A Side Journey to Titan

Side-Channel Attack on the Google Titan Security Key

(Revealing and Breaking NXP's P5x ECDSA Implementation on the Way)

Victor LOMNE and Thomas ROCHE

NinjaLab

161 rue Ada, 34095 Montpellier, France

firstname@ninjalab.io

January 7, 2021

ABSTRACT

The *Google Titan Security Key* is a FIDO U2F hardware device proposed by Google (available since July 2018) as a two-factor authentication token to sign in to applications (*e.g.* your Google account). We present here a side-channel attack that targets the *Google Titan Security Key*'s secure element (the *NXP A700X* chip) by the observation of its local electromagnetic radiations during ECDSA signatures (the core cryptographic operation of the FIDO U2F protocol). This work shows that an attacker can clone a legitimate *Google Titan Security Key*.

To understand the NXP ECDSA implementation, find a vulnerability and design a key-recovery attack, we had to make a quick stop on *Rhea* (*NXP J3D081* JavaCard smartcard). Freely available on the web, this product looks very much like the *NXP A700X* chip and uses the same cryptographic library. *Rhea*, as an open JavaCard platform, gives us more control to study the ECDSA implementation.

We could then show that the electromagnetic side-channel signal bears partial information about the ECDSA ephemeral key. The sensitive information is recovered with a non-supervised machine learning method and plugged into a customized lattice-based attack scheme.

Finally, 4000 ECDSA observations were enough to recover the (known) secret key on *Rhea* and validate our attack process. It was then applied on the *Google Titan Security Key* with success (this time with 6000 observations) as we were able to extract the long term ECDSA private key linked to a FIDO U2F account created for the experiment.

Cautionary Note Two-factor authentication tokens (like FIDO U2F hardware devices) primary goal is to fight phishing attacks. Our attack requires physical access to the *Google Titan Security Key*, expensive equipment, custom software, and technical skills.

Thus, as far as the work presented here goes, it is still safer to use your *Google Titan Security Key* or other impacted products as FIDO U2F two-factor authentication token to sign in to applications rather than not using one.

Nevertheless, this work shows that the *Google Titan Security Key* (and other impacted products) would not avoid unnoticed security breach by attackers willing to put enough effort into it. Users that face such a threat should probably switch to other FIDO U2F hardware security keys, where no vulnerability has yet been discovered.

Contents

Chapter 1

Introduction

1.1 Context

1.1.1 Study Motivation

This journey begins in Amsterdam, Netherlands, during the international conference CHES in September 2018 [7].

The second keynote of the conference was given by Elie Bursztein, Google's anti-abuse research team leader. At the end of his talk, Elie made some advertising about the new Google security product, the *Google Titan Security Key* [17]. He also promoted that *the hardware chips are designed to resist physical attacks aimed at extracting firmware and secret key material.*

During the Q&A session of the keynote, we asked if his team tried to apply the machine-learning based side-channel methods he was promoting during his keynote to their new product, and how resistant it was. We did not get any clear answer, but at the end of his talk, he offered to some of the attendees few samples of their new product, which was at that time only commercially available in the US. We managed to get one, with the idea to check by ourselves its robustness against side-channel analysis !

1.1.2 Product Description

The *Google Titan Security Key* is a hardware FIDO U2F (universal second factor) device. It can then be used, in addition to your login and password, to sign in to your Google account[1].

The original *Google Titan Security Key* box [17] (which has been available in the US market since July 2018, and in the EU market since February 2020) contains two versions:

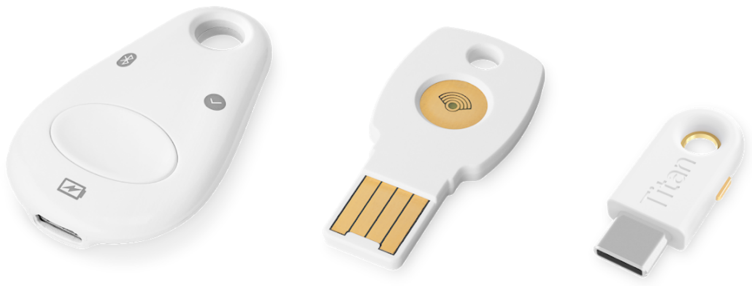
- one with three communication interfaces, micro-USB, NFC and BLE (Bluetooth Low Energy), see Figure 1.1[2], left side;
- one with two communication interfaces, USB type A and NFC (Figure 1.1 in the middle).

[1]see `https://www.yubico.com/works-with-yubikey/catalog/` for a list of almost all applications supporting FIDO U2F protocol

[2]pictures credit: `https://store.google.com/product/titan_security_key`

Furthermore, a third version has been released in October 2019, with only one communication interface, USB type C (Figure 1.1, right side).

Figure 1.1: *Google Titan Security Key* - Left: version with micro-USB, NFC and BLE interfaces - Middle: version with USB type A and NFC interfaces - Right: version with USB type C interface



The *Google Titan Security Key* main functionality is to generate a unique secret key and keep it safe. This secret key will be used to sign in to the user account. Since no device or server knows the secret (except the *Google Titan Security Key* itself), nobody can sign in to the legitimate user account without physically possessing the device. The security hence resides in ensuring the confidentiality of the secret key, as stated by Google Cloud product manager Christiaan Brand[3]:

> *"Titan Security Keys are designed to make the critical cryptographic operations performed by the security key strongly resistant to compromise during the entire device lifecycle, from manufacturing through actual use.*
>
> *The firmware performing the cryptographic operations has been engineered by Google with security in mind. This firmware is sealed permanently into a secure element hardware chip at production time in the chip production factory. The secure element hardware chip that we use is designed to resist physical attacks aimed at extracting firmware and secret key material.*
>
> *These permanently-sealed secure element hardware chips are then delivered to the manufacturing line which makes the physical security key device. Thus, the trust in Titan Security Key is anchored in the sealed chip as opposed to any other later step which takes place during device manufacturing."*

1.1.3 Contributions and Document Organization

In this report, we will show how we found a side-channel vulnerability in the cryptographic implementation of *Google Titan Security Key*'s secure element (we assigned CVE-2021-3011). Our contribution is threefold:

- use side-channel analysis to reverse-engineer the cryptographic primitive implementation and reveal its countermeasures (this part is presented in Chapter 2);
- discover a previously unknown vulnerability in the (previously unknown) implementation (see Chapter 3);

[3]`https://cloud.google.com/blog/products/identity-security/titan-security-keys-now-available-on-the-google-store`, last accessed the 31 Dec 2020

- exploit this vulnerability with a custom lattice-based attack and fully recover an ECDSA private key from the *Google Titan Security Key* (see Chapter 4).

Finally, in Chapter 5 we discuss the impact of our work, list the impacted products and provide the project timeline.

But first things first, let us have a look at the public information we have on the *Google Titan Security Key* and on the FIDO U2F protocol.

1.2 Preliminaries

1.2.1 FIDO U2F Protocol

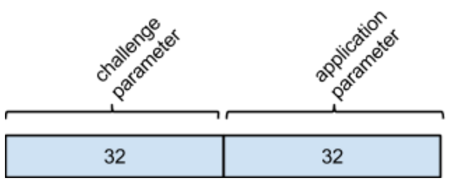
The FIDO U2F protocol, when used with a hardware FIDO U2F device like the *Google Titan Security Key*, works in two steps: *registration* and *authentication*.

Three parties are involved: the *relying party* (*e.g.* the Google server), the *client* (*e.g.* a web browser) and the *U2F device*. We summarize here how the different messages are constructed and exchanged, as explained in [15].

Registration

1. The FIDO *client* first contacts the *relying party* to obtain a challenge, and then constructs the `registration request message`, before sending it to the *U2F device*. As described in Figure 1.2[4], it has two parts:
   - the challenge parameter, which is the SHA-256 hash of the client data (containing among other things the challenge);
   - the application parameter, which is the SHA-256 hash of the application ID (the application requesting the registration).

Figure 1.2: FIDO U2F Registration Request Message



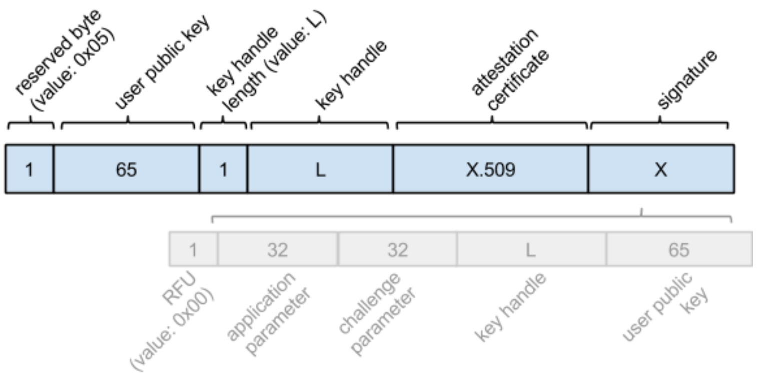
2. The *U2F device* creates a new Elliptic Curve Digital Signature Algorithm (ECDSA) key-pair in response to the `registration request message`, and answers the `registration response message`, as described in Figure 1.3[4]. Its raw representation is the concatenation of the following:
   - a reserved byte;

[4]pictures credit: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html

- a user public key, which is the (uncompressed) X-Y representation of a curve point on the P256 NIST elliptic curve [34];
- a key handle length byte;
- a key handle, which allows the U2F token to identify the generated key pair. Note that U2F tokens *may* wrap (*i.e.* encrypt) the generated ECDSA private key and the application ID it was generated for, and output that as the key handle (see section 2.2 of [14] for more details);
- an attestation certificate in X.509 DER format;
- an ECDSA signature on P256 encoded in ANSI X9.62 format, over the following byte string:
  - a byte reserved for future use;
  - the application parameter, from the `registration request message`;
  - the challenge parameter, from the `registration request message`;
  - the above key handle;
  - the above user public key.

3. Finally, the FIDO *client* sends back the `registration response message` to the *relying party*, which can store its different fields for later authentication.

Figure 1.3: FIDO U2F Registration Response Message



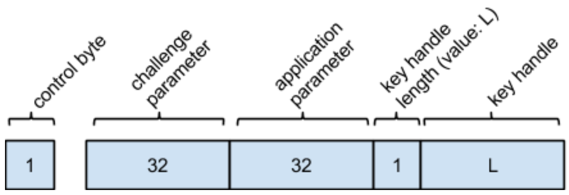
Authentication

1. The FIDO *client* first contacts the *relying party* to obtain a challenge, and then constructs the `authentication request message`, before sending it to the *U2F device*. As described in Figure 1.4[4], it is made of 5 parts:

   - control byte, which is determined by the FIDO client (the relying party cannot specify its value). The FIDO client will set the control byte to one of the following values:
     - 0x07 - *check-only*
     - 0x03 - *enforce-user-presence-and-sign*
     - 0x08 - *dont-enforce-user-presence-and-sign*

- the challenge parameter, which is the SHA-256 hash of the client data;
- the application parameter, which is the SHA-256 hash of of the application ID (the application requesting the authentication);
- a key handle length byte;
- a key handle, which is provided by the relying party, and was obtained during registration.

Figure 1.4: FIDO U2F Authentication Request Message

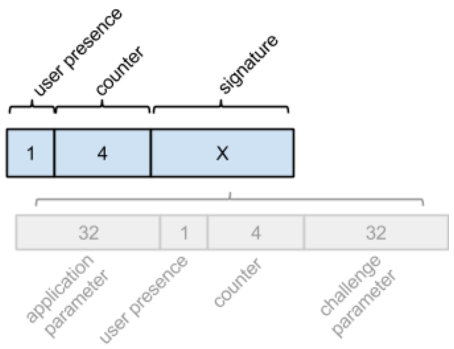


2. If the *U2F device* succeeds to process/sign the `authentication request message` described above, it answers the `authentication response message`. As described in Figure 1.5[4], it is made of 3 parts:

    - a user presence byte, where its first bit indicates whether user presence was verified or not;
    - a counter, which is the big-endian representation on 4 bytes of a counter value that the U2F device increments every time it performs an authentication operation. Note that this counter may be *global* (*i.e.* the same counter is incremented regardless of the application parameter in `authentication request message`), or per-application. See Section 2.6 of [14] for more details. Furthermore, as explained in Section 8.1 of [16], the counter *may* be used as a signal for detecting cloned U2F devices;
    - an ECDSA signature on P256 encoded in ANSI X9.62 format, over the following byte string:
        - the application parameter, from the `authentication request message`;
        - the above user presence byte;
        - the above counter;
        - the challenge parameter, from the `authentication request message`.

3. Finally, the FIDO *client* sends back the `authentication response message` to the *relying party*, which will verify the ECDSA signature using the public key obtained during registration.

Figure 1.5: FIDO U2F Authentication Response Message



1.2.2 A Side-Channel Attack Scenario on the FIDO U2F Protocol

From the study of the FIDO U2F protocol, we can imagine the following attack scenario:

1. the adversary steals the login and password of a victim's application account protected with FIDO U2F (*e.g.* via a phishing attack);

2. the adversary gets physical access to the victim's U2F device during a limited time frame, without the victim noticing;

3. thanks to the stolen victim's login and password (for a given application account), the adversary can get the corresponding *client data* and *key handle*, and then sends the authentication request to the U2F device as many time as necessary[5] while performing side-channel measurements;

4. the adversary quietly gives back the U2F device to the victim;

5. the adversary performs a side-channel attack on the measurements, and succeeds in extracting the ECDSA private key linked to the victim's application account;

6. the adversary can sign in to the victim's application account without the U2F device, and without the victim noticing. In other words the adversary created a clone of the U2F device for the victim's application account. This clone will give access to the application account as long as the legitimate user does not revoke its second factor authentication credentials.

Note that the *relying party* might use the counter value to detect cloned U2F devices and then limit (but not totally remove) the attack impact.

Practical Considerations

To apply the above scenario, we need to find a side-channel vulnerability in the ECDSA implementation. The cryptographic primitives implementation is not open-source and we have no information on its side-channel countermeasures. This is standard procedure for secure elements: in this field the secrecy of the implementation is still believed to add an extra layer of security.

[5]it might be limited to several millions of requests (the counter being encoded on 4 bytes).

As we have seen, the FIDO U2F protocol is very simple, the only way to interact with the U2F device is by registration or authentication requests. The registration phase will generate a new ECDSA key pair and output the public key. The authentication will mainly execute an ECDSA signature operation where we can choose the input message and get the output signature.

Hence, even for a legitimate user, there is no way to know the ECDSA secret key of a given application account. This is a limitation of the protocol which, for instance, makes impossible to transfer the user credentials from one security key to another. If a user wants to switch to a new hardware security key, a new registration phase must be done for every application account. This will create new ECDSA key pairs and revoke the old ones.

This limitation in functionality is a strength from a security point-of-view: by design it is not possible to create a clone. It is moreover an obstacle for side-channel reverse-engineering. With no control whatsoever on the secret key it is barely possible to understand the details of (let alone to attack) a highly secured implementation. We will have to find a workaround to study the implementation security in a more convenient setting.

1.2.3 *Google Titan Security Key* Teardown

Once plugged into a computer's USB port, `lsusb` outputs:

```
Bus 001 Device 018:  ID 096e:0858 Feitian Technologies, Inc.
```

As a matter of fact, the company who designed the *Google Titan Security Key* is Feitian [13][6]. Indeed Feitian proposes generic FIDO U2F security keys, with customization for casing, packaging and related services [12].

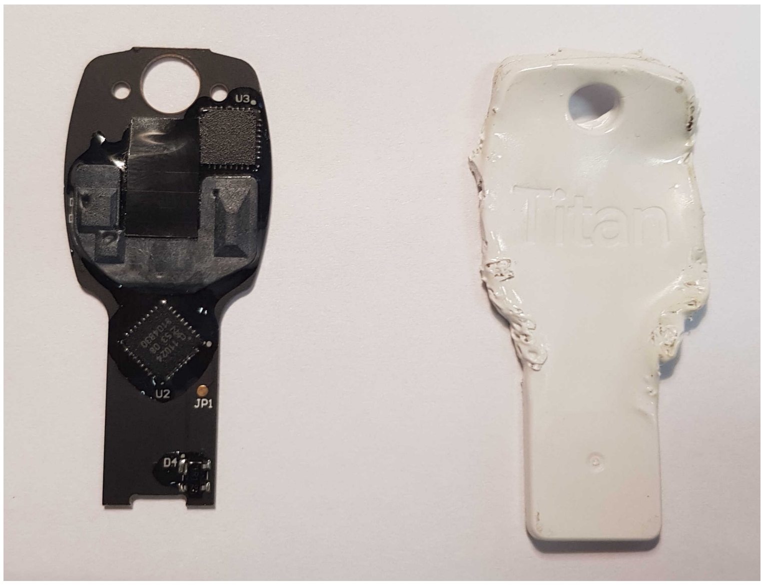
Removing the Casing

We decided to perform a teardown of the *USB type A Google Titan Security Key* (the middle one in Figure 1.1).

The plastic casing is made of two parts which are strongly glued together, and it is not easy to separate them with a knife, cutter or scalpel. We used a hot air gun to soften the white plastic, and to be able to easily separate the two casing parts with a scalpel. The procedure is easy to perform and, done carefully, allows to keep the Printed Circuit Board (PCB) safe. Figure 1.6 shows the extracted *Google Titan Security Key* PCB and one part of the casing, soften due to the application of hot air.

[6]this information is publicly available since https://www.cnbc.com/2018/08/30/google-titan-made-by-chinese-company-feitian.html

Figure 1.6: *Google Titan Security Key* Opened

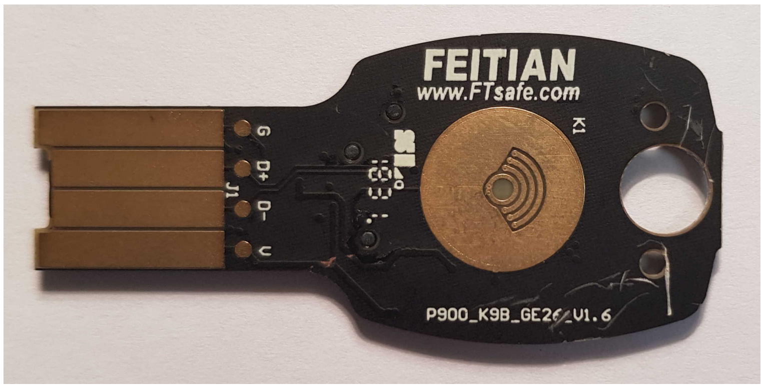


An interesting future work could be to find a way to open the *Google Titan Security Key* casing without damaging the two parts, such that it could be possible to re-assemble them after physical tampering.

PCB Analysis

Figure 1.7 shows the recto of the *Google Titan Security Key* PCB.

Figure 1.7: *Google Titan Security Key* PCB - Recto

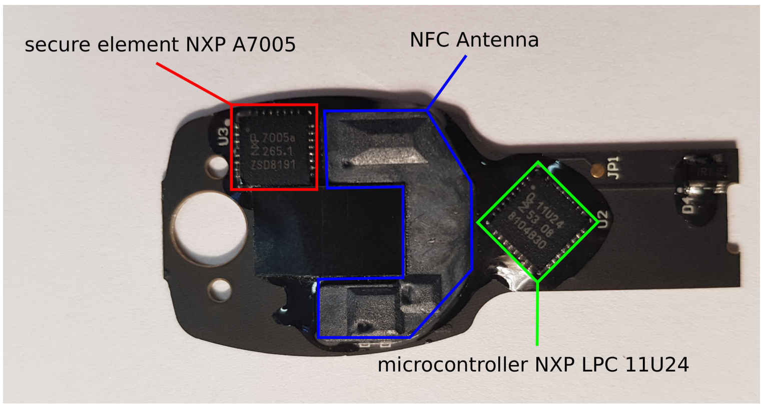


On Figure 1.8, one can see the verso of the *Google Titan Security Key* PCB, where the different circuits are soldered. The Integrated Circuit (IC) package markings allow to guess the IC references:

- the first IC (in green on Figure 1.8) is a general purpose microcontroller from NXP, the LPC11u24 from the LPC11U2x family [35]. It acts as a router between the USB and NFC interfaces and the secure element;

- the second IC (in red on Figure 1.8) is a secure authentication microcontroller also from NXP, the A7005 from the A700X family [30]. It acts as the secure element, storing cryptographic secrets and performing cryptographic operations (we validated this point by probing electric signals between the two ICs while processing an `authentication request message`).

Figure 1.8: *Google Titan Security Key* PCB - Verso



Similar Teardowns by Hexview

- a similar teardown of the *Google Titan Security Key* [19] confirms our observations;
- a similar teardown of the *Yubico Yubikey Neo* [20] shows that its hardware architecture is very similar to the one of the *Google Titan Security Key*.

1.2.4 NXP A700X Chip

Datasheet Analysis

As one can see on Figure 1.8, the package marking of the secure element is `NXP A7005a`. From its public datasheet [30], we get the following interesting information:

- it runs the NXP's JavaCard Operating System called JCOP, in version JCOP 2.4.2 R0.9 or R1 (JavaCard version 3.0.1 and GlobalPlatform version 2.1.1);
- technological node is 140nm;
- CPU is `Secure_MX51`;
- EPPROM size is of 80kB;
- 3-DES and AES hardware co-processors;
- public-key cryptographic co-processor is the NXP `FameXE`;
- RSA available up to 2048 bits and ECC available up to 320 bits.

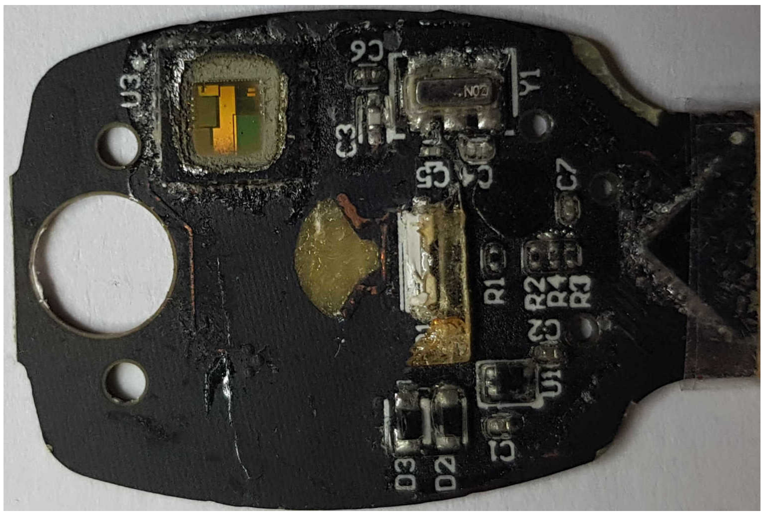
From the `NXP A7005a` RSA and ECC key length limitations, the JCOP version and the technological node, it is clear that this is not a very recent chip.

IC Optical Analysis

In order to perform an IC optical analysis, we first performed a package opening procedure of the `NXP A7005a` via wet chemical attack, as its package is made of epoxy. Luckily, we have access less than 100 meters away from our offices to the clean room of the university of Montpellier [44].

We first protected the PCB by sticking some aluminium tape around it, and cut a square just above the `NXP A7005a` package. Then we warmed some fuming nitric acid, and put carefully some drops of acid on the package, until we see the die appear. Figure 1.9 depicts the result. The device is still alive, it will be useful for ElectroMagnetic (EM) side-channel measurements.

Figure 1.9: Verso of *Google Titan Security Key* PCB, with `NXP A7005a` die visible after wet chemical attack of its package



Similarities with other NXP Products

As explained before, trying to perform a black-box side-channel attack on a cryptographic implementation of a commercial product with potentially dedicated countermeasures is usually really hard if no sample with known key is available. So with the information gathered from the `NXP A700X` datasheet and its IC optical analysis, we tried to find similar NXP products where we have more control on the ECDSA operations.

We found out that several NXP JavaCard platforms have similar characteristics with the `NXP A700X`. Note that these NXP JavaCard platforms are based on NXP P5x chips.

1.3 NXP Cryptographic Library on P5x Chips

1.3.1 The NXP P5x Secure Microcontroller Family

The NXP P5x secure microcontroller family is the first generation of NXP secure elements, also called `SmartMX` family [36], with the following characteristics:

- technological node of 140nm;

- CPU is `Secure_MX51`;
- contact and/or contactless interface(s);
- 3-DES and AES hardware co-processors;
- public-key cryptographic co-processor `FameXE`;
- optionnally NXP Cryptolib for RSA and ECC operations;
- Common Criteria (CC) and EMVCo certified (last CC certification found in 2015).

1.3.2 Available NXP JavaCard Smartcards on P5x Chips

We went through the public data that can be found online and figured out that several NXP JavaCard smartcards are based on P5x chips and have similar characteristics with the `NXP A700X`. Thanks to BSI and NLNCSA CC public certification reports[7], we were able to gather the following (non-exhaustive) list of NXP JavaCard smartcards based on P5x chips:

- Product Family A
  - J3A081, J2A081, J3A041
  - JCOP 2.4.1 R3, JavaCard 2.2.2 and GlobalPlatform 2.1.1
  - Secure MCU: P5CD081V1A / P5CC081V1A (die marking: T046B)
  - CC certification report BSI-DSZ-CC-0675
  - Cryptolib V2.7 (CC certification report BSI-DSZ-CC-0633-2010)
- Product Family B
  - J3D145_M59, J2D145_M59, J3D120_M60, J3D082_M60, J2D120_M60, J2D082_M60
  - JCOP 2.4.2 R2, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5CD145V0B / P5CC145V0B (die marking: T051A)
  - CC certification report BSI-DSZ-CC-0783-2013
  - Cryptolib V2.7/2.9 (CC certification report BSI-DSZ-CC-0750-V2-2014)
- Product Family C
  - J3D081_M59, J2D081_M59, J3D081_M61, J2D081_M61
  - JCOP 2.4.2 R2, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5CD081V1A (die marking T046B)
  - CC certification report BSI-DSZ-CC-0784-2013
  - Cryptolib V2.7 (CC certification report BSI-DSZ-CC-0633-2010)
- Product Family D
  - J3D081_M59_DF, J3D081_M61_DF
  - JCOP 2.4.2 R2, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5CD081V1D (die marking: T046D)

[7] https://www.bsi.bund.de/EN/Topics/Certification/certified_products/Archiv_reports.html

  - CC certification report BSI-DSZ-CC-0860-2013
  - NIST FIPS 140-2 certified
  - Cryptolib V2.7 (CC certification report BSI-DSZ-CC-0864-2012)

- Product Family E
  - J3E081_M64, J3E081_M66, J2E081_M64, J3E041_M66, J3E016_M66, J3E016_M64, J3E041_M64
  - JCOP 2.4.2 R3, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5CD016/021/041/051 and P5Cx081V1A/V1A(s) (die marking: T046B or s046B)
  - CC certification report NSCIB-CC-13-37761-CR
  - Cryptolib V2.7 (CC certification report BSI-DSZ-CC-0633-2010)
- Product Family F
  - J3E145_M64, J3E120_M65, J3E082_M65, J2E145_M64, J2E120_M65, J2E082_M65
  - JCOP 2.4.2 R3, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5Cx128/P5Cx145 V0v/V0B(s) (die marking: T051A, T051B or s051B)
  - CC certification report NSCIB-CC-13-37760-CR
  - Cryptolib V2.7/2.9 (CC certification report BSI-DSZ-CC-0750-V2-2014)
- Product Family G
  - J3E081_M64_DF, J3E081_M66_DF, J3E041_M66_DF,J 3E016_M66_DF, J3E041_M64_DF, J3E016_M64_DF
  - JCOP 2.4.2 R3, JavaCard 3.0.1 and GlobalPlatform 2.2.1
  - Secure MCU: P5CD016V1D/ P5CD021V1D/ P5CD041V1D/ P5CD081V1D (die marking: T046D)
  - CC certification report NSCIB-CC-13-37762-CR
  - Cryptolib V2.7/2.9 (CC certification report BSI-DSZ-CC-0864-2012)

1.3.3 *Rhea*

Most NXP JavaCard smartcards are available for purchase on the web thanks to various resellers for about 20€ per sample, we ordered cards from three families, namely `J3A081`, `J3D081` and `J2E081`. By observing their die markings, we could verify from the previous list that they respectively correspond the product families A, D and E.

We chose to start with product family D as its characteristics are the closest to `NXP A700X`'s. And decided to name it *Rhea*, as it is the name of the second largest moon of Saturn, right after *Titan*.

Open JavaCard products, like *Rhea*, are generic platforms for developers to load their own application (a JavaCard *applet*) on the smartcard. The JavaCard OS takes care of low level interactions with the hardware and offers high level APIs for the applets. Hence, an applet needs to comply with the JavaCard OS API independently of the underlying hardware.

On *Rhea*, the JavaCard OS happens to follow JavaCard 3.0.1 specifications [37], we hence developed and loaded a custom JavaCard applet allowing us to freely control the JavaCard ECDSA signature engine on *Rhea*. More precisely, we can now load chosen long term ECDSA secret keys, perform ECDSA signatures and ECDSA signature verifications.

Our JavaCard development was made easy thanks to the great job of Martin Paljak and other contributors of an open-source project for building JavaCard applets [27]. Moreover, for the use of JavaCard cryptographic API, our development was inspired by the open-source Wookey project from ANSSI [3] that, among many other things, implements an ECDSA signature/verification applet. Interestingly enough, they chose the same `J3D081` card for their tests.

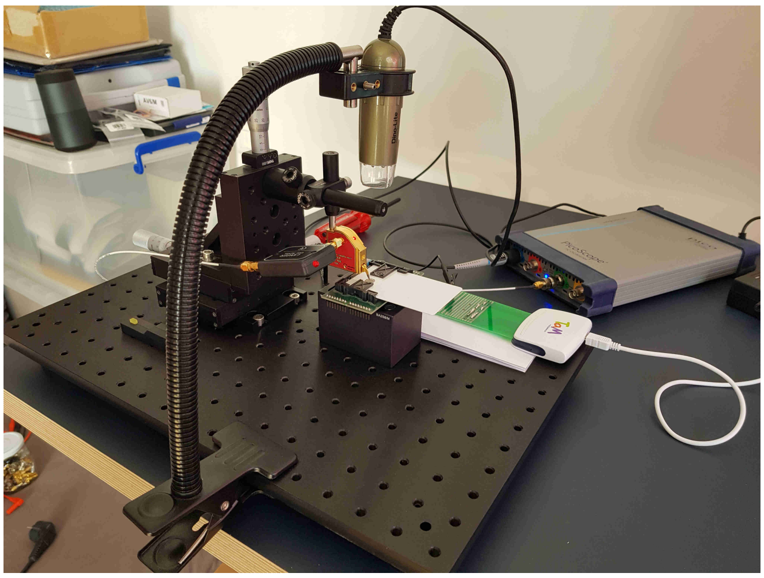
1.4 Side-Channel Observations

1.4.1 Side-Channel Setup

In order to perform EM side-channel measurements, we used the following side-channel analysis hardware setup (Figure 1.10 depicts the side-channel analysis platform while performing measurements on *Rhea*):

- Langer ICR HH 500-6 near-field EM probe with an horizontal coil of diameter 500μm and a frequency bandwidth from 2MHz to 6GHz [25];
- Thorlabs PT3/M 3 axes (X-Y-Z) manual micro-manipulator with a precision of 10μm [43];
- Pico Technology PicoScope 6404D oscilloscope, with a 500MHz frequency bandwidth, sampling rate up to 5GSa/s, 4 channels and a shared channel memory of 2G samples [39].

Figure 1.10: SCA Platform used for this study



For triggering the side-channel measurements, we proceeded as follows:

- for the side-channel measurements performed on *Rhea*, we used a modified commercial smartcard reader where we tap the I/O line, so we could trig on the sending of the APDU command;
- for the side-channel measurements performed on *Titan*, we used the triggering capabilities of our oscilloscope to trig on a pattern present at the beginning of the EM activity of the command processing the `authentication request message`.

Finally, note that the cost of this setup is about 10k€ (including the cost of the computer used for processing side-channel measurements).

1.4.2 First Side-Channel Observations on *Titan*

Figure 1.11 depicts the spatial position of the EM probe above the die of the *Google Titan Security Key* `NXP A7005a` die, whereas Figure 1.12 depicts the EM activity of the ECDSA signature performed during the processing the `authentication request message`.

Figure 1.11: *Titan* EM Probe Position

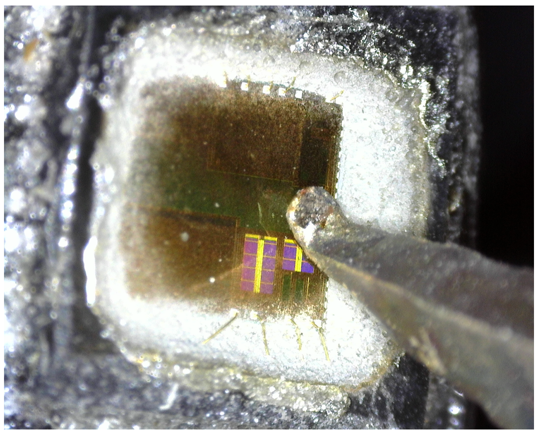
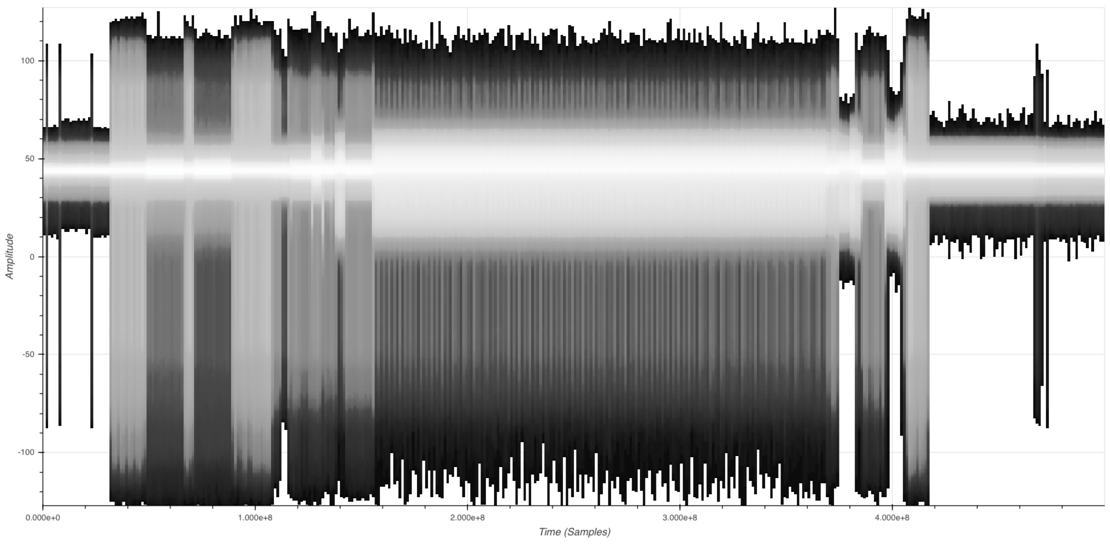


Figure 1.12: *Titan* EM Trace - ECDSA Signature (P256, SHA256)



As a side note, we should indicate that we use the `pyu2f` library[8] to send commands to *Titan*. Surprisingly, the library only implements the `authentication request message` such that user presence is checked (the user needs to touch the security key to validate his presence at each authentication request). This is a bit annoying for our task since we will need to observe several thousand of authentication requests. We then slightly modified the `pyu2f` library to remove the user presence check (see Section 1.2.1). A simple way to make the attacker life harder would be for the security keys to not support such requests, however this would make them not fully compliant with FIDO U2F protocol.

1.4.3 First Side-Channel Observations on *Rhea*

Figure 1.13 depicts the spatial position of the EM probe above the die of *Rhea*, whereas Figure 1.14 depicts the EM activity of the ECDSA signature performed during the processing of our APDU command launching the ECDSA signature available in the JavaCard cryptographic API of *Rhea*.

The similarity between the two EM activities of the ECDSA signature of *Titan* and *Rhea* confirms our hypothesis that both implementations are very similar.

[8]`https://github.com/google/pyu2f`

Figure 1.13: *Rhea* EM Probe Position

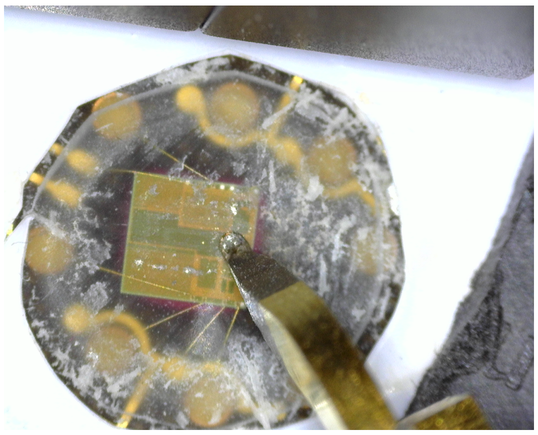
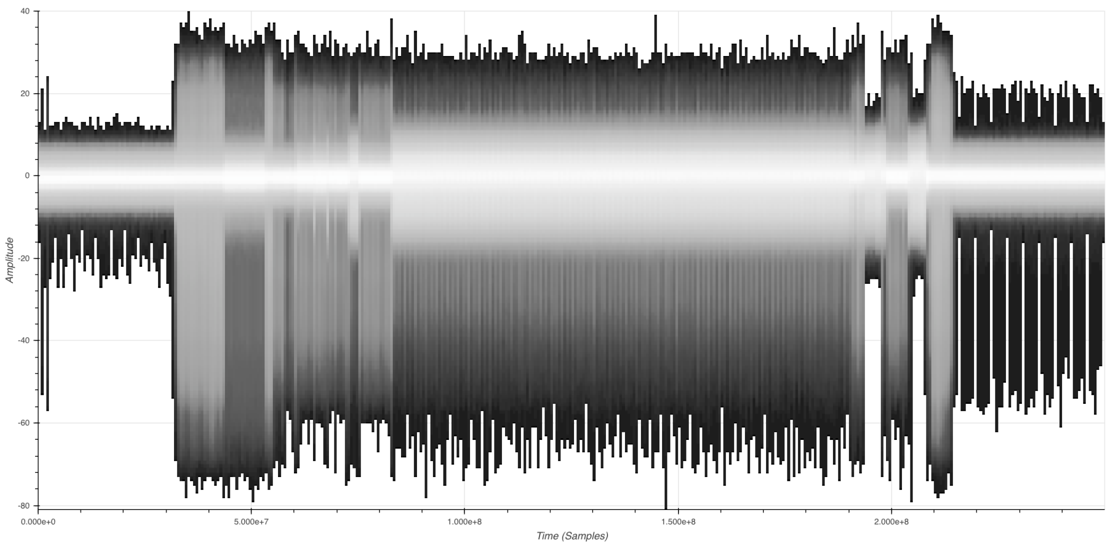


Figure 1.14: *Rhea* EM Trace - ECDSA Signature (P256, SHA256)



Chapter 2

Reverse-Engineering of the ECDSA Algorithm

2.1 ECDSA Signature Algorithm

We have seen in the previous chapter that the ECDSA signature operation looks very similar on *Titan* and *Rhea*. Furthermore we can fully control the inputs of the ECDSA signature and verification operations on *Rhea*, therefore we will first focus our efforts on *Rhea*.

2.1.1 Basics about the ECDSA Signature Algorithm

Let us briefly recall the ECDSA signature algorithm and introduce the notations we will use in this document:

- elliptic curve E over prime field $\mathbb{F}_p$, elliptic curve base point is $G_{(x,y)}$ of order q
- inputs: secret key d, hash of the input message to sign $h = H(m)$
- randomly generate a nonce k in $\mathbb{Z}/q\mathbb{Z}$
- scalar multiplication $Q_{(x,y)} = [k]G_{(x,y)}$
- denote by r the x-coordinate of Q: $r = Q_x$
- compute $s = k^{-1}(h + rd) \bmod q$
- output: (r,s)

First Remark: We can know k from the knowledge of (d, h, r, s):

$$k = s^{-1}(h + rd) \bmod q$$

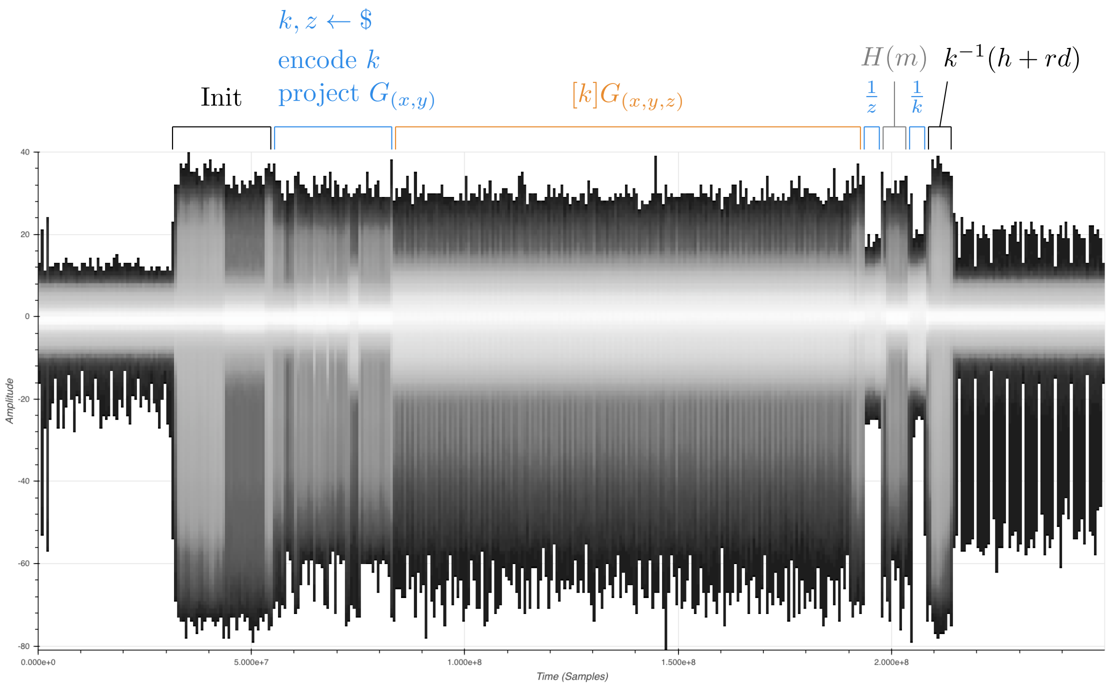
Second Remark: An usual countermeasure against side-channel analysis is to randomize the base point at each scalar multiplication (see [10]). So instead of computing directly the scalar multiplication $[k]G_{(x,y)}$ on the affine coordinates of G, one usually uses the projective coordinates of G:

- randomly generate a random z in $\mathbb{F}_p$
- send $G_{(x,y)}$ to the projective coordinates (xz, yz, z)
- compute $Q_{(x,y,z)} = [k]G_{(x,y,z)}$
- get the x affine coordinate of $Q_{(x,y,z)}$: $r = x/z \bmod p$

2.1.2 Matching the Algorithm to the Side-Channel Traces

Figure 2.1 presents a full EM trace of the ECDSA signature at sampling rate 2.5GSa/s. The whole execution time is approximatively 73ms. Our first goal here is to try to identify the different steps of the ECDSA algorithm on the trace.

Figure 2.1: *Rhea* EM Trace - ECDSA Signature (P256, SHA256)



After an initialization phase, where ECDSA inputs are processed and stored in the right places, the first step is to generate the randoms k (the nonce or ephemeral key) and z (the z coordinate of G in randomized projective coordinates). The call to a pseudo-random number generator (PRNG) is clear in the identified area, there are 48 calls to the PRNG to generate a 256-bit random and the PRNG re-initializes itself every 60 calls. There must also be at least two modular multiplications in this step to get G in projective coordinates given the random z. Also, the nonce k is *encoded*, meaning it is pre-processed to be used by the scalar multiplication algorithm, we will see how below.

Next comes the scalar multiplication itself, pretty easy to identify as this is the longest operation in ECDSA and its stable iterative process stands out clearly.

Finally, there are still two modular inversions to compute (k^{-1} mod q for the evaluation of the second part of the signature s and z^{-1} mod p to get the first part of the signature r), the hash of the input and the final computation of s with two modular multiplications and one addition. We propose to fill out as depicted on Figure 2.1 but we do not have strong arguments to show that these operations are actually performed in this order. It is worth mentioning that the overall process is pretty similar to what was observed in [31]. The authors were working on a P5 chip with an older version of the NXP cryptographic library.

2.1.3 Study of the Scalar Multiplication Algorithm

In side-channel analysis, there are many ways to attack an ECDSA implementation. In fact, any leakage inside one of the previously mentioned operations involving the nonce or the secret key would lead to an attack. In the literature, the most studied operation is the scalar multiplication, let us have a closer look.

The full scalar multiplication takes approximatively 43ms. This is an iterative process and every scalar multiplication contains exactly 128 iterations. Figure 2.2 displays the first 6 iterations of a trace (sampling rate is now set to 5GSa/s).

Figure 2.2: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - First Scalar Multiplication Iterations

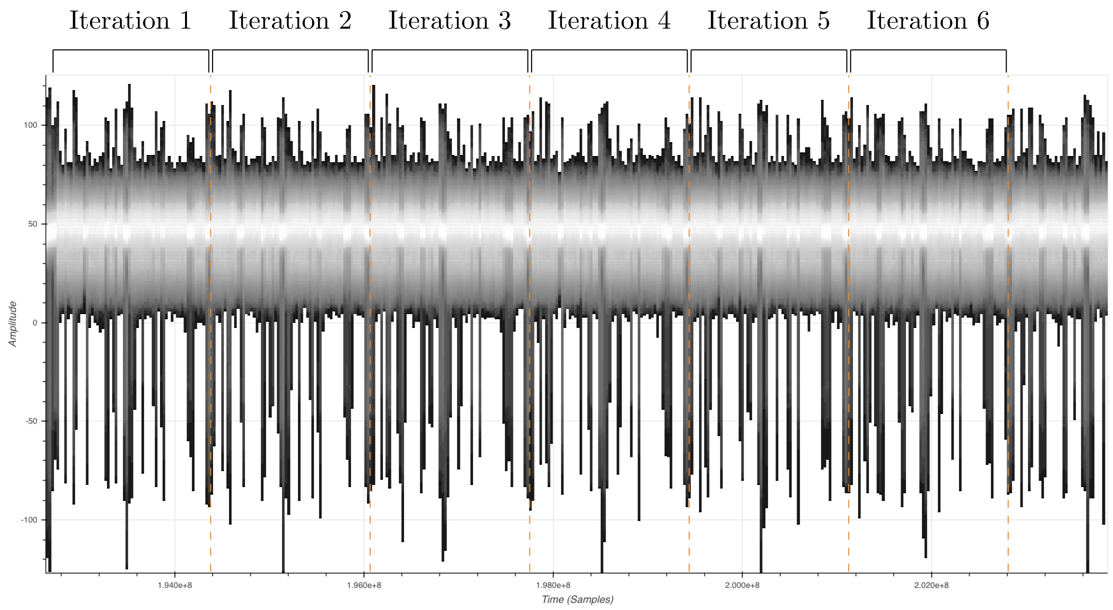
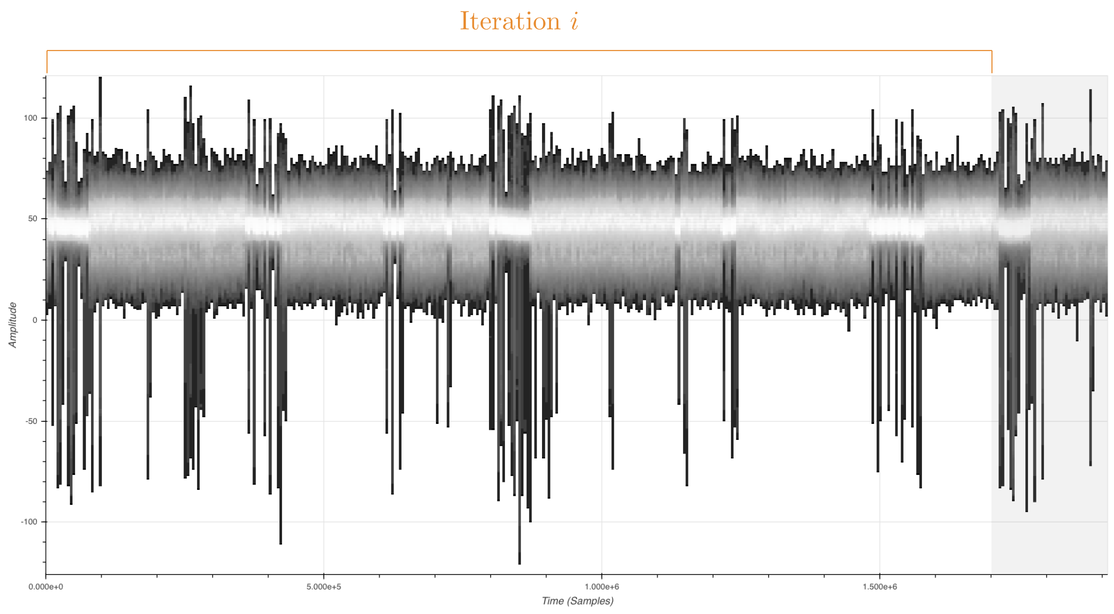


Figure 2.3 displays a single iteration. Some part of the iteration change from one iteration to the others and then the iteration length is not perfectly stable but it takes roughly 340us, which corresponds to about 1.7M samples (with 5GSa/s sampling rate).

Figure 2.3: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Scalar Multiplication Single Iteration



So the scalar multiplication algorithm does not seem to be a *binary* one, *i.e.* a scalar multiplication iteration is not related to a single bit of the nonce. This might then be a windowed algorithm with a window size of at least 2 bits, meaning that 2 bits of the nonce are used at each iteration (or more than that, for instance if we consider a blinded nonce, *e.g.* [10]). There are many different windowed algorithms for the scalar multiplication with many possible tweaks, which makes a lot of different directions to look at. Better understanding the iteration itself (*e.g.* identify the *Double* and *Add* operations) would help but there are so many ways to implement these things and we have no good starting point.

The good idea here was to look at the scalar multiplication in the ECDSA signature verification operation.

2.2 ECDSA Signature Verification Algorithm

As mentioned before, one great advantage of working on *Rhea* is the possibility to run the ECDSA signature verification algorithm (and not only the signature algorithm as on *Titan*). As we will see, the signature verification algorithm requires to compute similar operations than the signature algorithm, this might provide additional information on their implementation. Moreover, developers might downgrade countermeasures to improve the execution time. Indeed, the signature verification algorithm does not involve any secret and then side-channel or fault injection countermeasures seem useless speed reducers. For reverse engineering however, such a countermeasure downgrade is a windfall, it provides the opportunity to learn a lot on the implementation and its countermeasures.

2.2.1 Basics about the ECDSA Signature Verification Algorithm

Let us briefly recall the ECDSA signature verification algorithm:

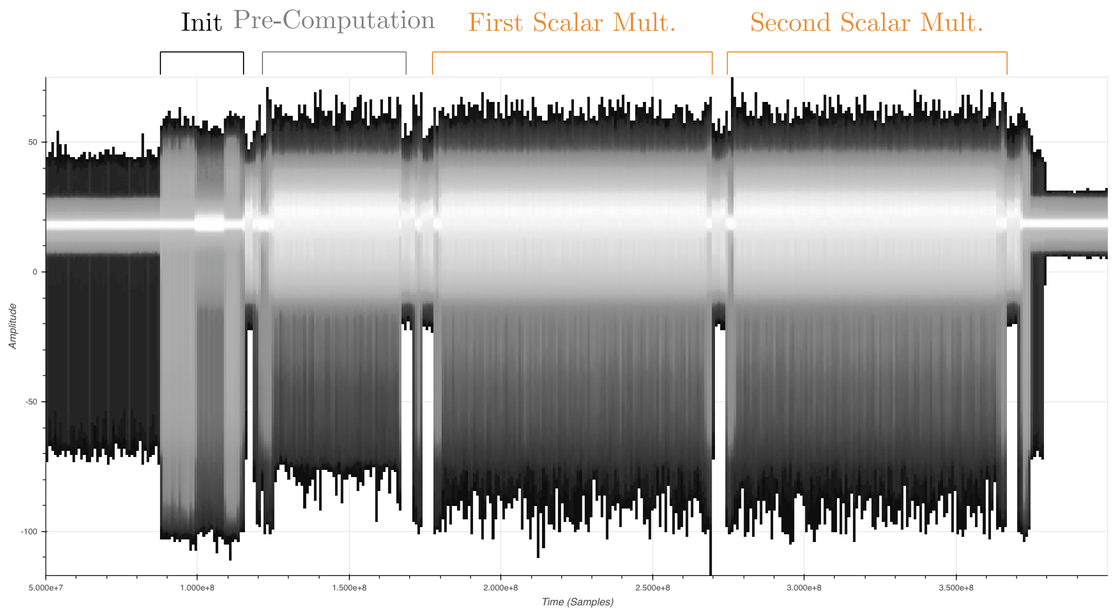
- elliptic curve E over prime field $\mathbb{F}_p$, elliptic curve base point is $G_{(x,y)}$ and order is q
- inputs: public key $P_{(x,y)}$, the hash of the signed input message $h = H(m)$
- inputs: the signature to be verified (r, s)
- first scalar $k^{(1)} = s^{-1}r \bmod q$
- second scalar $k^{(2)} = s^{-1}h \bmod q$
- first scalar multiplication $Q^{(1)}_{(x,y)} = [k^{(1)}]P_{(x,y)}$
- second scalar multiplication $Q^{(2)}_{(x,y)} = [k^{(2)}]G_{(x,y)}$
- compute $\bar{r} = Q^{(1)}_x + Q^{(2)}_x \bmod q$
- check that $\bar{r} = r$

Remark: we can know $k^{(1)}$ and $k^{(2)}$ from the public inputs

2.2.2 Matching the Algorithm to the Side-Channel Traces

Figure 2.4 shows a full signature verification EM trace (2.5GSa/s sampling rate) where we try to match the main operations. After a initialization phase very similar to the one in the signature trace, there is a large step we called *Pre-Computation* followed by the two expected scalar multiplications.

Figure 2.4: *Rhea* EM Trace - ECDSA Signature Verification (P256, SHA256)



2.2.3 Study of the Scalar Multiplication Algorithm

A closer inspection of the scalar multiplication trace shows that it slightly differs from the one in the signature algorithm. Indeed, as shown in Figure 2.5, there are two distinct patterns that can be identified (colored in orange and blue on the figure). Moreover, the concatenation of an orange and a blue pattern forms something very similar to a single iteration of the signature's scalar multiplication algorithm (Figure 2.6 shows few iterations of the signature's scalar multiplication algorithm where the orange and blue patterns are identified).

Figure 2.5: *Rhea* EM Trace - ECDSA Signature Verification (P256, SHA256) - Scalar Multiplication First Iterations

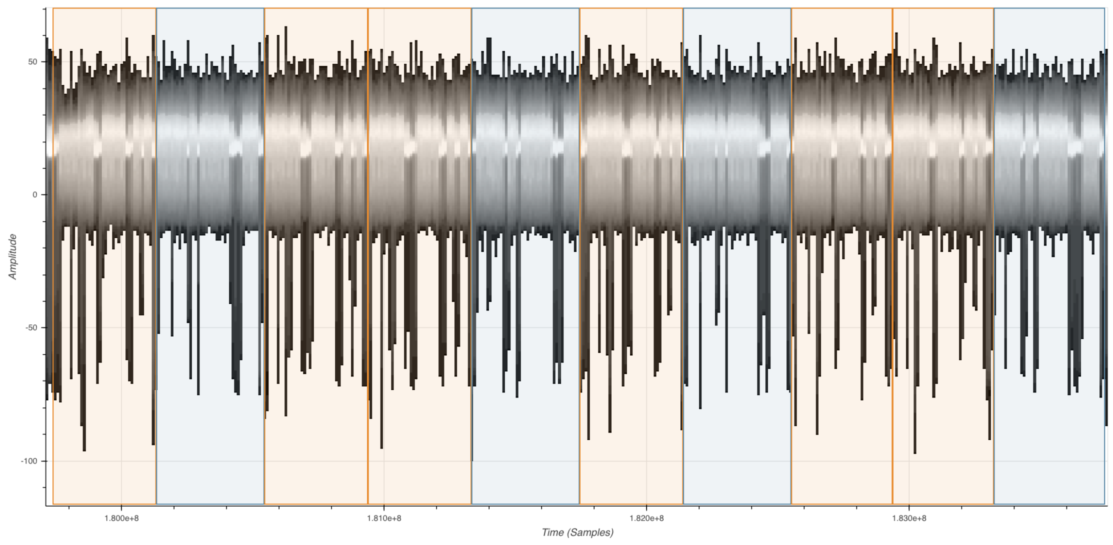
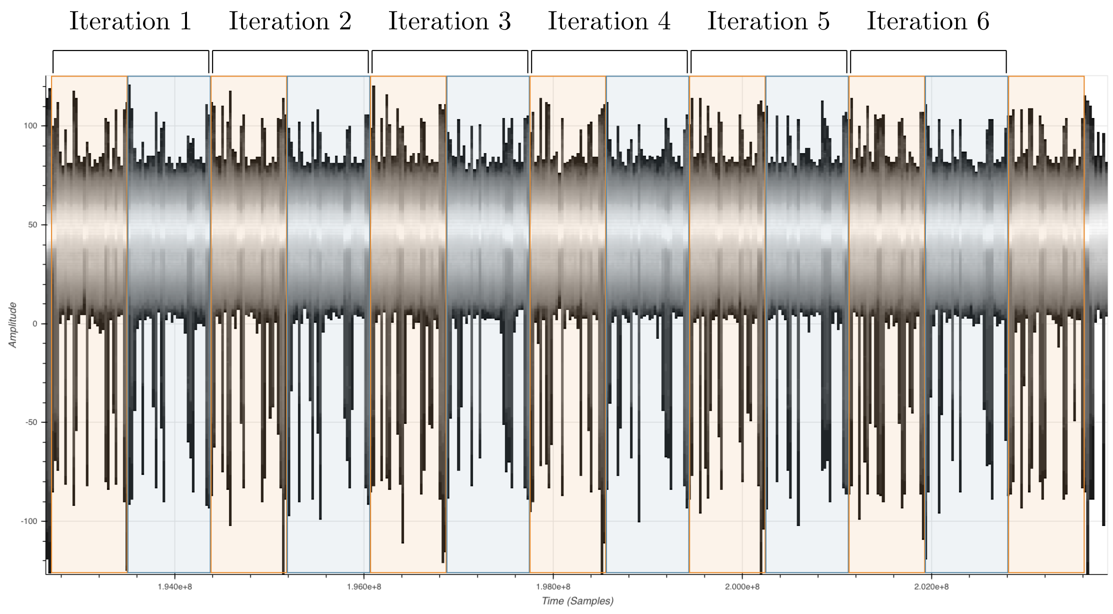


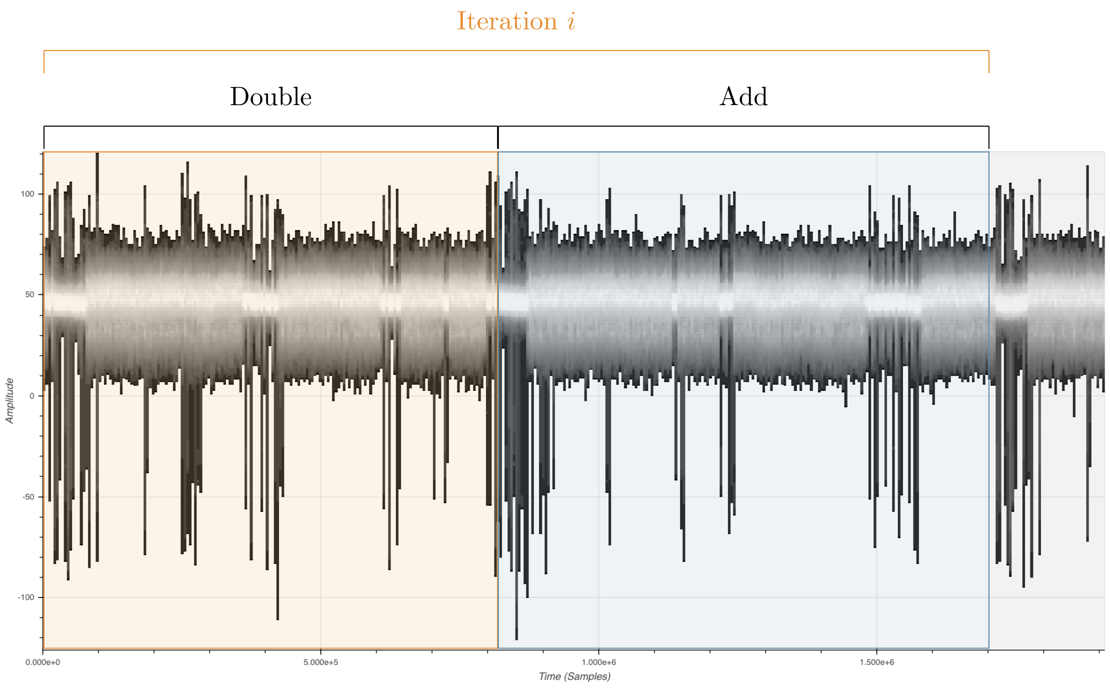
Figure 2.6: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Scalar Multiplication First Iterations



At this point, the implementation becomes much clearer: the scalar multiplication is implemented in a *Double&Add Always* fashion in the signature algorithm while it is a simple *Double&Add* in the signature verification algorithm. This observation gives critical information on the implementation:

- the *Double* and the *Add* operations are easily distinguishable, see Figure 2.7;
- the scalar multiplication is a left-to-right algorithm (meaning it starts with the most significant bit of the scalar down to the least significant bit);
- from the relative sizes of the *Double* and *Add* operations, it is clear that the *Double* is a *single* doubling operation. This might be surprizing since, from the observation that we have a kind of windowed implementation, we would expect at least two doubling operations by iteration (*i.e.* a 4×). This remark narrows down the possible scalar multiplication algorithms;
- the use of a scalar blinding countermeasure (see [10]) for the scalar multiplications of the signature verification algorithm is not very likely since we have seen that countermeasures are disabled for these operations (considered public). Then, the number of *Double* and *Add* operations shows that the windowed scalar multiplication implementation has a window size of 2, *i.e.* the scalar is manipulated 2-bit by 2-bit.

Figure 2.7: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Scalar Multiplication Single Iteration



Even if *Double* and *Add* operations seem very much alike in both scalar multiplications (from signature and signature verification), it must be noted that these operations are slightly smaller in the signature verification case, certainly because other countermeasures (apart from the *Double&Add Always*) are downgraded. More precisely, we observed that for the signature algorithm:

- *Double* operation takes about 159us;
- *Addition* operation takes about 179us;

while in the signature verification algorithm:

- *Double* operation takes about 155us;
- *Addition* operation takes about 166us.

2.2.4 Study of the Pre-Computation Algorithm

The pre-computation algorithm takes approximatively 17ms, it contains a varying number (but close to 128) of iterations of the same pattern. Figure 2.8 shows the full pre-computation trace while Figure 2.9 displays three iterations.

Figure 2.8: *Rhea* EM Trace - ECDSA Signature Verification (P256, SHA256) - Pre-Computation Algorithm

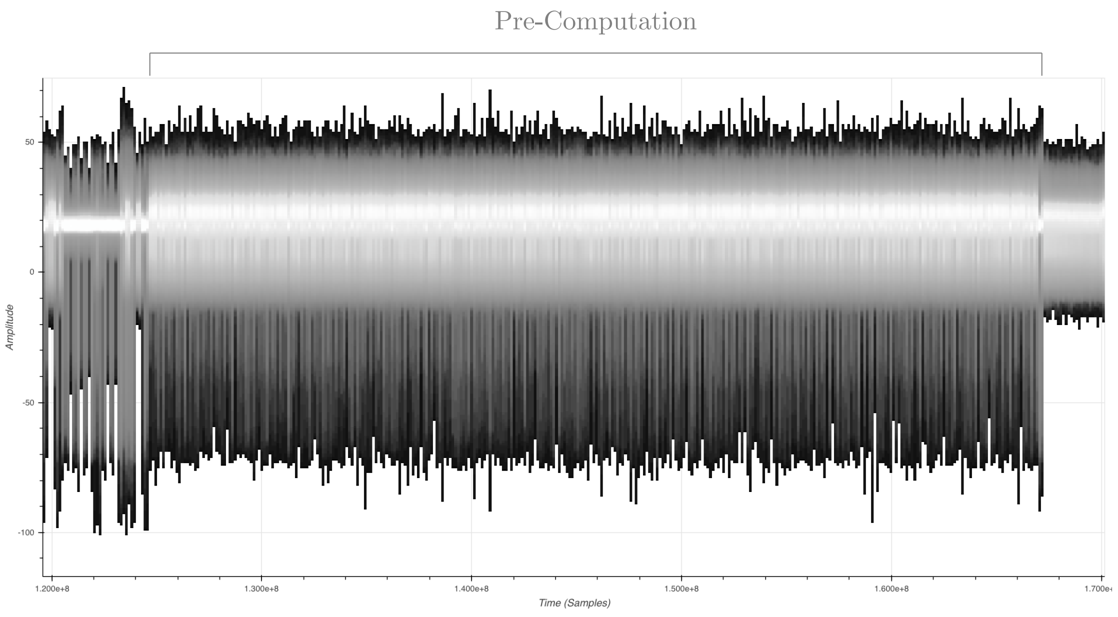
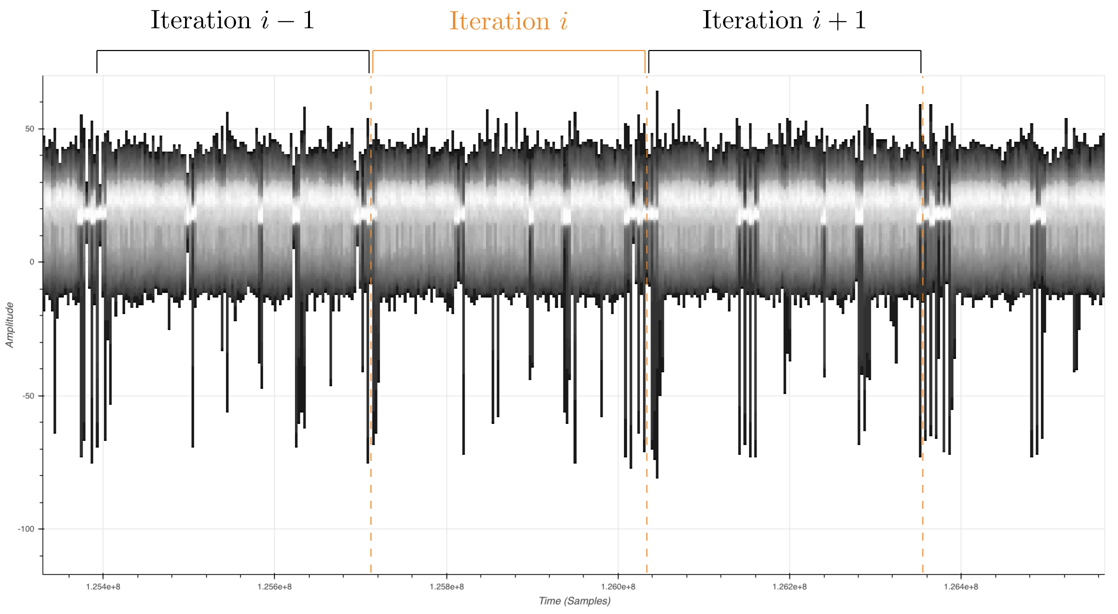


Figure 2.9: *Rhea* EM Trace - ECDSA Signature Verification (P256, SHA256) - Pre-Computation Algorithm First Iterations



The pre-computation algorithm looks very much like the scalar multiplication algorithm but with iterations (taking about 130us each) that are slightly smaller than the *Double* operations found inside the actual scalar multiplication.

Considering these iterations as efficient doubling operations in order to compute a scalar multiplication where the scalar is a power of 2, the pre-computation step is actually computing something close to $[2^{128}]P$.

2.3 High-Level NXP Scalar Multiplication Algorithm

There are many ways to implement a scalar multiplication algorithm, but the costly pre-computation observed in the previous section and the fact that there is a single doubling operation for each addition suggests the *comb* implementation (see [26]).

2.3.1 Pre-Computation and First Scalar Multiplication in Signature Verification Algorithm

Our best guess is then a *comb* method with window size 2. To compute $[k]P_1$ using this method, one has to do as follows[1]:

- First, let us consider the minimal binary form of $k = \{k_1, \cdots, k_{l_k}\}$ where l_k is even[2] with k_1 the most significant bit and k_{l_k} the least significant bit.

  We define the encoding of k as follows:

  $$Comb_2(k) : \{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{l_k/2}\} = \{k_1 k_{l_k/2+1}, \cdots, k_i k_{l_k/2+i}, \cdots, k_{l_k/2} k_{l_k}\},$$

  where $\tilde{k}_i$ is the 2-bit value created by the concatenation of k_i and $k_{l_k/2+i}$ (*i.e.* $\tilde{k}_i = k_i k_{l_k/2+i} = 2k_i + k_{l_k/2+i}$)

- The pre-computation phase is the computation of $P_2 = [2^{l_k/2}]P_1$ by $l_k/2$ doubling operations and $P_3 = P_1 + P_2 = [2^{l_k/2} + 1]P_1$.

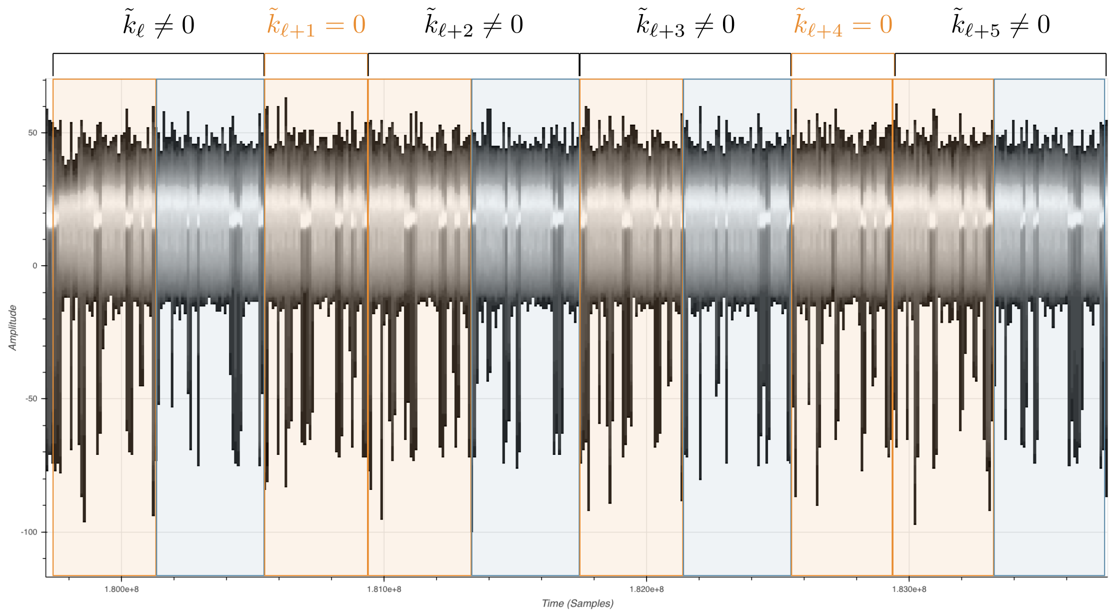
The *comb* method then processes as in Algorithm 1. From the side-channel traces, the number of iterations in the pre-computation scalar multiplication ($l_k/2$ doubling) and in the first scalar multiplication (assuming $k = (rs^{-1}) \bmod q$) match perfectly the algorithm. Moreover, the sequel of *Double* and *Add* in the scalar multiplication also matches perfectly the expected sequence from the value of the encoded scalar $\{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{l_k/2}\}$. Figure 2.10 shows what can be learned from the sequence of *Double* and *Add* operations about encoded digits.

[1]In the case of the first scalar multiplication in the signature verification algorithm, $k = (rs^{-1}) \bmod q$ and P_1 is the public key.

[2]*i.e.* If the minimal bit length of k is odd, then add a leading 0 bit to the binary form

```
Input  : $\{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{l_k/2}\}$: the encoded scalar
Input  : $P_1, P_2, P_3$: the pre-computed points
Output: $[k]P$: the scalar multiplication of scalar $k$ by point $P$
// Init Register $S$: point at infinity;
$S \leftarrow \mathcal{O}$;
// Find the first non null digit;
for $\ell \leftarrow 1$ to $l_k/2$ do
    if $\tilde{k}_\ell \neq 0$ then
        break;
    end
end
for $i \leftarrow \ell$ to $l_k/2$ do
    $S \leftarrow [2]S$;
    if $\tilde{k}_i > 0$ then
        $S \leftarrow S + P_{\tilde{k}_i}$;
    end
end
Return: $S$
```
Algorithm 1: Scalar Multiplication Algorithm used in Signature Verification Operation

Figure 2.10: *Rhea* EM Trace - ECDSA Signature Verification (P256, SHA256) - Scalar Multiplication First Encoded Digits



Another interesting observation can be made from Figure 2.10. Looking closely at the first *Double* operation, one can see a clear signal amplitude decline at the beginning of the operation that cannot be seen for other *Double* operations (the orange patterns). This amplitude drop is certainly due to the fact that the first *Double* operation is done on the infinity point of the elliptic curve, which has sparse coordinates $\mathcal{O}_{(x,y,z)} = (0, y, 0)$.

2.3.2 Second Scalar Multiplication in Signature Verification Algorithm

Given a scalar $k = (hs^{-1}) \bmod q$ and the curve base point G, compute $[k]G$.

The second scalar multiplication does not come with a pre-computation part (similarly to the signature scalar multiplication), this is because the input point of the scalar multiplication is the elliptic curve base point and therefore is always the same for all signature verifications (and for all signatures as well). Therefore, the pre-computation step can be done once for all (and this is actually why the *comb* method is a great choice for ECDSA signature: it is very efficient when pre-computation is done once for many scalar multiplications).

This creates a difference with the above algorithm (for the scalar multiplication over P_1): since the pre-computation is done once for all, it cannot be tuned for specific scalars k, therefore the computation of $G_2 = [2^{l_k/2}]G_1$ (with $G_1 = G$ the elliptic curve base point) must be done with l_k set to the maximum possible value, in our case 256.

Then, to compute $[k]G_1$, the considered binary form of $k = \{k_1, \cdots, k_{l_k}\}$ is constructed by adding enough leading 0 bits to match the desired length l_k. The rest of the algorithm goes unchanged.

When doing so, we found out that the sequence of iterations in the second scalar multiplication of the signature verification trace does not match the expected one. We found out that the bit length l_k was actually set to $l_k = 258$, *i.e.* at least two leading 0 bits are systematically added to the binary form of k. Doing this, we have a correct match between the $\tilde{k}$ sequence of digits and the sequence of *Double* and *Add* operations of the scalar multiplication.

2.3.3 Scalar Multiplication in Signature Algorithm

The signature's scalar multiplication algorithm is clearly the *Double&Add Always* version of the signature verification scalar multiplication algorithm. As mentioned before, the scalar multiplication contains exactly 128 consecutive *Double&Add* operations, making clear that the leading zero bits are not skipped anymore (contrary to the previous algorithms), hence avoiding leaking the nonce length. Moreover, we have seen in the previous section that the manipulation of the infinity point should be avoided as the side-channel signal could easily inform the attacker of such a manipulation.

Algorithm 2 combines these constraints and provides the best hypothesis we have so far of the scalar multiplication algorithm. In Algorithm 2, *Dummy* represents a register or memory address which will not be read and therefore stores useless computation results, G_0 is any point on the elliptic curve, $G_1 = G$ (the elliptic curve base point), $G_2 = [2^{129}]G_1$, $G_3 = G_1 + G_2$ and $G_4 = [2^{128}]G_1$.

Since G_0 is solely used for dummy computation, it could take any point on the curve, it could even change over time. Most likely G_0 takes its value in $\{G_1, G_2, G_3, G_4\}$, since these points coordinates are already computed.

```
Input  : $\{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{129}\}$: the encoded scalar
Input  : $G_0, G_1, G_2, G_3, G_4$: the pre-computed points
Output: $[k]G$: the scalar multiplication of scalar $k$ by point $G$
// Init Register $S$ to the point $G$ $(= G_1)$;
$S \leftarrow G_1$;
for $i \leftarrow 2$ to $129$ do
    $S \leftarrow [2]S$;
    if $\tilde{k}_i > 0$ then
        $S \leftarrow S + G_{\tilde{k}_i}$;
    else
        $Dummy \leftarrow S + G_0$;
    end
end
if $\tilde{k}_1 = 0$ then
    $S \leftarrow S - G_4$;
else
    $Dummy \leftarrow S - G_4$;
end
Return: $S$
```

Algorithm 2: Scalar Multiplication Algorithm used in Signature Operation

We now have a good explanation of the two extra leading zero bits added for the encoding of k. Thanks to them, $\tilde{k}_1$ can only take the values 0 or 1. In the former case, the initialization of S should be the infinity point. To avoid this, $\tilde{k}_1$ if forced to value 1, it is corrected by the last operations in Algorithm 2 assuming the G_4 point is also stored during the pre-computation step (in addition to G_2 and G_3). This process is confirmed by the presence of an *Add* operation following the scalar multiplication sequence of *Double&Add* iterations.

Chapter 3

A Side-Channel Vulnerability

In the previous chapter, we could identify the high-level implementation of the scalar multiplication algorithm used in *Rhea*'s ECDSA signature operation. With this knowledge, and the fact that, for each signature, we can know the nonce k (see Section 2.1.1 for the details) we will try to correlate the side-channel traces to the values of the encoded nonce digits.

3.1 Searching for Sensitive Leakages

The first step of the statistical side-channel analysis starts by the acquisition of the EM radiations of the *Rhea* chip during the ECDSA execution. Even though many configurations (choice of EM probe, EM probe position, sampling rate, amplitude ranges, etc.) have been tested during our work, only the final one is presented here as we believe the rest does not present real interest. For similar reasons, the research of a sensitive leakage has been a tedious work with many failed attempts and disillusions but only the output of this work is presented here.

Details of the acquisition setup are provided in Table 3.1 while the probe position is depicted on Figure 1.13.

Table 3.1: Acquisition parameters on *Rhea*

operation	ECDSA signature
equipment	PicoScope 6404D, Langer ICR HH 500-06
inputs	Messages are random, Key is constant (randomly chosen)
number of operations	4000
length	100ms
sampling rate	5GSa/s
samples per trace	500MSamples
channel(s)	EM activity
channel(s) parameters	DC 50ohms, ±50mV
file size	2TB
acquisition time	about 4 hours

With this acquisition campaign, what we need to do first is to identify each and every *Double&Add* operation inside every ECDSA scalar multiplication EM trace. After this step, we get

4000×128 sub-traces (since there are 4000 ECDSA executions and exactly 128 *Double&Add* operations in each scalar multiplication).

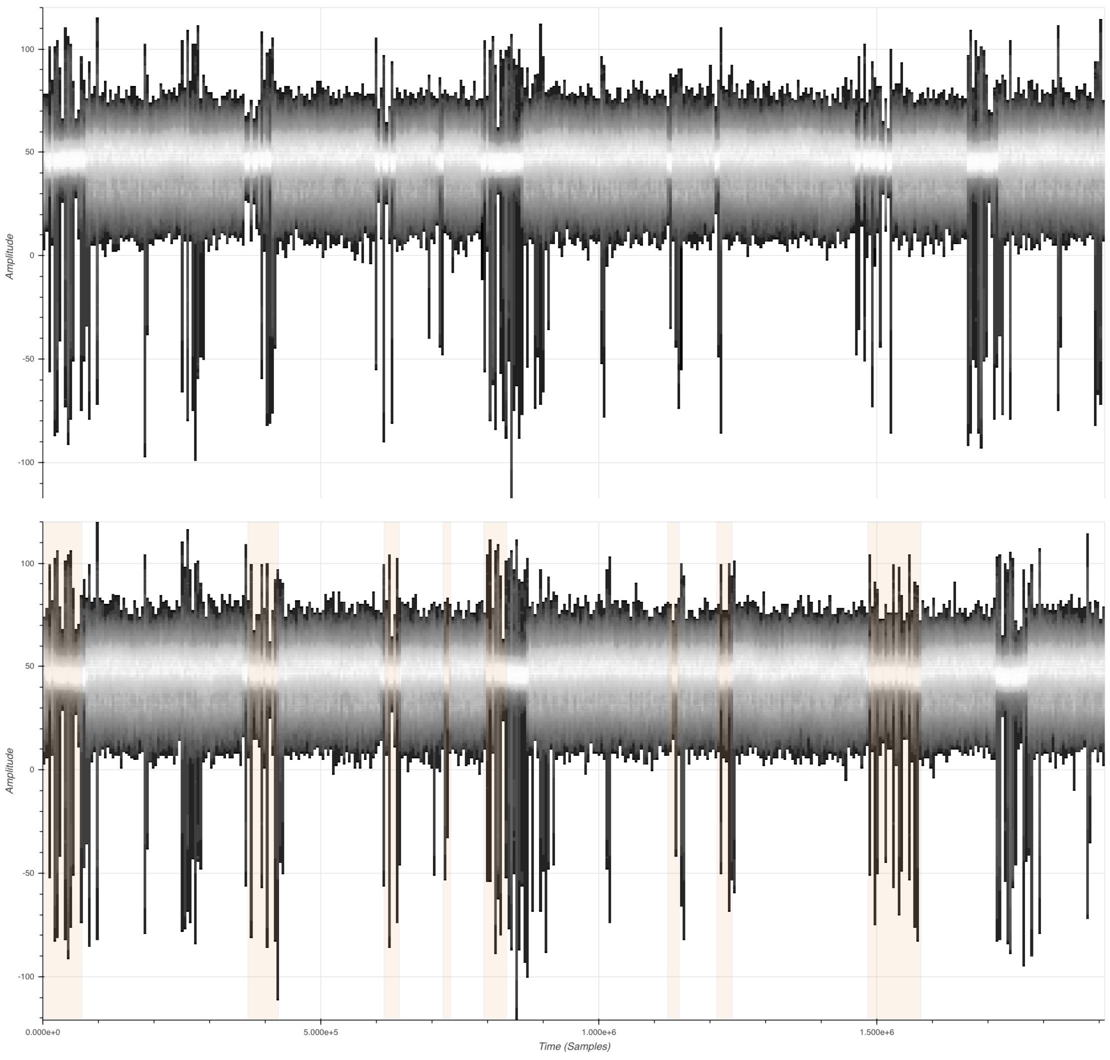
However, these sub-traces are not *aligned*, meaning that for two different sub-traces, the execution time is not perfectly synchronized. There are three main reasons for that:

- the *Rhea* internal clock is not perfectly stable (for this kind of chips, a natural clock jitter is usual, we can also expect an artificial clock jitter as a countermeasure against side-channel and fault injection attacks where, for instance, the clock slightly changes its frequency during the computation);
- the exact starting point of each iteration is not very clear, so we might not have the exact same starting point for each iteration;
- random delays are inserted during the computation (this is also a classical side-channel and fault injection countermeasure).

Figure 3.1 illustrates this with two different iterations of a scalar multiplication. The *Double&Add* operations take approximatively 1.7M samples (*i.e.* about 340us). The orange rectangles identify eight areas where the execution length changes (seemingly randomly) from one iteration to the other. These areas seem to correspond to pauses in the computation, where the microcontroller's main CPU takes back the control from the arithmetic co-processor and do some stuff (*e.g.* reconfigure the co-processor for next operation, move some values in memory, etc.). Since these areas behave randomly, they might correspond to countermeasures against side-channel or fault injection attacks (like random delays). In any cases, they do create misalignment (additionally to the clock jitter).

The misalignment together with the length of the sub-traces makes a global re-alignment algorithm very hard to design. We went back and forth for a while, each time re-aligning a different portion of the sub-traces and trying to correlate the re-aligned traces with the known encoded scalar digits.

Figure 3.1: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Scalar Multiplication Iterations Misalignment



3.2 A Sensitive Leakage

Figure 3.2 identifies the area where a sensitive leakage was detected whereas Figure 3.3 shows four signal peaks that bear the sensitive leakage inside the area defined in Figure 3.2.

Figure 3.2: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Sensitive Leakage Area

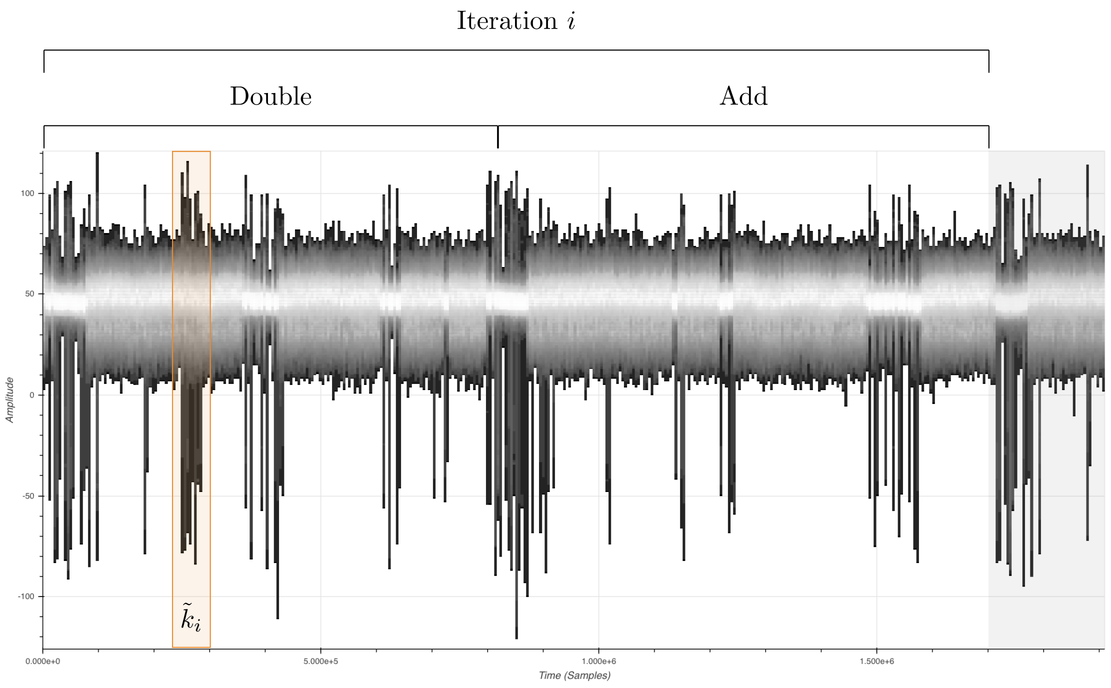


Figure 3.3: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - Sensitive Leakage

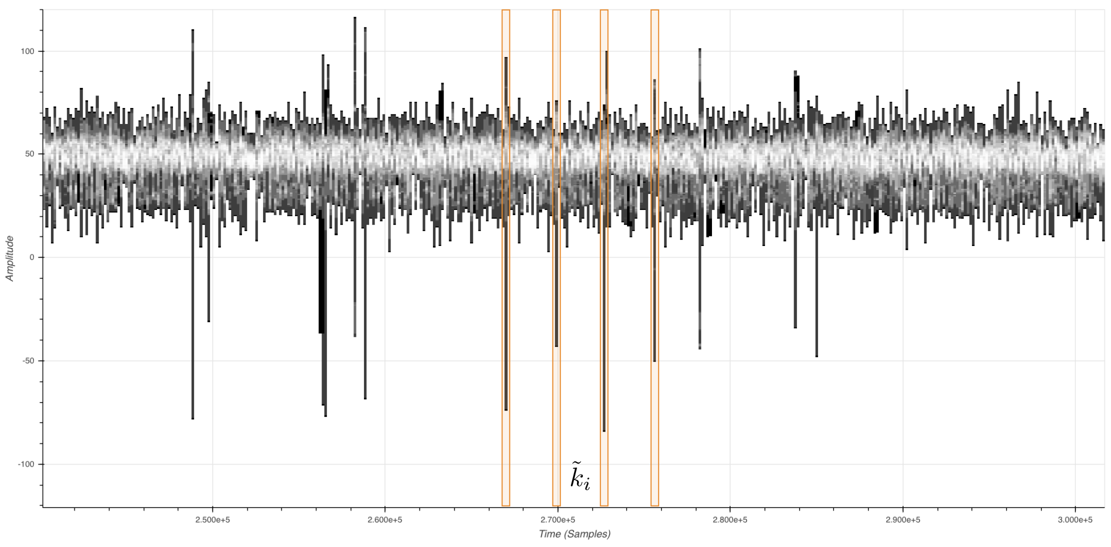


Figure 3.4 (first sub-figure) depicts 1000 superposed traces after re-alignment, only 400 samples are kept around each of the four identified signal peaks. To evaluate the statistical relation between the re-aligned traces and the encoded scalar digits, we compute the *Signal-To-Noise*

Ratio (SNR for short). More precisely, each of 4000×128 re-aligned sub-traces are classified with respect to its corresponding $\tilde{k}_i$ 2-bit digit. We then end up with four sets of sub-traces. For each set s and at each time sample t, we estimate the sub-traces mean $\mu_s(t)$ and variance $v_s(t)$. The SNR computed independently for each time sample t is then:

$$\texttt{SNR}(t) = \frac{Var(\mu_s(t))}{E(v_s(t))},$$

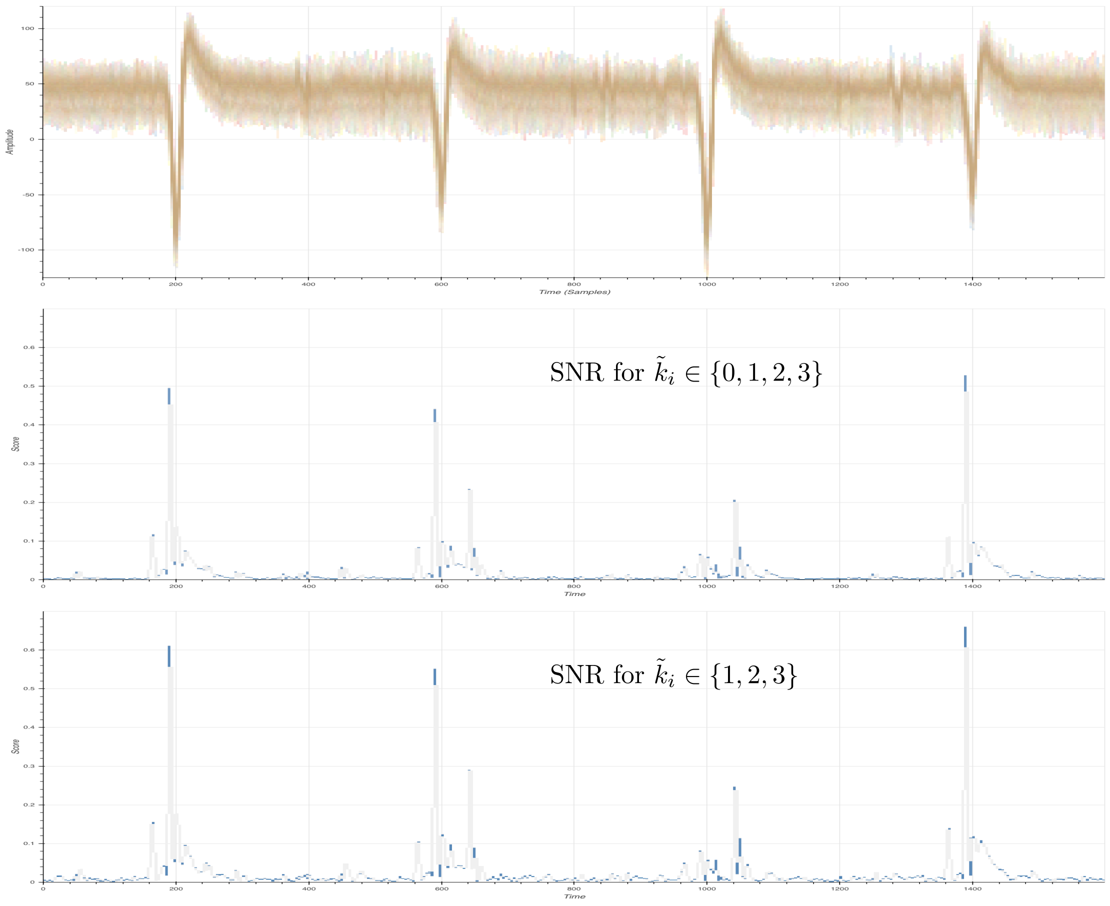
where $Var(\mu_s(t))$ is the estimated variance over the four estimated means and $E(v_s(t))$ is the estimated mean of the four estimated variances.

Figure 3.4 (second sub-figure) provides the SNR results for the four sets, best SNR value is about 0.53, clearly the amplitude of the side-channel traces are strongly related to the sensitive values $\tilde{k}_i$.

In our best guess on the scalar multiplication algorithm (Algorithm 2, Section 2.3.3) we have no way to know, when $\tilde{k}_i = 0$, what is the chosen dummy addition (*i.e.* what is the chosen point G_0 in Algorithm 2). In fact, G_0 could be chosen to be G_1, G_2, G_3, G_4 or any point on the elliptic curve (and G_0 could change at each iteration). We therefore also estimated the SNR without considering the cases where $\tilde{k}_i = 0$, *i.e.* the corresponding sub-traces are then just discarded from the SNR computation. Results are given in the third sub-figure of Figure 3.4. The SNR results gets to 0.65, significantly improving the previous SNR score. These results tend to show that G_0 takes varying values among G_1, G_2 and G_3, we will see in the next section what is actually going on here.

Using standard noise reduction techniques, based on filtering and principal component analysis, allowed us to further improve this SNR to 0.78.

Figure 3.4: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - SNR results (y-axis range $[0, 0.7]$)



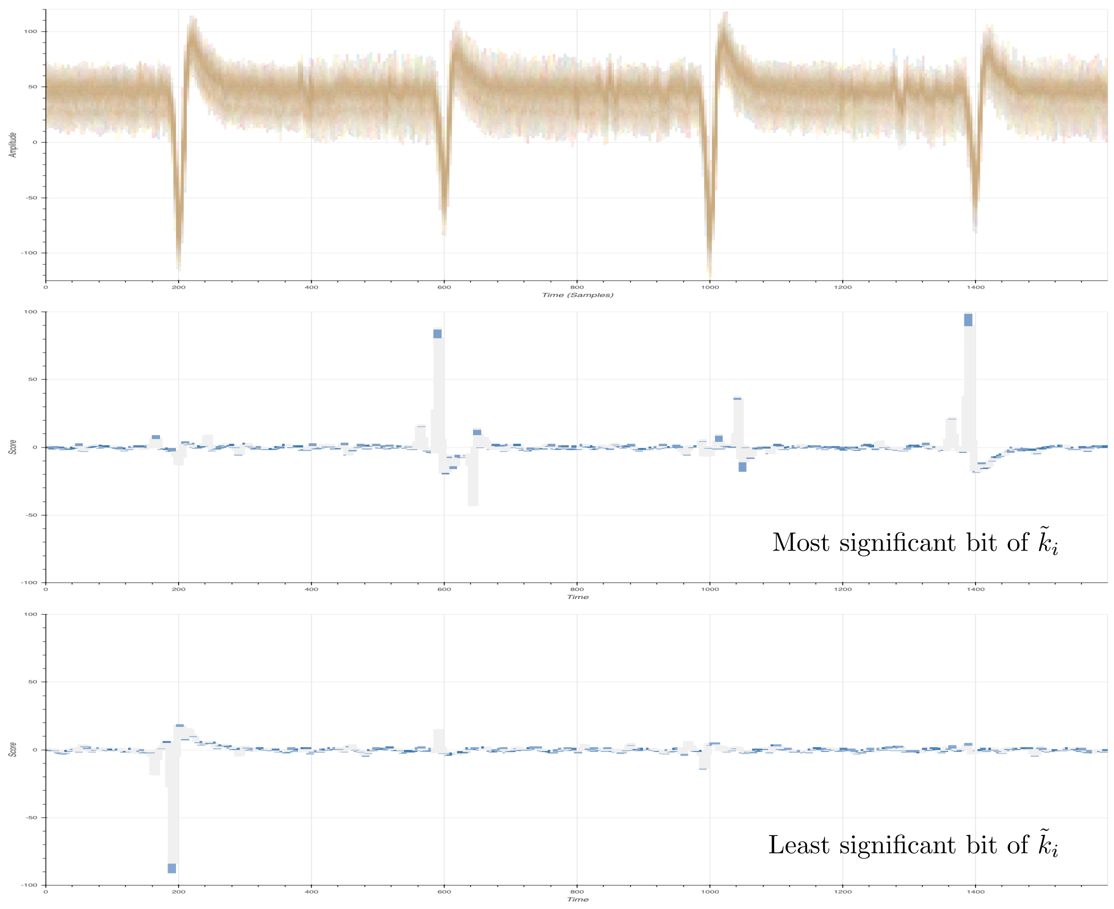
Let us go a bit further in the understanding of what is leaking. Considering only the subtraces where $\tilde{k}_i \neq 0$, we estimated the leakage strength with respect to the two bit values of $\tilde{k}_i$ individually.

To do so we use a binary test, the Welch T-Test [46]. Given two univariate data sources the T-Test will tell us if we can reject the *null hypothesis* with confidence, *i.e.* if these two sources are far enough from two independent sources.

1. $\tilde{k}_i = 1$ vs. $\tilde{k}_i = 3$ (*i.e.* test msb leaving lsb constant)
2. $\tilde{k}_i = 2$ vs. $\tilde{k}_i = 3$ (*i.e.* test lsb leaving msb constant)

A T-Test score is computed for each time sample independently, they are depicted in Figure 3.5. These scores clearly show that the two bits of $\tilde{k}_i$ do not leak at the same time. Furthermore, the most significant bit (msb) of $\tilde{k}_i$ shows a significant leakage on three of the four identified peaks whereas the least significant bit (lsb) of $\tilde{k}_i$ significant leakage is mainly located on a single peak.

Figure 3.5: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - T-Test results (y-axis range $[-100, 100]$)



3.3 Improving our Knowledge of the NXP's Scalar Multiplication Algorithm

In the previous section, we have removed the sub-traces related to the case $\tilde{k}_i = 0$ as they showed to deteriorate our SNR computation. Our hypothesis is that, when $\tilde{k}_i = 0$, since the corresponding addition (with G_0) has no effect on the scalar multiplication result (the addition output is sent to a dummy register), the developers might have decided to randomly choose (at each iteration) a point from the available pre-computed points (G_1, G_2, G_3) as value for G_0.

To try validate our hypothesis, we designed the following experiment based on supervised Expectation-Maximization clustering (to this end, we use the `GaussianMixture` class from `Scikit-learn` Python library [38]).

The idea is simple, we have many sub-traces with correct label $\tilde{k}_i$ (*i.e.* when $\tilde{k}_i \neq 0$), we will use them to train our clustering algorithm, *i.e.* define precisely the three clusters using maximum likelihood. And then match the un-labeled sub-traces (*i.e.* when $\tilde{k}_i = 0$): find for each un-labeled sub-trace the closest cluster, *i.e.* the value j such that $G_0 = G_j$ for this iteration. The Expectation-Maximization cultering is a multivariate process as it will use multi-dimensional data (*i.e.* our sub-traces with several time samples) and infer multivariate Gaussian distributions from them. To ease this work, we need to reduce the sub-traces to avoid adding useless time samples (*i.e.* time samples where the signal does not relate strongly to the sensitive variable $\tilde{k}_i$). The overall process is summarized below:

1. Reduce all sub-traces to the time samples where SNR is larger than a specific threshold (the best threshold choice is not something we know a priori, we applied the process for different threshold values until it gave consistent results).
2. With the sub-traces for which we know the corresponding encoded scalar digit (*i.e.* $\tilde{k}_i \neq 0$), estimate the three cluster centers, each cluster related to a value of $\tilde{k}_i$.
3. For each labeled sub-trace (corresponding to $\tilde{k}_i \neq 0$), find the closest cluster. This phase allowed us to control the success rate of the matching process.
4. For each un-labeled sub-trace (corresponding to $\tilde{k}_i = 0$), find the closest cluster.

The matching phase showed that about half of the un-labeled sub-traces matched the $\tilde{k}_i = 1$ case while the other half was divided equitably between $\tilde{k}_i = 2$ and $\tilde{k}_i = 3$.

This was validated by a new experiment: two sets of sub-traces are created. In the first one, we put the $\tilde{k}_i = 0$ iterations and in the other a mix of sub-traces with $\tilde{k}_i \neq 0$ where half of them correspond to $\tilde{k}_i = 1$ iterations and the rest is divided equitably between $\tilde{k}_i = 2$ and $\tilde{k}_i = 3$ iterations. The T-Test evaluation between these two sets could not reject the null hypothesis (best absolute T-Test value was less than 3), hence confirming the Expectation-Maximization experiment results.

With these experiments we have now improved our understanding of the scalar multiplication algorithm, Algorithm 3 gives the details. In Algorithm 3, $G_0 = G_1 = G$ (the elliptic curve base point), $G_2 = [2^{129}]G_1$, $G_3 = G_1 + G_2$ and $G_4 = [2^{128}]G_1$

Since $G_0 = G_1 = G$, one can check that the $Dummy \leftarrow S + G_{rand}$ addition is operated on G half the time and on G_2 or G_3 the rest of the time. We would like to emphasize that this algorithm is only our interpretation of the real algorithm implemented on *Rhea*, it might be different in many ways while doing roughly the same thing. Details of the real implementation are not our concern here, a high-level understanding of the countermeasures is good enough.

```
Input  : $\{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{129}\}$: The encoded scalar
Input  : $G_0, G_1, G_2, G_3, G_4$: The pre-computed points
Output: $[k]G$: The scalar multiplication of scalar $k$ by point $G$
// Init Register $S$ to the point $G(= G_1)$;
$S \leftarrow G_1$;
for $i \leftarrow 2$ to $l_k/2$ do
    $S \leftarrow [2]S$;
    $rand \leftarrow$ random element from $\{0, 1, 2, 3\}$;
    if $\tilde{k}_i > 0$ then
        $S \leftarrow S + G_{\tilde{k}_i}$;
    else
        $Dummy \leftarrow S + G_{rand}$;
    end
end
if $\tilde{k}_1 = 0$ then
    $S \leftarrow S - G_4$;
else
    $Dummy \leftarrow S - G_4$;
end
Return: $S$
```
Algorithm 3: Improved Version of Scalar Multiplication Algorithm used in Signature Operation

Chapter 4

A Key-Recovery Attack

We have showed a side-channel vulnerability in the ECDSA signature implementation of *Rhea*. This allowed us to better understand the scalar multiplication algorithm. Now, is this vulnerability exploitable in key-recovery attack ? In this chapter we will answer by the affirmative. But first, let us have a look at the options we have.

4.1 Directions to Exploit the Vulnerability

We will here refer to Algorithm 3, our reverse-engineering of the scalar multiplication algorithm of *Rhea* ECDSA signature operation.

4.1.1 A Closer Look at the Sensitive Information

We have seen that the value of the encoded scalar digits, denoted $\tilde{k}_i$ for the i^{th} iteration, leaks through the side-channel sub-trace related to the i^{th} iteration. In a less protected implementation, and assuming this leakage was noise free, one could recover the scalar, *i.e.* the ECDSA nonce. And from this nonce, recover the secret key, indeed for each ECDSA signature we have:

$$d = r^{-1}(ks - h) \bmod q$$

In the present case, we have seen in the previous chapter that the scalar value cannot be fully recovered from the leakage: when $\tilde{k}_i = 0$, it is set to $\tilde{k}_i = rand$ with $rand$ taking its values in $\{1, 2, 3\}$ with respective probabilities $\{.5, .25, .25\}$ and there is no leakage (to our knowledge) that could tell the attacker if $\tilde{k}_i$ was actually a 0 before its modification. Therefore, even in a noise free scenario, the attacker will recover an encoded scalar where no digit is null and that will not give the correct scalar when removing the encoding. The main issue is not that all the scalar bits are not recovered correctly, but the fact that the erroneous bits are indistinguishable from the correct ones and then there is no encoded scalar digit where one can know with high probability its value.

Let us consider an encoded nonce $\tilde{k}$, we recall that:

$$\tilde{k} = \{\tilde{k}_1, \cdots, \tilde{k}_i, \cdots, \tilde{k}_{129}\},$$

where each $\tilde{k}_i$ is the 2-bit digit built as the concatenation of the bits k_i and k_{129+i} (k_i lies in the upper half[1] of the 258-bit nonce k binary form and k_{129+i} in the lower half). In a noise free

[1] *upper half* means here in the 129 most significant bits

scenario, from the observed leakage of $\tilde{k}_i$, the attacker recovers the digit $\hat{k}_i$. Table 4.1 summarizes what can be deduced from the value $\hat{k}_i$.

Table 4.1: Information on Scalar Bits from Noise Free Sensitive Leakage

$\hat{k}_i$	Probability $\hat{k}_i$	Possible values for $\tilde{k}_i$	Probability for $\tilde{k}_i$	Binary value $(k_i k_{129+i})$
1	3/8	1	2/3	01
		0	1/3	00
2	5/16	2	4/5	10
		0	1/5	00
3	5/16	3	4/5	11
		0	1/5	00

In Table 4.1, the *Probability* $\hat{k}_i$ column gives the probability to observe the value $\hat{k}_i$: due to the non-uniform handling of $\tilde{k}_i = 0$, there is not an uniform distribution of the values $\hat{k}_i$. Moreover, two interesting cases appear in Table 4.1:

- When $\hat{k}_i = 1$ (which happens in 3/8 of the cases), the attacker can deduce that the non-encoded nonce bit k_i is equal to 0.
- When $\hat{k}_i = 2$ (which happens in 5/16 of the cases), the attacker can deduce that the non-encoded nonce bit k_{129+i} is equal to 0.

These are the only bit values that the attacker can know with certainty. All in all, in a noise free scenario, the attacker can expect to infer, in average, about 88 bits $(128 * (3/8 + 5/16)$ bits) of each nonce (*i.e.* one bit for each interesting case). These known bits being randomly scattered all over the nonce.

Finally, another issue (more classical in side-channel analysis) arises: the noise free scenario is not realistic. Even though we have found a strong leakage, it is still very noisy and matching a sub-trace to the corresponding $\hat{k}_i$ digit will be subject to errors.

4.1.2 Lattice-based ECDSA Attacks with Partial Knowledge of the Nonces

Since the seminal work of Howgrave-Graham and Smart [22], we know that the knowledge of few bits by nonce is enough to attack (EC)DSA schemes. This work was followed by many others that improved the understanding of this kind of attacks and/or successfully applied variants to practical settings (see *e.g.* [32, 33, 31, 6, 21, 18, 4, 11, 40, 23, 29, 2, 45, 28, 1]).

Roughly speaking (for a formal study and details, please refer to the literature), all these attacks work as follows:

1. Run N ECDSA signatures and, for each of them, record the input h, the signature output (r, s) and the leaked information $k^\star$ of the nonce k (let us denote by $k^{\bar\star}$ the unknown part of k: $k = k^{\bar\star} + k^\star$).
2. From the ECDSA equation $s = k^{-1}(h + rd) \bmod q$, one can build a linear equation over $\mathbb{Z}/q\mathbb{Z}$ where $k^{\bar\star}$ and d are unknowns: $Ak^{\bar\star} + Bd = C \bmod q$. Note that d is constant while $k^\star$ changes for each signature. In the following, we denote $k^{\bar\star(i)}$ the value $k^{\bar\star}$ for the i^{th} recorded signature.

3. Build a lattice where the vector (with potentially some extra leading and ending elements) $v = (k^{\bar{\star}(1)}, k^{\bar{\star}(2)}, \cdots, k^{\bar{\star}(N)})$ lies.

4. If the unknown part of each nonces $k^{\bar{\star}(i)}$ is *small* (*i.e.* the known part $k^{\star(i)}$ is *large* enough), then the vector v norm is small compared to the rest of the lattice vectors. One can then expect to find v (and then the nonce values) by solving the Shortest Vector Problem (SVP for short) in the lattice.

As shown in [22], this attack amounts to find a solution to the so-called *Hidden Number Problem* (HNP for short) introduced in [5]. In most of the studied cases by the literature, the known part of the nonces corresponds to its most significant bits (*i.e.* the attacker knows some leading bits of the nonces). But in a more general setting, sometimes referred to as the *Extended Hidden Number Problem* (EHNP for short), the known part of the nonces is a set of several blocks of consecutive known bits spread out over the nonce (seen as a vector of bits). In this case, the unknown $k^{\bar{\star}}$ is then a vector itself constituted of the unknown sections of the nonce. This more general setting did not draw so much attention (important papers are [32, 22, 21, 18]) but led to practical attacks nonetheless, mainly in the specific case a of windowed NAF implementation of the scalar multiplication ([11, 28]). We will see that EHNP fits well our context.

To solve (E)HNP, one needs to solve SVP in the lattice. To this end, one can use the LLL or BKZ algorithms (implementations can be found in Sage [42], we used Sage version 9.0)[2]. For these attacks to work in practice:

- the attacker should have enough known bits, let us have in mind a lower bound (approximative but simple): over all recorded signatures the number of known nonce bits must be larger than the bit length of the secret (*i.e.* the bit length of the elliptic curve order, 256 in our case)[3]. This means that the fewer known bits by nonce, the higher the number of signatures required for the attack to work (and then the higher is the lattice dimension);
- the number of known bits in a nonce should not be too small. For a 256-bit curve, practical HNP attacks were done with 4 known bits ([29, 23])[4]. In the EHNP setting, this means that known sections of the nonces must not be too small: a single known bit surrounded by two unknown bits does not help;
- the attack is not very tolerant to errors, *i.e.* the known part of the nonces should have very high probability to be correct.

4.1.3 How to Deal with Erroneous Known Bits

In Section 4.1.1, we have seen that only part of the nonces could be known from the attacker and we know that this knowledge is susceptible to errors due to side-channel noise. Previous work usually had to deal with the same issue (see *e.g.* [23, 18, 29, 11, 28]) and solve it with ad-hoc solutions depending on the context[5]. We can summarize the procedure as follows, pruning and brute-force:

[2]In a recent paper [1], the authors show that enumeration and sieve algorithms actually perform better than BKZ for this task. We were not aware of that during our work, using a sieve as in [1] would clearly improve our results.

[3]This is called the *information theoretic limit* in [32, 1]. In [1], the authors show that with a sieve algorithm this limit can actually be broken, *i.e.* attack can succeed with a number of known bits a little bit below the limit.

[4]The authors in [1] show that 3 bits are enough in practice with a sieve and 2 bits are theoretically reachable.

[5]In the recent paper [1], the authors show how this problem should be treated, avoiding the *folklore* of ad-hoc solutions. To our understanding, this corresponds to what is done in [28], very few errors could be tolerated (we would be in [28]'s worst case of "$1 \to 0$" errors). Our many attempts to reproduce this behaviour in our context

1. the pruning technique will select the best candidates and reduce the probability of error as low as possible. This step will usually select a small subset of the available signatures where the known part of the nonce is more likely to be correct. This is possible because most side-channel matching algorithms (*i.e.* the process that will infer the value $\hat{k}_i$ from the side-channel trace) will also provide a confidence level for the matching. Selecting the known parts of the nonce with confidence level above a given threshold will lower the probability of error. However, if the attack requires a certain number of signatures, say N, and the pruning techniques selects a signature with probability ε, then the attacker must acquire about N/ε signatures for the attack to work. In practice this technique is very effective to significantly lower the probability of error but is not practical to remove all errors: there usually exists some erroneous matching with very high level of confidence;
2. the brute-force process will finish the work on the selected candidates with very low error rate. The basic idea is to have slightly more selected candidates than necessary (say $N' > N$) and apply the attack on random subsets of candidates of size N. If the probability of choosing N error-free candidates among the N' candidates is high enough, the attacker will eventually find one by chance and succeed the attack.

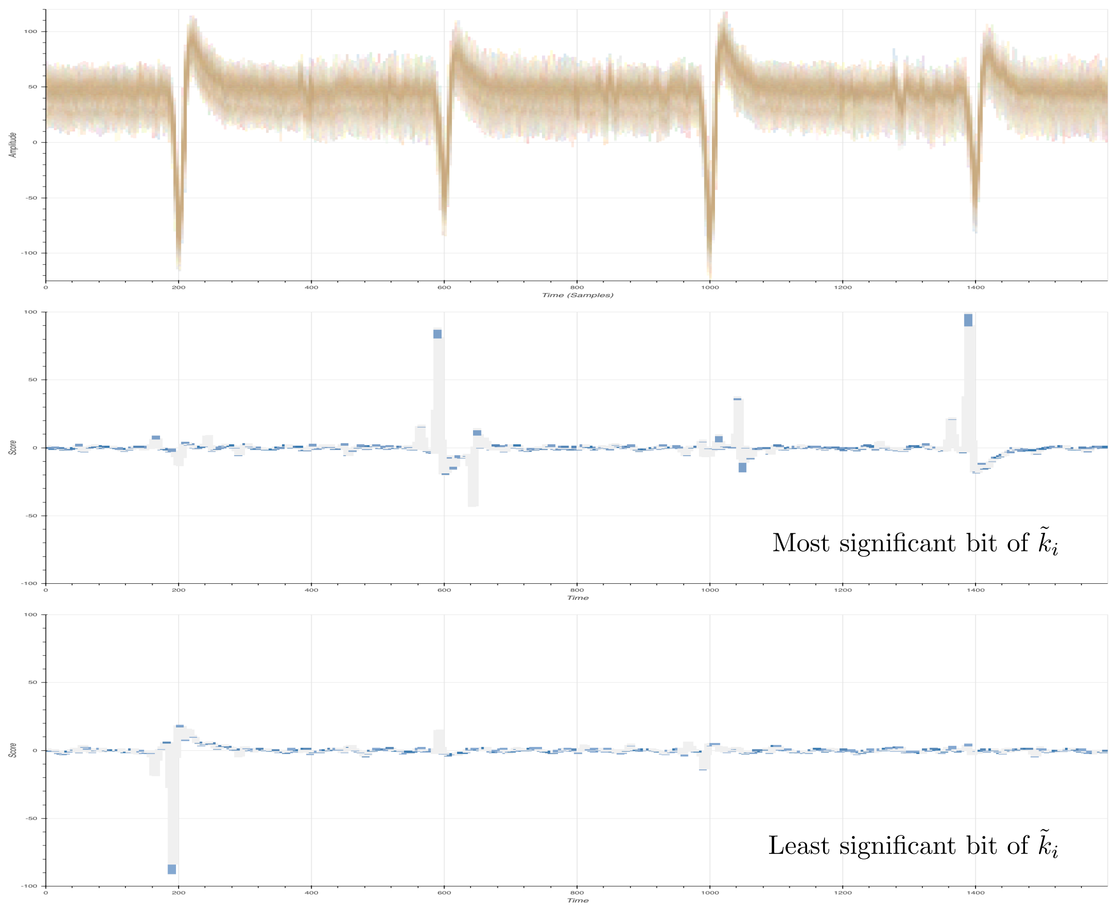
In the next section will see what are the matching success rates we have on *Rhea* leakage and how it can be improved with pruning. Then, we will describe our choices for the EHNP-based attack. Finally we will see how this attack worked on *Rhea*, and then how it could be applied on *Titan*.

4.2 Recovering Scalar Bits with Unsupervised Machine Learning

In the previous chapter, T-Test results – on carefully re-aligned sub-traces around four EM signal peaks – gave us very precise time samples where the encoded scalar digits are leaking. Figure 4.1 recalls the T-Test results: this computation is done after having removed all scalar multiplications with $\tilde{k}_i = 0$ (as we know that they lead to erroneous matching) and the two bits of $\tilde{k}_i$ are tested separately.

did not succeed and we believe this might be a question of computational power, which becomes prohibitive in our case.

Figure 4.1: *Rhea* EM Trace - ECDSA Signature (P256, SHA256) - T-Test results



We can easily see that the two bits are handled at different time samples, therefore there is no reason to work on the whole $\tilde{k}_i$ digit for the attack. Moreover, we have seen in Section 4.1.1 that for these two bits only the matching to 0 is useful: assuming that no error is due to side-channel noise, if the attacker infers that:

- the msb of the encoded scalar digit at iteration i is 0, then $k_i = 0$;
- the lsb of the encoded scalar digit at iteration i is 0, then $k_{129+i} = 0$.

Let us first consider the msb (we will denote it b in the following), the process will be similar for the other bit. Thanks to the T-Test analysis, it is easy to precisely select the time samples that are the most relevant regarding the handling of b: choose a threshold and select all time samples where the T-Test absolute value is above this threshold. The best threshold value is not known *a priori*, it is a parameter that will be found by brute-force. But first, let us choose an arbitrary value (say t). We now have all sub-traces reduced to only include time samples that are relevant *w.r.t.* b. As mentioned in Section 3.2, the reduced sub-traces are signal processed to slightly improve SNR.

From there, the most classical direction for the matching process is a supervised technique: using Templates [9] or DeepLearning (see *e.g.* [8]), build a matching reference from a training set. For us, this would mean to acquire a set of traces on *Rhea* (where we can know secret and nonces), re-align and reduce the corresponding sub-traces and create a template (or train a neural network) for $b = 0$ and $b = 1$ from these traces. Then, acquire a set of traces where the secret is not known (*e.g.* for us acquisitions on *Titan*), and for each re-aligned and reduced sub-traces, estimate its distance from the two templates (or the neural network output). This gives, for each sub-trace, the best candidate value for b as well as a level of confidence in the matching.

This approach is certainly the most effective in theory but it has a major drawback in practice: the side-channel signal must be identical in the training and testing set. This makes sense when the attacked target device can be also used for training, for instance if our final target was *Rhea*. In our case, even though the two chips (P5 and A7x) look very similar, they might have some design differences. Also, they do have different packages and we do not open them the same way (see Chapter 1), positioning the EM probe at the exact same location with the exact same orientation would make the attack hardly doable in practice. For all these reasons, we decided to try another direction – unsupervised clustering – and then increase our chances to transpose the attack on *Titan*.

In unsupervised clustering, there is no training data, we let the algorithm classify the traces in two categories in the hope that it will actually divide them between $b = 0$ and $b = 1$. Since we provide traces that are already re-aligned and reduced to the very time samples relevant *w.r.t.* b, this is not completely hazardous. We used the Expectation-Maximization algorithm (`Scikit-learn` GaussianMixture class[6]) to do the job.

In fact, even without knowing the secret, it is pretty easy to tell if the classification worked: from Table 4.1 we know that b takes value 0 in 3/8 of the cases and 1 in the other cases (this comes from the non-uniform handling of the $\tilde{k}_i = 0$ case). Therefore, if the unsupervised classification algorithm outputs two sets with the right respective sizes the attacker is likely to have found the correct parameters, moreover it is pretty obvious which set is corresponding to which value of b. This remark was a great help in attacking *Titan*.

Table 4.2 summarizes the matching success rates for $b = 0$ on the 4000×128 sub-traces of *Rhea* for various value of t (the threshold parameter). For a threshold t, the table gives the resulting sub-traces length after samples selection and signal processing, the probability of success when a sub-trace is labeled $b = 0$ and the overall number of sub-traces labeled $b = 0$ over the 4000×128 sub-traces. More precisely, the clustering algorithm will choose two cluster centers (*i.e.* two multivariate Gaussian distributions) and output, for each sub-trace, the probability of fitting each cluster. We will call *confidence level* the probability for a sub-trace to fit the cluster corresponding to $b = 0$. In Table 4.2, all sub-traces with confidence level over 0.5, *i.e.* sub-traces for which the multivariate Gaussian distribution associated with $b = 0$ seems a better fit, are labeled as $b = 0$.

[6]Exact parameters are `GaussianMixture(n_components=2, covariance_type='tied')`

Table 4.2: Confidence level 0.5

t	sub-trace length	success rate (%)	# sub-traces
9	767	49.7	217380
10	697	92.7	184364
11	650	92.7	184203
12	591	92.7	183864
13	554	92.3	184498
14	520	92.4	184405
15	484	92.3	184158

Table 4.3 (resp. Table 4.4) summarizes the matching success rates for $b = 0$ when only considering matching with confidence level over 0.95 (resp. 0.98). The highlighted row was the chosen setting for the attack, it provides high success rate while keeping the number of considered sub-traces high enough (which is not the case for confidence level 0.98).

Table 4.3: Confidence level 0.95

t	sub-trace length	success rate (%)	# sub-traces
10	697	99.0	110054
11	650	99.0	109714
12	591	99.0	108451
13	554	99.0	106990
14	520	99.1	106691
15	484	99.1	105911

Table 4.4: Confidence level 0.98

t	sub-trace length	success rate (%)	# sub-traces
10	697	99.5	82872
11	650	99.5	82456
12	591	99.6	81111
13	554	99.5	79157
14	520	99.6	78959
15	484	99.6	78289

We applied the same process to the lsb. However, since the leakage is not as strong as for the msb, the matching success rates were much lower: about 80% (without pruning, *i.e.* confidence level set to 0.5). We did not spend so much time on this since the attack was possible with the msb only. Maybe the leakage could be improved, but we believe that the gap in strength between the two leakages will not be overcome, there is simply a difference in the way the developers handle the two bits and one is manipulated a little bit more than the other. We hence drop the lsb information and focus on the msb (and when its value can safely be labeled as a 0).

4.3 Solving the Extended Hidden Number Problem

This report has no intention to get into the details of solving the Extended Hidden Number Problem, great literature exists and will provide all the information to build the lattice base and run the lattice reduction. We would like to emphasize that the heavy lifting of our lattice attack development was preformed by our intern, Camille Mutschler[7]. Together with her academic supervisor Dr. Laurent Imbert (LIRMM, CNRS), they did a tremendous work in the practical study of these kind of attacks, vastly outside the scope of this work on the *Google Titan Security Key*.

The building of the Lattice is the same than [22] (which was re-used in [18]), more precisely:

- we remove the secret key d from the equations, hence slightly reducing the lattice dimension;
- we use the embedding technique from [24], as it has been shown to be more efficient that way: we hence end up solving the SVP instead of a Closest Vector Problem (CVP);
- we apply to EHNP the modification presented in [32] and recalled in [29, 1]: recentering the unknown nonce parts around 0[8]. This showed to be a significant improvement in the attack success rate. The idea is very simple: in EHNP, each unknown nonce part is defined by an a priori integer interval in which it fits. By construction this interval is between 0 and a positive bound B. By recentering this interval between $-B/2$ and $B/2$, one reduces the lattice basis values and then makes the lattice reduction easier.

We use the LLL algorithm, although recent tests with BKZ confirmed better performances. And, as mentioned earlier, the use of a sieve (as in [1]) is expected to perform even better.

4.4 Touchdown on *Rhea*

From the pruning parameters chosen in previous section (see Table 4.3), we have extracted about 110K sub-traces (exactly 109714) that match a nonce bit value to 0 with high probability (99%). This makes, in average, 27.5 known 0 bit values in each of the 4000 256-bit nonces, all located in the upper half of the nonces. This might look a lot, however a vast majority of this information cannot be used. Indeed, [18]'s equation (26)[9] tells us that in the case of EHNP, any known block of less than three consecutive bits is not helping (actually it is worse than that, it is deteriorating the success rate by increasing the lattice dimension for no gain).

In fact, if we follow [18]'s equation (26) to accept a 3-bit long known block of bits (*i.e.* consecutive bits), there should be at least three such known blocks in a nonce. For 4-bit long known block of bits, there should be at least two and it is only starting from 5-bit long known block of bits that one can accept a nonce with a single known block. After few experiments we chose to look for a single block of 5 or more consecutive 0 bits in a nonce to select one. This process dramatically reduces the number of available nonces: from 4000 we end up with 180. Together, the 180 selected nonces gathers 948 known bits. For 5 of these 180 nonces, the known part was wrongly estimated (*i.e.* 5 bits among the 948 are erroneous).

[7]She is now PhD student at NinjaLab

[8]To our knowledge, this is the first time that this optimization is used for EHNP

[9]The advantage of this equation is its simplicity, it gives the number of bits that has to be known with respect to the nonces bit-length and the number of unknown sections in the nonces. However, this equation comes from approximations that are conservative and practical results show that fewer known bits are usually enough.

In simulation, with such a configuration, 80 error free signatures are enough to get about 50% chances to find the secret. It is worth mentioning that without the recentering optimization mentioned above, the success rate with these parameters would drop to 0%. Also, later experiments using BKZ with block size 25 improved this result to 60 error free signatures and 100% success rate.

We hence could run our brute-force process: among the 180 available nonces, we randomly choose 80 and test the attack until the secret key was found.
The attack on 80 signatures takes about 100 seconds to complete (on a 3,3GHz Intel Core i7, with 16GB RAM), the process was successfully completed after few tens of attempts.

4.5 Touchdown on *Titan*

To apply the attack on the *Google Titan Security Key*, we will need to go through all the steps, hoping for the sensitive leakage to be there. First, we place the EM probe at approximatively the same spatial position, with the same orientation (see Figure 1.11 in Introduction) and acquire 6000 side-channel traces of the U2F *authentication request* command. Details of the acquisition campaign are provided in Table 4.5.

Table 4.5: Acquisition parameters on *Titan*

operation	ECDSA Signature
equipment	PicoScope 6404D, Langer ICR HH 500-06
inputs	Messages are random, Key constant (unknown)
number of operations	6000
length	100ms
sampling rate	5GSa/s
samples per trace	500MSamples
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $\pm$50mV
file size	3TB
acquisition time	about 6 hours

Re-alignment, samples selection and signal processing We apply exactly the same process than for *Rhea* (the same four signal peaks are clearly visible). Once re-aligned around the four signal peaks, we use the *Rhea* T-Test results to select the time samples and apply exactly the same signal processing on the sub-traces.

By reusing the *Rhea* T-Test results for selecting the time samples, we here assume that *Rhea* and *Titan* share the same clock frequency and instructions order. These are not strong hypotheses since the clock frequency can be easily checked and the NXP cryptographic library version seems to be the same on both devices.

Unsupervised clustering We used the same algorithm (Expectation-Maximization) than for *Rhea*. As mentioned earlier, we can evaluate the correctness of the clustering with respect to the msb as the relative sizes of the two output clusters get closer to $(3/8, 5/8)$. Moreover, the smallest set corresponds to a msb equal to 0. With this predicate, we brute-force the T-Test threshold for time samples selection. The best threshold is $t = 8$ (whereas for *Rhea* is was $t = 11$), the sub-trace length after signal processing and samples selection with this threshold is 854.

Pruning and nonces selection We choose the highest confidence level that keeps enough nonces with 5 or more consecutive zeros. Since we have more traces than for *Rhea*, we can increase the confidence level to 0.98. After selection of nonces with 5 or more consecutive zeros we end up with 156 nonces.

Key recovery attack Our EHNP solver is run on random subsets of size 80 among the selected nonces. It took few tens of attempts before finding a correct subset and output the secret key.

A posteriori analysis From the secret key, we can compute the values of the nonces and verify that, among the 156 selected nonces, 7 were erroneous. The attack was then a little more challenging than for *Rhea* but still possible. Again, the use of BKZ with medium or large block size would do the work with much less nonces.

Chapter 5

Conclusions

5.1 Impact on *Google Titan Security Key*

The attack presented in this document allows to physically extract an ECDSA private key linked to an application secured by FIDO U2F from the *Google Titan Security Key*, whereas this private key should never leave the *Google Titan Security Key* in cleartext.

So the adversary could make a clone allowing her to sign in to the targeted application, assuming she previously had stolen the login and password of the victim's application account, without the victim noticing.

However, it requires that the adversary steals during several hours the device of the victim without the victim noticing, open or thin the IC package, access to expensive side-channel setup equipment (about 10k€) and custom attack softwares.

Note also that depending on how the counter use is implemented by the *relying party* to detect cloned U2F devices, it could limit the validity of the attack.

Thus it is still clearly far safer to use your *Google Titan Security Key* (or other impacted products) as FIDO U2F two-factor authentication token to sign in to applications like your Google account rather than not using one.

Nevertheless, this work shows that the *Google Titan Security Key* (or other impacted products) would not avoid unnoticed security breach by attackers willing to put enough effort into it. Users that face such a threat should probably switch to other FIDO U2F hardware security keys, where no vulnerability has yet been discovered.

5.2 List of Impacted Products

We performed the full attack (side-channel attack and lattice-based attack, leading to the recovery of the full long term ECDSA private key) on the following products:

- NXP J3D081_M59_DF and variants (product family D in the list of section 1.3.2, also called *Rhea* in this work);

- *Google Titan Security Key* (US version based on NXP A7005a).

We only validated that the EM activity is similar to the previous products, but did not performed the side-channel attack, on the following products:

- NXP J3A081 and variants (product family A in the list of section 1.3.2);
- NXP J2E081_M64 and variants (product family E in the list of section 1.3.2);
- *Google Titan Security Key* (EU version based on NXP A7005c).

Furthermore, our research suggests that other products are also impacted by our attack (we did not check via side-channel measurements, but for Feitian products we performed a partial teardown to get package markings):

- NXP J3D145_M59 and variants (product family B in list of section 1.3.2);
- NXP J3D081_M59 and variants (product family C in list of section 1.3.2);
- NXP J3E145_M64 and variants (product family F in list of section 1.3.2);
- NXP J3E081_M64_DF and variants (product family G in list of section 1.3.2);
- Yubico Yubikey Neo (based on NXP A7005c, see [20]);
- Feitian FIDO NFC USB-A / K9 (product very similar to the *Google Titan Security Key* in version with NFC and USB-A interfaces, nevertheless the package marking is `NXP 3E81G ZSD8161`, and we could not validate that it corresponds to the NXP A700X);
- Feitian MultiPass FIDO / K13 (based on NXP A7005c, product very similar to the *Google Titan Security Key* in version with micro-USB, NFC and BLE interfaces);
- Feitian ePass FIDO USB-C / K21 (based on NXP A7005c);
- Feitian FIDO NFC USB-C / K40 (based on NXP A7005c).

Finally, in our exchanges with the NXP Product Security Incident Response Team (PSIRT), they confirmed that all *"NXP ECC Crypto Library up to version v2.9 on P5 and A7x products"* are vulnerable to the attack.

5.3 Attack Mitigations

Several measures can be implemented to thwart the proposed attack, at different levels.

5.3.1 Hardening the NXP P5x Cryptographic Library

Straightforward ways for hardening the NXP P5x cryptographic library:

- blinding of the scalar. This does not remove the sensitive leakage but makes the attack much harder. For instance, by addition of a random factor of the curve order (the random's bit length should be at least half of the curve order's bit length);
- re-randomizing the table lookup of precomputed points in the comb implementation at each new access and hence completely remove the sensitive leakage.

5.3.2 Use the FIDO U2F Counter to Detect Clones

As explained in section 8.1 of [16], the counter *may* be used as a signal for detecting cloned U2F devices. Thus if a *relying party* of an application protected with FIDO U2F receives a cryptographically correct `authentication response message`, but with a counter value smaller or equal to the previous counter value recorded, it means that a clone of the U2F device has been created and used. Then the *relying party* should not validate the authentication request, and lock the account.

This countermeasure would reduce the usability of the clone to a unique time after giving the security key back to the legitimate user. Once the clone has been used (say one month after the attack), the account will be locked by the next access from the legitimate user.

5.4 Impact on Certification

We followed the rating rules of the Common Criteria, the international standard for rating hardware attacks. More precisely, we used the document *Application of Attack Potential to Smartcards and Similar Devices* [41]. Table 5.1 provides a rating of the full attack (side-channel attack and lattice-based attack).

Table 5.1: Common Criteria rating of the full attack

Factor	Comments	Identification	Exploitation
Elapsed Time	Identification phase took us more than 4 months. Exploitation phase would take less than one day.	6	3
Expertise	For Identification phase, a multiple expert is required. Only an expert is required for Exploitation phase.	7	4
Knowledge of the TOE	As the attack has been performed in *black-box*, no knowledge of the TOE is required for both phases.	0	0
Open Samples / Known Key	No Open Sample is required for both phases.	0	NA
Access to TOE	Less than ten samples were used for both phases.	0	0
Equipement	An electromagnetic SCA platform and custom softwares are required for both phases.	3	4
Sub Total		16	11
Total		27	

Our attack gets a rating of 27 points, meaning that the device has a resistance to attackers with *moderate* attack potential. It is worth mentionning that the NXP P5 chips and their cryptographic libraries are no more covered by CC certificates and therefore our work does not revoke any ongoing certification.

5.5 Project's Timeline

The following timeline summarizes the different phases of this work and the associated responsible disclosure:

- September 11th, 2018:
  - Elie Bursztein's keynote talk at CHES 2018 and obtention of first samples of the *Google Titan Security Key*;
- June 29th, 2020:
  - full attack validated on *Rhea*;
- July 2nd, 2020:
  - full attack validated on *Titan*;
- October 1st, 2020:
  - contact Google VRP, Feitian, NXP, ANSSI and BSI, with a short technical description of our work and our responsible disclosure plan (3 months);
  - acknowledgment of reception from NXP PSIRT team and ANSSI CERT-FR;
- October 2nd, 2020:
  - acknowledgment of reception from Feitian and BSI CERT-BUND;
  - bug accepted by the Google VRP team;
- October 21st, 2020:
  - Google VRP panel decided to acknowledge our contribution to Google Security in their Hall of Fame (honorable mentions), but no reward as they consider that it is a NXP issue;
- October to December 2020:
  - several technical exchanges with NXP and Google about our work;
- January 4th, 2021:
  - CVE ID assigned: CVE-2021-3011 [1];
- January 7th, 2021:
  - publication of this report on the NinjaLab website and on the IACR eprint archive.

Thanks

We would like to thank:

- Ryad Benadjila for his technical support about the development of JavaCard applets;
- Camille Mutschler and Dr. Laurent Imbert for working with us in developing a library implementing lattice-based attacks.

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3011

Bibliography

[1] M. R. Albrecht and N. Heninger. On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem. Cryptology ePrint Archive, Report 2020/1540, 2020. https://eprint.iacr.org/2020/1540.

[2] A. C. Aldaya, C. P. Garcia, and B. B. Brumley. From A to Z: Projective Coordinates Leakage in the Wild. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):428–453, Jun. 2020.

[3] R. Benadjila, M. Renard, P. Trebuchet, P. Thierry, A. Michelizza, and J. Lefaure. Wookey Project Github repository. https://wookey-project.github.io/javacard/index.html. [online; accessed 31-December-2020].

[4] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014.

[5] D. Boneh and R. Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 129–142, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[6] B. B. Brumley and N. Tuveri. Remote Timing Attacks Are Still Practical. In V. Atluri and C. Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[7] I. Buhan and P. Schwabe. Conference on Cryptographic Hardware and Embedded Systems 2018. https://ches.iacr.org/2018/index.shtml, 2018. [online; accessed 31-December-2020].

[8] M. Carbone, V. Conin, M.-A. Cornélie, F. Dassance, G. Dufresne, C. Dumas, E. Prouff, and A. Venelli. Deep Learning to Evaluate Secure RSA Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):132–161, Feb. 2019.

[9] S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[10] J. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[11] S. Fan, W. Wang, and Q. Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1505–1515, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Feitian. Feitian FIDO solutions. https://www.ftsafe.com/Products/FIDO/. [online; accessed 31-December-2020].

[13] Feitian. Feitian website. https://www.ftsafe.com. [online; accessed 31-December-2020].

[14] FIDO Alliance. FIDO U2F Implementation Considerations. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-implementation-considerations-v1.2-ps-20170411.pdf. [online; accessed 31-December-2020].

[15] FIDO Alliance. FIDO U2F Raw Message Formats. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html. [online; accessed 31-December-2020].

[16] FIDO Alliance. Universal 2nd Factor (U2F) Overview. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf. [online; accessed 31-December-2020].

[17] Google. Google Titan Key. https://cloud.google.com/titan-security-key/. [online; accessed 31-December-2020].

[18] D. Goudarzi, M. Rivain, and D. Vergnaud. Lattice Attacks Against Elliptic-Curve Signatures with Blinded Scalar Multiplication. In R. Avanzi and H. M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 120–139. Springer, 2016.

[19] HexView. Google Titan Key teardown. http://www.hexview.com/~scl/titan/. [online; accessed 31-December-2020].

[20] HexView. Yubikey Neo teardown. http://hexview.com/~scl/neo/. [online; accessed 31-December-2020].

[21] M. Hlavác and T. Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *Lecture Notes in Computer Science*, pages 114–133. Springer, 2006.

[22] N. Howgrave-Graham and N. P. Smart. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptogr.*, 23(3):283–290, 2001.

[23] J. Jancar, V. Sedlacek, P. Svenda, and M. Sys. Minerva: The Curse of ECDSA Nonces (Systematic Analysis of Lattice Attacks on Noisy Leakage of Bit-Length of ECDSA Nonces). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):281–308, 2020.

[24] R. Kannan. Minkowski's Convex Body Theorem and Integer Programming. *Mathematics of Operations Research*, 12(3):415–440, 1987.

[25] Langer. ICR HH 500-6. https://www.langer-emv.de/en/product/near-field-microprobes-icr-hh-h-field/26/icr-hh500-6-near-field-microprobe-2-mhz-to-6-ghz/108, 2019. [online; accessed 31-December-2020].

[26] C. H. Lim and P. J. Lee. More Flexible Exponentiation with Precomputation. In Y. G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 95–107, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[27] Martin Paljak. Ant JavaCard Project Github repository. https://github.com/martinpaljak/ant-javacard. [online; accessed 31-December-2020].

[28] G. D. Micheli, R. Piau, and C. Pierrot. A Tale of Three Signatures: Practical Attack of ECDSA with wNAF. In A. Nitaj and A. M. Youssef, editors, *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*, volume 12174 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2020.

[29] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[30] MOUSER. NXP A700x datasheet, secure authentication microcontroller. https://www.mouser.fr/datasheet/2/302/a700x_fam_sds-1187735.pdf. [online; accessed 31-December-2020].

[31] E. D. Mulder, M. Hutter, M. E. Marson, and P. Pearson. Using Bleichenbacher's Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-bit ECDSA: extended version. *J. Cryptogr. Eng.*, 4(1):33–45, 2014.

[32] N. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptol.*, 15(3):151–176, 2002.

[33] P. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.

[34] NIST. FIPS 186-2, Digital Signature Standard (DSS). https://csrc.nist.gov/csrc/media/publications/fips/186/2/archive/2000-01-27/documents/fips186-2.pdf, 2001. [online; accessed 31-December-2020].

[35] NXP. NXP LPC11U2x datasheet, 32-bit ARM Cortex-M0 microcontroller. https://www.nxp.com/docs/en/data-sheet/LPC11U2X.pdf. [online; accessed 31-December-2020].

[36] NXP. NXP SmartMX family brochure. https://www.nxp.com/docs/en/brochure/75017515.pdf. [online; accessed 31-December-2020].

[37] Oracle. JavaCard Connected Platform Specifications 3.0.1. https://www.oracle.com/java/technologies/javacard/platform-specification-3-0-1.html. [online; accessed 31-December-2020].

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[39] Pico Technology. PicoScope 6000 Series datasheet. https://www.picotech.com/download/datasheets/PicoScope6000CDSeriesDataSheet.pdf, 2019. [online; accessed 31-December-2020].

[40] K. Ryan. Return of the Hidden Number Problem.: A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):146–168, Nov. 2018.

[41] SOGIS. Application of Attack Potential to Smartcards and Similar Devices v3.0. https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v3-0.pdf. [online; accessed 31-December-2020].

[42] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020. https://www.sagemath.org.

[43] Thorlabs. Manual 3-axes Stage PT3/M. https://www.thorlabs.com/thorproduct.cfm?partnumber=PT3/M#ad-image-0, 2019. [online; accessed 31-December-2020].

[44] University of Montpellier. CTM website (technological center of the university of Montpellier). http://www.iesengineering.fr/centrale-de-technologie-de-montpellier. [online; accessed 31-December-2020].

[45] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer. Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1767–1784. USENIX Association, Aug. 2020.

[46] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947.