FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Side-channel Attacks on Supersingular Isogeny Diffie–Hellman Key Exchange
Student:	Bc. František Kovář
Supervisor:	Ing. Jiří Buček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2022/2023

Instructions

Supersingular isogeny Diffie–Hellman key exchange (SIDH) is a post-quantum asymmetric scheme for symmetric key establishment.

* Study the algorithm SIDH for establishing a shared key.
* Research existing implementations and attacks, focus on side-channel attacks.
* Find a suitable implementation and demonstrate its function on a 32-bit microcontroller such as ARM Cortex M4 or similar.
* Design a side channel attack that will allow you to obtain secret information about the private key.
* Implement the proposed attack and experimentally evaluate the results.
* Analyze possible countermeasures.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 24 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Side-channel Attacks on Supersingular Isogeny Diffie–Hellman Key Exchange

František Kovář

Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.

June 23, 2022

Acknowledgements

I would like to thank everyone who supported me during my work on this thesis. First of all, I would like to thank my supervisor Ing. Jiří Buček, PhD. for his incredible patience and every valuable advice, that helped me to finish my thesis. An exclusive thanks belong to my sister, Ing. Michaela Kovářová, for her endless support and unbelievable motivation during all my studies.

Declaration

In Prague on June 23, 2022 . . . .

Citation of this thesis

Kovář, František. *Side-channel Attacks on Supersingular Isogeny Diffie–Hellman Key Exchange.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

V této diplomové práci jsme se zaměřili na aktuálně alternativního kandidáta pro proces postkvantové standardizace NIST. Supersingular Ksogeny Key Encapsulation (SIKE) je jediným kryptosystémem založeným na izogeniích v procesu standardizace. Hlavní důraz byl kladen na analýzu postranních kanálů (SCA) SIKE a experimentální testování možných útočných vektorů pomocí CPA vůči oficiálně předložené referenční implementaci. K tomu jsme použili ChipWhisperer-Lite, který obsahuje STM32F303. Kromě toho jsme se zaměřili i na možná protiopatření proti SCA. Přestože byl útok neúspěšný, shromáždili jsme mnoho zajímavých informací o SCA proti SIKE.

Klíčová slova SIKE, SIDH, Analýza postranními kanály, Post-kvantové zřízení klíče, Korelační odběrová analýza, Eliptické křivky, Kryptografie založena na isogeniích, ARM Cortex-M4

Abstract

In this master's thesis, we aimed at the currently alternative candidate for the NIST post-quantum standardization process. Supersingular Ksogeny Key Encapsulation (SIKE) is the only isogeny-based cryptosystem in the standardization process. The main focus was on the side-channel analysis (SCA) of SIKE and experimentally testing the possible attack vectors, using CPA, in the officially submitted reference implementation. For that, we used ChipWhisperer-Lite, which features an STM32F303. Apart from that, we also focused on the possible countermeasures against SCA. Although the attack was unsuccessful, we have gathered a lot of interesting information regarding SCA against SIKE.

Keywords SIKE, SIDH, Side-channel analysis, Post-quantum key exchange, Correlation power analysis, Elliptic curve, Isogeny-based cryptography, ARM Cortex-M4

Contents

List of Figures

List of Tables

Introduction

One of the essential things these days is cryptography. Most people are not even aware there exists such a thing. One building block uses symmetric cryptography to encrypt the transmitted data. But to achieve that, we would have to apriori share a secret key with everyone else, which is not that much possible. Luckily we have other options to do so. There are algorithms such as Diffie-Hellman or Elliptic curve Diffie-Hellman. Both are key agreement protocols that allow two or more parties to establish a shared secret key over an insecure channel. With the rise of quantum computers and threats coming from building one, such as Shor's algorithm, towards any public key infrastructure built over factoring numbers or discrete logarithm problem [6].

That is why NIST announced a standardization process toward post-quantum cryptography, meaning algorithms that will provide asymmetric encryption, shared key agreement, signatures, and will be able to withstand so far known quantum algorithms. One alternative candidate is the Supersingular Isogeny Key Encapsulation algorithm based on Supersingular Isogeny Diffie-Hellman [3].

SIKE, or rather the SIDH, is the only isogeny-based algorithm in the standardization process. It is based on the hardness of finding large order isogenies between supersingular elliptic curves. The algorithm seems to be secure, but there are still possibilities to break an algorithm not only by focusing on its core problem. So we will look into its reference implementation and elaborate on whether and how this algorithm could be broken using side-channel analysis. We will analyze a 32-bit ARM Cortex M4 with the reference implementation or similar microcontroller because of the rise of embedded devices and the requirement of securing them. The result of this attack should yield part or the entire secret key. At last, we will research possible countermeasures toward already made attacks using the side-channels analysis.

Chapter 1

Mathematical background

1.1 Groups and fields

Before we jump into the definition of finite fields and their extension fields, we will introduce the concept of a field itself. We will need to define a group as a mathematical object and an operation with that group. This operation must fulfill some properties to form a group and potentially an abelian group. Suppose we use two operations instead of one, and these two operations also have the required properties. In that case, we can talk about that mathematical object as a field, or a commutative field. More detailed information are available in [7, 8, 9, 10].

1.1.1 Definition of a group

We need two ingredients to define a group or rather a commutative group. First of them is a non-empty set of objects denoted M furthermore named as a carrier set or domain, M can be finite or infinite. A binary operation $\circ$ that maps every possible pair of elements of M to another element of M. It is a function such as $\circ : M \times M \to M$.

1.1.2 Forming a group

To form a group, we need the ordered pair $(M, \circ)$ to have these fundamental axiom properties satisfied.

1. The operator $\circ$ has to be associative, such that for any elements $a, b, c \in M$ we have $(a \circ b) \circ c = a \circ (b \circ c)$.

2. There exists an element $e \in M$ such that for any element $a \in M$, $a \circ e = e \circ a = a$.

3. For every element $a \in M$, there exists an element $b \in M$ such that $a \circ b = b \circ a = e$. Further more this element will be denoted as a^{-1}, or $-a$, or simply an inverse element of a.

Instead of our general operator $\circ$, suppose we are using $\oplus$, an addition as defined in elementary school $5 \oplus 3 = 8$ and an operator $\odot$ the same way around but multiplication $5 \odot 3 = 15$. Without thinking, we used this in a group if we were to define a sufficient carrier set M for it. For the operator $\oplus$ it is enough to use only $M = \mathbb{Z}$. As for the operator $\odot$ we would miss one part of the group property, which is having an inversion to every element of the carrier set. For that, we will use a bigger carrier set, which is the set of all numbers in $M = \mathbb{Q} \setminus \{0\}$. For the additive group, the neutral element $e = 0$, for the multiplicative group, the natural element $e = 1$.

To shorten extended expressions where $a \in M$ appears in a composition with itself n-times, where $n \in \mathbb{Z}$, we will use a shortened version:

$$n \cdot a = \underbrace{a \oplus \cdots \oplus a}_{n \text{ summands}}$$

for the additive operator and for the multiplicative operator we will use:

$$a^n = \underbrace{a \odot \cdots \odot a}_{n \text{ factors of } a}$$

1.1.3 Properties of a group

A few group properties that we are interested in are the following:

1. commutative group, or also called abelian, the group operation must support the property of commuting the elements, such that for every $a, b \in M, a \circ b = b \circ a$.

2. A subgroup is a non-empty subset N of M, where there is the same binary operation $\circ$, and the elements together with the operation behave as a group. There always exist two trivial subgroups, M itself and a set containing only the neutral element e.

3. Order of a group is defined as the number of elements in the set the group is using. This will be denoted as $|M|$.

4. Suppose we have a multiplicative group. An element a is called a generator of the group, if any element $b \in M$ can be created in a way that $b = a^n$, for some integer n, we write it as $\langle a \rangle$. Any element can be a generator of a group or at least a subgroup. If we were to take multiple subgroup generators $b, c \in M$, that would together generate the whole group, we could write $\langle b, c \rangle$.

1.1.4 Definition of a field

A triplet $K = (M, \oplus, \odot)$, is a nonempty set M with two binary operations $\oplus, \odot$. The operations must satisfy these following conditions, so the underlying triplet K is a **field**.

1. Suppose we take only $(M, \oplus)$, as we need that this form an abelian group as defined above in the section 1.1.2 and in section 1.1.3.

2. The ordered pair $(M \setminus \{0\}, \odot)$ must form a group.

3. The distribution law must apply, such that for every $a, b, c \in M$, $(a \oplus b) \odot c = a \odot c \oplus b \odot c$, and the other way around $c \odot (a \oplus b) = c \odot a \oplus c \odot b$.

The results $a \odot c \oplus b \odot c$ and $c \odot a \oplus c \odot b$, don't necessarily have to be equal as we have just built only a field. If it were true that these two results were equal for any $a, b, c \in M$, we would have a commutative field.

For example let's use the domain $\mathbb{R}$ with the two binary operations $\oplus, \odot$. We could easily show the group $(\mathbb{R}, \oplus)$ and also $(\mathbb{R} \setminus \{0\}, \odot)$ are abelian groups and so they together form a field. In this case as both are commutative, the entire field is also commutative.

1.1.5 Extension fields

We we will use the example from the previous section, to understand an extension field. We now have a field K which is a sub-field, similarly defined as a subgroup but for fields, of $F = (\mathbb{C}, \oplus, \odot)$, where $\mathbb{C}$ represents complex numbers. We could show all the required properties to prove that F is a field, but this part is not interesting. So now F is called an extension field of K, as both are fields, and K is a subset of F.

1.1.6 Finite field $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$

So far we have looked upon groups and fields whose carrier set M were infinite. Now we will reduce our view only to a finite subset of M. Suppose our set $M = \mathbb{Z} \,/\, p$, which denotes the integers modulo some prime number p. An example of a multiplicative finite subgroup would be $p = 11, M = \{1, 3, 4, 5, 9\}$. Carrier set M can also be written as $\langle 3 \rangle$, since this element could be the generator of this group.

$\odot$	1	3	4	5	9
1	1	3	4	5	9
3	3	9	1	4	5
4	4	1	5	9	3
5	5	4	9	3	1
9	9	5	3	1	4

Table 1.1: Finite subgroup F_{11}, generator 3.

We could also look into an extension field F as mentioned beforehand. Instead of using modulo some prime p number, we will use an irreducible polynomial $P(x)$. An irreducible polynomial works somehow similarly to prime numbers as we know them. No other polynomial except for polynomial $P(x)$ and 1 divides this polynomial without any remainder.

Suppose we have $p = 2$, our irreducible polynomial $P(x) = x^2 + x + 1$. So our carrier set would look like $M = \{0, 1, x, x + 1\}$. The operations defined for both binary operators $\oplus, \odot$ are as follows:

$\oplus$	0	1	x	$x+1$
0	0	1	x	$x+1$
1	1	0	$x+1$	x
x	x	$x+1$	0	1
$x+1$	$x+1$	x	1	0

$\odot$	0	1	x	$x+1$
0	0	0	0	0
1	0	1	i	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

Table 1.2: Finite field F_{2^2}, modulo $x^2 + x + 1$.

1.1.7 Properties of finite fields

As we discussed some properties with the groups, we are also interested in some properties of the fields. Such as:

- Characteristics of a field is the smallest possible number $n \in \mathbb{N}$, not including zero, such that if we take the multiplicative identity 1 and use the additive binary operation in a way $n \cdot 1$, we get the neutral element 0 of the binary operation. If the field was not finite, the characteristics would be 0. Furthermore denoted as $char(F)$.
- The order of a field is the number of all elements the field contains. A field of order p^k has characteristics equal to p.

1.2 Elliptic curves

Most of the background material comes from [10, 11]. To define an elliptic curve, we need an equation. Usually, we will be using the Weierstrass equation over some field K.

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad \text{with } a_1, a_2, a_3, a_4, a_6 \in K \tag{1.1}$$

This equation looks rather ugly so if we choose the field K carefully, that its characteristics, as defined in section 1.1.7, is $char(K) \neq 2$ and at the same time $char(K) \neq 3$, we can use a linear map to transform it to a different form.

With use of substitution $(x, y) \to \left(\frac{x-3b_2}{36}, \frac{y}{108}\right)$, we can rewrite the above equation to

$$y^2 = x^3 - 27c_4x - 54c_6.[10]$$

The only reason we could have achieved this is by using a field, where we can divide by 2 and by 3 at the same time. That is why we required the specific characteristics. Not only we could have simplified the Weierstrass equation, but we have to be careful. As for now, there are two classes of elliptic curves.

1. There are singular elliptic curves. A singular elliptic curve is such a curve that its discriminant is equal to zero.

   $$\Delta_E = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 = 0$$

   This gives us information about its roots. If the discriminant $\Delta_E = 0$, then the curve has a singularity.

2. And there are non-singular elliptic curves, which we will be using from now on. These curves have its discriminant $\Delta_E \neq 0$.

The discriminant is calculated over the field K, that was used when defining the curve.

There are many more forms of writing the equation for an elliptic curve, there are also another types of curves such as Edwards curves[12].

The first part is the equation, but that is not all. The next part is the interesting part about them. We will be focusing on all solutions of a given curve E over a field K, or its sub-field and a point at infinity $\mathcal{O}$, such that:

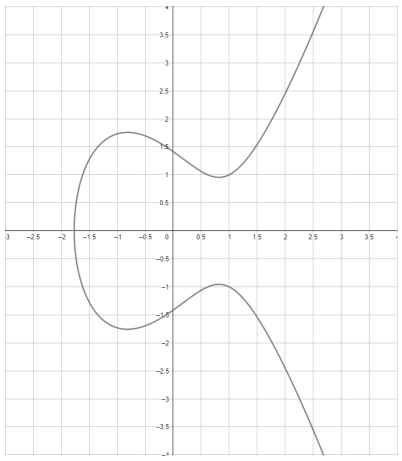
$$E(K) = \Big\{(x, y) \in K^2 \text{ is a solution to } eq. \text{ (1.1)}\Big\} \cup \{\mathcal{O}\}.$$

The point at infinity plays the role of a neutral element e.

Elliptic curves usually have different shape and they consist of one or two components, such curves can be seen in fig. 1.1.

1.2.1 Elliptic curve on a group

Since we defined an elliptic curve E, it is time to use it in a group. This group doesn't have to be finite. To note, if we were to use graphical representation, then three points P, Q, and R that are lying on the same line, their sum is equal to zero. Thus the sum of $P + Q = -R$, where $R = (x_R, y_R)$, $-R = (x_R, -y_R)$,


(a) $y^2 = x^3 - 2x + 2$

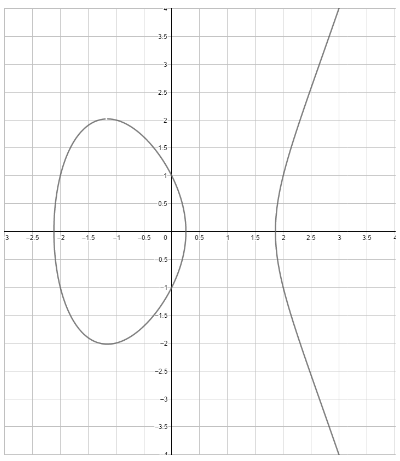
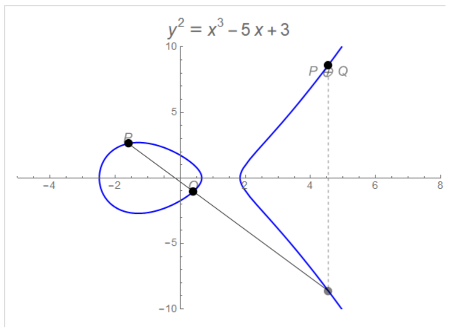

(b) $y^2 = x^3 - 4x + 1$

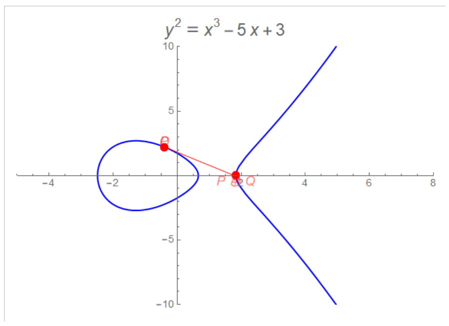
Figure 1.1: Elliptic curves with different equations over $\mathbb{R}$.

when we use the simplified Weierstrass equation. To add the same point P twice, we will use a tangent line to this point P, this line, thanks to the curve being non-singular, will always intersect the curve at another one point.

As shown above, we need three points to form a line. Since our equation is in a cubic form, there will always be three points on a line, including multiplicities of one or the other point. For this, we will use a graphical representation. The figure fig. 1.2 shows us how the point addition works with two different points and a point itself.



(a) Two diffferent points.



(b) Same point.

Figure 1.2: Addititon of points on an elliptic curve over $\mathbb{R}$.

Not only can we add two different points, P and Q, from the same elliptic curve, but we can also add one point to itself multiple more times. As we used with the group definition section 1.1.2 for shortening the addition of the same element, we will also use the same thing here. Be careful, though, we are working in a group of points on an elliptic curve, and there is only one operation. With that, there is no definition of multiplying two points like $P \odot Q$; this is not defined.

With that being said, we will use a shortened version of writing an $m \in \mathbb{N}$ multiple of point P, such as $[m]P = P + P + \cdots + P$, where there are m terms of point P.

1.2.2 Montgomery curves

A Montgomery curve over a field K is in the form of $by^2 = x^3 + ax^2 + x$. Its usage is primarily due to fast differential addition in this representation. The terms a and b are elements of underlying field K, and they must satisfy the $b(a^2 - 4 \neq 0$ in the field K. Often, a point P of a Montgomery curve are referred to only with their x-coordinate, we will be using the notation x_p, respectively y_p for their y-coordinate.

1.2.3 J-invariant of an elliptic curve

Two elliptic curves are isomorphic, if and only if their j-invariants are the same[11]. But what is the j-invariant? A j-invariant is a function that gives every elliptic curve a value. These values determine a class of elliptic curves that are isomorphic. These curves will be necessary in the upcoming algorithm for section 2.2.

The j-invariant is calculated differently according to the used form of an elliptic curve. If we were to use the form in eq. (1.1). Note: don't forget the elliptic curve is defined over some field K. The j-invariant is calculated from the parameters of the curve, and the calculation would look like the following:

$$\begin{aligned} b_2 &= a_1^2 + 4a_4, \\ b_4 &= 2a_4 + a_1a_3, \\ j(E) &= \frac{(b_2^2 - 24b_4)^3}{\Delta_E}[10]. \end{aligned}$$

1.2.4 Supersingular elliptic curves

So far we have seen an elliptic curve in section 1.2 already, these elliptic curves are from now on called ordinary elliptic curves. Another category of elliptic curves are supersingular elliptic curves. By definition, they are not singular and the term *supersingular* will be explained later on in section 2.1.

Let E be an elliptic curve defined over some field K with $char(K) = p > 0$. We use a map that multiplies every point in the E $n-$times, where $n \in \mathbb{N}$. This is written as $E[n]$. The number n is chosen as some (all) power $e > 1$, of the $char(K) = p$, thus we have a map in the form of $n = p^e$, similarly $E[p^e]$.

A curve is supersingular iff the result is isomorphic to only the point at infinity $\mathcal{O}$, otherwise it is an ordinary elliptic curve.

$$E[p^e] \cong \{\mathcal{O}\}[10]$$

1.3 Isogenies

In section 1.2.4, we used a map that multiplies all points in a given elliptic curve by a scalar m. What could have been said there is that this map preserves the binary operator and also the group structure. We will use this as the beginning of our example for the upcoming definition. Silverman and Washington [13, 14] are showing more detailed descriptions, but both define their isogenies a bit differently.

An isogeny is a particular map between two elliptic curves that is also a group homomorphism. Given two elliptic curves E and E'

$$\phi : E \to E',$$

this also is a surjective map, meaning that given any point $P' \in E'$, there exists at least one element $P \in E$, that $\phi(P) = P'$. For our cases, this map should not be a non-constant rational map. [15]

We are also interested in the kernel of such an isogeny. This map preserves the kernel points of the original curve and potentially expands them to more elements. Furthermore, we define the kernel of the isogeny ϕ as

$$ker(\phi) = \{P \in E : \phi(P) = \mathcal{O}\}.$$

Not only can we have any map, but we will also use mapping in the form of endomorphism, meaning that we have a map from the elliptic curve E to itself. The elliptic curve E' is defined using the map ϕ and the subgroup H of $E(K)$, where $H = ker(\phi)$. We create the new elliptic curve $E' = E/H$. The transformation can be made using the Vélu's formula [16]. Also the formula can be used to create the map ϕ.

1.3.1 Isogeny graph

An undirected graph G is an ordered pair (V, E), where V is a set of vertices and E is a set of ordered pairs (v_1, v_2), where $v_1, v_2 \in V$. The meaning of the ordered pair is that there is an edge between vertices v_1 and v_2, or differently there is a way how one can get from the vertex v_1 to v_2. Suppose we have the shortest path between nodes v_1 and v_2. Distance between these two nodes is the shortest number of edges we have to travel through to get from the beginning to the end. If there is no path from the vertex v_1 to v_2, then the distance is usually said to be infinite. Otherwise, it is finite. If there exists at least one path from every vertex to another, the graph is called a connected graph. A degree of a vertex is the number of edges going from or to this vertex. Similarly, we could talk about in-degree, the number of in-going edges, and out-degree, the number of out-going edges. A graph is called $k-$regular if all its vertices have the same degree k.

An isogeny graph is a type of an expander graph [11]. These graphs have wonderful properties. The graph's diameter is bounded by $\mathcal{O}\left(log(n)\right)$, where the diameter is the longest path amongst all of the shortest paths, which means that with a logarithmic number of steps in this graph, we have a high probability of ending up in any other vertex of this graph.

In our case, we have supersingular elliptic curves, which are identified by their j-invariant, and we are able to create a map ϕ between them. So the set of vertices V are the j-invariants of elliptic curves, and the edges between vertices are the isogenies between them.

Chapter 2

Shared key establishment

In this chapter, we will discuss the importance of using supersingular elliptic curves instead of ordinary ones. This is followed by the definition of the general core problem used to create the same secret between two parties. Then describe an encapsulation mechanism that uses this problem and builds a key encapsulation mechanism (KEM) over it.

2.1 Supersingular elliptic curves versus normal elliptic curves

First, let's start with why we use supersingular elliptic curves instead of ordinary ones. A similar algorithm was proposed in [17] that uses isogenies to create a post-quantum algorithm. After that, other refined algorithms used the same mechanisms [18, 19].

For some time, it was believed that there was no sub-exponential quantum algorithm for finding the isogeny between two elliptic curves. This algorithm aimed to be a post-quantum candidate in the standardization process that would one day be used instead. Until [20] found a sub-exponential algorithm, that breaks this type of cryptography thanks to its structure. So far, everything looks bad for isogeny-based cryptography. The computation is slow, unlike other post-quantum schemes like [21], and is also broken with a quantum computer thanks to this abelian group structure.

The supersingular elliptic curves were chosen to replace the ordinary elliptic curves, and it was proposed in [22]. The weak spot of having an abelian group was removed as it is characteristic of the supersingular elliptic curves. So far, there is no known sub-exponential algorithm for finding the isogenies between two supersingular elliptic curves.

2.2 Supersingular Isogeny Diffie-Hellman

Diffie-Hellman key exchange is old as it gets, nevertheless, it is still being used but with a different group structure. More details regarding the scheme and the algorithm can be found in [22, 3]. This time, we will use it to compute a shared key between two parties. To define the whole cryptosystem, we will first define its public parts. One of them is the public prime number $p = l_a^{e_a} \cdot l_b^{e_b} \cdot f \pm 1$. Where l_a, and l_b are small prime numbers, e_a, and e_b are any natural number, and f is a co-factor such that p is a prime number.

An example of such parameters would be:

- $l_a = 2, l_b = 3$,
- $e_a = 5, e_b = 3$,
- $f = 1$, and -1.
- This will yield the value $p = 2^2 \cdot 3^5 \cdot 1 - 1 = 971$, which is a prime number.

From this, there is created a field F_{p^2} and a supersingular elliptic curve E of cardinality $(p \mp 1)^2 = (l_a^{e_a} \cdot l_b^{e_b} \cdot f \pm 1 \mp 1)^2 = 972^2$.

Both parties then find any basis points $\{P_a, Q_a\}$, for Alice, respective $\{P_b, Q_b\}$, for Bob, or rather a generating group of their respective sup-groups $E[l_a^{e_a}] = \langle P_a, Q_a \rangle$, respectively $E[l_b^{e_b}] = \langle P_b, Q_b \rangle$. Both Alice's points must be independent, and their order matches their torsion group. Both parties will share their basis points.

Both parties will now randomly choose their secret values m_a, n_a, that are not divisible by l_a, respectively m_b, n_b, that are also not divisible by l_b. With that, they both compute their own isogenies $\phi_a : E \to E_a$, with kernel $ker(E_a) = \langle [m_a]P_a + [m_b]Q_a \rangle$, respectively ϕ_b. Alice then computes the images of Bob's base points $\{\phi_a(P_b), \phi_a(Q_b)\}$. Bob does the same but with the basis points of Alice and his secret values m_b, n_b.

When both parties finish computing their isogenies and their newly created supersingular elliptic curves E_a, respectively E_b, they exchange their supersingular elliptic curves and the images of the opposing basis points. When Alice receives Bob's curve E_b and the images of her basis points $\{\phi_b(P_a), \phi_b(Q_a)\}$, she then creates a new isogeny

$$\phi_a^{'} : E_b \to E_a b,$$

having its kernel $ker(\phi_a^{'}) = \langle [m_a]\phi_b(P_a) + [m_b]\phi_b(Q_a) \rangle$. Bob as usually does the same on his side. With that, they both should end up with a supersingular elliptic curve. This curve doesn't have to be the same curve, but what is more important is, that they both belong to the same isomorphic class, and thus, their j-invariant will be the same since

$$\begin{aligned} E_{ab} = \phi_a^{'}(\phi_b(E)) = E/\langle [m_a]P_a + [n_a]Q_a, [m_b]P_b + [n_b]Q_b \rangle = \\ = \phi_b^{'}(\phi_a(E)) = E_{ba} \end{aligned}$$

Alice	$p = {l_a}^{e_a} \cdot {l_b}^{e_b} \cdot f \pm 1$	**Bob**
	$E(\mathbb{F}_{p^2})$	
Chooses base $\{P_a, Q_a\}$		Chooses base $\{P_b, Q_b\}$
	$\xrightarrow{\{P_a,Q_a\}}$	
	$\xleftarrow{\{P_b,Q_b\}}$	
Choose a random		Choose a random
$m_a, n_a \in (0, {l_a}^{e_a})$		$m_b, n_b \in (0, {l_b}^{e_b})$
$R_a = [m_a]P_a + [n_a]Q_a$		$R_b = [m_b]P_b + [n_b]Q_b$
Calculates public values:		Calculates public values:
$\phi_a : E \to E_a$		$\phi_a : E \to E_b$
$E_a = E/\langle R_a \rangle$		$E_b = E/\langle R_b \rangle$
$\phi_a(P_b), \phi_a(Q_b)$		$\phi_b(P_a), \phi_b(Q_a)$
	$\xrightarrow{E_a, \phi_a(P_b), \phi_b(Q_b)}$	
	$\xleftarrow{E_b, \phi_b(P_a), \phi_b(Q_a)}$	
Calculates the final		Calculates the final
isogeny and curve		isogeny and curve
$R_a^{'} = [m_a]\phi_b(P_a) + [n_a]\phi_b(Q_a)$		$R_b^{'} = [m_b]\phi_a(P_b) + [n_b]\phi_a(Q_b)$
$\phi_a^{'} : E_b \to E_{ab}$		$\phi_b^{'} : E_a \to E_{ba}$
$E_{ab} = E_b/R_a^{'}$		$E_{ba} = E_a/R_b^{'}$
	$E_{ab} \cong E_{ba}$	
	$j(E_{ab}) = j(E_{ba})$	

Figure 2.1: SIDH shared key exchange protocol.

What parameters are not a good idea to share? The public parameters at the end are:

- prime number p,
- the supersingular elliptic curves E, E_a, E_b,

- basis points P_a, Q_a, P_b, Q_b,
- their images in the given isogenies $\phi_b(P_a), \phi_b(Q_a), \phi_a(P_b), \phi_a(Q_b)$.

The secret parameters are:

- secret values m_a, n_a, m_b, n_b,
- secret isogenies $\phi_a, \phi_a^{'}, \phi_b, \phi_b^{'}$.

$$\begin{tikzcd}
E \arrow[r, "\varphi_a"] \arrow[d, "\varphi_b"'] & E_a \arrow[d, "\varphi'_b"'] \\
E_b \arrow[r, "\varphi'_a"] & E_{ab}
\end{tikzcd}$$

Figure 2.2: Shared elliptic curve computation.

From the figure fig. 2.2 above, the public and private parameters, we can see that if only one, in other case the two secret values m_a, n_a, parameters would be leaked, an attacker would endanger the whole shared secret key exchange. Simply, if the secret isogeny $\phi_a^{'}$ were to be revealed, the attacker can easily calculate the image of Alice's point in Bob's isogeny and thus compromising the final elliptic curve E_{ab}.

2.2.1 Parameter generation

Suppose we have fixed parameters l_a, l_b, e_a, and e_b. In [22], it is shown that finding the prime number $p = l_a^{e_a} \cdot l_b^{e_b} \cdot f \pm 1$, is relatively easy as we test random values of f and check if the given value is a prime or not. The next step is to find a supersingular curve E over the field F_{p^2}, from there we can choose the starting curve E_0 as any other random supersingular curve that can be found as randomly walking in a graph of the isogenies.

When we have the starting curve, Alice and Bob have to find their basis points of $E_0[l_a^{e_a}]$ for Alice, respectively $E_0[l_b^{e_b}]$ for Bob. As we know from before, the basis points have to be of the order of these torsion subgroups, and the points have to be independent. We will use Weil pairing $e(P_a, Q_a)$ in $E_0[l_a^{e_a}]$ to check both conditions. If the result is valid and has order $l_a^{e_a}$, we have found our basis points, otherwise, we choose another point Q_a, and repeat the process. The same applies to Bob, but with the use of his parameters.

2.2.2 Isogeny finding

To find the final isogeny, our first step is calculating the kernel of the isogeny. Before we go straightforward to the computation, we are trying to find the elements described by the generator $\langle R \rangle = \langle [m_a]P_a + [n_a]Q_a \rangle$. For this, we would need to do some computation twice over an elliptic curve. This could be as twice as slow. To ease the calculation, we will use any other generator of $\langle R \rangle = \langle [m_a]P_a + [n_a]Q_a \rangle$, as is shown in [22]; we can use without loss of generality that the element m_a is invertible modulo the order of the group. Thus we can use $\langle R^{'} \rangle = \langle P_a + [m_a^{-1} \cdot n_a]Q_a \rangle$

When we acquire the kernel of our isogeny, we need to calculate the final isogeny. The computation of the isogeny is disassembled into computing smaller isogenies and then combining them all together to create one isogeny of the required degree. For that, we use an algorithm as described in [22]:

Algorithm 1 Isogeny computation for Alice.

```
$E_0 \leftarrow E$
$R_0 \leftarrow R'$
$e \leftarrow e_a$
$l \leftarrow l_a$
for $0 \leq i < e$ do
    $E_{i+1} = E_i / \langle l^{e-i-1} R_i \rangle$
    $\phi_i : E_i \rightarrow E_{i+1}$
    $R_{i+1} = \phi_i(R_i)$
end for
$\phi \leftarrow \phi_{e-1} \circ \cdots \circ \phi_0$
return $E_e, \phi$
```

We will use the Velu's formula [16] to compute the E_{i+1} curve and its ϕ_i. It is easy to see that the above algorithm has quadratic complexity in the length of e; this could also be lowered if using a better approach.

To fasten this process, we will usually use different strategies over this naive approach. These strategies aim to achieve overall faster computation time for the price of computing not only $2-$isogenies and $3-$isogenies, but also $4-$isogenies and so on.

2.3 Supersingular Isogeny Key Encapsulation

Supersingular isogeny key encapsulation (SIKE) is an algorithm that is currently under the National Institute of Standards and Technology (NIST) post-quantum standardization process available at [23]. This algorithm is the only one isogeny-based algorithm. As described in section 2.2 and in [3], it uses pseudo-random walks on supersingular isogeny graphs, which is the heart of this KEM. This algorithm, versus all the other types of algorithms based on code-based, lattice-based, and others, uses very small keys and thus is very likely to be the future standard for shared key exchange. Before this happens, the algorithm has one disadvantage, and it is its speed because of the exhaustive isogeny computation.

There is a public-key encryption scheme that is CPA-secure (chosen plaintext attack) and can be transformed into a CCA-secure (chosen ciphertext attack) KEM scheme as well.

2.3.1 Security of supersingular isogeny key encapsulation

There are two ways for security measurement. As we are looking towards the future, not only classical computers are a threat, but also the rising threat of quantum computers. That is why security has to be measured from both directions. If the scheme were not secure one or another, we would always be able, if we had a large enough quantum computer, to break the encryption of that scheme or with a powerful enough classical computer. Detailed information is available at [3].

The early release of SIKE algorithm had proposed bigger key sizes than there are now; as it was shown, the initial analysis was a bit conservative, but rather than propose smaller keys and have a worse security level, they proposed bigger keys that guaranteed a certain level of security. [24]

Currently, SIKE is in NIST standardization process with four different target levels of security [25]. Let's start with the classical ones. SIKE proposed implementations aims for levels 1,2,3 and 5.

- Level 1: requires the algorithm to be able to withstand an exhaustive search for the AES-128 key.
- Level 2: requires the algorithm to be able to withstand a collision search of SHA256.
- Level 3: requires the algorithm to be able to withstand an exhaustive search for the AES-192 key.
- Level 4: requires the algorithm to be able to withstand a collision search of SHA384.
- Level 5: requires the algorithm to be able to withstand an exhaustive search for the AES-256 key.

The theoretical quantum safety of this algorithm was firstly used with respect to using the claw-finding algorithm [26] to analyze its security. This algorithm has $\mathcal{O}\left(p^{\frac{1}{6}}\right)$ time complexity. But as was previously mentioned, this estimate was very conservative, and in [27], it was shown that the attack requires $\mathcal{O}\left(p^{\frac{1}{3}}\right)$ (RAM) operations. This means they could change the system's public parameters, even more, lower, and the cryptosystem would still be sufficiently safe against other attacks.

2.3.2 Reference implementation

When a proposal to the NIST standardization process is submitted, a person or a group of people have to include a reference implementation of a given algorithm. The main reason of providing such a reference implementation is to publish it together with the proposal to the general public. These reference implementations are then tested in a broad spectrum of analysis. They are starting with its effectiveness in time and memory complexity, key sizes, etc. Also, not only to conclude how the implementations will behave but also to have a closer hands-on experience working with this algorithm to find possible back-doors, vulnerabilities, and side-channel analysis. This method of sharing the reference implementation helps detect the mentioned issues and keeps researchers from including their own weaknesses.

Of course, there is not only one implementation; there are a lot of different implementations of the SIKE algorithm. One of them is purely written in C, some are written in VHDL, some are partly written in assembly to use specific instructions on a given processor, and so on. There are also versions with compressed keys, but we are not interested in them right now.

There are available implementations distinguishable primarily by the bit length of their given prime number p. Currently, SIKE supports 434-bit, 503-bit, 610-bit, and 751-bit long prime numbers as their public parameter.

The following fig. 2.3 shows how the SIKE protocol works. The protocol works with three hash functions F, G, H at three places [3], all these three places can have a different hash function but also the same. The figure contains only two hash functions. A function for encryption is also used, and we could find this function in [3]. With this being said, the key encapsulation and decapsulation work in the following way.

Bob can either be a server, or he can be the one initiating the communication. With that, he computes his public parts of the SIDH protocol. Alice is then either initiator or Bob wants to communicate with her. Alice then generates a random binary string m that is combined with Bob's public key using a hash function. This way, Alice gets her (almost) secret number n_a, and uses it to compute her part of the SIDH protocol and the invariant of the final curve. She creates two cipher texts, which tie her secret m, with a xor operation with the secret j-invariant of the final curve. Everything is, in the end, combined together to be hashed and forms a shared key K.

When Bob receives the public parameters from Alice, he also computes the secret isogeny, its final elliptic curve, and its j-invariant. If everything went correctly, Bob will end up with a message m', that should be equal to the

m Alice generated, and thus, he can proceed with the same steps of hashing information together and create the same shared key K.

Bob		**Alice**
KeyGen:		
$\texttt{pk}_B = [E_B, \phi_B(x_{P_A}), \phi_B(x_{Q_A})]$		
$s \in_R \{0,1\}^\ell$		
		Encapsulation:
		$m \in_R \{0,1\}^\ell$
		$r = G(\texttt{pk}_B, m)$
		$\texttt{pk}_A(r) = [E_A, \phi_A(x_{P_B}), \phi_A(x_{Q_B})]$
		$j_{\texttt{inv}} = j(E_{AB})$
		$\texttt{Enc}(\texttt{pk}_B, m, r) \rightarrow (c_0, c_1)$
		$(c_0, c_1) = (\texttt{pk}_A(r), G(j_{inv}) \oplus m)$
		$K = H(m \parallel (c_0, c_1))$
	$\xleftarrow{(c_0, c_1)}$	
Decapsulation:		
$j_{\texttt{inv}} = j(E_{BA})$		
$m' = c_1 \oplus G(j_{\texttt{inv}})$		
$r' = G(\texttt{pk}_B, m')$		
$\texttt{pk}_A(r') = c_0 \rightarrow K = H(m' \parallel (c_0, c_1))$		
$\texttt{pk}_A(r') \neq c_0 \rightarrow K = H(s \parallel (c_0, c_1))$		

Figure 2.3: SIKE protocol, encapsulation and decapsulation. [1]

2.3.3 Existing side channel attacks

Not only is an algorithm vulnerable to some casual attacks, but it can be vulnerable to a side-channel analysis. The side-channel analysis focuses on different physical states of a measured device and provides additional information, not including the device's output. Usually, these side-channel are unintentional, as it weakens the device in the way that some part of a secret or even the whole secret can be discovered. These side-channels are introduced with the implementation of a given algorithm. There are different kinds of side channels:

- power consumption, a device uses more energy to process different data.
- Time spending, such as cache miss or doing some operations, takes longer than others.
- Speculative execution: the CPU executes some instructions without knowing if it should or needs to do these instructions.
- And so on...

For example, the leakage in power consumption consists of two parts and exists only due to the imperfection of creating transistors.

- Static that is a passive consumption the circuit or device would do anyway;
- and dynamic, that part is caused by switching or other activity of a circuit.

According to [3], since the cryptosystem is isogeny-based, the SIDH is only vulnerable to the side-channel channel analysis at two points.

- The first is the discovery of the secret kernel point denoted before as $\langle R \rangle$, respectively its computation.
- The second is to figure out the secret isogeny walk from the starting curve to the final curve.

We could attack on the secret kernel point, [28] showed some examples of how to attack. It attacks the double and add algorithm, an efficient algorithm for multiplying a point on an elliptic curve. It is a deterministic function and only behaves differently when the secret scalar is different. The secret scalar could be found using simple power analysis (SPA) if the algorithm doesn't use any countermeasure. There is a simple solution; a Montgomery ladder could be used as it should eliminate this issue [29].

Since the SIDH uses the Montgomery ladder and also Montgomery curves, it is required to use some sophisticated strategy. That comes with a correlation power analysis [30] (CPA). CPA uses traces of consumption of an attacked device. These traces should correlate with the hypothetical consumption of a processor which executes a deterministic function. This hypothetical consumption resembles the consumption of change of bits in a register and is estimated using the Hamming weight,

There are more attacks and leakages [31, 32, 33, 34] regarding the SIKE algorithm. They are based on zero value attacks or targeting a specific implementation on an arm-based, attacks that overcome specific countermeasures, and more. Some attacks aim at an ephemeral key. Such an attack can be found here [35]. It uses a horizontal CPA to discover the secret scalar by splitting one trace into smaller ones.

Chapter 3

Attack design

The attacked implementation is the official SIKE implementation adapted for the (32-bit) ARM Cortex-M4 microcontroller. We will elaborate on the hardware and software solution, eventually modified for this thesis.

3.1 Hardware and software selection

There are four versions of SIKE to choose from. Since we are going to use a 32-bit microcontroller from an ARM Cortex family, We chose to attack the smallest key size of SIKE, which is the one satisfying level 1 NIST security level. It is due to the expected slow computation process. Even though if we were to choose the largest and most secure version of SIKE and we would attack on that implementation, the only difference would be in the time spent waiting for data. The software we chose is from [36, 4]. From there, we started adapting this code to our needs, still with keeping the system functional as it would be without any adaptation.

3.1.1 Hardware

We have to use a 32-bit microcontroller. Lately, one of the popular devices to use is to use the ChipWhisperer [2]. We used the lite version as it has most of the needed things included. ChipWhisperer-Lite (CW1173) 32-bit basic board which features an STM32F303, the target is a Cortex-M4 microcontroller which we will program, and can be seen in fig. 3.1. There is also a serial port for communication and programming. The interface and control are implemented using an Atmel SAM3U microcontroller and Xilinx Spartan6 FPGA.

There are some disadvantages to measuring the traces; the sample count is at most 24400 and can't be increased more. There are two options, though. The first option is to enable the decimate property to some value of x. ChipWhisperer will now keep only each x-th sample and discard the samples in

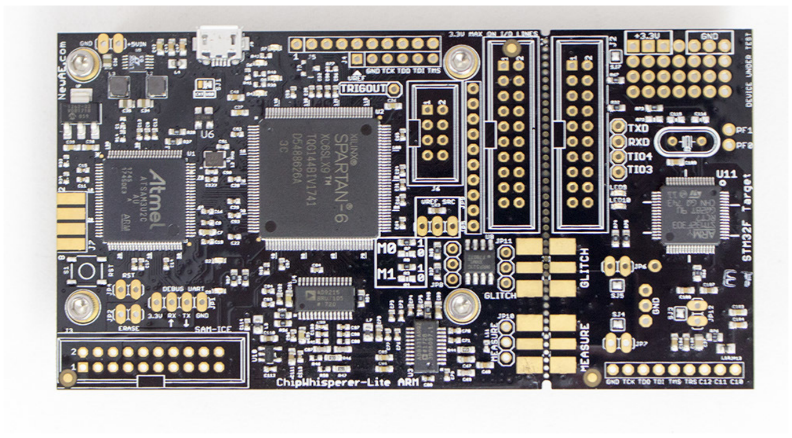


Figure 3.1: ChipWhisperer Lite. [2]

between. This will increase the time span the ChipWhisperer can record over, but for the price of potentially losing important information. This lack could have been neglected using an external oscilloscope. Also, there is a built-in signal amplifier, so we don't have to worry about noise that much.

To control the ChipWhisperer, we are using python and a Jupyter notebook. This notebook will handle all communication with the use of ChipWhisperer API.

3.1.2 Software

The ChipWhisperer software is as stated above. We also had to make an interface so the device could communicate with the computer. The interface contains commands to

- upload the private key,
- compute the public key,
- send the public key to the computer,
- receive the opposite side public key,
- compute the shared key,
- send the shared key back to the computer.

There had to be some modifications done as the length of the public key is 330 bytes, but we were only able to send 255 bytes at the same time, so some functions contain a slight change in their data processing, such as the first two bytes represent the length of the data and the other the real start of the data. All of these were made to test and measure all comfortably.

To generate data for the SIDH, we also used some software base points from a GitHub repository [37]. There are three modifications to the code.

First, we needed to ensure the hardware and software worked properly on both the microcontroller and the testing computer. To do that, we created a simple program that randomly generates all required random parameters of the SIDH protocol, goes through all the steps of the public key exchange, and generates every step of the SIDH protocol. When all computations were done, the secret shared key was printed to the terminal, and all the necessary intermediate steps, such as printing from Alice's perspective her private key, her public key, and the shared key she computed. Then from Bob's perspective, the other way around. All of these values are then used to test against the ChipWhisperer and its implementation. The included python file contains such a test. This functionality is now disabled; the explanation of why is in the section 3.2.

The second program we used was to generate n random private keys and their respective public keys. The program takes as an argument a whole number $n \geq 0$. If the given argument is not valid, the program finishes and does not write anything to the standard output. The program goes through the steps of Alice. It generates a random $n_a \in (0, l_a^{e_a})$, where l_a, e_a are public parameters for SIKE-434. Then it computes the kernel point, the final isogeny, and encodes the j-invariant value of that curve using the official interface of the SIDH algorithm. In the end, it formats the private and public keys as hexadecimal values and prints them to the standard output on a new line. The private keys are discarded every time except for the ChipWhisperer initial phase when there we need to fill the ChipWhisperer with a new secret key.

The last piece of code for our attack to work is to know some intermediate values or their internal state. This program, rather oracle, has access only to public keys, so any mention of a secret key is for a hypothetical key we think is correct. This program takes three arguments. First is the number of bits we know about the private key, the hypothetical key, then the bytes of the private key, and lastly, the public key. Except for the number of known bits, all values are encoded in hex, and the program processes them as such. If all parameters are passed and are valid, the program processes all the known bits first and then prints out the two possible states that could have happened in the hardware. Either there was a binary zero or a binary one. It saves both possible results and prints them to the standard output.

3.2 Implementation related issues

Before we try to find any weak spot, we should try to check if there is any leakage of so far known data. The easiest way of doing so is to measure n traces where we use n random public keys, and we will try to use 8-bit words, these words are taken as bytes from the public keys, and correlate their Hamming weights with the measurement. As the microcontroller has a 32-bit processor, we can try to go up to 32-bit long words. Surprisingly enough, we found a correlation between all bytes of the public keys. The next fig. 3.2 shows such correlations of 8-bit words and fig. 3.3 shows correlation of 32-bit long words.

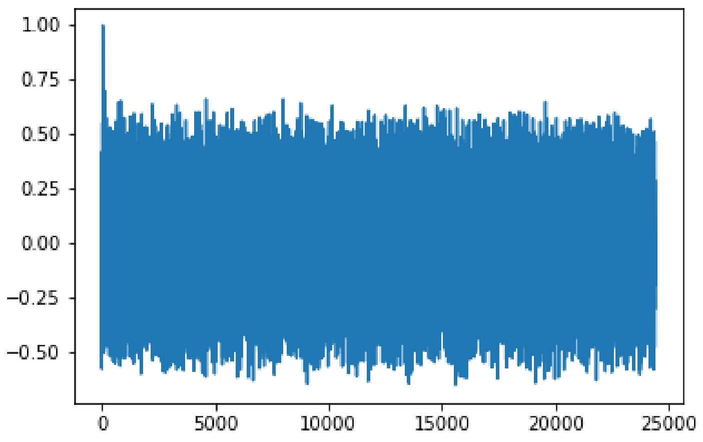


Figure 3.2: Correlation of power consumption using 8-bit words.

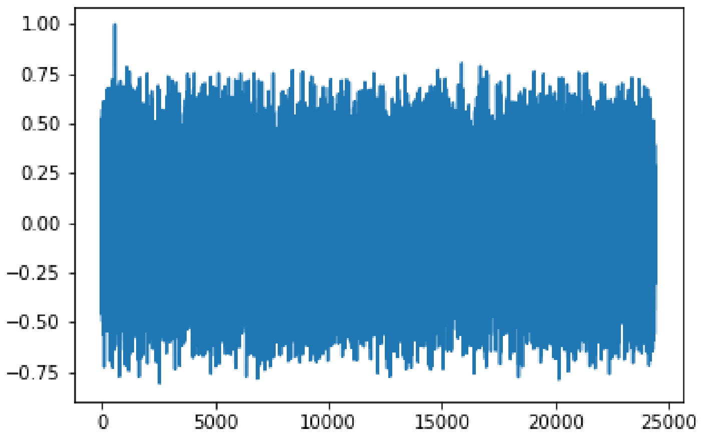


Figure 3.3: Correlation of power consumption using 32-bit words.

While doing these experiments, the execution speed was very slow. For one trace, or instead, for one computation, it took about 6.97 seconds. Because of this, we decided to reduce the computation complexity and removed some parts of the algorithm that would not be so much important to us. This reduced the execution speed to about 1.92 seconds, which makes capturing $n = 1000$ traces feasible.

As the isogeny computation differs for both Alice and Bob, we decided that the microcontroller will always be the Bob side, and it will do only Bob's part of the SIDH protocol. The attacker will be in the position of Alice and thus will be using her computations of isogenies.

3.3 Looking for leaking spots

The inspiration for our attack comes from [35] where they successfully attacked on SIKE with an ephemeral key where they split one long trace into smaller ones, so it is a horizontal attack. Both SIKE and SIDH share the same background, but as shown in chapter 2., the key encapsulation is very different.

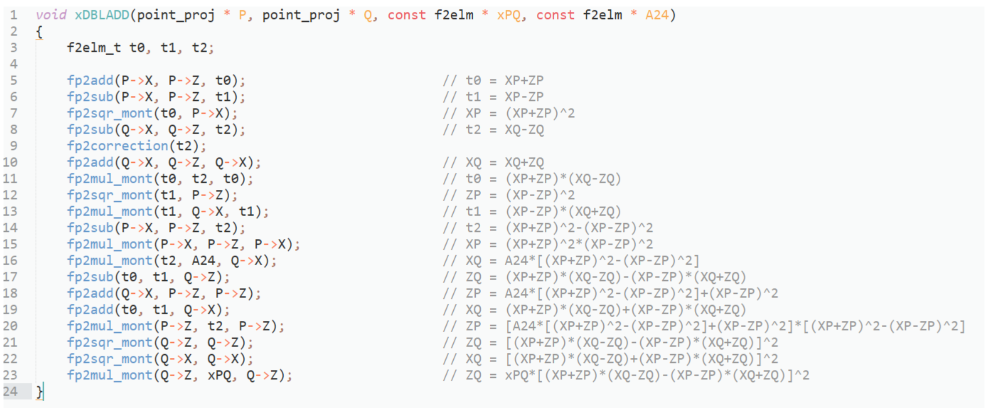
We are attacking on the SIDH so that we will make a vertical attack against this protocol. Thus instead of only requiring one trace, we will need hundreds of them or maybe thousands. We will be using a correlation power analysis to distinguish between a wrong guess and a correct guess. The attack will be made on the Montgomery curve's three-point ladder.

Algorithm 8: Three point ladder

```
function Ladder3pt
    Input: $m = (m_{\ell-1}, \ldots, m_0)_2 \in \mathbb{Z}$, $(x_P, x_Q, x_{Q-P})$, and $(A : 1)$
    Output: $(X_{P+[m]Q} : Z_{P+[m]Q})$
$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$
$a_{24}^{+} \leftarrow (A + 2)/4$
for $i = 0$ to $\ell - 1$ do
    if $m_i = 1$ then
        $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^{+} : 1))$    // Al . 5
    else
        $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^{+} : 1))$    // Al . 5
return $(X_1 : Z_1)$
```

Figure 3.4: Three point ladder.[3]

A conditional $xDBLADD$ is highlighted in the figure fig. 3.4, the reference implementation contains a simple constant-time swap using xor function instead of if and else branching. Since we are using the ChipWhisperer, as described in section 3.1.1, with its oscilloscope, its memory isn't enough, not even for the swapping part. As the swap operation affects only the double and add function, we will focus on this operation. Also this is recommended in [35]. With this in mind, we will first try to deduce how many cycles some operations take in that function. Then we will limit ourselves only to collecting as many useful samples as possible that work with one or the other



```c
void xDBLADD(point_proj * P, point_proj * Q, const f2elm * xPQ, const f2elm * A24)
{
    f2elm_t t0, t1, t2;

    fp2add(P->X, P->Z, t0);                                // t0 = XP+ZP
    fp2sub(P->X, P->Z, t1);                                // t1 = XP-ZP
    fp2sqr_mont(t0, P->X);                                 // XP = (XP+ZP)^2
    fp2sub(Q->X, Q->Z, t2);                                // t2 = XQ-ZQ
    fp2correction(t2);
    fp2add(Q->X, Q->Z, Q->X);                              // XQ = XQ+ZQ
    fp2mul_mont(t0, t2, t0);                               // t0 = (XP+ZP)*(XQ-ZQ)
    fp2sqr_mont(t1, P->Z);                                 // ZP = (XP-ZP)^2
    fp2mul_mont(t1, Q->X, t1);                             // t1 = (XP-ZP)*(XQ+ZQ)
    fp2sub(P->X, P->Z, t2);                                // t2 = (XP+ZP)^2-(XP-ZP)^2
    fp2mul_mont(P->X, P->Z, P->X);                         // XP = (XP+ZP)^2*(XP-ZP)^2
    fp2mul_mont(t2, A24, Q->X);                            // XQ = A24*[(XP+ZP)^2-(XP-ZP)^2]
    fp2sub(t0, t1, Q->Z);                                  // ZQ = (XP+ZP)*(XQ-ZQ)-(XP-ZP)*(XQ+ZQ)
    fp2add(Q->X, P->Z, P->Z);                              // ZP = A24*[(XP+ZP)^2-(XP-ZP)^2]+(XP-ZP)^2
    fp2add(t0, t1, Q->X);                                  // XQ = (XP+ZP)*(XQ-ZQ)+(XP-ZP)*(XQ+ZQ)
    fp2mul_mont(P->Z, t2, P->Z);                           // ZP = [A24*[(XP+ZP)^2-(XP-ZP)^2]+(XP-ZP)^2]*[(XP+ZP)^2-(XP-ZP)^2]
    fp2sqr_mont(Q->Z, Q->Z);                               // ZQ = [(XP+ZP)*(XQ-ZQ)-(XP-ZP)*(XQ+ZQ)]^2
    fp2sqr_mont(Q->X, Q->X);                               // XQ = [(XP+ZP)*(XQ-ZQ)+(XP-ZP)*(XQ+ZQ)]^2
    fp2mul_mont(Q->Z, xPQ, Q->Z);                          // ZQ = xPQ*[(XP+ZP)*(XQ-ZQ)-(XP-ZP)*(XQ+ZQ)]^2
}
```

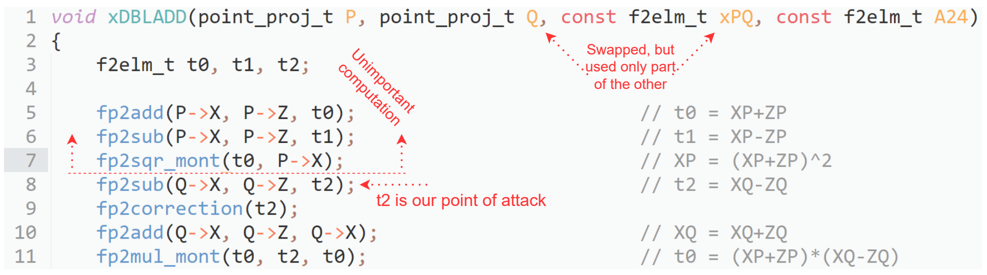
Figure 3.5: Double and add algorithm. [4]

swapped value. The table table 3.1 shows approximately how many samples each operation takes.

Operation	Samples
fp2add	2200
fp2mul_mont	13400
fp2sqr_mont	8600
fp2sub_mont	1700
fp2correct	8000

Table 3.1: Samples count for different operations.

Suppose $T = (X_0, Z_0), S = (X_1, Z_1)$, and $U = (X_2, Z_2), \alpha = (A+2)/4$ as written in fig. 3.4. Let's illustrate the variable propagation when the swap does not occur; we would call the double and add the operation as $xDBLADD(S, T, U, \alpha)$. When the swap occurred, the variable names would be different, but the illustration here is meant for the content of these variables, and we would have called it like $xDBLADD(S, U, T, \alpha)$. The following figure fig. 3.6 show these changes and where our point of attack is.



```c
void xDBLADD(point_proj_t P, point_proj_t Q, const f2elm_t xPQ, const f2elm_t A24)
{
    f2elm_t t0, t1, t2;

    fp2add(P->X, P->Z, t0);                    // t0 = XP+ZP
    fp2sub(P->X, P->Z, t1);                    // t1 = XP-ZP
    fp2sqr_mont(t0, P->X);                     // XP = (XP+ZP)^2
    fp2sub(Q->X, Q->Z, t2);                    // t2 = XQ-ZQ
    fp2correction(t2);
    fp2add(Q->X, Q->Z, Q->X);                  // XQ = XQ+ZQ
    fp2mul_mont(t0, t2, t0);                   // t0 = (XP+ZP)*(XQ-ZQ)
```

Unimportant computation
Swapped, but used only part of the other
t2 is our point of attack

Figure 3.6: Change of variables of $xDBLADD$ with swap.

From the illustration of variable propagation in fig. 3.6, we can see the first three operations do not matter as they are not operating with the second and third argument, which are being swapped in the algorithm fig. 3.4. If we take into consideration how many samples we can record and the length of each operation, we can roughly record the operations on lines 8-11. Because of this, we will focus only on the first bit of the secret key, and if we wanted to discover the whole key, we would have to do some offsetting to record only those double and add operations that are present during the $k - th$ bit.

Since everything is deterministic except for the unknown secret key sk, we will use this for our hypothesis. As we can see, there is a good correlation with the use of 8-bit words from the public key, and we will continue with them as well. Suppose we know the first l bits of the secret key sk, which has n bits. Since we don't know the next bit $sk[l + 1]$, we will try both possible combinations of the next bit being zero or one. Since we know there should be a leakage in the result $t2$, according to their Hamming weight, as part of our hypothetical consumption, this theoretical consumption should correlate with the actual measured consumption. Since the data are random, the data will behave as a random variable. Suppose our hypothesis about the theoretical power consumption with the use of the Hamming weight of the intermediate values is correct. In that case, there will be a significant correlation with one or the other key guess set. As we are measuring multiple operations, our oracle can choose whichever intermediate state it wants.

Chapter **4**

Attack implementation and evaluation

This chapter concludes how the attack was made and what we did. How much the attack was successful or not.

4.1 Setup and attack implementation

As described in the sections above, we already have two programs prepared for this attack. One acts as an oracle and the other as a random data generator that gives us n random public keys. Also, we already have the oracle, which will provide us with both possible answers. Then we have a Jupyter notebook which serves as a master controlling unit that calls required subprograms or communicates with the ChipWhisperer.

Currently, the oracle saves the result of fig. 3.5 line 8 and provides the result to the standard output for both next bit being equal to zero and one. Then the master notebook collects all the data, meaning all the random public keys we used, and tries to separate them into two categories. One category is where we tried the next bit being equal to one and the other being equal to zero. We then compute the Hamming Weights of these two categories and try to correlate them with the measured traces. We expect a high correlation between the power traces with the correct guess.

4.2 Results

Sometimes it happened that the measured data were not aligned correctly. It could have been due to some difference in the internal state of the measurement cycle, resulting in differences in the first trace of the measurement in the beginning, and all subsequent traces were perfectly aligned. For that, there is a prepared command in the Jupyter notebook to remove a single trace from all the traces and its respective public key.

Sadly, the results were inconclusive. Although we have got pretty lucky and usually guessed the right bit correctly, it was roughly in 62 % of the time, and the correlation looked like noise rather than a real correlation between two dependent groups. The number of traces was usually 1000, but we tried to take up to 5000 traces with the same results. The expected Hamming weight distribution is binomial, and this can be seen in the fig. 4.1. These correlation can be seen in fig. 4.2.

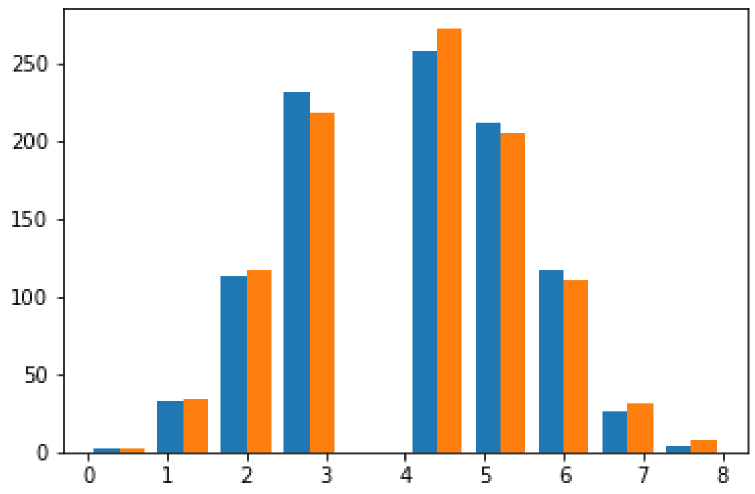


Figure 4.1: Distribution of Hamming weight in the intermediate values.

Even though we started using only 8-bit long words and their hamming weights to experimentally test the hypothesis, as the processor uses 32-bit long words, we also tried to use 32-bit long words from the internal state, but this didn't help either. The correlation still looked like playing over some independent groups. The expected Hamming weight distribution is binomial, and this can be seen in the fig. 4.3. The correlation can be seen here fig. 4.4.

We believe this result is due to a poorly chosen oracle. Although we tried many of them and even tried to measure different parts of the double and add algorithm with their intermediate values for the oracle, it had no effect.

The natural question would arise if this attack were to be successful: how do we attack the other bits? This would be fairly easy to change. Instead of measuring the first double and adding operation, we would look iteratively over every bit and measure all traces repeatedly. If we were to find a specific point in the traces that leaks the information, even the measurement and following computations would be much easier. Suppose the leakage really

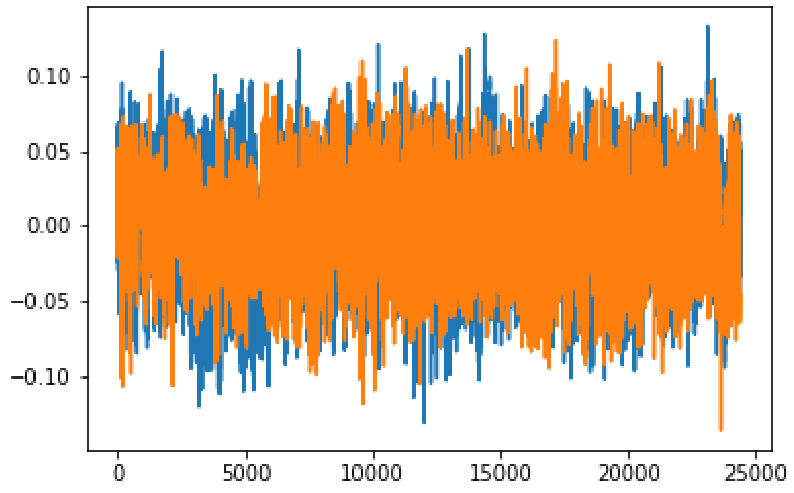


Figure 4.2: Correlation of power consumption with theoretical consumption.

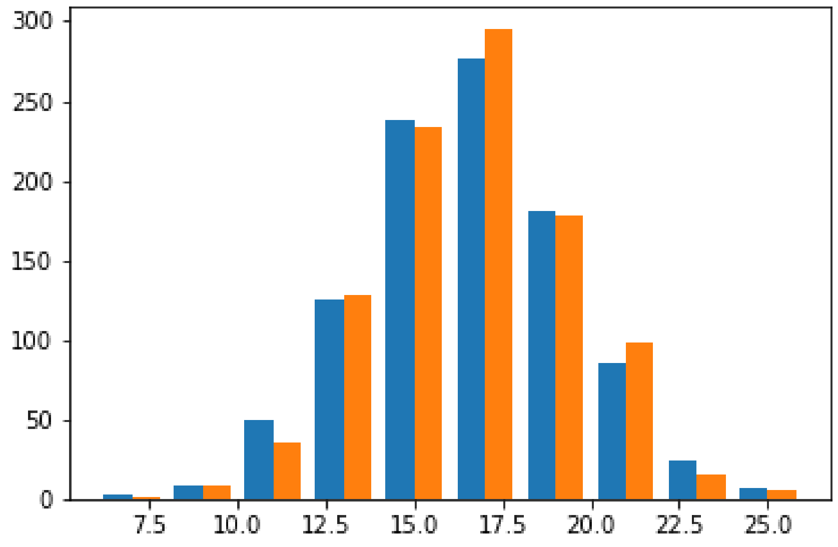


Figure 4.3: Distribution of Hamming weight in the intermediate values.

comes from the result of t2 as shown in the fig. 3.6 and since all the traces are aligned, we can figure out which sample is the leaking one. ChipWhisperer comes with a feature called segmented capture. Since when the trigger high sets, the measurement doesn't end unless we fill the required buffer size. This

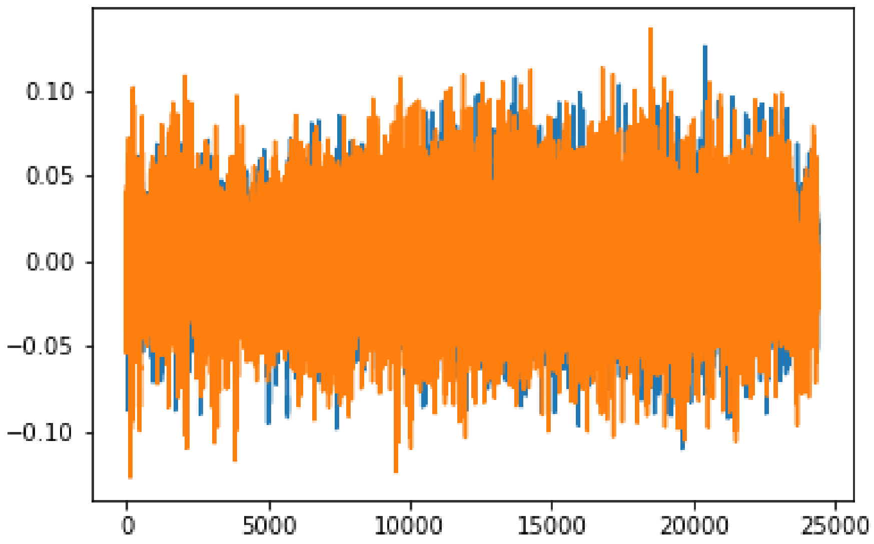


Figure 4.4: Correlation of power consumption with theoretical consumption.

isn't necessarily bad. If we were to limit our capture buffer for one trace to a small number of samples, or even with the offset function, we could trace every important sample in the computation. Since the domain here is time and the operations are constant, we could save one trace with all the time regions where the leakage function works at. Then the ChipWhisperer would give us back a 2D array of samples for every triggered trigger.

Chapter **5**

Countermeasures against side channel attacks

There are many ways of countermeasure side-channel analysis. We will focus on a few of them and how to achieve better security. Many DPA-based attacks are expected to have some victim that uses the same set of private information. Suppose there is only one secret information K. The DPA can be successful if we measure enough traces of processing random data with the same secret information K. If the victim were to use ephemeral secret information, DPA alone would not be 100 % eliminated, but the attacker would have to use a more sophisticated attack, such as horizontal DPA. To apply any countermeasure, we have to have an idea of a potential attack that we are using such a countermeasure against. There is no specific countermeasure to counter all possible side-channel attacks.

There are a few types of countermeasures. Typical types are hiding and masking. As for hiding, there are these types:

- hiding in amplitude that describes a model where all operations require the same amount of power for any data, or a model where there is generated random switching noise which will disrupt the valuable information regarding the power consumption based on data;

- hiding in time that means we could put some random periods of sleep inside our hardware or program, so it executes its operations randomly spread in time. Including dummy operations or randomly changing the order of operations if we prevent functions that depend on some other. If we apply this strategy, as for DPA, it will be unable to get some useful secret information without any preprocessing. It could be done that we realign all the traces using cross-correlation.

As for masking, we also have two types:

- logic masking that means instead of processing raw data and having deterministic functions, we use a random mask and xor it to the intermediate values; we could even xor it on top of the processed data. These masks are trying to disrupt the dependency between data and its consumption.

- Arithmetic masking uses multiplication homomorphism; we use a random scalar to modify the intermediate values in an unpredictable way. Usually, the scalar is required to have an inverse so it can be removed at the end or when the data needs to be processed.

5.1 Attacks on double and add

5.1.1 Simple power analysis

As we discussed a bit in section 2.3.3, a naive implementation of double and add can be relatively easily exploited. In [5], it is advised that the algorithm double and add does the addition part every time, not only in the conditional case of the bit is one. This algorithm is called *double and add always*. The algorithm can be seen below:

```
Algorithm 2: Double-and-add-always method
Input: $d = (d_{n-1} \cdots d_1 d_0)_2,\ P \in E(K)\ \ (d_{n-1} = 1)$.
Output: $dP$.
1. $Q[0] \leftarrow P$
2. For $i = (n-2)$ downto 0 do:
    $Q[0] \leftarrow$ ECDBL($Q[0]$),
    $Q[1-d_i] \leftarrow$ ECADD($Q[0], P$),
3. Return($Q[0]$).
```

Figure 5.1: Double and add always algorithm [5].

Also, we could use Montgomery curves and Montgomery ladder instead of using curves in the Weierstrass model. These methods also prevent SPA.

5.1.2 Differential power analysis

To prevent DPA attacks it is necessary to include some random value within the secret point multiplication dP. In [38] it is shown to generate a random scalar k and create a new $d^{'} = d + k \cdot ord(E)$, where $ord(E)$ is the order of the elliptic curve. With that, the resulting point will be $d^{'}P = dP + k \cdot ord(E) \cdot P = dP + k \cdot \mathcal{O} = dP$. This seemingly small change will lead the attacker to not being able to correlate the consumption as the k is always different, even though the result is still correct.

The other way around, we can use randomized projective coordinates [39]. Different coordinate systems can have various benefits or downfalls. Some are used for their efficiency as there is no need to calculate inverses directly, but rather with the use of some additional point addition and subtraction.

In the case of zero value attacks, it is shown in [5] that the zero value points, such as $(0, y)$ or $(x, 0)$ drastically decrease the power consumption, and they are able to detect this vulnerability.

On the other hand, it is shown how to attack an uncompressed SIKE version with a static key. This version can also contain the countermeasure of randomizing points; still, it is vulnerable to their attack, and they are able to recover the whole key. They achieve this by manipulating the kernel point of the secret isogeny, so the casual computations are undefined or random. Since this attack is based on maliciously tampering with the public key, or rather the SIKE ciphertexts, a simple countermeasure would be to validate these parameters, but that task is hard as it should be as hard as breaking the SIKE. Since their ciphertexts have a recognizable pattern, they made a countermeasure regarding this. [40]

5.1.2.1 Attacking ephemeral key

In [35] we can see that they were able to obtain an ephemeral key even with use of DPA. So far we have seen DPA only with multiple traces with a static key but there is an attack that targets ephemeral key.

They achieved this results by exploiting horizontal CPA. Since the field elements are long and they do not necessarily fit into a word as they are hundreds of bits long, they can have from every operation with these values several traces separated at the point where next word is processed as the entire word has to be for example multiplied by two, so each part will be and some corrections will be made. Since all internal functions are still deterministic they exploit this structure with only one trace.

In the paper [35] are also available some relatively cheap countermeasures. Suppose we have a secret key sk, and the two basis points of respective torsion group $\langle Q, P\rangle$, and a random point P. The countermeasure lies in masking the point $Q^{'} = Q+R$, then the almost kernel point is calculated using $P+[sk]Q^{'} = P+[sk](Q+R)$, final calculation is made by subtracting $[sk]R$ from the result. This has its downsides, as we will have to leave the Montgomery representation and calculate a very expensive square root.

5.2 Attacks on the isogeny computation

To recover the secret isogeny, we focus on the secret kernel point that is used to walk in the graph of isogenies, not to mention that the graph has a small degree. If the attacker gets at least partial knowledge of the walk, the security of the cryptosystem is weakened. [3]

5.2.1 Fault injection attack

With the fault attacks, we assume we have physical access to a given cryptosystem, and we are able to emulate or insert a fault into the typical function of the cryptosystem and the implementation will run under unexpected circumstances. Thus, if choosing the correct values, revealing part of the secret information.

In [41], it is shown that we can recover the secret isogeny from receiving an image of a random point under the secret isogeny. The attack uses a random point on that particular curve E, which was brute-forced to match their specific needs, and the probability of laying on the curve is relatively high. Since the isogeny is a group homomorphism, they find a linear combination of their points to then help them to construct a dual isogeny and then recover the original isogeny. It is also said that you should never reveal the image of a random point under the secret isogeny.

To eliminate this attack, it is advised to implement order checking of the given public key, respectively, the basis points of the torsion subgroups.

Another type of attack could be implementing loop aborts [42]. These work in the way that we force a loop to an unexpected end. For example, a glitch could happen when a condition is being checked if some control variable is over the required limit. Even though the loop should have continued, it unexpectedly ended, for example due to skipping a jump instruction.

Attack mentioned in [42] explores the possibility of breaking the computation of the large degree isogeny. They also let Alice use validation methods to check the received public key, so they limit themselves with this. These attacks recover bit by bit from the least significant bit. They use an oracle that checks the $k-th$ iteration of the isogeny computation, to remind it of the calculation of E_k and then compare this faulty elliptic curve j-invariant to their own within the oracle. So they basically test bit by bit, guess and check if they hit.

To defend against this type of attack, it is not only necessary to validate the input and check whether the other side is trying to be malicious or not, but also not to give away any free information about our secret isogeny. It is proposed to check the counter after the loop if it is really the expected value or not. That is a case for those attackers who don't have the capability to introduce perfect and precise faults into the algorithm. Also, there is a recommendation to use multiple counters to have more precise control over the loop's iterations, and it could also lead to checking against random faults not only invoked by some attacker.

The last countermeasure from [43] is about possible unknowing exposure of the kernel point in the middle of computing the large degree isogeny. They suppose they could divide the large degree isogeny computation into two smaller isogeny computations. The loop break could have induced the exposure. Determining the order of the intermediate kernel point, they can then use brute force with the use of the generalized elliptic curve logarithm. This will yield the secret value m, which was the secret scalar. To avoid this attack, they suggest generating some random walk in the isomorphism class of the starting curve E_0, so the relation between the basis point and the isogeny is obfuscated.

Conclusion

In this thesis, we designed and experimentally evaluated a side-channel attack. The attack on the key exchange SIDH protocol was designed to use a correlation power analysis. We used an uncompressed 32-bit reference implementation from the NIST standardization process. This version was slightly modified to fit our time constraints. In the thesis, we describe an attack on the first bit of the secret key due to limited memory on the used ChipWhisperer-Lite and show instructions on how to reveal the whole secret key if we can break the first bit.

The measurements were made on an ARM Cortex-M4 microcontroller. Unfortunately, the designed attack did not work, and we could not reveal the first bit of the secret key. The steps after not being able to break the first bit were:

- we conducted a broad trial and error phase of testing whether the intermediate values we calculate are the same both on the microcontroller and the computer, which acted as the oracle. Both intermediate devices worked perfectly fine, so we believe this was not the case.

- Next, we carefully measured and researched if what we really measured were the first steps of the SIDH algorithm. This conducted once more experimentally remeasuring cycle counts of the operation $xDBLADD$, so we knew the length of each function and how many operations we could sample. Experimentally we understood that the measurements we were doing were correct.

- Next, the hypothesis of being able to correlate the Hamming weight of data using 8-bit words, respectively 32-bit words, is incorrect. We conducted a brief correlation testing with the public key components, and we were able to get the correlation. So this was not probably the cause.

- The last item on our list is the oracle function itself. Even though it gave us the correct intermediate values, we probably were misusing it, or we were processing the data in the Jupyter notebook improperly.

The aim was to get to know and understand the underlying SIDH and SIKE algorithms, have some hands-on experience with a side-channel attack on the SIDH algorithm, and to research possible counter-measurements to possible side-channel analysis. We have gathered a lot of interesting information regarding SCA against SIKE. All the aspects of the thesis were completed, although we could not show our attack is convincingly working with high probability.

Bibliography

[1] Jalali, A.; Azarderakhsh, R.; et al. NEON SIKE: Supersingular Isogeny Key Encapsulation on ARMv7. In *Security, Privacy, and Applied Cryptography Engineering*, edited by A. Chattopadhyay; C. Rebeiro; Y. Yarom, Cham: Springer International Publishing, 2018, ISBN 978-3-030-05072-6, pp. 37–51.

[2] NewAE Technology Inc. ChipWhisperer - the complete open-source toolchain for side-channel power analysis and glitching attacks. Available from: `https://github.com/newaetech/chipwhisperer`

[3] Campagna, M.; Costello, C.; et al. Supersingular isogeny key encapsulation. 2019.

[4] Fabio Campos. Safe Error Attacks on SIKE and CSIDH. Available from: `https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC`

[5] Akishita, T.; Takagi, T. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In *Information Security*, edited by C. Boyd; W. Mao, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-39981-0, pp. 218–233.

[6] Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, volume 41, no. 2, 1999: pp. 303–332.

[7] Lidl, R.; Niederreiter, H. *Finite Fields.*, *Encyclopedia of Mathematics and Its Applications*, volume Second edition. Cambridge University Press, 1997, ISBN 9780521392310. Available from: `https://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=569266&lang=cs&site=ehost-live`

[8] Riley, K. F.; Hobson, M. P.; et al. *Group theory*. Cambridge University Press, second edition, 2002, p. 883–917, doi:10.1017/CBO9781139164979.026.

[9] Kibler, M. R. 1 - The Structures of Ring and Field. In *Galois Fields and Galois Rings Made Easy*, edited by M. R. Kibler, Elsevier, 2017, ISBN 978-1-78548-235-9, pp. 1–32, doi:https://doi.org/10.1016/B978-1-78548-235-9.50001-4. Available from: `https://www.sciencedirect.com/science/article/pii/B9781785482359500014`

[10] Mullen, G. L.; Panario, D. *Handbook of Finite Fields*. Chapman & Hall/CRC, first edition, 2013, ISBN 143987378X.

[11] De Feo, L. Mathematics of Isogeny Based Cryptography. 2017, doi:10.48550/ARXIV.1711.04062. Available from: `https://arxiv.org/abs/1711.04062`

[12] Lange, T. *Edwards Curves*. Boston, MA: Springer US, 2011, ISBN 978-1-4419-5906-5, pp. 380–382, doi:10.1007/978-1-4419-5906-5_243. Available from: `https://doi.org/10.1007/978-1-4419-5906-5_243`

[13] Silverman, J. H. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.

[14] Washington, L. C. *Elliptic curves: number theory and cryptography*. Chapman and Hall/CRC, 2008.

[15] Shumow, D. Isogenies of Elliptic Curves: A Computational Approach. 2009, doi:10.48550/ARXIV.0910.5370. Available from: `https://arxiv.org/abs/0910.5370`

[16] Vélu, J. *Isogénies entre courbes elliptiques*. CR Acad. Sci. Paris Sér. AB, 273:A238–A241, 1971.

[17] Couveignes, J.-M. Hard Homogeneous Spaces. *IACR Cryptology ePrint Archive*, volume 2006, 01 2006: p. 291.

[18] Stolbunov, A. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Advances in Mathematics of Communications*, volume 4, no. 2, 2010: pp. 215–235.

[19] Stolbunov, A. *Cryptographic Schemes Based on Isogenies*. Dissertation thesis, 01 2012, doi:10.13140/RG.2.2.20826.44488.

[20] Childs, A. M.; Jao, D.; et al. Constructing elliptic curve isogenies in quantum subexponential time. 2010, doi:10.48550/ARXIV.1012.4019. Available from: `https://arxiv.org/abs/1012.4019`

[21] Hermans, J.; Vercauteren, F.; et al. Speed Records for NTRU. In *Topics in Cryptology - CT-RSA 2010*, edited by J. Pieprzyk, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-11925-5, pp. 73–88.

[22] Jao, D.; De Feo, L. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In *Post-Quantum Cryptography*, edited by B.-Y. Yang, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-25405-5, pp. 19–34.

[23] Moody, D.; Alagic, G.; et al. Status report on the second round of the NIST post-quantum cryptography standardization process. 2020.

[24] Costello, C. The Case for SIKE: A Decade of the Supersingular Isogeny Problem. Cryptology ePrint Archive, Paper 2021/543, 2021, `https://eprint.iacr.org/2021/543`. Available from: `https://eprint.iacr.org/2021/543`

[25] Moody, D. Round 2 of NIST PQC competition. *Invited talk at PQCrypto*, 2019.

[26] Tani, S. Claw finding algorithms using quantum walk. *Theoretical Computer Science*, volume 410, no. 50, 2009: pp. 5285–5297, ISSN 0304-3975, doi:https://doi.org/10.1016/j.tcs.2009.08.030, mathematical Foundations of Computer Science (MFCS 2007). Available from: `https://www.sciencedirect.com/science/article/pii/S0304397509006136`

[27] Jaques, S.; Schanck, J. *Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE*. 08 2019, ISBN 978-3-030-26947-0, pp. 32–61, doi:10.1007/978-3-030-26948-7_2.

[28] Joye, M. Elliptic curves and side-channel analysis. *ST Journal of System Research*, volume 4, no. 1, 2003: pp. 17–21.

[29] Wu, K.; Li, H.; et al. Simple Power Analysis on Elliptic Curve Cryptosystems and Countermeasures: Practical Work. In *2009 Second International Symposium on Electronic Commerce and Security*, volume 1, 2009, pp. 21–24, doi:10.1109/ISECS.2009.7.

[30] Brier, E.; Clavier, C.; et al. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, edited by M. Joye; J.-J. Quisquater, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN 978-3-540-28632-5, pp. 16–29.

[31] De Feo, L.; El Mrabet, N.; et al. SIKE Channels: Zero-Value Side-Channel Attacks on SIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, volume 2022, no. 3, Jun. 2022: p. 264–289, doi:10.46586/tches.v2022.i3.264-289. Available from: `https://tches.iacr.org/index.php/TCHES/article/view/9701`

[32] Koziel, B.; Azarderakhsh, R.; et al. Side-Channel Attacks on Quantum-Resistant Supersingular Isogeny Diffie-Hellman. In *Selected Areas in Cryptography – SAC 2017*, edited by C. Adams; J. Camenisch, Cham: Springer International Publishing, 2018, ISBN 978-3-319-72565-9, pp. 64–81.

[33] Villanueva-Polanco, R.; Angulo-Madrid, E. Cold Boot Attacks on the Supersingular Isogeny Key Encapsulation (SIKE) Mechanism. *Applied Sciences*, volume 11, no. 1, 2021, ISSN 2076-3417, doi:10.3390/app11010193. Available from: `https://www.mdpi.com/2076-3417/11/1/193`

[34] Zhang, F.; Yang, B.; et al. Side-Channel Analysis and Countermeasure Design on ARM-Based Quantum-Resistant SIKE. *IEEE Transactions on Computers*, volume 69, no. 11, 2020: pp. 1681–1693, doi: 10.1109/TC.2020.3020407.

[35] Genêt, A.; de Guertechin, N. L.; et al. Full Key Recovery Side-Channel Attack Against Ephemeral SIKE on the Cortex-M4. In *Constructive Side-Channel Analysis and Secure Design*, edited by S. Bhasin; F. De Santis, Cham: Springer International Publishing, 2021, ISBN 978-3-030-89915-8, pp. 228–254.

[36] David Jao, et al. Supersingular Isogeny Key Encapsulation. Available from: `https://sike.org/#implementation`

[37] FastSIKE2019. FastSIKE2019. Available from: `https://github.com/FastSIKE2019/generic`

[38] Coron, J.-S. Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems*, edited by Ç. K. Koç; C. Paar, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, ISBN 978-3-540-48059-4, pp. 292–302.

[39] Menezes, A. J. *Elliptic curve public key cryptosystems*, volume 234. Springer Science & Business Media, 1993.

[40] De Feo, L.; El Mrabet, N.; et al. SIKE Channels: Zero-Value Side-Channel Attacks on SIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, volume 2022, no. 3, Jun. 2022: p. 264–289, doi:10.46586/tches.v2022.i3.264-289. Available from: `https://tches.iacr.org/index.php/TCHES/article/view/9701`

[41] Ti, Y. B. Fault Attack on Supersingular Isogeny Cryptosystems. In *Post-Quantum Cryptography*, edited by T. Lange; T. Takagi, Cham: Springer International Publishing, 2017, ISBN 978-3-319-59879-6, pp. 107–122.

[42] Gélin, A.; Wesolowski, B. Loop-Abort Faults on Supersingular Isogeny Cryptosystems. In *Post-Quantum Cryptography*, edited by T. Lange; T. Takagi, Cham: Springer International Publishing, 2017, ISBN 978-3-319-59879-6, pp. 93–106.

[43] Koziel, B.; Azarderakhsh, R.; et al. An Exposure Model for Supersingular Isogeny Diffie-Hellman Key Exchange. In *Topics in Cryptology – CT-RSA 2018*, edited by N. P. Smart, Cham: Springer International Publishing, 2018, ISBN 978-3-319-76953-0, pp. 452–469.

Appendix **A**

Acronyms

KEM	Key Encapsulation Mechanism
SIDH	Supersingular Isogeny Diffie-Hellman
SIKE	Supersingular Isogeny Key Encapsulation
CCA	Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
NIST	National Institute of Standards and Technology
AES	Advanced Encryption Standard
SHA	Secure Hash Algorithm
RAM	Random Access Memory
VHDL	VHSIC Hardware Description Language
SPA	Simple Power Analysis
DPA	Differential Power Analysis
CPA	Correlation Power Analysis
API	Application Programming Interface
$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of all neutral numbers
$\mathbb{Q}$	Set of all rational numbers
$\mathbb{R}$	Set of all real numbers
$\mathbb{C}$	Set of all complex numbers

$\mathbb{Z}/p$	Integers modulo prime p
$\|\mathbf{M}\|$	Size, or order of a given set M
$\langle a \rangle$	a is generator of some subgroup or group
$P(x)$	A polynomial with variable x
$char(F)$	Characteristics of a field F
$E(K)$	An elliptic curve E over field K
$[m]P$	Multiple of point P on an elliptic curve
$E[m]$	m torsion subgroup over elliptic curve E
$\mathcal{O}$	Point at infinity
$ker(\phi)$	Kernel of a map ϕ
C	Programming language C
CPU	Central Processing Unit
ARM	Advanced RISC Machines
CW	ChipWhisperer
$xDBLADD$	Double and add function

Appendix **B**

Contents of enclosed CD

`measurements`	the directory with measured data
`src`	the directory of source codes
`  generate`	the source code of the codes to generate data
`  jupyter`	the jupyter notebook
`  sike`	the source code for the chipwhisperer
`  thesis`	the directory of LaTeX source codes of the thesis
`text`	the thesis text directory
`  Frantisek_Kovar_DT_2022.pdf`	the thesis text in PDF format