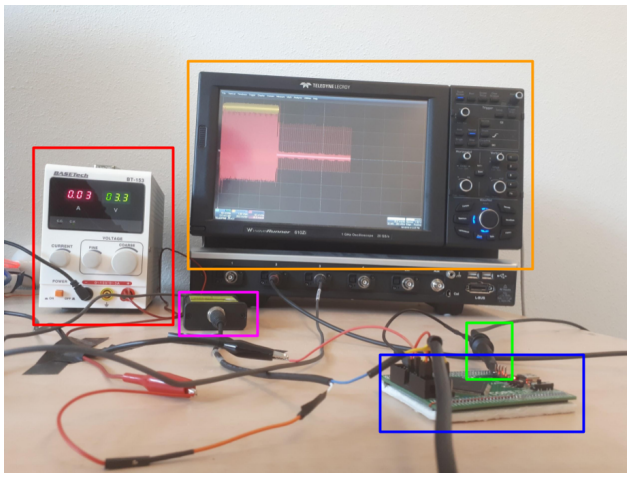

Figure 3.1: Setup for the attack, different parts are highlighted and explained below

3.2 Adapting OTA

The paper on online template attack discusses various situations where such an attack can be used and how to adapt the attack. The most basic application of this attack is performed on the double-and-add-always algorithm. For this algorithm, there are only two templates per iteration (2P and 3P), which means only two traces per iteration. In an ideal situation, when there is not much noise, even one template could suffice. If the correlation of the correct template is significantly higher than the correlation for the incorrect template, a threshold could be introduced that separates the correct and wrong templates. This would remove the necessity to get a trace for both templates.

However, FourQ works differently: there are a 8 precomputed points (P, 3P, 5P, ..., 15P) that are either added or subtracted at every iteration. This makes a total of 16 possibilities for every iteration and therefore 16 templates. After the recoding phase there are 63 digits and corresponding masks, with the main loop of the scalar multiplication consisting of 62 iterations. The last digit mask pair (d_{63}, m_{63}) is always equal to $(0, 1)$ because this is used to initialise point P, which is the point will be multiplied.

Ultimately, this means that 16 templates have to be constructed for 61 iterations, with the first iteration being the exception. This iteration has fewer templates because not all digits and masks are possible (a direct result of the modular reduction). In particular, we know that the scalar is in the range $(1, 2\#E(\mathbb{F}_p) - 1)$. Only scalars build with the digit/mask pairs (5, 0), (6, 0) and (7, 0) in the first iteration yield integers that are in this range, therefore there are only 3 templates for the first iteration.

Construction of these templates (scalars) given the digits and masks is rather straightforward. It is just a matter of mimicking the windowed routine that is used for the scalar multiplication. However, instead of working on some point P we just use an integer k.

Algorithm 3.1: Constructing a scalar

```
1  input: int[] d, int[] m
2  output: int k
3  begin
4    k ← 1
5    for i ← 62 to 1
6      k ← k * 16
7      if m[i] = 1 then
8        k ← k + d[i] * 2 + 1
9      else
10       k ← k − d[i] * 2 + 1
11   end
12   return k
13 end
```

3.3 The attack

The first step is to retrieve the target trace, this is the trace contains the computation with the (target) scalar. After that 62 iterations are performed to retrieve all the digit/mask pairs. For every iteration, all 16 possible templates are generated and corresponding traces are obtained, these traces are compared with the target trace by computing the Pearson correlation coefficient. The digit/mask pair with the highest correlation is chosen and stored so that it can be used in the next iteration. Ultimately, when all 62 iterations have completed we can build the scalar and check whether it is equal to our target scalar.

Algorithm 3.2: Online template attack

```
1   input: int target_scalar
2   output: bool success
3   begin
4     target_trace = trace(target_scalar)
5
6     key_digits ← [0]
7     key_masks ← [1]
8     for i ← 1 to 62
9       correlations ← []
10      for d ← 1 to 7
11        for m ← 0 to 1
12            digits ← [d] + key_digits
13            masks ← [m] + key_masks
14            pad(digits, 7, 62)
15            pad(masks, 0, 62)
16
17            template_scalar ← recode(digits, masks)
18            template_trace = trace(template_scalar)
19
20            correlation ← correlate(template_trace, target_trace)
21            append(correlations, correlation)
22          end
23        end
24        best_digit, best_mask = max(correlations)
25        prepend(key_digits, best_digit)
26        prepend(key_mask, best_mask)
27    end
28    scalar = recode(key_digits, key_masks)
29    return scalar == target_scalar
30  end
```

After the key digits and masks are retrieved, reconstructing the scalar is not always trivial. The scalar may have been reduced modulo the subgroup order and subsequently converted to odd. The conversion to odd is simply an addition with the subgroup order, which is very convenient because the modular reduction makes sure that all scalars are lower than the subgroup. Scalars that are higher than the subgroup are scalars that are converted to odd, this allows us to distinguish odd scalars and scalars that were even and converted to odd.

The modular reduction is more difficult, there is no trick we can apply to return the scalar that was used. However, the difference between the scalar length (256 bits) and the curve order (246 bits) is only 10 bits which corresponds to about 2^{10} possible variations. This can be brute forced by consumer grade hardware.

3.4 Results

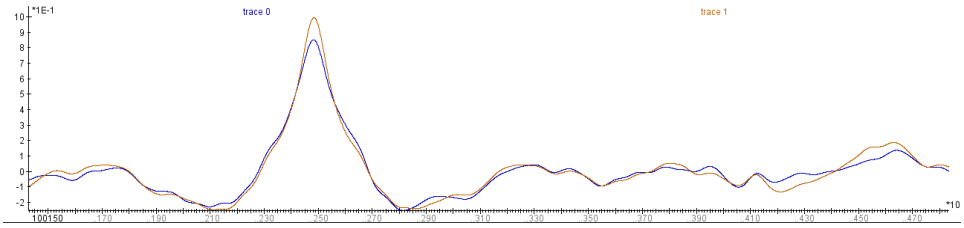

Figure 3.2: The difference in correlation between a correct template (red line) and an incorrect template (blue).

The initial results when running the aforementioned algorithm were somewhat unsatisfying, eventhough the difference in correlation between correct templates and incorrect templates was often significant (i.e. $0.98 - 1$ correlation for correct template and < 0.93 for incorrect) there were also some deviations. That is, the correlation with the correct templates is not consistently ~ 0.99, but irregular correlations of ~ 0.35 also occur. This makes it hard to "retrieve" the scalar consistently, because one wrong digit or mask immediately makes the entire online template attack invalid.

To overcome this problem, more than one trace per template is retrieved and the trace with the highest correlation is selected. This will make sure that the correct template will have an correlation close to 1. However, this approach also has a few downsides. For starters, the number of traces increases significantly, especially because in our case just adding one extra trace is not enough.

Also, because the trace with the highest correlation is chosen for every template the correlations are going to be high for every template. This might not be a problem for the first couple of iterations, but as the iterations increase the offsets become somewhat misaligned reducing the correlation with the correct template.

Ultimately, applying this procedure yielded a success rate close to 50%, but this success rate can be improved by finding the cause of the deviations. One could also further increase the number of traces per template, but as

this does not fix the underlying problem therefore some might call this bad practice.

3.5 Suggestions for further improvement

The fact that there are 61 iterations with 16 templates each iteration means that roughly 1000 traces have to be measured, processed and correlated. This is a very time-consuming process, it would be advantageous if we could reduce the number of traces. Similar to the situation in (2.1) we could come up with a threshold that instantly classifies a template as correct. Assuming an even distribution of digits and masks, this would mean that the run time is halved. Care must be taken when choosing such a threshold, as overfitting might be a concern.

Chapter 4

Attack on Four$\mathbb{Q}$ with endomorphisms

4.1 Adapting OTA

4.1.1 Analyzing the routine

The goal is to attack the efficient Four$\mathbb{Q}$ implementation in a similar fashion. Again, the first step perform the actual online template attack. If the attack succeeds then it is known which digits and masks were used. The next step is scalar reconstruction, i.e. converting the digits and masks into the original scalar, as this is the secret value of interest.

The scalar multiplication routine has a main loop that consists of 64 iterations. In every iteration the point is first doubled and subsequently a point from the lookup table is added. Thus, the doubling of iteration $i + 1$ depends on the addition of iteration i, which in theory makes the loop vulnerable for a online template attack. The table consists of 8 points in total, which means that including the negative points there are again 16 templates for every iteration.

Template generation however, is significantly more difficult. This begins by the fact that the points in the lookup table are less predictable, i.e. where in the previous case the lookup table consisted of the points $P, ..., 15P$ whereas the lookup table now has the points $P, ..., P + \phi(P) + \psi(P) + \psi(\phi(P))$.

Additionally, it is important to keep in mind that the templates are in fact scalars. That being said, the template scalar that is given as input is not equal to the scalar that is being used at the recoding phase because scalar decomposition is applied before the recoding. This means that in order to generate correct templates, not only the recoding phase has to be reversed

but also the decomposition phase. This is also necessary to reconstruct the scalar when the attack is finished, i.e. after the correct digits and masks have been retrieved.

4.1.2 Reversing the decomposition

Given a multiscalar (a_1, a_2, a_3, a_4) the goal is to find the original scalar k. The multiscalar (a_1, a_2, a_3, a_4) is defined as

$$\begin{aligned} a_1 &= k - \tilde{a_1} \cdot b_1[1] - \tilde{a_2} \cdot b_2[1] - \tilde{a_3} \cdot b_3[1] - \tilde{a_4} \cdot b_4[1] \\ a_2 &= \quad - \tilde{a_1} \cdot b_1[2] - \tilde{a_2} \cdot b_2[2] - \tilde{a_3} \cdot b_3[2] - \tilde{a_4} \cdot b_4[2] \\ a_3 &= \quad - \tilde{a_1} \cdot b_1[3] - \tilde{a_2} \cdot b_2[3] - \tilde{a_3} \cdot b_3[3] - \tilde{a_4} \cdot b_4[3] \\ a_4 &= \quad - \tilde{a_1} \cdot b_1[4] - \tilde{a_2} \cdot b_2[4] - \tilde{a_3} \cdot b_3[4] - \tilde{a_4} \cdot b_4[4] \end{aligned}$$

where b_i are constants and $\tilde{\alpha_i} = \left\lfloor \frac{\ell_i \cdot k}{\mu} \right\rfloor$ for which $\mu = 2^{256}$ and ℓ_i are constants. The first equation a_1 is the sign-aligner (mask) column which is slightly different than the other columns which are used to generate the digits. The problem with using these equations to find k is that you have to find a solution for k which holds for all equations. Simply using a_1 to find k will not be sufficient as this solution is only correct for a_1. Additionally, the floor rounding is used for computing $\tilde{a_i}$ which may be a problem when rewriting these equations.

Alternatively, one could use the equivalence $k \equiv a_1 + a_2\lambda_\phi + a_3\lambda_\psi + a_4\lambda_\psi\lambda_\phi \pmod{\text{N}}$ to find the original k. It seems to be a matter of simply plugging in the values a_1, a_2, a_3, a_4 as the eigenvalues λ_ψ and λ_ϕ can explicitly be computed using the curve's parameters:

$$\lambda_\psi = 4 \cdot \frac{p+1}{r} \pmod{\text{N}} \qquad\qquad \lambda_\phi = 4 \cdot \frac{(p-1)r^3}{(p+1)^2 V} \pmod{\text{N}}$$

where $p = 2^{127} - 1$, N is the order of the subgroup and V and r are fixed constants.

4.1.3 Reversing the recoding

Reversing the scalar recoding can also be done in two ways: rewriting the digits and masks as $b_1 = (m_{64}, ..., m_0)$ and $b_j = (d_{64}[j-1], ..., d_0[j-1])$ where $d_i[k]$ is the kth bit of the ith digit. Another approach is to use the pairs of digits and masks directly. The first approach has the disadvantage that roundings are used which can be difficult in a reversing situation. The latter approach is easier, the implementation version of the algorithm contains solely simple algebraic operations such as additions, subtractions and bit shifts.

The masks are computed using only a_1, therefore with only the masks it should be possible to retrieve a_1. The loop in the recoding reverses the bits, i.e. the least significant bits become the most significant bits. Additionally, for every iteration i the (i + 1)th bit is selected as resulting bit. This means that the 0th bit is not in m, however since a_1 is odd we already know this bit. Ultimately, this means we can define a_1 as follows:

$$a_1 = (0, m_{63}, ..., m_1, 1)_2$$

Converting the digits to a_2, a_3, a_4 is similar. Every digit is defined as $d_i = 4a_4 + 2a_3 + a_2$, but in order to find the original a_2, a_3, a_4 it is necessary to find out how the scalar is being reduced. The scalar is reduced by $a_j = (a_j >> 1) + c$, which means to get the original a_j the value of c has to be subtracted and shifted back, i.e. $a_j = 1 << (a_j - c)$.

However, the bit that was shifted during the recoding is lost. Simply shifting back once does not restore this value, it simply appends a zero. Instead of only doing a shift back, the bit that was lost has to be added. This bit is stored in d_i as described earlier, and can thus be extracted using a mask: $a_j = d_i >> j$.

The full reversing routine (4.2) is given below, next to the original recoding scheme (4.1).

Algorithm 4.1: Original recoding

```
1  input: a_j = (0, a_j[63], ..., a_j[0]) for 1 ≤ j ≤ 4
2  output: (d_64, ..., d_0), (m_64, ..., m_0)
3  begin
4    m_64 = −1
5    for i from 0 to 63
6      d_i ← 0
7      m_i ← −a_1[i + 1]
8      for j from 2 to 4
9        d_i ← d_i + (a_j[0] << (j − 2))
10       c ← (a_1[i + 1]|a_j[0])^a_1[i + 1]
11       a_j ← (a_j >> 1) + c
12
13   return (d_64, ..., d_0), (m_64, ..., m_0)
14 end
```

Algorithm 4.2: Reverse recoding

```
1  input: (d_64, ..., d_0), (m_64, ..., m_0)
2  output: a_j = (0, a_j[63], ..., a_j[0]) for 1 ≤ j ≤ 4
3  begin
4    a_0 ← 1
5    for i from 0 to 63
6      a_1 ← −m_1[i + 1]
7      for j from 2 to 4
8        bit ← (d[i] >> (j − 2))&1
9        a_i ← a_i − (a_1[i]|bit) + bit
10
11   return
12       a_j = (0, a_j[63], ..., a_j[0]) for 1 ≤ j ≤ 4
13 end
```

4.2 The attack

As noted before, the main loop consists of 64 iterations starting with the highest indexed digits. These digits correspond to the least significant bits in the multiscalar, e.g. d_{63} corresponds to $a_j[1]$ for $j \in 2, 3, 4$. The goal is to find all digits and masks, starting with this d_{63} and m_{63} as it is the

digit/mask pair used in the first iteration. In order to find the correct value for this pair all possible pairs of digits and masks have to be generated, these pairs are then converted to scalars by first applying the reverse recoding and subsequently applying the reverse decomposition. Similar to before, these scalars are used as input to retrieve the template traces that are matched with the target trace to find the correct digit/mask pair.

When the implementation without endomorphism was attacked, it was sufficient to fill in unknown digits with 7's and masks with 0's, which would directly correspond to 0's in the template scalar. This was perfect in a sense that the resulting scalars were very predictable which made it easier to make the scalars fit a specific format (e.g. smaller than the subgroup order N). In this case, padding is more difficult because of the intermediate decomposition phase. For the attack, the digits/mask pairs that are generated are converted into a scalar which is used as input. The Four$\mathbb{Q}$ routine will decompose and recode this scalar, but as it turns out the resulting digit/mask pairs are not always equal to the initial digit/mask pairs.

Experiments show that for low scalars, i.e. scalars lower than 2^{64}, the

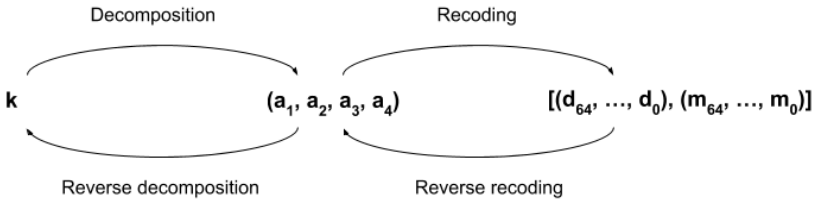

Figure 4.1: Overview of the different forms of the scalar, the conversions on top are performed by *Four*$\mathbb{Q}$ while the conversions below are necessary to build the template scalars

reverse recoding and decomposition is almost always correct, which means that decomposing and recoding this scalar again will result in the original digit and mask values. However, when higher scalars are generated the resulting digits and masks are almost always incorrect. For example, scalars bigger than 2^{246} will always result in incorrect digit/mask pairs. This inconsistency stems from the fact there are multiple scalars whose decomposition result in the same multiscalar. This is really problematic as for OTA to work the generated template scalars need to be reliable.

The main problem is that it is not possible to predict whether a scalar will be correct. In particular, when trying to find the first digit/mask pair, all values for d_{63} and m_{63} have to be tried. To make a template, the remaining digit and masks could for example be randomized, but the chance that converting these digits and masks result in a correct scalar is very slim. Brute forcing to find a correct template is not an option as the number of options is simply too big. Additionally, it would mean brute forcing 16 templates for all 64 iterations. This is simply an infeasible amount of possibilities.

Nevertheless, lets suppose that it would be possible to find such digits, in that case we could launch an attack similar like the attack that was used on the windowed scalar multiplication version of Four$\mathbb{Q}$. The following algorithm could function as a boilerplate for such an attack. In this boilerplate, *unrecode* is the reverse recoding function that was defined earlier and *undecompose* is the equivalence to compute k. The *pad* function is what still needs to be defined.

Algorithm 4.3: Boilerplate for an Online template attack on Four$\mathbb{Q}$

```
1  input: int target_scalar
2  output: bool success
3  begin
4    target_trace = trace(target_scalar)
5
6    key_digits ← []
7    key_masks ← []
8    for i ← 1 to 64
9      correlations ← []
10     for d ← 1 to 7
11       for m ← 0 to 1
12         digits ← [d] + key_digits
13         masks ← [m] + key_masks
14         pad(digits, ?, 64)
15         pad(masks, ?, 64)
16
17         multiscalar ← unrecode(digits, masks)
18         template_scalar ← undecompose(multiscalar)
19         template_trace = trace(template_scalar)
20
21         correlation ← correlate(template_trace, target_trace)
22         append(correlations, correlation)
23       end
24     end
25     best_digit, best_mask = max(correlations)
26     prepend(key_digits, best_digit)
27     prepend(key_mask, best_mask)
28   end
29   scalar = undecompose(unrecode(key_digits, key_masks))
30   return scalar == target_scalar
31 end
```

Chapter 5

Related Work

As of now, there are not many papers discussing the practicalities of online template attacks, however there are papers which discuss side channel analysis of Four$\mathbb{Q}$. Z. Liu et al. [11] have written an extensive paper on side channel analysis of Four$\mathbb{Q}$. In their paper, they implement the three countermeasures proposed by Coron [2] in such a way that overhead on the scalar mulitplication is minimal. These countermeasures(point blinding, scalar randomization and projective coordinate randomization) are mostly useful against vertical attacks. The authors further attempt to perform DPA on their protected scheme and show that the countermeasures significantly increase the effort necessary to perform such an attack. Using this knowledge, they claim that the countermeasures also protect the implementation against similar (vertical) attacks, such as refined power attack, doubling attack and several correlation and collision attacks.

Chapter 6

Future work

The attack on the efficient version of Four$\mathbb{Q}$ was unsuccessful. However, this does not mean that it is impossible. The only thing that makes this attack difficult is the decomposition, without it executing the attack would be very similar to the attack on the first version of the algorithm.

One approach at making the attack work is by reversing the decomposition using the derived formulas. This approach at reversing the decomposition is more difficult because there is a set of equations for which the scalar k has to hold. Alternatively, one could take another look at generating correct templates by using the equivalence, there could exist a method which cleverly chooses the digits and masks to ensure a correct scalar.

Additionally, it would be interesting to see if an online template attack can be used if one or more of the countermeasures (point blinding, scalar blinding, etc.) is enabled. The original paper mentions that is indeed possible to bypass scalar blinding because the retrieved (blinded) scalar is in fact equivalent to the original scalar.

Chapter 7

Conclusions

It was already known that online template attacks could be performed on a wide range of elliptic curve schemes, but it is now confirmed that it can also be used on a windowed scalar multiplication scheme. Most of the time the difference in correlation between the correct and incorrect template was significant. That being said, the fact that it seemed not possible to make the attack work consistently was rather disappointing. It was very unfortunate that the ARM optimized version of the algorithm in combination with the wrapper code did not work in the first place. This in combination with the anomalies as described in the Results section also made the attack much slower than anticipated.

As for the efficient Four$\mathbb{Q}$ implementation, the decomposition phase turned out to be a real obstacle. It is not proven that it is impossible to perform OTA, but the fact that scalar decomposition makes the attack much more difficult is a good aspect of Four$\mathbb{Q}$ either way. Being able to reliably generate template scalars is a fundamental part of OTA and decomposition seems to be a good first defense. The boilerplate presents an first attempt to do an attack, but does not offer a solution to the problem on hand. This solution of reliably generating correct template scalars might require a very different approach.

Bibliography

[1] L. Batina, Ł. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall. Online template attacks. *Journal of Cryptographic Engineering*, Aug 2017.

[2] J.S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. 1999.

[3] C. Costello and P. Longa. Fourq: four-dimensional decompositions on a q-curve over the mersenne prime. 2015.

[4] H. Edwards. A normal form for elliptic curves. 2007.

[5] Armando Faz-Hernández, Patrick Longa, and Ana H Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on glv-gls curves (or keep calm and stay with one (and p¿ 3)).

[6] Robert P Gallant, Robert J Lambert, and Scott A Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Annual International Cryptology Conference*, pages 190–200. Springer, 2001.

[7] H. Hisil, K. Koon-Ho Wong, G. Carter, and E. Dawson. Faster group operations on elliptic curves. 2009.

[8] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[9] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. 1998.

[10] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[11] Z. Liu, P. Longa, G.C.C.F. Pereira, O. Reparaz, and H. Seo. Fourq on embedded devices with strong countermeasures against side-channel attacks. 2017.

[12] P. Longa. Fourqlib. `https://github.com/Microsoft/FourQlib`, 2017.

[13] Patrick Longa and Francesco Sica. Four-dimensional gallant–lambert–vanstone scalar multiplication. *Journal of Cryptology*, 27(2):248–283, 2014.

[14] P.L. Montgomery. Modular multiplication without trial division. 1985.