EUCLEAK

Side-Channel Attack on the YubiKey 5 Series

(Revealing and Breaking Infineon ECDSA Implementation on the Way)

Thomas Roche

NinjaLab, Montpellier, France
thomas@ninjalab.io

Revision History

Revision	**Date**	**Author(s)**	**Description**
1.0	03.09.24	T. Roche	Initial Version
1.1	22.10.24	T. Roche	- Add revision history table - Minor typo corrections - Correct bandpass filter spec. - Correct the Infineon SC300 40/65nm family - Add/Discuss new certificates - Update Project Timeline

ABSTRACT

Secure elements are small microcontrollers whose main purpose is to generate/store secrets and then execute cryptographic operations. They undergo the highest level of security evaluations that exists (Common Criteria) and are often considered inviolable, even in the worst-case attack scenarios. Hence, complex secure systems build their security upon them.

FIDO hardware tokens are strong authentication factors to sign in to applications (any web service supporting FIDO); they often embed a secure element and the FIDO protocol uses *Elliptic Curve Digital Signature Algorithm* (ECDSA for short) as its core cryptographic primitive. `YubiKey 5 Series` are certainly the most widespread FIDO hardware tokens, their secure element is an Infineon `SLE78`.

This document shows how – finding a JavaCard open platform (the `Feitian A22`) based on a similar Infineon `SLE78` – we understood the Infineon ECDSA implementation, found a side-channel vulnerability and designed a practical side-channel attack. The attack is then demonstrated on a `YubiKey 5Ci`. Finally, we show that the vulnerability extends to the more recent Infineon `Optiga Trust M` and Infineon `Optiga TPM` security microcontrollers.

Our work unearths a side-channel vulnerability in the cryptographic library of *Infineon Technologies*, one of the biggest secure element manufacturers. This vulnerability – that went unnoticed for 14 years and about 80 highest-level Common Criteria certification evaluations – is due to a non constant-time modular inversion.

The attack requires physical access to the secure element (few local electromagnetic side-channel acquisitions, *i.e.* few minutes, are enough) **in order to extract the ECDSA secret key**. In the case of the FIDO protocol, this allows to create a clone of the FIDO device.

All `YubiKey 5 Series` (with firmware version below 5.7) are impacted by the attack and in fact all Infineon security microcontrollers (including TPMs) that run the Infineon cryptographic library (as far as we know, any existing version) are vulnerable to the attack. These security microcontrollers are present in a vast variety of secure systems – often relying on ECDSA – like electronic passports and crypto-currency hardware wallets but also smart cars or homes. However, we did not check (yet) that the EUCLEAK attack applies to any of these products.

Cautionary Note: Authentication tokens (like FIDO hardware devices) primary goal is to fight the scourge of phishing attacks. The EUCLEAK attack requires physical access to the device, expensive equipment, custom software and technical skills. **Thus, as far as the work presented here goes, it is still safer to use your `YubiKey` or other impacted products as FIDO hardware authentication token to sign in to applications rather than not using one**.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Context

Secure elements are the *root of trust* of secure systems. These little pieces of hardware should be able to generate, store and manipulate secrets while ensuring their full confidentiality and integrity. And, more importantly, they must be able to do so even when falling inside the adversary hands. To fulfill this requirement, the secure elements follow the *simpler-is-safer* rule: keep the hardware simple, keep the software simple, keep the communication protocols simple. By keeping the functionalities to a bare minimum (generate and store cryptographic keys and certificates, sign documents, verify signatures and wrap secret keys), secure elements manufacturers must fight the endless urge of making things more powerful, more complex, more everything.

By keeping things simple, the attack surface is small and easier to audit. The most stringent certification scheme of security devices is under the umbrella of *Common Criteria* (CC for short) and was preliminary designed for smart cards evaluation [1]. This evaluation, organized around certification bodies (public institutions from the signatory countries of the SOGIS [2]), is executed by an independent and accredited ITSEF laboratory. These evaluations are tedious and extremely time consuming, especially when aiming at the highest security level: `AVA_VAN_5` [3]. Hence, secure elements manufacturers must face the marketing challenge of selling what might seem outdated hardware technology [4] along with epsilonesque functionality updates but for a important time-to-market production due to certification delays.

Integrated in a public key infrastructure, secure elements allow to safely – with unparalleled security guarantees – authenticate a genuine product (*e.g.* for access control) or validate that a piece of software was not modified (*e.g.* for boot purpose). Secure elements are then everywhere, from the Trusted Platform Modules (TPM for short) that can be found managing the secure boot of most computers (personal or server), to the banking or ID cards that can be found in everybody's pocket. They are also omnipresent in high-end smartphones (notably for payment purpose), in cars and more generally in any IoT device that requires (or advertises) high level of security.

[1] https://www.commoncriteriaportal.org/files/ppfiles/ssvgpp01.pdf

[2] https://www.sogis.eu/

[3] The `Vulnerability Assessment Number`, it goes from 1 to 5. The CC *EAL* level aggregates, among other things, the `AVA_VAN_X` level

[4] Most common technology node for smart cards is 40nm nowadays.

In the last decade secure elements made their way inside two new kind of products: cryptocurrency hardware wallets and FIDO hardware tokens. Both of these products claim high level of security while being easily in the hand of an adversary: the perfect use case for secure elements. As secure elements penetration testers, NinjaLab is obviously interested in these products and more particularly into FIDO hardware tokens. Indeed, while undergoing the most thorough security evaluations from highly skilled ITSEF laboratories, some vulnerabilities have been found in the past by independent security researchers. The ROCA attack [22] and the TPM-Fail attack [21] are certainly the most important ones in the recent years.

In 2021 NinjaLab published *A Side Journey to Titan* [19, 27], reporting a side-channel vulnerability in the P5x family of security microcontrollers. The P5x are old secure elements from NXP [5] manufacturer that still could be found in some products (they are slowly replaced by the P6 and P7 families from the same manufacturer), and notably in the Google FIDO hardware token: the *Google Titan Security Key* [6]. Note that the Google Titan Security Key was recently updated to support *passkey* [7]: passwordless FIDO authentication. In [19], NinjaLab reported a side-channel vulnerability in the *Elliptic Curve Digital Signature Algorithm* (ECDSA for short) implementation of NXP that could be exploited with a physical access to the device [8] for about 10 hours (and an offline phase of about 6 hours, that can be done when the device is already given back to the legitimate user).

In the present work, NinjaLab unveils a new side-channel vulnerability in the ECDSA implementation of Infineon [9] on any security microcontroller family of the manufacturer.This vulnerability lies in the ECDSA ephemeral key (or nonce) modular inversion, and, more precisely, in the Infineon implementation of the *Extended Euclidean Algorithm* (`EEA` for short). To our knowledge, this is the first time an implementation of the `EEA` is shown to be vulnerable to side-channel analysis (contrarily to the `EEA` binary version, *e.g.* [9, 2, 1]). The exploitation of this vulnerability is demonstrated through realistic experiments and we show that an adversary only needs to have access to the device for few minutes. The offline phase took us about 24 hours; with more engineering work in the attack development, it would take less than one hour.

After a long phase of understanding Infineon implementation through side-channel analysis on a Feitian [10] open JavaCard smartcard, the attack is tested on a `YubiKey 5Ci`, a FIDO hardware token from Yubico. All `YubiKey 5 Series` (before the firmware update 5.7 [11] of May 6th, 2024) are affected by the attack. In fact all products relying on the ECDSA of Infineon cryptographic library running on an Infineon security microcontroller are affected by the attack. We estimate that the vulnerability exists for more than 14 years in Infineon top secure chips. These chips and the vulnerable part of the cryptographic library went through about 80 CC certification evaluations of level `AVA_VAN_4` (for TPMs) or `AVA_VAN_5` (for the others) from 2010 to 2024 (and a bit less than 30 certificate maintenances).

A coordinated responsible disclosure has been conducted with Infineon (we naturally included

[5] https://www.nxp.com/

[6] https://store.google.com/fr/product/titan_security_key?hl=fr

[7] See https://blog.google/technology/safety-security/titan-security-key-google-store/

[8] or more generally any device running the ECDSA on a P5x secure element

[9] https://www.infineon.com/

[10] https://www.ftsafe.com/Products/Card_OS

[11] https://www.yubico.com/blog/empowering-enterprise-security-at-scale-with-new-product-innovations-yubikey-5-7-and-yubico-authenticator-7/

Yubico and Feitian, as well as the certification bodies BSI [12] and ANSSI [13] in the responsible disclosure). The responsible disclosure started on April the 19th 2024 and Infineon confirmed (July 26^{th}, 2024) that they have implemented and tested a patch to their library, eliminating the attack threat.

Necessary information about FIDO protocol, ECDSA and our side-channel acquisition chain are presented in the remainder of this introduction while the following chapters present, in all technical details:

- a first side-channel reverse-engineering of the ECDSA implementation, and more particularly of the nonce modular inversion, leading to the identification of a sensitive leakage (Chapter 2);
- the design of a generic and flexible side-channel attack on the Extended Euclidean Algorithm (Chapter 3);
- a deeper reverse-engineering of Infineon implementation to uncover stronger sensitive leakages (Chapter 4);
- the successful application of the attack on `Feitian A22 JavaCard` and `YubiKey 5Ci` products (Chapter 5);
- the demonstration that this vulnerability is not limited to `SLE78` but also to all the subsequent families of Infineon security microcontrollers (Chapter 6).

Finally, Chapter 7 presents the attack impact analysis (by no means comprehensive), a simple mitigation solution that Infineon chose to follow, several directions of research opened by this work and finally the whole project timeline.

1.2 FIDO Hardware Tokens

FIDO Hardware tokens are USB and/or NFC devices (sometimes Bluetooth) that allow a user to setup a *factor of authentication* while sign in to a web service account that supports FIDO protocol [14] (*e.g.* a Google account). This *authentication factor* can be added to a traditional login/password authentication (in that case our attack requires that the attacker also knows the login/password of the target, *e.g.* using phishing) or as a unique factor of authentication (as specified in the FIDO2 protocol and often refereed to as the *FIDO2 Passwordless Authentication* or *Passkey* functionality).

For the purpose of this document, we do not need to go into the details of the FIDO protocol – interested readers can have a look to [19] where we took some time to explain the basic principles. Here, what we need to know can be summarized as follows [15]:

- when registering a new FIDO token in the user account, an ECDSA key pair is generated inside the device. The device sends to the remote server the public key and stores the private key;

[12] `https://www.bsi.bund.de/DE/Home/home_node.html`

[13] `https://cyber.gouv.fr/`

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

[15] The ECDSA scheme is formally described in Section 1.4

- when signing in, the remote sever sends a challenge that is signed by the FIDO token using the associated private key. The remote server can confirm the presence of the token by verifying the signature (using the public key shared during the registration phase).

Side-channel attack scenario Here is a natural side-channel attack scenario (as proposed initially in [19]):

1. the adversary steals the login and password of a victim's application account protected with FIDO (*e.g.* via a phishing attack);
2. the adversary gets physical access to the victim's 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 sends the authentication request to the device as many time as necessary[16] while performing side-channel measurements;
4. the adversary quietly gives back the FIDO device to the victim;
5. the adversary performs a side-channel attack over 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 FIDO device, and without the victim noticing. In other words the adversary created a clone of the FIDO 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 authentication credentials.

While the first version of the Google Titan security key embeds a P5x secure element from NXP, Yubico – the leader of FIDO hardware tokens – relies on the more recent `SLE78` secure element from Infineon. One of the advantages of the `SLE78` chip is that there exists a USB version, allowing to build a USB secure token with a single chip (the `SLE78` handling both the secure element functionalities and the USB communication). This is the *"industry's first FIDO2 certified Reference Design based on the `SLE78` single-chip solution"* [17] and seems pretty attractive for designing FIDO hardware token. Also, compared to NXP P5x secure elements, the `SLE78` family is more recent and has still active `AVA_VAN_5` (and in fact EAL 6+) CC certificates.

For the interested readers of an overview of the internals of many FIDO hardware tokens, we strongly encourage to have a look to Victor Lomné presentation at Hardwear.io NL 2022 conference [18]. Figure 1.1 is directly taken from this presentation and shows the teardown of all `YubiKey 5 Series`. All of them embed an Infineon `SLE78CLUFX5000` secure microcontroller, this will be our new target of investigation. However, the FIDO protocol does not allow to choose (or even know) any private key generated inside the device. To ease the understanding of Infineon implementation and the research of a vulnerability, we first need to find a product where one can know the private key, also called a *training device* (this was also the direction chosen in [19], with success). We then first look for a *JavaCard open platform* that embed a `SLE78`, indeed JavaCard open platforms offer a richer user cryptographic API through the development of JavaCard applets.

[16] This can be done while actually signing in to the remote server or not. In the former case, this would make the FIDO counter countermeasure useless but necessitate to have either a small number of ECDSA executions to perform or a lot of time.

[17] https://fidoalliance.org/tech-industry-leaders-ship-fido2-certified-solutions-to-reduce-password-use-on-the-web/

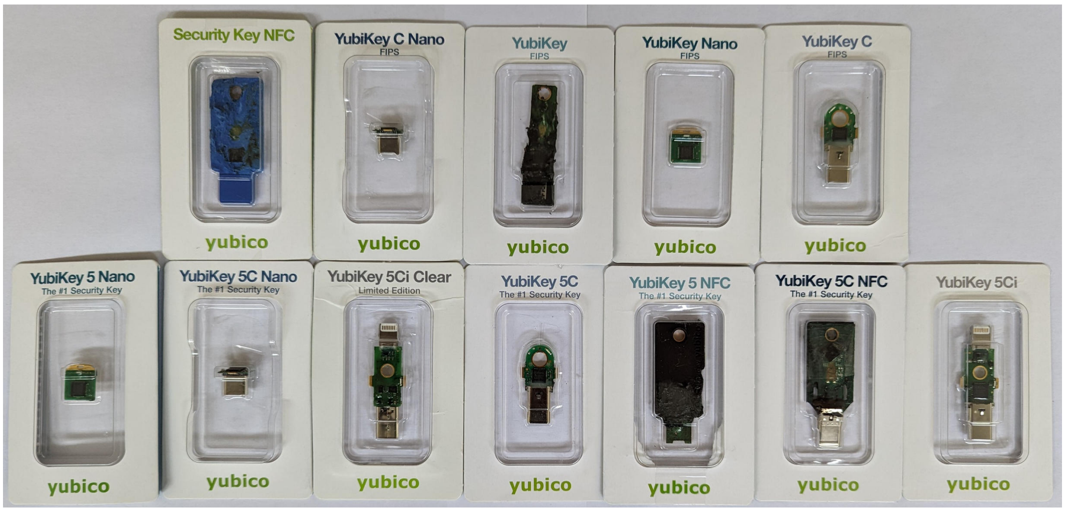


Figure 1.1: `YubiKey 5 Series` Teardowns

1.3 Infineon `SLE78`

When considering only `AVA_VAN_5` secure elements (*i.e.* the most secure chips that exist), the Infineon `SLE78` family is one of the most common in the field. It has been designed for banking and ID applications and successfully took an important market share. The microcontroller is a proprietary design from Infineon, from the many public CC certification reports of chips from the SL78 family [18], we have:

> *”The [`SLE78`] provides a real 16-bit CPU-architecture and is compatible to the Intel 80251 architecture. The major components of the core system are the dual CPU (Central Processing Units), the MMU (Memory Management Unit) and MED (Memory Encryption/Decryption Unit). The dual interface controller is able to communicate using either the contact based or the contactless interface.”*

Infineon `SLE78`, like any secure element, embeds several dedicated hardware cryptographic co-processors, notably a modular arithmetic co-processor which can be used, though a cryptographic library (we will sometimes use *cryptolib*, for short), to securly (and efficiently) execute public-key cryptography like RSA or ECC. Infineon provides an optional cryptolib implementing such cryptographic operations.

The `YubiKey 5 Series` are based on a SLE78 IC version *M7893 B11* with Infineon EC cryptolib version 1.03.006 [19], information that can be found in the ANSSI CSPN (Certificat de Sécurité de Premier Niveau) security target document [20].

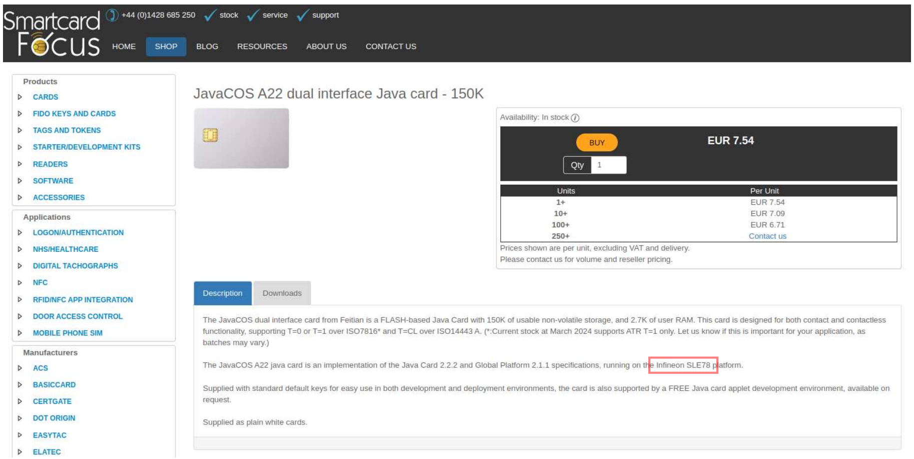
[18] See *e.g.* `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/Reporte07/0782V5a_pdf`

[19] M7893 B11 with EC library v1.03.006: last CC certification (EAL6+) in 2018, `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/Reporte08/0879V3a_pdf`

[20] `https://cyber.gouv.fr/sites/default/files/2021/09/anssi-cible-cspn-2021_18en.pdf`, page 18, Table 5

1.3.1 `Feitian A22 JavaCard`

When looking for JavaCard open platforms that embed a `SLE78`, the `Feitian A22 JavaCard` product was cheap and easy to purchase (see Figure 1.2 from SmartCard Focus reseller site [21]).



JavaCOS A22 dual interface Java card - 150K

Availability: In stock

EUR 7.54

Units	Per Unit
1+	EUR 7.54
10+	EUR 7.09
100+	EUR 6.71
250+	Contact us

Prices shown are per unit, excluding VAT and delivery.
Please contact us for volume and reseller pricing.

The JavaCOS dual interface card from Feitian is a FLASH-based Java Card with 150K of usable non-volatile storage, and 2.7K of user RAM. This card is designed for both contact and contactless functionality, supporting T=0 or T=1 over ISO7816* and T=CL over ISO14443 A. (*:Current stock at March 2024 supports ATR T=1 only. Let us know if this is important for your application, as batches may vary.)

The JavaCOS A22 java card is an implementation of the Java Card 2.2.2 and Global Platform 2.1.1 specifications, running on the Infineon SLE78 platform.

Supplied with standard default keys for easy use in both development and deployment environments, the card is also supported by a FREE Java card applet development environment, available on request.

Supplied as plain white cards.

Figure 1.2: `Feitian A22 JavaCard` – Screenshot from SmartCard Focus

On a JavaCard open platform, we can develop and push our own JavaCard applet and run all cryptographic primitives supported by the JavaCard API and implemented by the JavaCard OS of the card. This JavaCard (or a similar one from Feitian) has been certified EAL5+ under Common Criteria [22]. This certification is from 2018 and involves `SLE78` IC version *M7892 B11* with Infineon EC cryptolib version 1.02.013 [23].

This is not exactly the same `SLE78` hardware version nor cryptolib version than the `YubiKey 5 Series`, we will assume it is close enough.

On `Feitian A22 JavaCard`, the JavaCard OS happens to follow JavaCard 2.2.2 specifications [24], we hence developed and loaded a custom JavaCard applet [24] allowing us to freely control the JavaCard ECDSA signature engine on `Feitian A22 JavaCard`. More precisely, we can load chosen long term ECDSA private keys, perform ECDSA signatures and ECDSA signature verifications.

1.4 Elliptic Curve Digital Signature Algorithm

In this section we recall some basic information about ECDSA as well as introduce the notations we will use in this document. It is worth noting here that the FIDO protocol is specified on the 256-bit elliptic curve P256 [23], and then, *w.l.o.g.*, all our tests are done on this curve. Hence, for our simulations, the elliptic curve order N is that of P256 curve and the nonce k is always a random of binary size up to 256 bits.

[21] https://www.smartcardfocus.com/shop/ilp/id~712/javacos-a22-dual-interface-java-card-150k/p/

[22] https://www.commoncriteriaportal.org/files/epfiles/SERTIT-091CRFeitianv1.0.pdf

[23] M7892 B11 with EC library v1.02.013: last CC certification (EAL6+) in 2020, https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/Reporte07/0782V5a_pdf

[24] Thanks notably to the great open-source project for building JavaCard applets [20]

1.4.1 ECDSA Signature Scheme

The ECDSA signature scheme is the main target of our attack. Here is a rough sketch of the scheme (taken from [19] with slightly different notations):

- elliptic curve E over prime field $\mathbb{F}_p$, elliptic curve base point is $G_{(x,y)}$ of order N
- inputs: long term private key d, hash digest of the input message to sign $h = H(m)$

1. randomly generate a nonce k in $\mathbb{Z}/N\mathbb{Z}$
2. scalar multiplication $Q_{(x,y)} = [k]G_{(x,y)}$
3. denote by r the x-coordinate of Q: $r = Q_x$
4. compute $s = k^{-1}(h + rd) \bmod N$

- output: (r,s)

First Remark: We can find the nonce k from the signature (r, s) of digest h with the knowledge of the private key d:

$$k = s^{-1}(h + rd) \bmod N \tag{1.1}$$

Inversely, knowing the nonce k is enough to find the private key d:

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

Second Remark: An usual countermeasure against side-channel analysis is to randomize the base point at each scalar multiplication (see [3]). So instead of computing directly the scalar multiplication $[k]G_{(x,y)}$ on the affine coordinates of G, one might uses the projective coordinates of G:

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

1.4.2 ECDSA Signature Verification Scheme

As mentioned before, one great advantage of working on the `Feitian A22 JavaCard` is the possibility to run the ECDSA signature verification algorithm (and not only the signature algorithm as on `YubiKey 5 Series`). As we will see, the signature verification algorithm requires to compute similar operations than in 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.

Here is a rough sketch of the ECDSA signature verification scheme (taken from [19] with slightly different notations):

- elliptic curve E over prime field $\mathbb{F}_p$, elliptic curve base point is $G_{(x,y)}$ and order is N
- inputs: public key $P_{(x,y)}$, the hash of the signed input message $h = H(m)$
- inputs: the signature to be verified (r, s)

1. compute first scalar $k^{(1)} = s^{-1}r \bmod N$
2. compute second scalar $k^{(2)} = s^{-1}h \bmod N$
3. first scalar multiplication $Q^{(1)}_{(x,y)} = [k^{(1)}]P_{(x,y)}$
4. second scalar multiplication $Q^{(2)}_{(x,y)} = [k^{(2)}]G_{(x,y)}$
5. compute $\bar{r} = Q^{(1)}_x + Q^{(2)}_x \bmod N$
6. check that $\bar{r} = r$

First Remark: One can compute $k^{(1)}$ and $k^{(2)}$ from the public inputs.

Second Remark: The two scalar multiplications can be executed in a single pass using a *double scalar multiplication* approach. One way to do so is to pre-compute $Q_{(x,y)} = P_{(x,y)} + G_{(x,y)}$ and then follow a simple double-and-add algorithm where, at each iteration, one bit of $k^{(1)}$ and one bit of $k^{(2)}$ are processed.

1.5 Side-Channel Setup and First Observations

Let us briefly present the side-channel setup we used for our analysis.

1.5.1 Side-Channel Setup

In order to perform electromagnetic (EM for short) side-channel measurements over the two targets, we used the following hardware parts:

- Langer ICR HH 500-6 near-field EM probe with an horizontal coil of diameter 500μm and a frequency bandwidth from 2MHz to 6GHz with its Langer BT 706 bias-tee [15];
- Thorlabs PT3/M three axes (X-Y-Z) manual micro-manipulator with a precision of 10μm [28];
- Dino-Lite digital microscope AM4113TL [4];
- Pico Technology PicoScope 6424E oscilloscope [25], with a 500MHz frequency bandwidth, sampling rate up to 5GSa/s, 4 channels, a shared channel memory of 4G samples max. and 8-bit to 12-bit ADC resolution;
- LeCroy WavePro 254HD oscilloscope, with a 2.5GHz frequency bandwidth, sampling rate up to 20GSa/s, 4 channels and a memory of 5G samples max. per channel with 12-bit ADC resolution [16];
- Ledger Scaffold electronic board [17].

Note that the cost of this setup is about 10k€ (including the cost of the computer used for processing side-channel measurements). The LeCroy WavePro oscilloscope with 12-bit resolution raises the cost (it has been used for the Yubikey acquisitions) by about 30k€, but we are confident that the PicoScope set with 8-bit ADC resolution would have been completely sufficient for the attack.

1.5.2 YubiKey 5Ci

Accessing the SLE78 on a YubiKey requires to open the plastic case and access the logic board. From Victor Lomné teardowns of the whole family of YubiKey 5 Series (see Figure 1.1 and [18]), the YubiKey 5Ci seemed in good shape, so we selected it (see Figure 1.3). At the end of this project, we wanted to estimate the difficulty for an attacker to remove the plastic case of a YubiKey. However, by that time (two years have passed), Victor did not remember how hard it was. From Figure 1.1 one can tell that the NFC and FIPS version of the YubiKey were quite difficult to open and the result leads (in the case of Victor's not so careful opening) to destroy the product. However, for the rest, it is difficult to tell. So we bought two new YubiKey 5C from Yubico. The package opening of these two devices is presented in Annex A. This study was done after the YubiKey 5.7 firmware update of the 6th May 2024 that moves the firmware to a new cryptography library [25].

We hence could validate, by the observation of the side-channel execution traces of these new devices, that the new implementation is not *vulnerable to the attack presented in this report.*

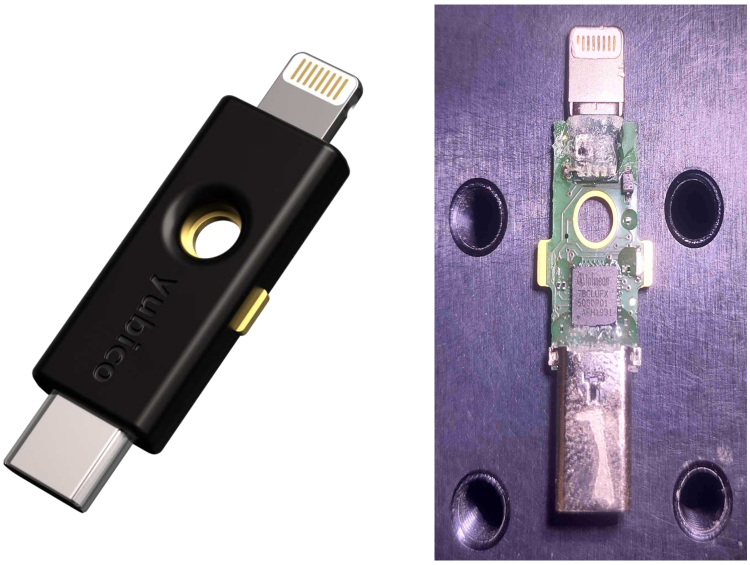


Figure 1.3: YubiKey 5Ci – Teardown

The EM acquisition setup is depicted on Figure 1.4, with the correct position of the EM probe to acquire the signal.

[25] https://www.yubico.com/blog/empowering-enterprise-security-at-scale-with-new-product-innovations-yubikey-5-7-and-yubico-authenticator-7/

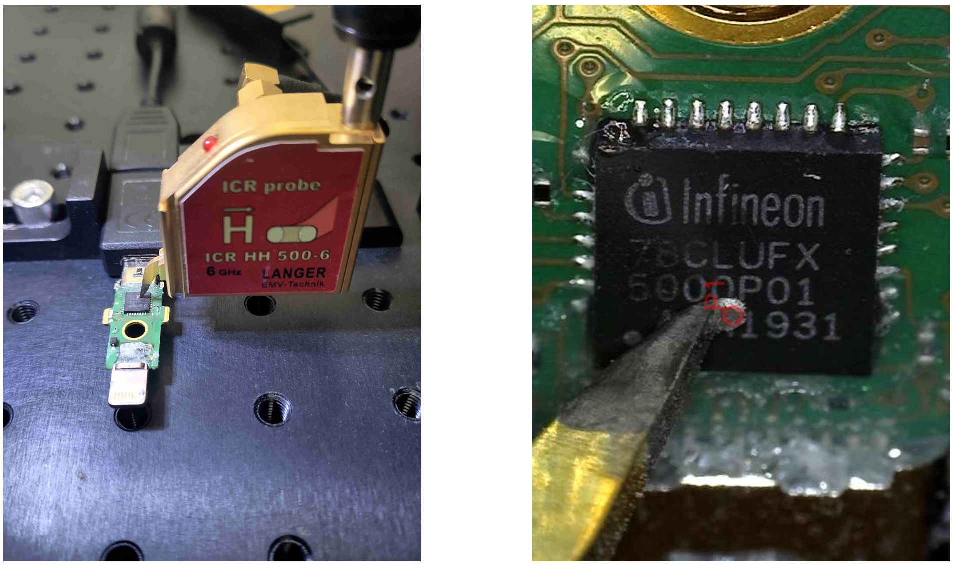


Figure 1.4: `YubiKey 5Ci` – EM Acquisition Setup

Figure 1.5 shows a side-channel execution trace of an ECDSA signature acquired with the above setup. The acquisition is triggered directly from the EM signal, on a pattern present at the beginning of the EM activity of the command processing the `authentication request message` of the FIDO protocol.

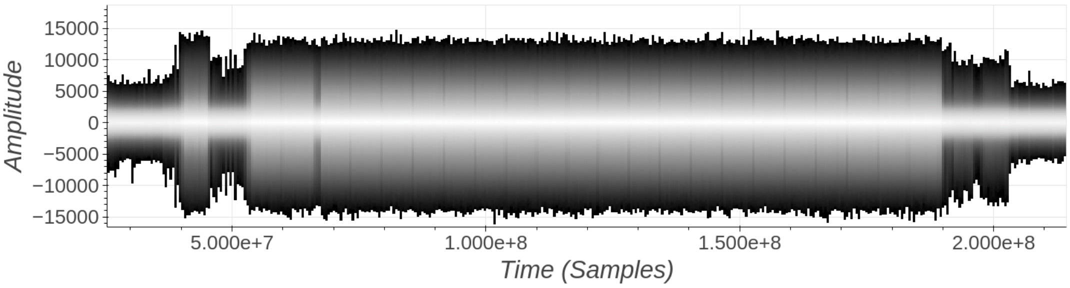


Figure 1.5: `YubiKey 5Ci` – EM Traces – ECDSA Signature

1.5.3 `Feitian A22 JavaCard`

To communicate and acquire both power and EM side-channels from the `Feitian A22 JavaCard`, we use the Ledger Scaffold electronic board [17]. It provides several features really useful for side-channel attacks:

- native support of ISO7816, with possibility to easily spy all digital signals of the protocol thanks to SMA connectors (RESET, CLOCK and DATA);
- triggering capabilities easily configurable either on the APDU header processing, or on the APDU command processing;

- power measurement capabilities thanks to a clean PCB design, and a potentiometer allowing to manually adapt the gain of the measured power consumption thanks to a screw. Furthermore it is possible to activate 1kOhm resistors placed in serial on the ISO7816 digital signals, allowing to reduce the activity of the ISO7816 clock in the measured power consumption;

- the Ledger Scaffold is physically made of two parts: one *motherboard* (main part of Scaffold including the FPGA driving the board, all the electronic circuitry and the different connectors) and one *daughter board* (used to connect the target, and being interchangeable). Thus one can easily develop new *daughter boards*, for instance for having access to the different sides of the target. This is what NinjaLab has done with a custom smartcard daughter board that gives access to the opened side of the smartcard (see Figure 1.6).

The full setup is shown on Figure 1.6, to access the die we simply remove the white plastic of the smartcard above the die with a scalpel, we hence have access to its front side. The right subfigure shows the EM probe position over the die.

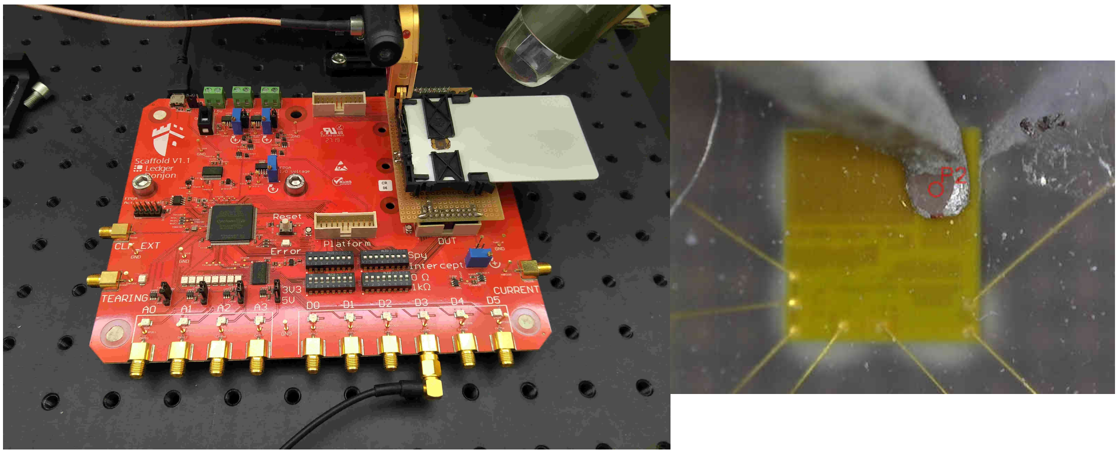


Figure 1.6: `Feitian A22 JavaCard` – EM Acquisition Setup

Figure 1.7 depicts a side-channel execution trace of an ECDSA signature acquired with the above setup. The acquisition is triggered thanks to the Scaffold board on the last byte of the ECDSA signature APDU command sent. On the figure, we identified the three major steps of ECDSA signature:

1. Initialization and generation of nonce k and random projective coordinate z
2. Scalar multiplication $Q_{(x,y,z)} = [k]G_{(x,y,z)}$
3. Computation of the signature (r, s) from $Q_{(x,y,z)}, h, d, k$

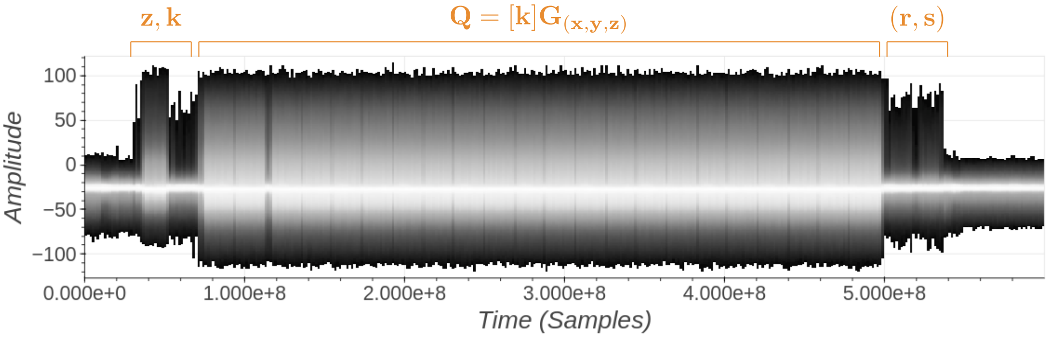


Figure 1.7: `Feitian A22 JavaCard` – EM Traces – ECDSA Signature

The `Feitian A22 JavaCard` and `YubiKey 5Ci` ECDSA signature EM activities look very similar, we will then work on `Feitian A22 JavaCard` and try to find a weakness in Infineon implementation. After fruitless attempts to find a vulnerability in the scalar multiplication, we focused our effort on the end of the computation and more precisely the nonce modular inversion because of a surprisingly *not* constant-time operation.

1.5.4 Focus on the Nonce Modular Inversion

Figure 1.8 is a zoom into the end of the ECDSA signature EM activity. The computation of (r, s) from $[k]G_{(x,y,z)}, k, h, d$ amounts to:

- $r = z^{-1}x \bmod p$
- $s = k^{-1}(h + rd) \bmod N$

It seems pretty obvious to us that the two similar shaped operations (identified on Figure 1.8) correspond to the two modular inversions (which are the most complicated operations in the computation of (r, s)). Moreover, when observing several ECDSA signature executions, these operations seem to have different execution times.

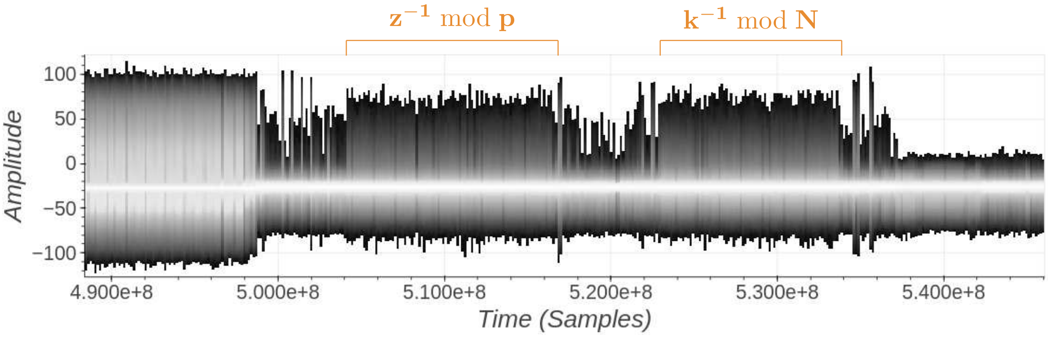


Figure 1.8: `Feitian A22 JavaCard` – EM Traces – ECDSA Signature – (r, s) Computation

This observation is the beginning of a journey that will take two years to complete and a lot of efforts and disappointments but finally a success. This document tries to report this journey in its many details.

Chapter 2

Reverse-Engineering of the Modular Inversion

As mentioned in Chapter 1, we decided to analyze the nonce modular inversion of the Infineon ECDSA implementation due to the presence of a timing leakage. A timing leakage does not necessary mean that there is a vulnerability but it is surprising for such a sensitive operation. Indeed, revealing even a small number of bits of the nonce for several ECDSA signatures is enough for a complete key-recovery attack thanks to lattice reduction techniques (initially proposed in [10] and applied in many practical attacks since).

2.1 ECDSA Signature Traces

First, 1000 EM side-channel traces of ECDSA signature are acquired with the `Feitian A22 JavaCard` setup. For completeness, the acquisition details are given in Table 2.1. 1000 EM side-channel traces of the full ECDSA signature execution are acquired in about 7 hours.

2.1.1 Acquisition Campaign

operation	ECDSA signature
equipment	PicoScope 6424E, Langer ICR HH 500-06
inputs	Messages are random, Key is constant (randomly chosen)
number of operations	1000
length	120ms
sampling rate	5GSa/s
samples per trace	600MSamples
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-120, 80]$mV
file size	600GB
acquisition time	about 7 hours

Table 2.1: Acquisition parameters on `Feitian A22 JavaCard` – ECDSA Signature

Figure 2.1 shows a single side-channel execution trace, where the modular inversion patterns are identified. We suppose that the second pattern corresponds to the nonce inversion and the first one (the closest to the scalar multiplication) relates to the inversion of the third projective coordinate of the resulting point Q. It just makes more sense but the operations could be done the other way around.

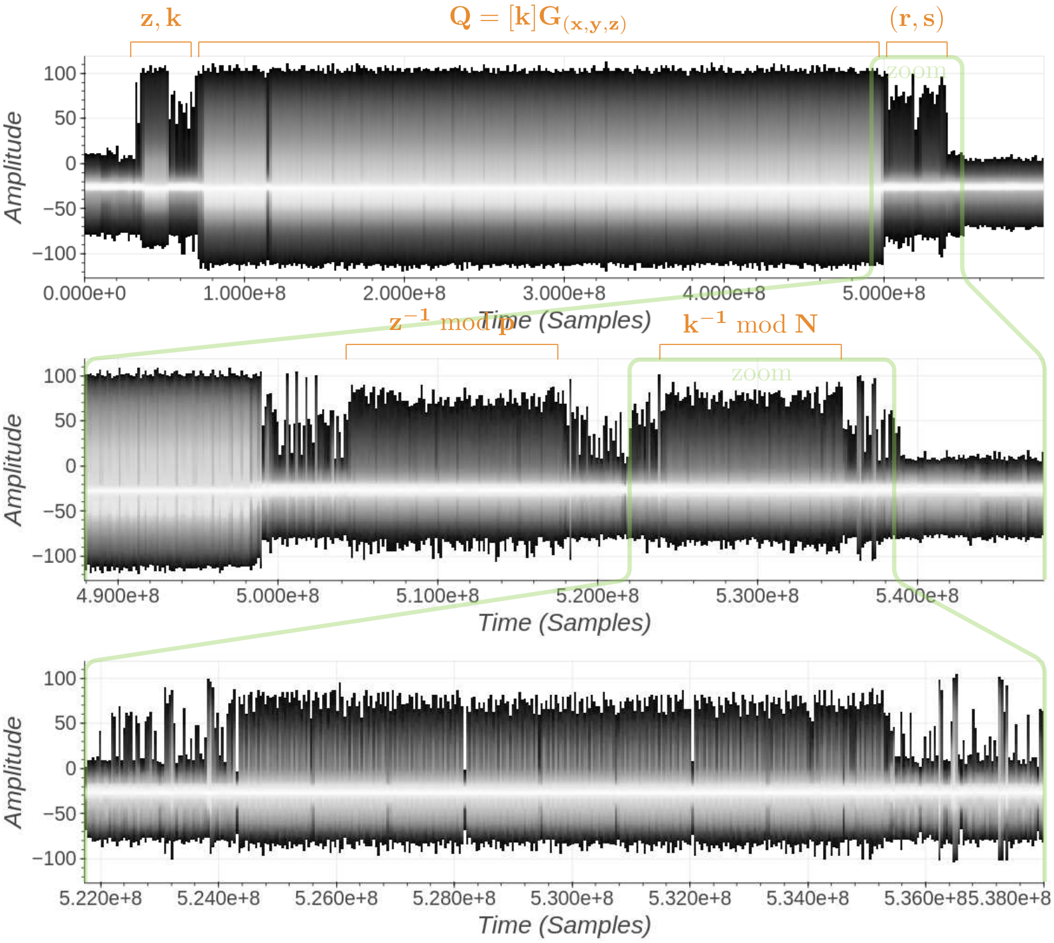


Figure 2.1: `Feitian A22 JavaCard` – ECDSA Signature – Full Trace (Top) - (r, s) Computation (Middle) - $k^{-1} \bmod N$ Computation (Bottom)

2.1.2 Side-Channel Analysis

Now that we have many executions for various values of the nonce k (which is known here since the private key d is known, thanks to equation 1.1), let us look into the EM signal. Figure 2.2 bottom subfigure shows that the computation is an iterative process. Moreover, the process is regularly paused (as we can see on the figure). These pauses are quite easy to automatically detect (specially because they occur at a pretty stable period), the two black dots on the figure illustrate this automatic detection.

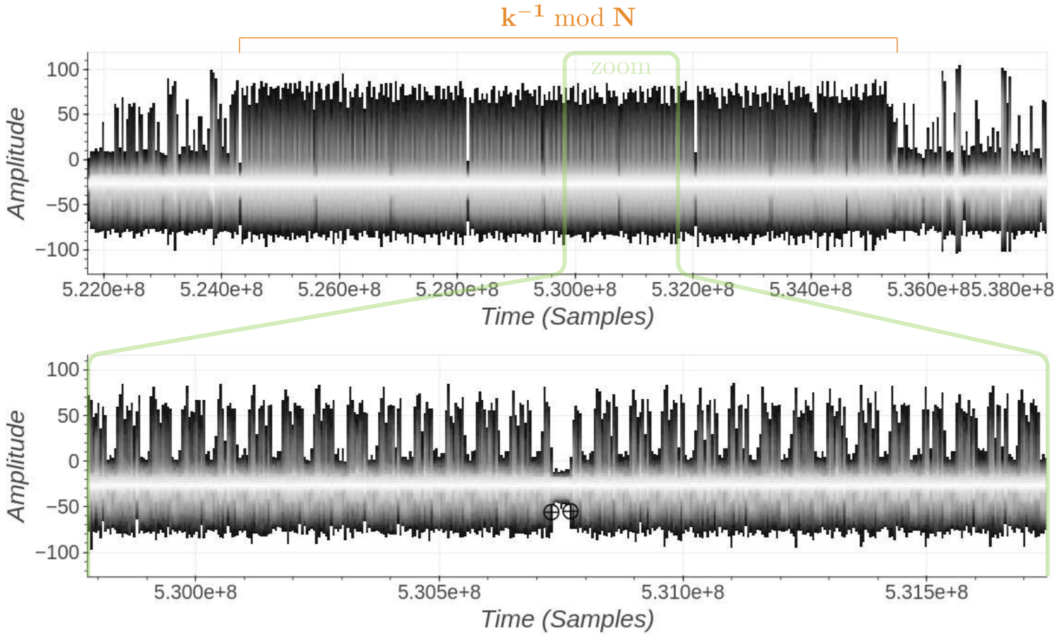


Figure 2.2: `Feitian A22 JavaCard` – ECDSA Signature – k^{-1} mod N Computation (Top) - Zoom on Few Iterations (Bottom)

Figure 2.3 shows the result of this detection over the whole operation (top subfigure). Our first step is simply to remove these pauses as we suspect they are not related to the computation but to regular interrupts of the microcontroller. The second subfigure depicts a single subtrace (corresponding to the suspected modular inversion) after the removal of all pauses.

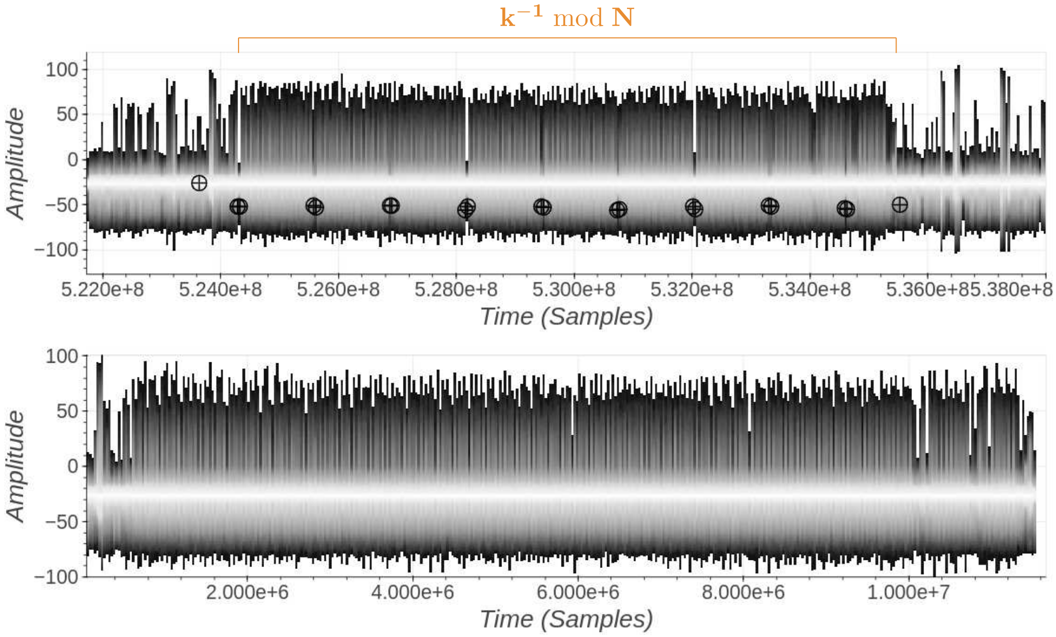


Figure 2.3: `Feitian A22 JavaCard` – ECDSA Signature – k^{-1} mod N Computation – Interruptions Detection (Top) - Cleaned Trace (Bottom)

The next step is to detect all the operation iterations, this is done over all the 1000 ECDSA executions. The result is illustrated on Figure 2.4 first subfigure where each iteration is detected through the research of two anchors: the blue and black dots (see the second subfigure). This detection process is simply based on the amplitude of the signal and approximate length of an iteration. We cannot tell if the detection works perfectly over the whole set of 1000 traces but random visual checks over some traces make us pretty confident that we are not too wrong. One can also remark that some iterations are longer than others (or are preceded by pauses), as we can see in the middle of the second subfigure.

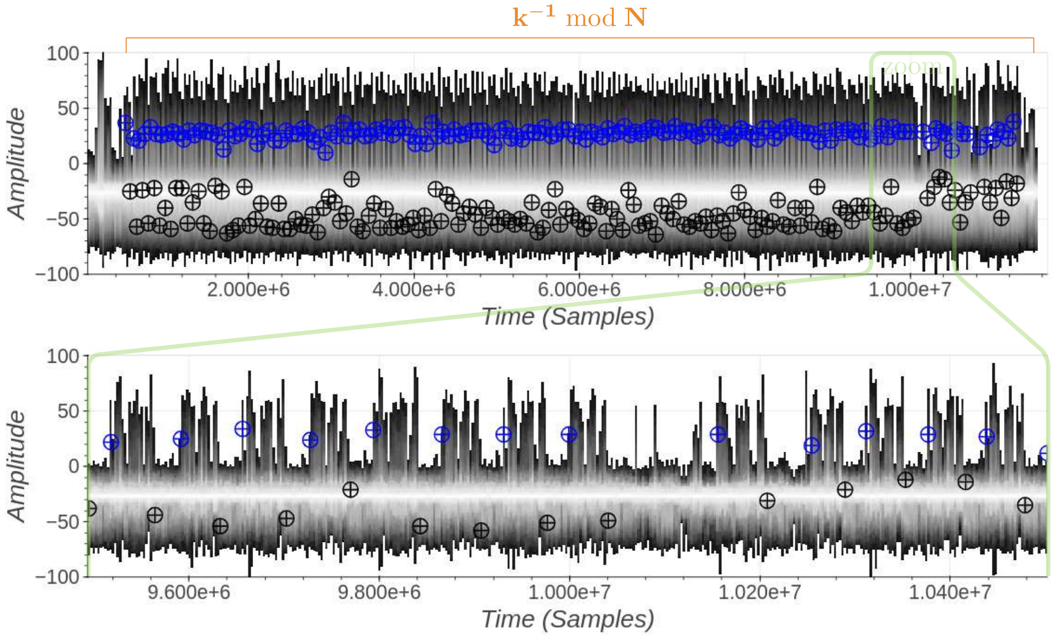


Figure 2.4: `Feitian A22 JavaCard` – ECDSA Signature – k^{-1} mod N Computation – Iterations Detection (Top) - Zoom on Few Iterations (Bottom)

Now we have a precise estimate of the number of iterations of a modular iteration, we can validate that this number changes from one ECDSA execution to the other. When considering a modular inversion that is not constant time, our first hypothesis is the Extended Euclidean Algorithm (`EEA` for short).

2.1.3 First Hypothesis: Extended Euclidean Algorithm

Algorithm 1 recalls the computation of a modular inversion with the textbook `EEA` (there are many versions of the `EEA` and notably the binary version, but here the number of iterations observed would better match the classical `EEA`). If our hypothesis is correct, then the number of iterations detected previously should match the Algorithm 1 while loop iteration number (for the right inputs (k, N), where k is the nonce and N is the order of the elliptic curve). However, while the iteration count of $\texttt{EEA}(k, N)$ (based on Algorithm 1) is always close to the number of detected iterations in the side-channel traces, it does not match exactly.

Algorithm 1: Extended Euclidean Algorithm for Modular Inversion

```
Input  : $v, n$: two positive integers with $v \leq n$ and $gcd(v, n) = 1$
Output: $v^{-1} \bmod n$: the inverse of $v$ modulo $n$
$r_0, r_1 \leftarrow n, v$
$t_0, t_1 \leftarrow 0, 1$
while $r_1 \neq 0$ do
    $q \leftarrow \texttt{div}(r_0, r_1)$
    $r_0, r_1 \leftarrow r_1, r_0 - q.r_1$
    $t_0, t_1 \leftarrow t_1, t_0 - q.t_1$
end
if $t_0 < 0$ then
    $t_0 \leftarrow t_0 + n$
end
return $t_0$
```

When plotting the distribution of the number of iterations over the 1000 observed operations with regard to the distribution of the number of iterations of random calls to $\texttt{EEA}(x, N)$ with x a random value of same size as k, one can tell that the distribution matches. This is illustrated on Figure 2.5, confirming that we are observing the execution of an `EEA`, but not with the value of k.

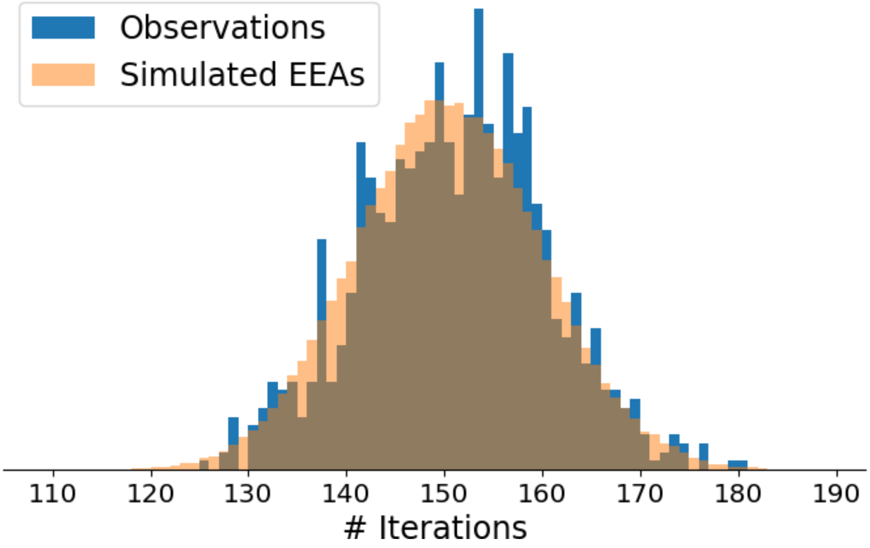


Figure 2.5: `Feitian A22 JavaCard` – ECDSA Signature – $k^{-1} \bmod N$ Computation – Distribution of the Number of Iterations – Observations (Blue) - EEA Simulations (Orange)

After checking that the first modular inversion pattern was not the correct one (the inversion of z and k could have been done in the unnatural way), the next hypothesis is the presence of a side-channel countermeasure:

Hypothesis: k is *blinded* before its inversion.

This would make an attack much harder (and maybe impossible if the mask is large enough). Validating this hypothesis seems quite far-reaching, we rather first validate once for all that the identified operation is really a modular inversion using EEA before going deeper in this direction. One way to do so is to study the ECDSA signature verification execution.

2.2 ECDSA Signature Verification Traces

The ECDSA signature verification scheme involves the modular inversion of a public value (see the $s^{-1} \bmod N$ operation in the introductory chapter, Section 1.4.2). Let us bet that the developer did not protect this operation since it does not involve any secret.

2.2.1 Acquisition Campaign

The acquisition campaign for the ECDSA signature verification operation is quite similar to that of the ECDSA signature operation, full details are given in Table 2.2. A single side-channel execution trace is displayed in Figure 2.6.

operation	ECDSA signature verification
equipment	PicoScope 6424E, Langer ICR HH 500-06
inputs	Messages are random, Signature generated from a constant Key
number of operations	1000
length	160ms
sampling rate	5GSa/s
samples per trace	800MSamples
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-120, 80]$mV
file size	800GB
acquisition time	about 7 hours

Table 2.2: Acquisition parameters on Feitian A22 JavaCard – ECDSA Signature Verification

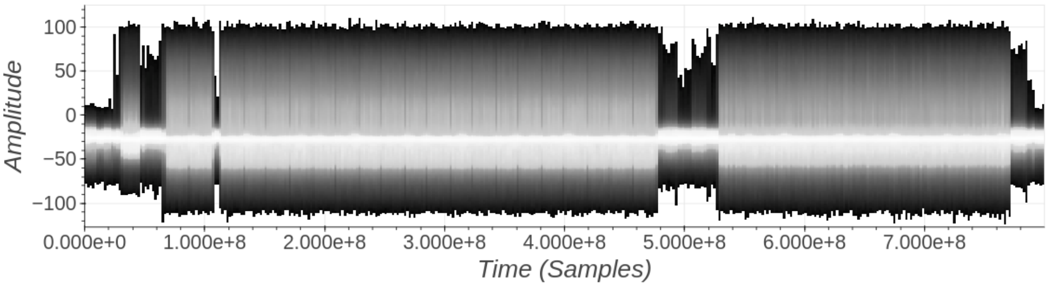


Figure 2.6: Feitian A22 JavaCard – ECDSA Signature Verification – Full Trace

2.2.2 Side-Channel Analysis

The analysis of the side-channel traces tells us the following (as illustrated in Figure 2.7):

- the first long operation is a secure scalar multiplication (very similar to the one observed in the ECDSA signature traces), we do not know its purpose. Maybe to check that the point P is on the curve (even though this seems a bit overkill);
- knowing the values $k^{(1)}$ and $k^{(2)}$ and observing the sequence of doubling and addition operations, it was easy to confirm that the last operation is a double scalar multiplication based on an unbalanced (*i.e.* naive) double-and-add algorithm;
- between the two scalar multiplication operations lies two modular inversion patterns similar to those of the signature algorithm. The first one is certainly owned by the previous scalar multiplication while the second one should be the computation of s^{-1} mod N.

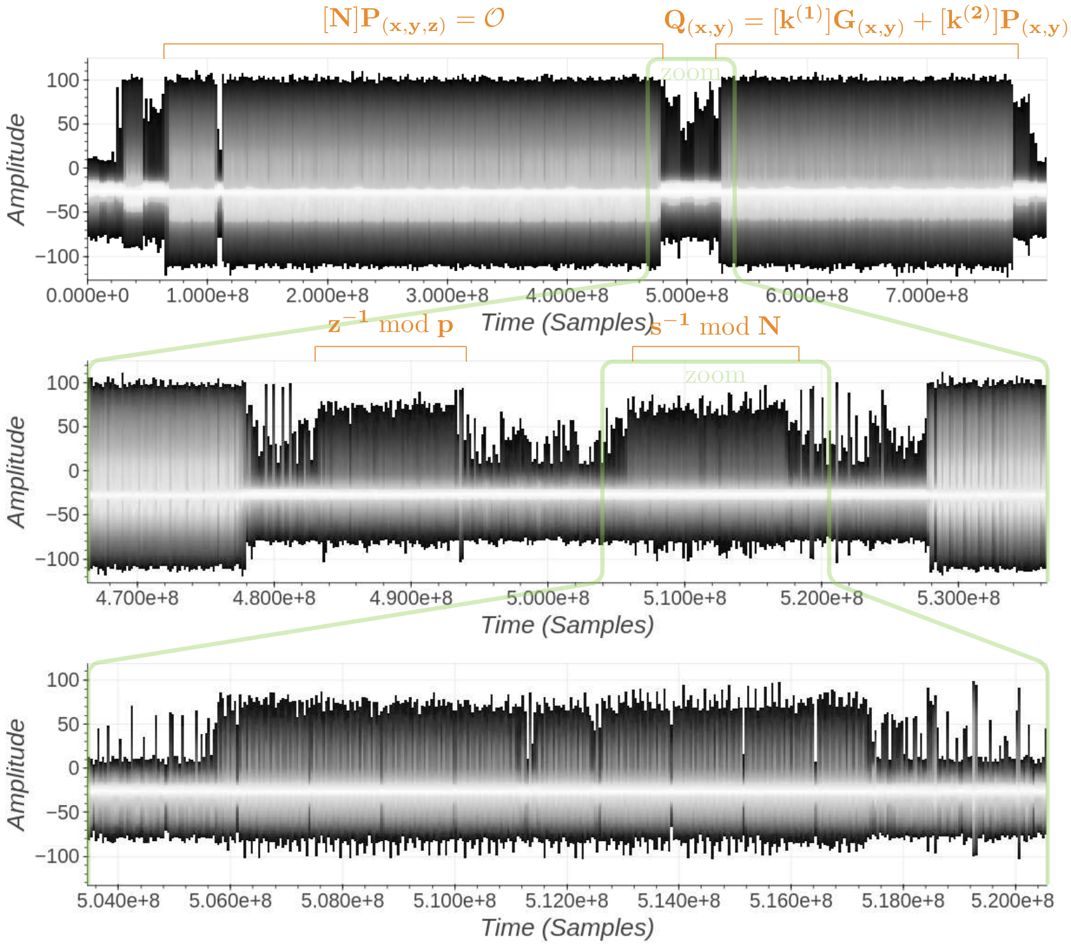


Figure 2.7: `Feitian A22 JavaCard` – ECDSA Signature Verification – Full Trace (Top) - Zoom Between the Scalar Multiplications (Middle) - s^{-1} mod N Computation (Bottom)

After the cleaning process of removing the interrupts (see Figure 2.8) and detecting each iteration (as illustrated in Figure 2.9) we could validate that the number of iterations perfectly matches the `EEA` number of iterations. This confirms without any doubt that the observed process is the `EEA` and let the hypothesis of a blinding countermeasure in the signature algorithm open.

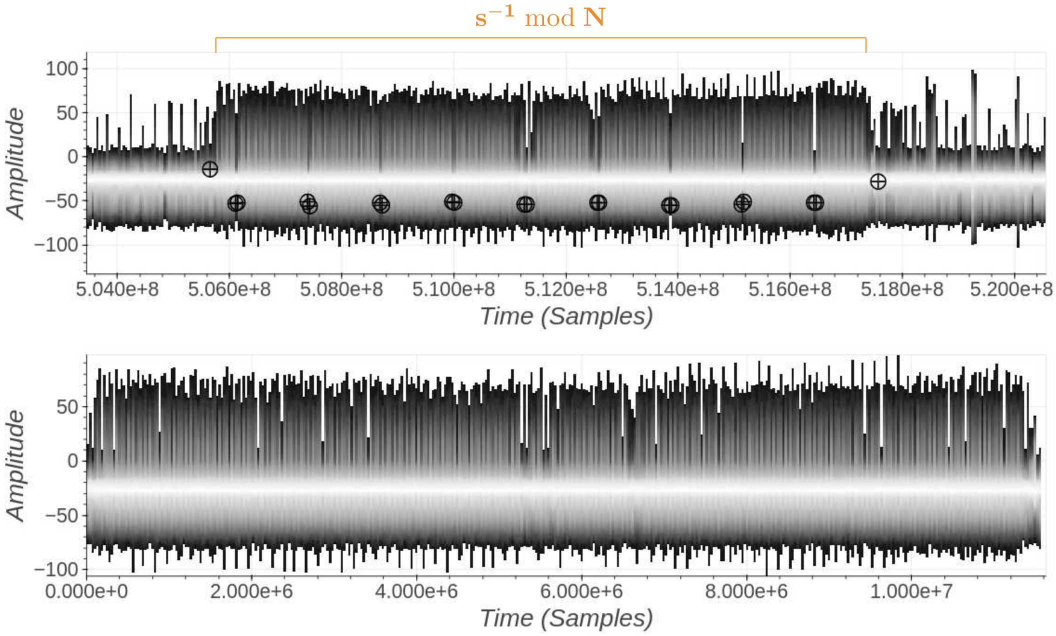


Figure 2.8: `Feitian A22 JavaCard` – ECDSA Signature Verification – s^{-1} mod N Computation – Interruptions Detection (Top) - Cleaned Trace (Bottom)

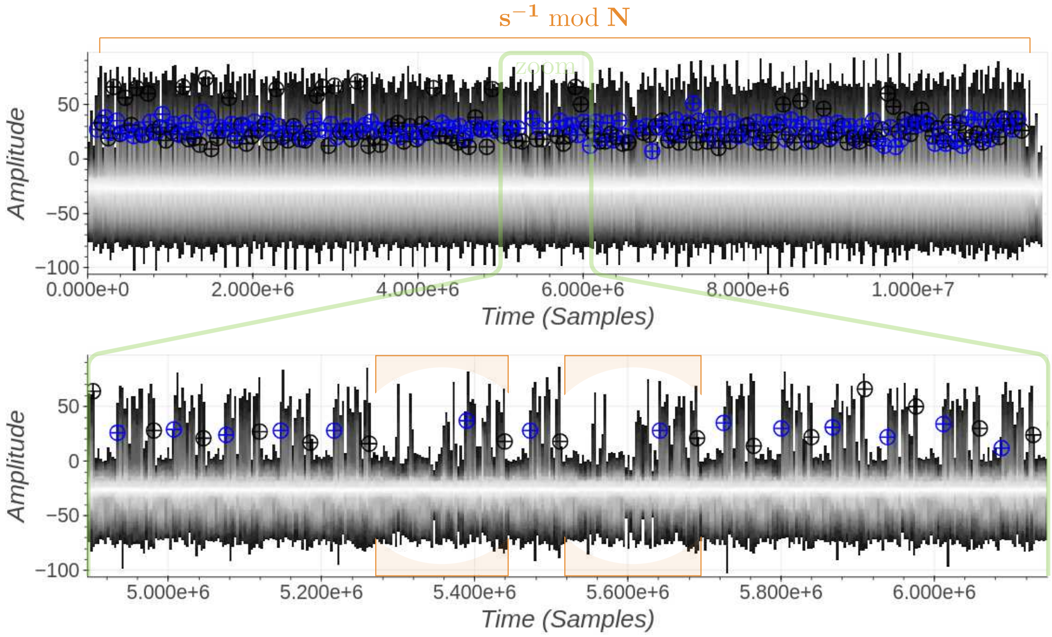


Figure 2.9: `Feitian A22 JavaCard` – ECDSA Signature Verification – s^{-1} mod N Computation – Iterations Detection (Top) - Zoom on Few Iterations (Bottom)

Before coming back to the signature algorithm, we could relate the *large* iterations occurrence (as identified by the orange boxes on Figure 2.9, second subfigure) to the inner values of the `EEA` computation: the iteration is large if (and only if) the binary length difference between r_0 and r_1 (as computed in Algorithm 1, line 4) is larger than 5. In all the following we will denote the binary length difference between r_0 and r_1 as:

$$\Delta_{r_0,r_1} = len(r_0) - len(r_1).$$

2.2.3 A Timing Leakage

We have seen that the computation length of the inner iterations of the `EEA` is related to the value Δ_{r_0,r_1} taken at the beginning of the iteration: the iteration is larger when $\Delta_{r_0,r_1} > 5$. In fact, the timing leakage is stronger than that. Figure 2.10 shows four consecutive iterations (none of them are *large*, *i.e.* $\Delta_{r_0,r_1} \leq 5$), two observations can be made:

- *Odd* versus *Even* iterations are different, we will have to treat them differently (and have no explanation for this observation);
- a small part at the beginning of each iteration is varying in length (identified with orange boxes).

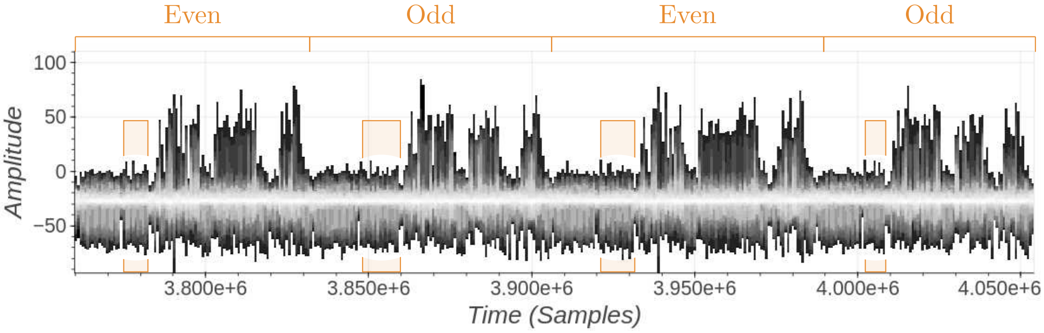


Figure 2.10: `Feitian A22 JavaCard` – ECDSA Signature Verification – s^{-1} mod N – Four Consecutive Iterations

Thanks to the $\Delta_{r_0,r_1} > 5$ timing leakage that stands out clearly, we easily found out that the execution time is directly proportional to the value Δ_{r_0,r_1} as displayed on Figure 2.11 where iterations are synchronized and regrouped with constant values of Δ_{r_0,r_1}.

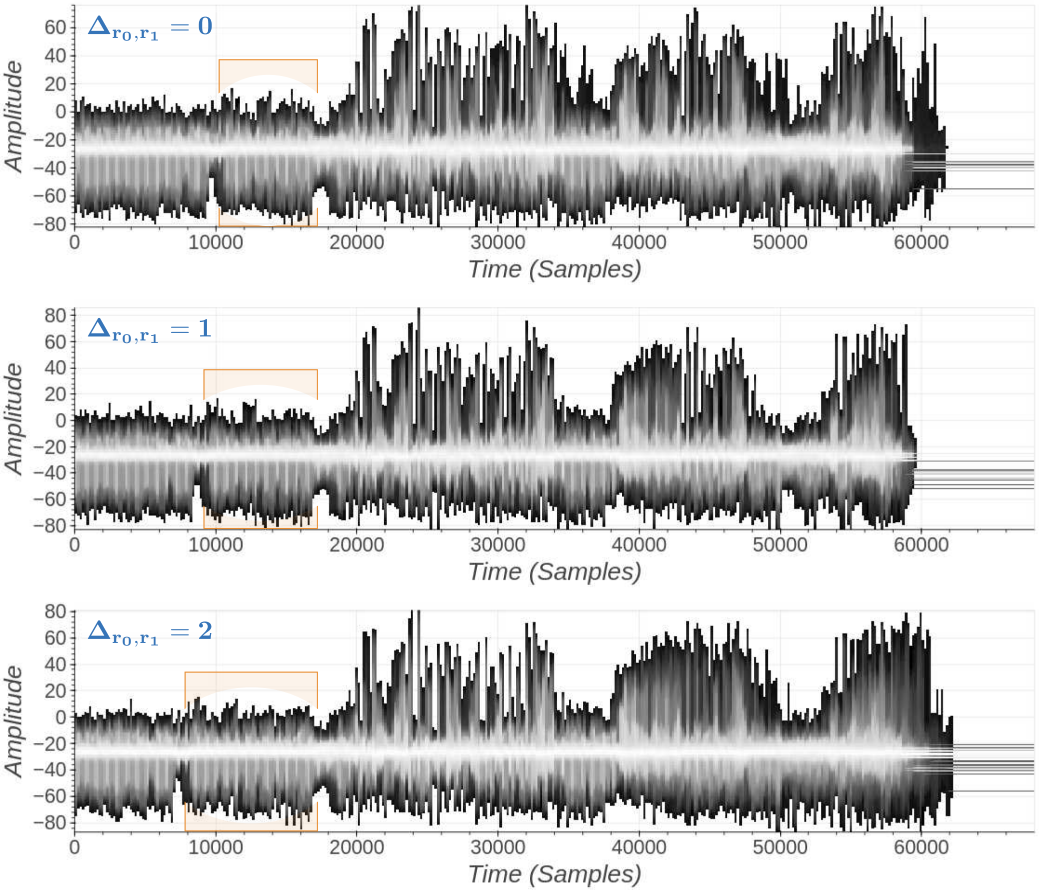


Figure 2.11: `Feitian A22 JavaCard` – ECDSA Signature Verification – 100 Superposed Odd Iterations – $\Delta_{r_0,r_1} = 0$ (Top) - $\Delta_{r_0,r_1} = 1$ (Middle) - $\Delta_{r_0,r_1} = 2$ (Bottom)

2.3 Reverse-Engineering of the Modular Inversion Countermeasure

Let us now get back to the blinding countermeasure hypothesis and try to confirm and better understand it.

2.3.1 Hypothesis

A simple and cost-effective way to protect the modular inversion of the nonce in ECDSA is to use a multiplicative mask. Hence, instead of computing directly the inverse of k, one can compute the inverse of k' defined as:

$$k' = m \times k \bmod N,$$

where m is a non-null positive integer. Once inverted, $k^{-1} \bmod N$ can be easily computed from $k'^{-1} \bmod N$ by another modular multiplication by m.

This countermeasure seems really interesting because it only costs 2 modular multiplications (a quite cheap operation when the secure microcontroller embeds a modular arithmetic co-processor) to protect a modular inversion (a much more expensive operation). Also, if we consider that m is generated with a similar entropy than k itself, retrieving k' by side-channel analysis would provide zero information about the nonce value k and then the attack would be meaningless. However, the real cost of the countermeasure lies in the generation of m and if the random number generator is expensive, it might be tempting to reduce the size of m.

Now, let us suppose that Infineon developers decided to reduce the length of m to something small enough to be brute-forced, then by retrieving k' by side-channel analysis one could brute force the value of k and then recover the ECDSA long term private key. So let us hope for the best (adversary-wise) and try to validate this.

2.3.2 Brute-force Experiments

We build an experiment that would validate that the multiplicative mask is small:

Since the private key is known on the `Feitian A22 JavaCard` setup, we can compute, for each ECDSA execution, the nonce value k (as per equation 1.1). We are looking for the value of the mask m such that $k' = m \times k \bmod N$. To discriminate the possible values of m we have the information extracted from the side-channel execution traces of $\texttt{EEA}(k', N)$. In a preliminary test, we decided to test all non-zero values of m of binary length 32 or less. We then select a mask candidate m if $\texttt{EEA}(m \times k \bmod N, N)$ has the exact same number of iterations than the side-channel execution trace of $\texttt{EEA}(k', N)$ and if the number and positions of the large iterations (*i.e.* iterations for which $\Delta_{r_0,r_1} > 5$) also match. We apply this process to the first 10 traces of the signature acquisition campaign. At the end of this 32-bit brute-force, we end-up with:

- 3 mask candidates for the first trace;
- 38035 mask candidates for the 10^{th} trace;
- a unique mask candidate for the other 8 traces.

The 10^{th} trace only shows a single large iteration ($\Delta_{r_0,r_1} > 5$), the first trace has 4 such iterations, all other traces have 4 or more large iterations. Since our mask detection is mainly based on the position of these large iterations, it makes quite sense that the more an `EEA` execution shows large iterations, the fewer mask candidates are left. The fact that, for all 10 traces, there exists at least one mask candidate left is very promising and comforts ourselves in the assumption that Infineon uses a multiplicative mask countermeasure with a 32-bit mask.

Now we have seen in the previous section that a finer-grain timing leakage appears for each iteration. We can use this leakage to try and discriminate one mask candidate among several. To do so, we automatically identify the timing leakage start and end positions (as illustrated in Figure 2.11) for each iteration of the first 10 ECDSA side-channel execution traces. This detection, without being perfect, works well enoug for our test. We then have, for all iterations, the length (in time samples) of the timing leakage area. We previously observed that this area length is proportional to the value Δ_{r_0,r_1}. A good way to capture this linear relation is to use the Pearson correlation [8], we then compute the Pearson correlation ρ between the sequence of Δ_{r_0,r_1} values computed from the execution simulation of $\texttt{EEA}(m \times k \bmod N, N)$ for a mask candidate m and the observed sequence of time sample lengths extracted from the side-channel execution trace of EEA(k', N) (and denoted $\{\mathcal{L}_i\}_{0<i\leq n}$ for an n iterations EEA).

Figure 2.12 displays the Pearson correlation results for each candidate of each trace [1] (each trace has a different color, *e.g.* the blue dots represent the 10th trace with all its 38K mask candidates). It appears clearly that, for each trace, a single mask candidate stands out clearly with a high correlation ($\rho > 0.7$) while all other candidates stay well below $\rho < 0.5$.

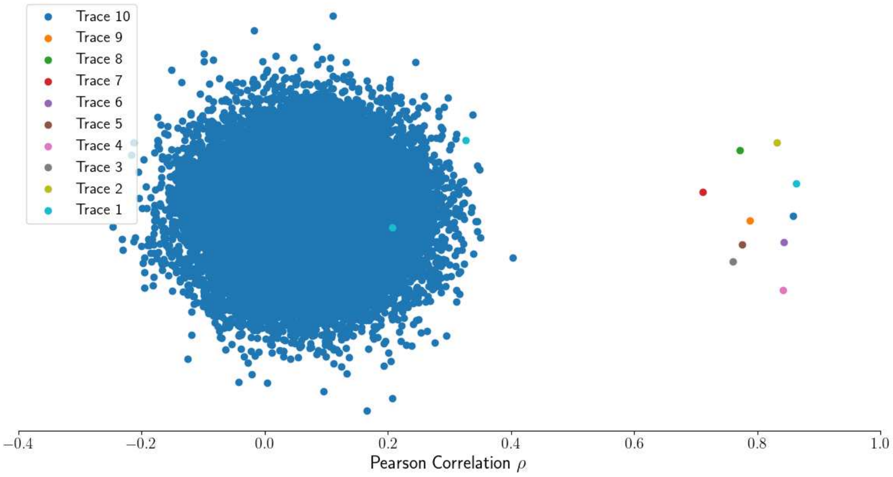


Figure 2.12: Feitian A22 JavaCard – Modular Inversion Multiplicative Mask Brute Force – Pearson Correlation Results $\rho(\{\Delta_{r_0,r_1 i}\}_{0<i\leq n}, \{\mathcal{L}_i\}_{0<i\leq n})$

This experiment, without being a formal proof, finishes to convince us that we have understood the masking countermeasure of Infineon to protect the modular inversion of the nonce.

2.4 Conclusions

Algorithm 1 is validated, the nonce is blinded with a multiplicative mask of size 32 bits (and odd). Moreover, we identified a timing leakage proportional with $\Delta_{r_0,r_1} = len(r_0) - len(r_1)$ (r_0 and r_1 appearing in line 4 of Algorithm 1). In the next chapter, we will consider a side-channel attacker able to soundly (*i.e.* without error) extract the value Δ_{r_0,r_1} for each and every iteration of an EEA execution.

[1] We randomly spread the correlation results over the y-axis following a Gaussian distribution of mean 0 and standard deviation 10^{-2} for illustration purpose.

Chapter 3

Input-Recovery Attack on the Extended Euclidean Algorithm

Let us recall the EEA algorithm here:

Algorithm 1: Extended Euclidean Algorithm for Modular Inversion (repeated from page 26)

```
   Input  : $v, n$: two positive integers with $v \leq n$ and $gcd(v, n) = 1$
   Output: $v^{-1}$ mod $n$: the inverse of $v$ modulo $n$
 1 $r_0, r_1 \leftarrow n, v$
 2 $t_0, t_1 \leftarrow 0, 1$
 3 while $r_1 \neq 0$ do
 4     $q \leftarrow$ div$(r_0, r_1)$
 5     $r_0, r_1 \leftarrow r_1, r_0 - q.r_1$
 6     $t_0, t_1 \leftarrow t_1, t_0 - q.t_1$
 7 end
 8 if $t_0 < 0$ then
 9     $t_0 \leftarrow t_0 + n$
10 end
11 return $t_0$
```

This algorithm is applied to the input pair (k', N), where k' is the unknown masked nonce and N is the public elliptic curve order (of elliptic curve P256 [23] for our tests).

3.1 First Observations

In the previous chapter, we identified a timing leakage that gives us, for each iteration of the EEA, the binary length difference between r_0 and r_1 (appearing as inputs of the div call in Algorithm 1, line 4) denoted Δ_{r_0,r_1}. Let us take a step back and consider the more general case where the side-channel adversary is able to recover a function f of the input-pair (r_0, r_1) for each iteration of the while loop.

It is quite easy to see that if the function f outputs enough information then an attacker would be able to recover the two inputs (v, n) of the EEA. For instance, a simple case is when the attacker is able to recover the quotient q (appearing in Algorithm 1, line 4) of the euclidean division of r_0 by r_1 without error (*i.e.* $f(r_0, r_1) = \frac{r_0}{r_1} = q$). From the sequence of quotients $\{q_i\}_{0<i\leq\ell}$, there is a simple way to recover the two inputs (v, n) of the `EEA`:

$$\begin{pmatrix} n \\ v \end{pmatrix} = \prod_{i=1}^{\ell} \begin{pmatrix} q_i & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

In fact, since one of the two inputs is already known to the attacker ($n = N$, the order of the elliptic curve), one can do even better than that. We only need about half of the quotient sequence $\{q_i\}_{0<i\leq\lceil\frac{\ell}{2}\rceil}$ as they are enough to estimate a good approximation of the rational number $\frac{n}{v}$ using continued fractions:

$$\frac{n}{v} \sim q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{\ddots}}}$$

Since n is known, v can be correctly estimated by computing the right hand side of the equation (up to $q_{\lceil\frac{\ell}{2}\rceil}$) and estimating the value of v with a small margin of error [1].

However the above solution requires that the attacker has, at least, the knowledge of half of the quotient sequence. Also, this information should not contain too many errors: a brute force could cope with only a small amount of erroneous or missing quotients.

In the case where the attacker has less information (as in our case where $f(r_0, r_1) = \Delta_{r_0,r_1}$) or an information that does not contains the quotient value, there is no clear way to mount an attack. Worse, if the attacker has access to more information than the quotient value, the solution is to reduce this information to the quotient value and somehow loose the extra information that could be helpful to handle some errors. When considering side-channel attack, the information is often noisy (even with timing leakages) and handling errors is usually critical for the practicality of the attack.

We then created a more flexible attack process that can take a large class of leakage functions f. It is based on the following assumption: the leakage function $f(r_0, r_1)$ can be properly estimated with the knowledge of the *most-significant-bits* (msb for short) of r_0 and r_1. For instance this is the case for $f(r_0, r_1) = \Delta_{r_0,r_1}$ or $f(r_0, r_1) = q$ where, for long integers r_0, r_1, only their msb will dictate the output of f (with high probability).

With this assumption, an attacker would be able to run through all the possible values that the s-msb of k' can take (*i.e.* 2^s candidates) and, for each of them, predict the leakage (given a leakage function f) for the few first iterations of $\texttt{EEA}(k', N)$. By comparing the predicted leakage to the observed leakage, the attacker might descriminate a small enough set of candidates. The ℓ remaining candidates are then extended with all possible values for the next s bits, creating a new list of candidates of size $2^s\ell$, each candidate being of length $2s$ bits. The prediction can now go further through the iterations of $\texttt{EEA}(k', N)$, and so on.

Before materializing this idea into an attack process, let us first recall the Jebelean's stop condition in the following Theorem 1. It will be very useful in telling us, for each candidate msb of k', up to which `EEA` iteration we can predict meaningful leakage. In Theorem 1:

[1] This solution was actually proposed by Franck Rondepierre and Guénaël Renault from ANSSI, during the responsible disclosure.

- the quotient sequence refers to the EEA quotient sequence;
- the $\{(a_i, a_{i+1})\}$ pairs sequence are the value taken by (r_0, r_1) in Algorithm 1;
- the $\{(v_i, v_{i+1})\}$ pairs sequence are the value taken by (t_0, t_1) in Algorithm 1;
- the $\{(u_i, u_{i+1})\}$ pairs sequence are the second Bézout coefficients sequence.

Theorem 1 ([14]' Theorem 1) *Let $a > b > 0$ be integers. Then the k-length quotient sequences of (a, b) and $(a2^h + A', b2^h + B')$ match for any $h > 0$ and any $A', B' < 2^h$*

if and only if, for all $i \leq k$

$$a_{i+1} \geq -u_{i+1} \text{ and } a_i - a_{i+1} \geq v_{i+1} - v_i \text{ and } i \text{ even}$$
$$\text{or}$$
$$a_{i+1} \geq -v_{i+1} \text{ and } a_i - a_{i+1} \geq u_{i+1} - u_i \text{ and } i \text{ odd}$$

From Theorem 1, one can easily tell when the EEA on the s-msb of (k', N) coincide (at least in terms of quotient sequence) with the EEA on (k', N) without needing more knowledge than the s-msb of (k', N).

3.2 Building a Generic Attack Algorithm

From the above observations, a basic attack algorithm can be sketched.

1. Select the brute-force parameter: a positive integer m.
2. Initialize an integer $t = m$ and a list of integers $L = \{0\}$.
3. Denote by N_t the integer composed by the t most-significant-bits of N and L' an empty list.
4. For all $x \in L$, for all $g \in \{0, \cdots, 2^m - 1\}$:
   (a) $v = x.2^m + g$;
   (b) execute the EEA on (N_t, v) and, *iff* $N_t < N$, stop when the Jebelean's stop condition is met;
   (c) for each executed iteration of the truncated EEA, predict the leakage and compare it to the extracted leakage;
   (d) if the comparison is successful, add v to L'.
5. If $N_t = N$ (*i.e.* t equals the bit length of N) then return L',
   else t is increased by m, L takes L' and go back to step 3.

Figure 3.1: A Generic Input-Recovery Side-Channel Attack on the EEA

Although arguably lacking elegance, the above attack algorithm has many advantages:

- it is completely generic with respect to the extracted leakages (one does not need to relate the leakage to the EEA inputs as it was necessary with the previous analytical approach);

- the matching process between the extracted leakage and the predicted leakage can be easily done so that some reasonable margin of errors is acceptable (in exchange to some computational effort). For instance, if for an iteration the leakage information is not extractable (say the side-channel trace is not readable), all v candidates for this iteration are accepted;

- at step 3, when t equals the bit length of n, the algorithm has merely checked the first half of the EEA iterations [2]. The second half is then checked in a single pass and then, with overwhelming probability, a unique element is added to L' during this last pass (or none, if there are too many errors in the extracted leakages) [3].

Let us now remark that this algorithm is more efficient as m is small. Intuitively, one can see that a small m provides a finer grain: by setting $m = 1$, the algorithm will remove wrong candidates as soon as possible and avoid unnecessary calls to the truncated `EEA`. Our simulations show that for small values of m, the impact is not huge but gets more and more important as m increases ($m = 8$ is already much more expensive than $m = 4$). So on the one hand, $m = 1$ should be the most interesting choice but on the other hand, the 2^m inner loop (step 4.) is the easiest part of the algorithm for parallelization. Hence, in our practical tests, $m = 4$ was the best choice.

This basic attack algorithm has however three issues that all come from the use of the Jebelean's stop condition:

- in [14, Theorem 1], the (a, b) integers must be strictly greater than zero. In our attack algorithm this means that – for the first iteration – m must be greater than the binary length difference between the two inputs (k', N) of the `EEA`, denoted $\Delta_{N,k}$. This binary length difference might not be known (depending on the leaked information), a first solution would be to set m to be large enough so that it has great chances to be larger than $\Delta_{N,k}$. However, as mentioned before, m should stay small (and is set to 4 in our tests). Our solution will then be to test the attack iteratively for an increasing value of $\Delta_{N,k}$ and handle the first iteration differently than the others (with a larger value for m when necessary);

- for specific choices of (a, b) the Jebelean's stop condition is met too soon and the leakage is not tested against the prediction. This artificially increases the candidate set with pathological cases and then the overall attack complexity. Since the attack algorithm is virtually scanning over all possible values of the unknown `EEA` input k', these pathological cases will necessary arise. This happens when $a = b$; when the case arises, we simply remove a candidate v if $v = N_t$ for $t > 2^{32}$ (we could increase to 2^{64}). The other pathological case is when very large quotients appear at the end of the quotient sequence. Similarly to the first case, we can add a threshold T_q on the quotient value and remove a candidate that creates a quotient sequence where a quotient is larger than T_q. Another, more subtle, way to bypass this issue is to select the next candidate based on its last quotient value (the smallest first). This is what is presented in the next algorithm 2;

- Jebelean's stop condition ensures that the quotient sequence is preserved. However, when dealing with leaked information different from the quotient values, (*e.g.* Δ_{r_0,r_1}), this might not be enough. This is the most problematic issue of using Jebelean's stop condition and we did not find a perfect solution for that. Our idea is to estimate when we are *far* from

[2] This is actually a corollary of [14, Theorem 1] since, during the `EEA` (see Algorithm 1), $|t_1|$ will become greater than r_1 around the middle of execution and then the stop condition will always be met, whatever the number of msb considered.

[3] This observation has led us to actually use a *depth-first* version of the attack algorithm.

the stop condition with a simple metric. If we are far enough, one can check the leakage against the prediction. Otherwise, if we are too close to the stop condition, then we can only rely on the quotient values: we extract the list of all possible quotient values that could lead to the observed leakage and check if the predicted quotient is in the list. For instance, for a leakage function $f(r_0, r_1) = \Delta_{r_0,r_1}$, all predicted quotients of binary length Δ_{r_0,r_1} or $\Delta_{r_0,r_1} + 1$ are accepted.

Taking all these different remarks into account, the following Algorithm 2 describes the full attack process. It is based on the so called *Truncated Extended Euclidean Algorithm*, defined in Algorithm 3, that stops when the Jebelean stop condition is met.

Algorithm 2: A Generic Attack Algorithm

```
Input  : $\{m_{start}, m_{next}\}$: a pair of brute-force parameters,
Input  : $\{\mathcal{L}_i\}_{0 \le i < n}$: the leaked information from the execution of EEA$(k', N)$
Input  : $f(.)$: the leakage function
Output: $c$: a candidate for $k$ or Error if none found
$L \leftarrow \{(0, 0, -1)\}$
while $True$ do
    $(c, t, q) \leftarrow L.pop()$                    // pop the first element of the list $L$
    if $t = len(N)$ then                              // $len(x)$ returns the binary length of $x$
        return c
    else if $t = 0$ then
        $s, m \leftarrow 1, m_{start}$
    else
        $s, m \leftarrow 0, min(m_{next}, len(N) - t)$
    end
    $N_{t+m} \leftarrow msb_{t+m}(N)$                                        // $t + m$ msb of N
    for $x \in [s, 2^m]$ do
        $v \leftarrow c.2^m + x$
        $success, \hat{q} \leftarrow$ TEEA$(v, N_{t+m}, \{\mathcal{L}_i\}_{0 \le i < n}, f)$          // call to Algorithm 3
        if $success$ then
            Add tuple $(v, t + m, \hat{q})$ to $L$ such that $L$ stays sorted $w.r.t.$ last element of the
              tuple $(\hat{q})$.
        end
    end
end
return Error
```

Algorithm 3: Truncated Extended Euclidean Algorithm

```
Input   : $v$: a candidate integer
Input   : $N_t$: the $t$ msb of $N$
Input   : $\{\mathcal{L}_i\}_{0 \leq i < n}$: the leaked information from the execution of EEA($k'$, $N$)
Input   : $f(.)$: the leakage function
Output: $success$: True if the leakage matches the prediction, False otherwise
Output: $\hat{q}$: the last quotient of the predicted quotient sequence
$t \leftarrow len(N_t)$                    // $len(x)$ returns the binary length of $x$
$T \leftarrow len(N)$
if $N_t = v$ then
    if $t > 32$ then return (False, -1)
    else return (True, 1)
end
$r_0, r_1 \leftarrow N_t, v$
$t_0, t_1 \leftarrow 0, 1$
$s_0, s_1 \leftarrow 1, 0$
$step \leftarrow 0$
$Failure \leftarrow 0$
while $r_1 \neq 0$ do
    $q \leftarrow r_0 // r_1$
    $\hat{\mathcal{L}} \leftarrow f(r_0, r_1)$            // Leakage Prediction from $(r_0, r_1)$
    $r_0, r_1 \leftarrow r_1, r_0 - q.r_1$
    $t_0, t_1 \leftarrow t_1, t_0 - q.t_1$
    $s_0, s_1 \leftarrow s_1, s_0 - q.s_1$
    if $t < T$ then                        // Test Jebelean's stop condition
        if $step \bmod 2 = 1$ then
            $stop\_cond \leftarrow not\ ((r_1 \geq -s_1)\ and\ ((r_0 - r_1) \geq (t_1 - t_0)))$
            $stop\_dist \leftarrow min(len(r_1) - len(-s_1), len(r_0 - r_1) - len(t_1 - t_0))$
        else
            $stop\_cond \leftarrow not\ ((r_1 \geq -t_1)\ and\ ((r_0 - r_1) \geq (s_1 - s_0)))$
            $stop\_dist \leftarrow min(len(r_1) - len(-t_1), len(r_0 - r_1) - len(s_1 - s_0))$
        end
        if $stop\_cond$ then
            return (True, q)
        end
    end
    if $stop\_dist > Thr\_dist$ then
        if $\hat{\mathcal{L}} \neq \mathcal{L}_{step}$ then $Failure \leftarrow Failure + 1$
    else
        $L_q \leftarrow \{\hat{q}\ s.t.\ \mathcal{L}_{step}\}$ // List all possible quotients compatible with $\mathcal{L}_{step}$
        if $q \notin L_q$ then $Failure \leftarrow Failure + 1$
    end
    if $(t < T\ and\ Failure > Thr\_Failure\_FH)\ or\ (t = T\ and\ Failure > Thr\_Failure)$
     then return (False, -1)
    $step \leftarrow step + 1$
end
if $step = n$ then return (True, 0)
else return (False, -1)
```

In Algorithm 2, adding a new tuple to the list is done such that the list stays sorted with respect to the last quotient value. For equal values of quotients, the strategy is *last added first out*, but it is not clear if this is the best approach.

In Algorithm 3, three threshold values must be chosen (the Thr_dist, line 30, the $Thr\_Failure\_FH$ and $Thr\_Failure$, line 36). The choice of these thresholds is clearly dependent on the nature (and quantity) of leaked information. In all our tests, we set Thr_dist value to 8, which seems to be enough in our cases. $Thr\_Failure\_FH$, the number of accepted errors during the first half for the EEA iterations was set to 0 while $Thr\_Failure$, the total number of accepted errors was arbitrary set to 10 (as mentioned before, many leakage extraction errors in the second half of the EEA execution can be tolerated).

3.3 Simulation Experiments

With the above attack process we can simulate the attack on random inputs k' of size 256 bits. We did that assuming the leakage of the quotient sequence (*i.e.* at iteration i, $f(r_0, r_1) = \mathcal{L}_i = q_i$). The attack process was always successful (which is not so surprising since we already have seen that the quotient sequence if enough to recover the EEA inputs), the complexity of the attack is given with respect to the number of calls to the Truncated EEA in Figure 3.2. In about 500 tests, the observed attack complexity ranges from 2^{20} to 2^{30} calls. Interestingly, this complexity seems to be directly related to the maximum quotient value appearing in the first half of the quotient sequence, more precisely the log complexity seems proportional to the log of the maximum quotient value.

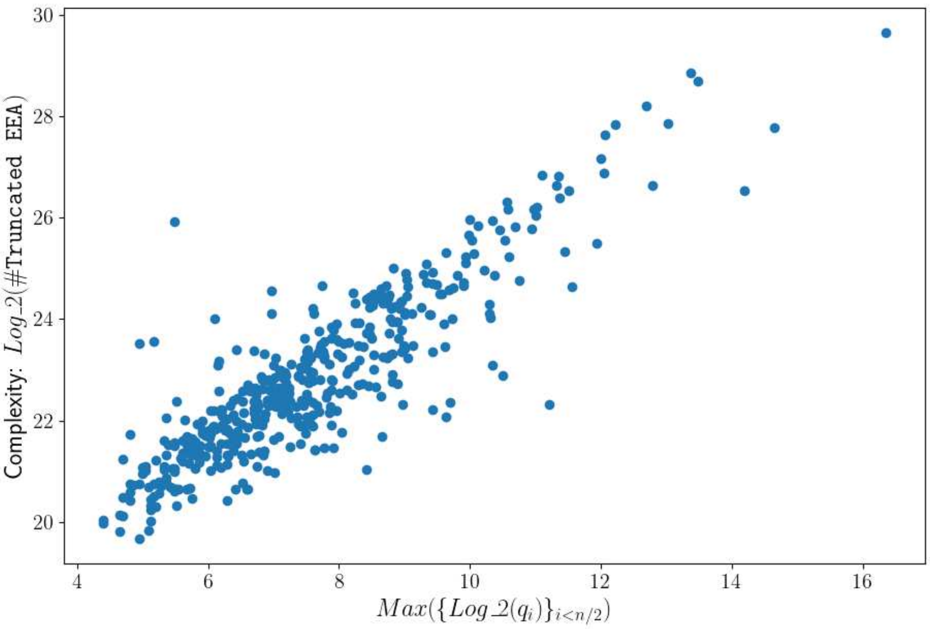


Figure 3.2: Simulation of the Generic Attack – $\mathcal{L}_i = q_i$

Of course, we also simulated the attack assuming $f(r_0, r_1) = \Delta_{r_0,r_1}$. None of these simulations ever ended: the attack with less information than the quotient seems out of reach of our computational power.

3.4 Conclusions

We have built a generic side-channel attack algorithm targeting the EEA. However, the identified leakage does not seem to be enough for a practical input recovery. We let for future work the theoretical analysis of this attack algorithm and the study of the minimum quantity of leaked information necessary for the attack to succeed in practice.

So while all this was incredibly fun to do, at this point it seems a complete waste of our time!

Fortunately this attack process was not built in vain. Indeed we eventually were able to improve our understanding of the side-channel traces and Infineon EEA implementation. This is described in the next chapter.

Chapter 4

Full Reverse-Engineering of Infineon EEA

While working on a generic side-channel attack process, we realized that the identified leak will not be enough for the attack to work in practice. We then came back to the side-channel traces and looked for additional leakages.

4.1 More Timing Leakages

In fact, it was pretty clear that other timing leakages were present in the traces. Figure 4.1 first subfigure shows 100 superposed subtraces corresponding only to odd iterations of EEA executions where $\Delta_{r_0,r_1} = 2$. Similar behavior can be found with the even iterations. One can see that, if the beginning of the subtraces are correctly aligned, the end of the iteration is desynchronized.

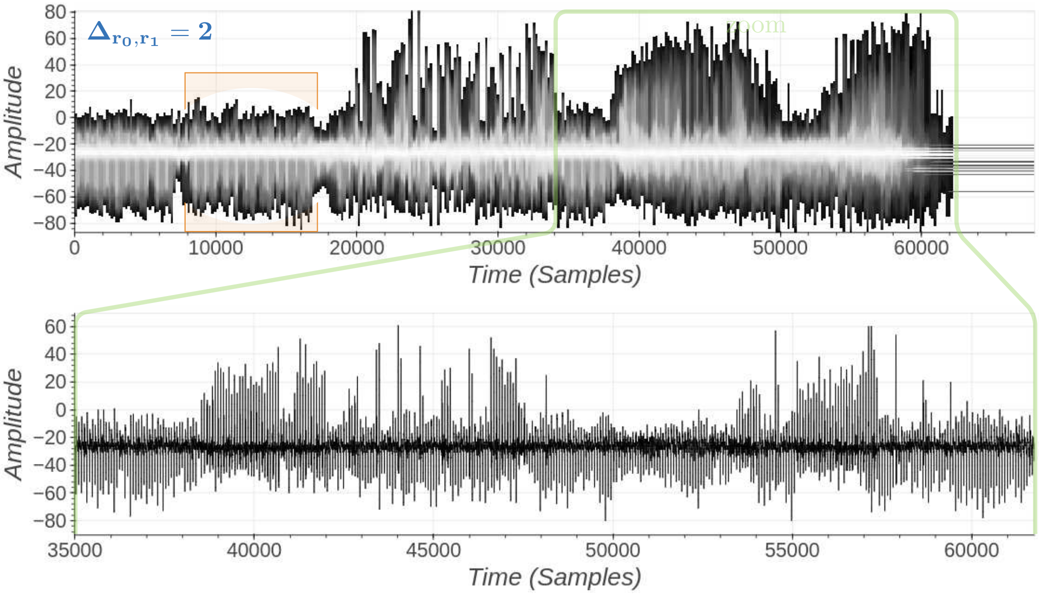


Figure 4.1: `Feitian A22 JavaCard` – ECDSA Signature Verification – $\Delta_{r_0,r_1} = 2$ – 100 Superposed Odd Iterations (Top) - A Single Trace, Zoom on the New Leakage Area (Bottom)

Focusing on the end of the computation, Figure 4.2 displays various shapes that can be found at the end of the odd iterations with $\Delta_{r_0,r_1} = 2$. This might well be additional leakage, but also could come from another side-channel countermeasure that forces de-synchronization. Again, let us hope for the best (adversary-wise) and try to give some sense to this behavior.

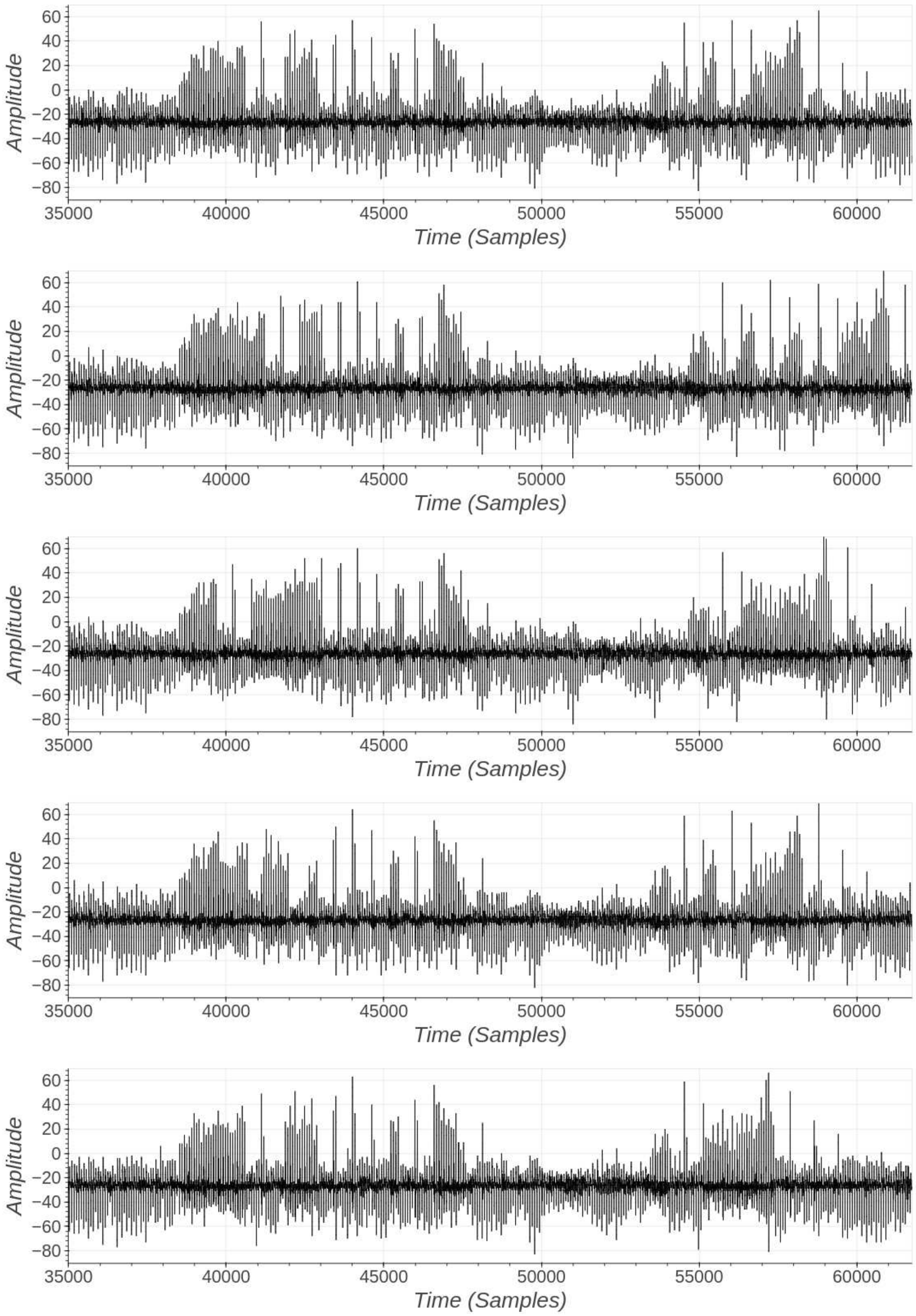


Figure 4.2: `Feitian A22 JavaCard` – ECDSA Signature Verification – Odd Iteration with $\Delta_{r_0,r_1} = 2$ – Various Shapes in the EM Signal

4.2 A Deep Dive into Euclidean Division Algorithms

4.2.1 First Steps

Let us first prepare the subtraces for the analysis, we will need to have well aligned subtraces and therefore improve our resynchronization algorithm. Figure 4.3 displays the side-channel trace of the first iterations of an `EEA` execution. One can see that the first iteration does not have the same shape as the others. This is always the case, we decided to remove the first subtrace from our analysis, in fact we will always consider that no information is extracted from the first iteration [1].

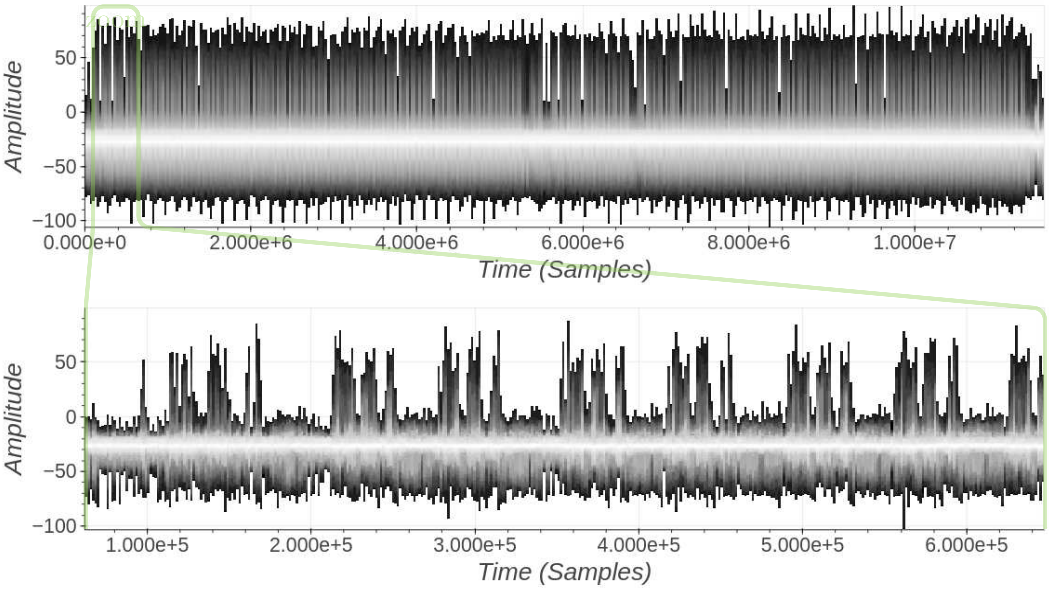


Figure 4.3: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Full Trace (Top) - Zoom around the First Iterations (Bottom)

Figure 4.4 illustrates the new detection algorithm: it now enters inside the iterations to automatically detect eight different anchors related to the four different patterns contained in an iteration. The first anchor (the blue dot) represent the start of the iteration. This process is more costly than before but also much more precise.

[1] In fact, working specifically on the first iteration would certainly show some sensitive leakage, this would require extra work to correctly capture it.

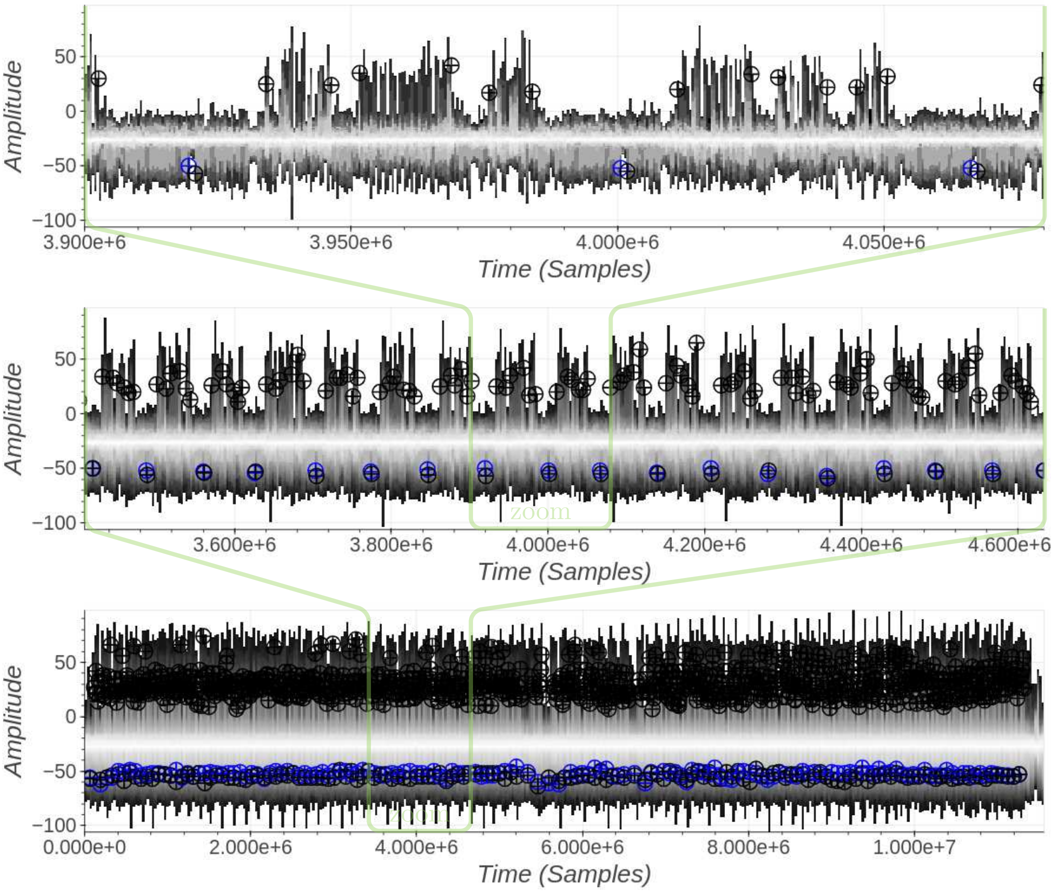


Figure 4.4: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation Pattern Detection – Two Consecutive Iterations (Top) - Several Iterations (Middle) - Full Trace (Bottom)

Thanks to the iteration detection algorithm, the `EEA` execution trace can be split into sub-traces, each of them has its sub-patterns realigned. Figure 4.5 displays the re-alignment result. The four patterns are identified in the first subfigure. The first one is already known, its length is directly related to the value Δ_{r_0,r_1}. The second one appears to be constant-time (for all odd and for all even operations). The two last patterns are not constant time and they are the focus of our analysis. We first restrict our work to the first one, as selected on the second subfigure. Furthermore, we will first only consider odd iterations as they seem to give a richer information than the even ones.

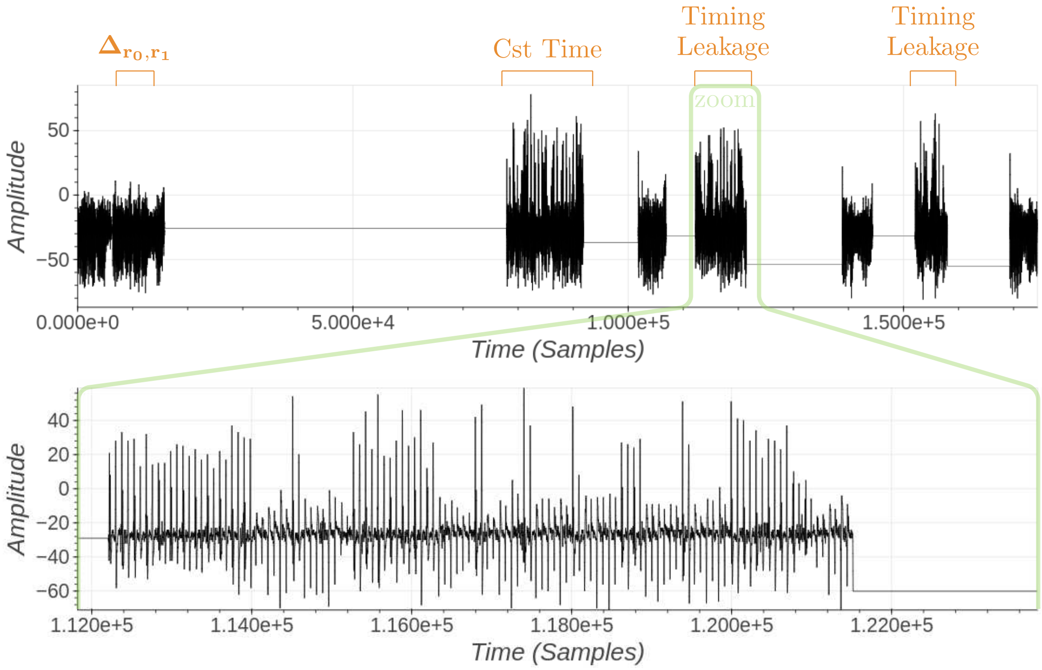


Figure 4.5: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Resynchronized Iteration – Full Iteration (Top) - Zoom (Bottom)

For reasons beyond our understanding, the shapes of two similar iterations (in terms of number of peaks) are sometimes different. It indeed happens that an iteration is more *compact* than it should be. Figure 4.6 depicts two similar iterations, the first one being the compact version and the second one being the regular version. In fact this appears at very specific places in the `EEA` sequence.

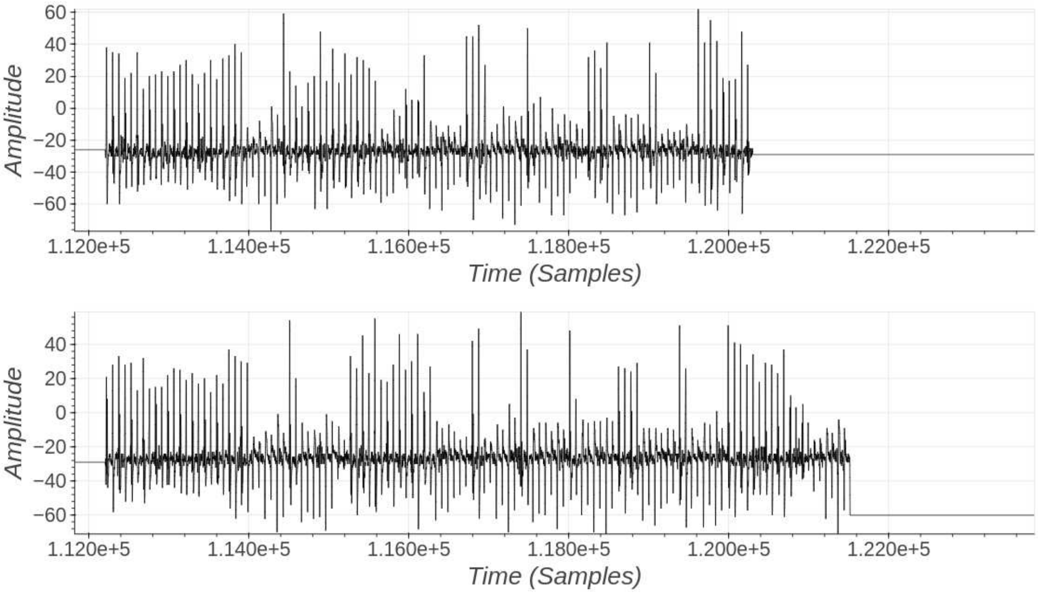


Figure 4.6: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Resynchronized Iteration – Compact Iteration (Top) - Regular Iteration (Bottom)

We observed that *compact* iterations occur when the following condition is met:

$$r_1 < 2^{31} \text{ or } r_1 > 2^{256-16}$$

Figure 4.7 illustrates, on a full `EEA` execution, the parts where *compact* iterations are found. These iterations bear the same information as regular ones, however we also remove them from the analysis for now as they would just make things more complicated.

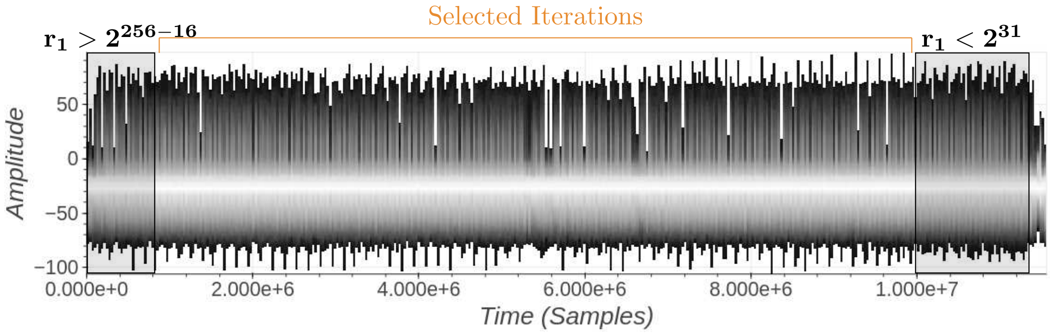


Figure 4.7: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Full Trace with *Compact* Areas

Finally, our experimental data is ready for the analysis, cleaned from artifacts and limited to a single case: odd regular iterations. Our next observation gives us some new information about the new leakage. Figure 4.8 shows three iterations (limited to the unknown leakage area) for which both Δ_{r_0,r_1} and q are equal to 2. These three iterations have clearly different shapes. Hence, the leakage provides more information than the value of q (at least in this case). This is pretty promising as we know already that if q is leaked, the attack will work.

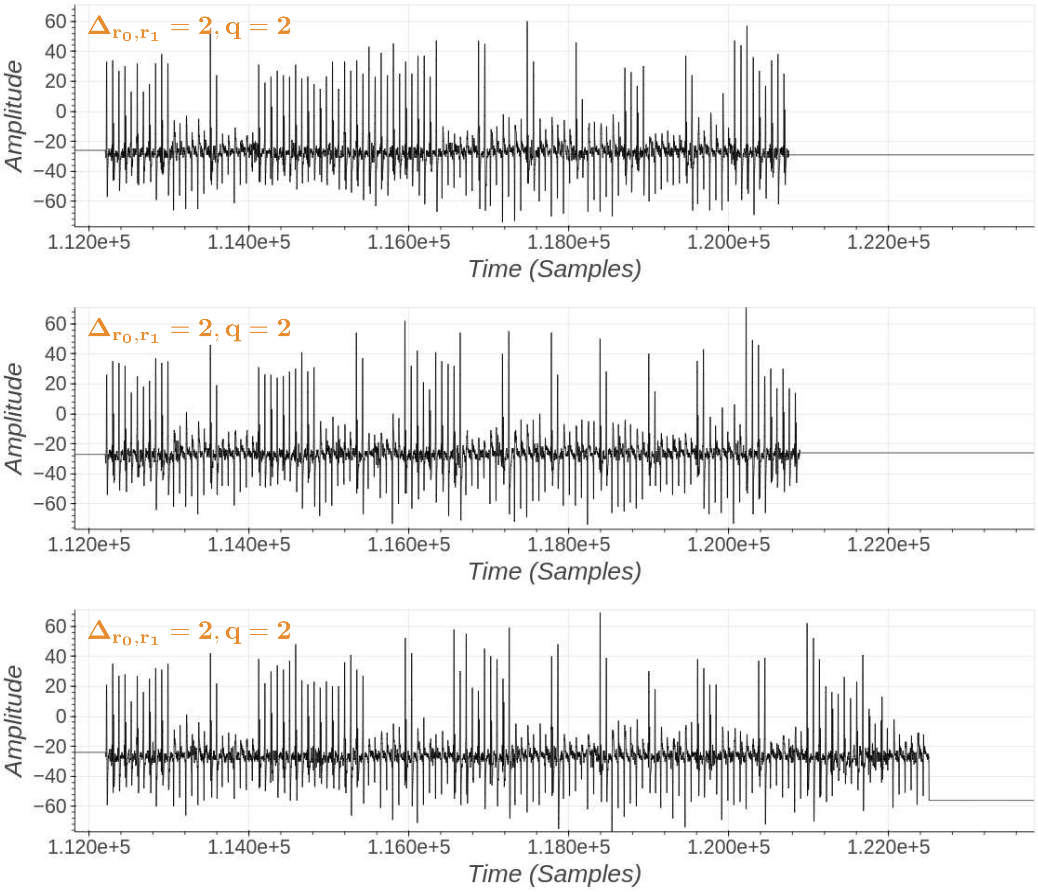


Figure 4.8: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Resynchronized Iteration – $\Delta_{r_0,r_1} = 2$, quotient $= 2$ – Case 1 (Top) - Case 2 (Middle) - Case 3 (Bottom)

This good news is diminished by the observation depicted in Figure 4.9 where two iterations look similar but are related to different values of q. It means that the new leakage (and in fact the entire leakage, including Δ_{r_0,r_1}) does not allow to fully distinguish the quotient q.

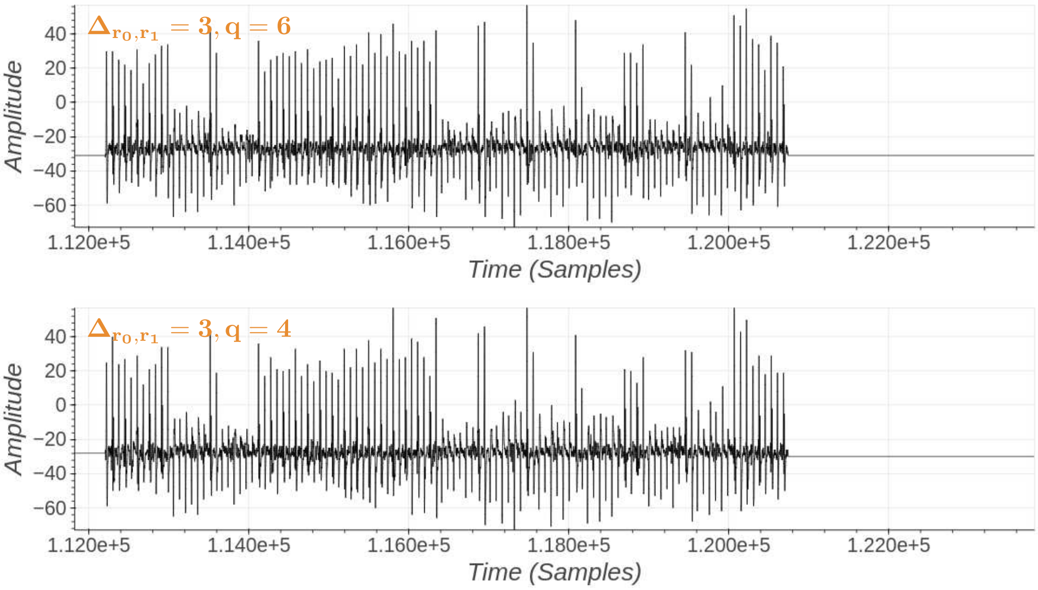


Figure 4.9: `Feitian A22 JavaCard` – ECDSA Signature Verification – EEA Computation – Resynchronized Iteration – $\Delta_{r_0,r_1} = 3$, quotient $= 6$ (Top) - $\Delta_{r_0,r_1} = 3$, quotient $= 4$ (Bottom)

4.2.2 Failed Attempts

Only considering the first 100 `EEA` traces from the ECDSA signature verification acquisition campaign (*i.e.* there is a total of 13893 `EEA` iterations), we manually regroup matching subtraces (for odd resynchronized iterations) for some small parameters choices $(\Delta_{r_0,r_1}, q) \in \{(1,1),(1,2),(1,3),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7)\}$.

This is a long and error prone process:

- for a parameter choice (Δ_{r_0,r_1}, q), we extract all subtraces that match the parameters (omitting the *compact* iterations, and only selecting the odd iterations);
- from this set of subtraces, manually divide them into matching groups where all subtraces look alike (this step requires a comfortable visualization tool). For our parameter choices, the number of groups can be 1 (*e.g.* $(\Delta_{r_0,r_1}, q) = (2,7)$), 2 (*e.g.* $(\Delta_{r_0,r_1}, q) = (1,1)$) or 3 (*e.g.* $(\Delta_{r_0,r_1}, q) = (2,2)$, see Figure 4.8).

Our first direction of research was to enumerate all euclidean division algorithms we could think of and see if one of them could explain these groups. By running a candidate division algorithm on the inputs (r_0, r_1) corresponding to all iterations of a set of subtraces, we produced labels (according to some inner computation or state of the candidate division).

We started with classical textbook divisions (long division, binary divisions, Knuth D algorithm, etc.) and for each one, tried to tweak it into something that would divide our sets into correct groups. It sometimes worked for some groups but always failed to scale. We even dug up an old patent from Infineon [6, 7] that explains how to execute the `EEA` without directly manipulating the sequence of quotients by leveraging the access to a long integer modular reduction hardware co-processor. This patent seemed like a great clue and we tried everything to squeeze it into what we observed ... profitless.

The most promising results were found with the most simple approach. The basic idea (which is at the basis of fast long division algorithms) is that when dividing two long integers, we get a good approximation of the quotient with the division of their most-significant-bits (msb for short). Depending on the number of msb selected for the approximate division, the approximation of the quotient is more of less precise and the effort to correct it into the real quotient could be the source of the timing leakage. This effort would be related to the distance between the real quotient and first approximation. Based on this simple idea we could properly divide some sets of subtraces and find quite good success with the other sets (the groups are not perfectly found but the error is small). However this approach does not scale: the more complicated become the sets (*i.e.* the larger the parameters (Δ_{r_0,r_1}, q)) the further go our observations from the simulations.

These attempts were excessively time consuming and did not allow to conclude on the real source of leakage. They however demonstrated that the observed timing leakages were clearly depending on the inputs values (r_0, r_1) since we managed to divide some sets almost perfectly. This was enough for us to not give up.

However the only reasonable direction left to us seemed even longer and more painful...

4.2.3 Perseverance is the Key

When we are not smart enough we need to be persevering. Our next approach was to:

1. for each parameters choice (Δ_{r_0,r_1}, q), enumerate all groups $\{G_{(\Delta_{r_0,r_1},q),i}\}_i$ (as before);
2. for each group $G_{(\Delta_{r_0,r_1},q),i}$, identify all associated inputs pairs (r_0, r_1);
3. try and find a function f of (r_0, r_1) that outputs its correct group label i, otherwise just keep the list of all input pairs.

This process looks actually very close to the our previous attempts, the crucial difference is that the function f does not have to be an euclidean division and the simplest the better. Here we are not trying to bridge the gap between the observation and a division algorithm, we are simply looking for a (set of) leakage function.

Of course, we started with the most simple set of subtraces, which happen to be $(\Delta_{r_0,r_1}, q) \in \{(1,1),(1,2)\}$. These two sets have exactly 2 groups and we quickly found the following matching function:

$$f : (r_0, r_1) \to \begin{cases} 0 & len(r_1) > len(r_0 - 2^{\Delta_{r_0,r_1}} * r_1) \\ 1 & \text{otherwise} \end{cases}$$

In fact the function works for all our observations for the following sets $(\Delta_{r_0,r_1}, q) \in \{(1,1),(1,2),(1,3),(2,3),(2,4),(2,5),(2,7),(3,7)\}$ (where the sets $(1,3)$ and $(2,7)$ have only a single group).

Then we found, for $(\Delta_{r_0,r_1}, q) \in \{(2,6),(3,4),(3,6),(3,8),(3,11),(3,12)\}$, that the following function does the trick.

$$f : (r_0, r_1) \to \begin{cases} 0 & len(r_1) - len(r_0 - 2^{\Delta_{r_0,r_1}} * r_1) = \Delta_{r_0,r_1} \\ 1 & len(r_1) - len(r_0 - (2^{\Delta_{r_0,r_1}} + 2^{\Delta_{r_0,r_1}-1}) * r_1) = \Delta_{r_0,r_1} \\ 2 & \text{otherwise} \end{cases}$$

Furthermore, we managed to handle the more complex set $(2,2)$ (with three different groups) by adding another case to the function:

$$f : (r_0, r_1) \to \begin{cases} 0 & len(r_1) - len(r_0 - 2^{\Delta_{r_0,r_1}} * r_1) = \Delta_{r_0,r_1} \\ 1 & len(r_1) - len(r_0 - (2^{\Delta_{r_0,r_1}} + 2^{\Delta_{r_0,r_1}-1}) * r_1) = \Delta_{r_0,r_1} \\ 2 & len(r_1) - len(r_0 - (2^{\Delta_{r_0,r_1}} - 2^{\Delta_{r_0,r_1}-1}) * r_1) = \Delta_{r_0,r_1} \\ 3 & \text{otherwise} \end{cases}$$

Once we get there, and even though many more sets had to be investigated, we could see a pattern in building a generic function f. Funny enough, when trying to write down the generic function f, we ended up with an euclidean division !

4.2.4 Infineon Euclidean Division Algorithm

Algorithm 4: Euclidean Division Algorithm

```
Input  : $a, b$: two positive integers
Output: $q$: the quotient of the division of $a$ by $b$

$r \leftarrow a$
$\ell \leftarrow len(r) - len(b)$            // $len(x)$ returns the binary length of $x$
$q \leftarrow 0$
while $\ell \geq 0$ do
    $g \leftarrow \mathtt{sign}(r).2^{\ell}$   // sign($x$) returns -1 if $x < 0$, 1 otherwise
    $r \leftarrow r - g.b$
    $q \leftarrow q + g$
    $\ell \leftarrow len(r) - len(b)$
end
if $r < 0$ then
    $q \leftarrow q - 1$                     // q is the quotient
    $r \leftarrow r + b$                     // r is the remainder
end
return $q$
```

Algorithm 4 is a slightly tweaked schoolbook long division algorithm in base 2, where intermediate remainders are allowed to be negative. We could not find a reference to this specific algorithm in the literature.

4.3 Summary of the Timing Leakages

The leaked sensitive information relates to the execution time of Algorithms 1 and 4. We summarize here the information inferred directly from the side-channel traces:

- the successive patterns of the side-channel trace allow to identify the iterations of Algorithm 1 while loop (and then their number);
- for each such iteration, a first timing leakage is proportional to the binary length difference of the (a, b) inputs of Algorithm 4, *i.e.* the value ℓ computed at line 2 of Algorithm 4, so-called Δ_{r_0,r_1} in the previous chapters;
- in the cases where this value ℓ is greater than 5, the process is slightly modified, the iteration takes significantly more time (these are the large iterations identified in Chapter 2). One interpretation might be that Algorithm 4 is executed only on the most-significant-bits of its inputs a and b. If $\ell > 5$, then the number of most-significant-bits considered is increased such that the result of Algorithm 4 stays correct;
- then, for all iterations of Algorithm 4 while loop, the side-channel trace shape changes when:
  - $\ell = 0$ compared to $\ell > 0$ (line 8 of Algorithm 4);
  - $r < 0$ compared to $r \geq 0$ (lines 5 and 10 of Algorithm 4);
- for reasons beyond our understanding, the *odd* and *even* EEA iterations are slightly different. For the *even* iterations, there is less information leaked (or it is harder to extract): the sign of r only leaks when $\ell > 0$ otherwise we do not get it.

Chapter 5

Key-Recovery Attack on ECDSA

We have finally understood the details of Infineon `EEA` implementation, making sense to the observed timing leakages. Let us try to apply the attack process developed in Chapter 3 on the newly identified leakages.

5.1 Input-Recovery Attack on the `EEA`

In the previous chapter, Section 4.3, we have precisely identified the sensitive leakage that we are able to extract from the side-channel traces. The generic attack algorithm decribed in Chapter 3, Section 3.2, Algorithms 2 and 3 can be applied straightforwardly. We first need to choose a way to correctly encode the leaked information. We simply use a tuple of variable length:

- the first cell encodes the iteration parity (0 for even iterations, 1 otherwise);
- the second cell encodes the Δ_{r_0,r_1} value;
- then, for each iteration of Algorithm 4, a cell stores the value $s * v$. where $s = \texttt{sign}(r)$ and $v = 1$ if $\ell = 0$ and $v = 2$ otherwise [1];
- the encoding supports the two following erasure cases (that can be simultaneous):
  - an encoding tuple of length 2 means that only the iteration parity and Δ_{r_0,r_1} value are known, the remaining is not readable (hence any simulated leakage matching with the Δ_{r_0,r_1} value is acceptable for this iteration);
  - if the Δ_{r_0,r_1} value is set to -1, this means that we could not properly extract the Δ_{r_0,r_1} value (then any value for Δ_{r_0,r_1} is acceptable).

Also, as observed in Section 4.2.1, the first iteration is considered unreadable and then is encoded as $(0, \Delta_{N,k})$, where $\Delta_{N,k}$ is the first (unknown) value of Δ_{r_0,r_1}. As explained in Section 3.2, the attack is iterated over increasing values of $\Delta_{N,k}$.

With this encoding we can run simulations and see if the leaked information allows for practical input-recovery attacks.

[1] Note that for even iterations, the leaked information is slightly different. The encoding follows a similar idea.

5.1.1 Simulation Experiments

The simulation experiments results are displayed in the following Figure 5.1, they represent about 50 attacks with the leakage function defined in the previous section. They appear in orange on the figure while the blue dots are the simulation results when the leakage function is directly the quotient value, taken from Chapter 3, Section 3.3, Figure 3.2.

As one can see, we limited our simulations to the case where the maximum quotient value in the first half of the EEA iterations is small (in fact we ensured that for the first half of the EEA iterations, $\Delta_{r_0,r_1} \leq 5$), so that the complexity of the attack stays small.

The attack is always successful in all our tests but, as we can see on the figure, it is slightly more expensive than the $\mathcal{L}_i = q_i$ case.

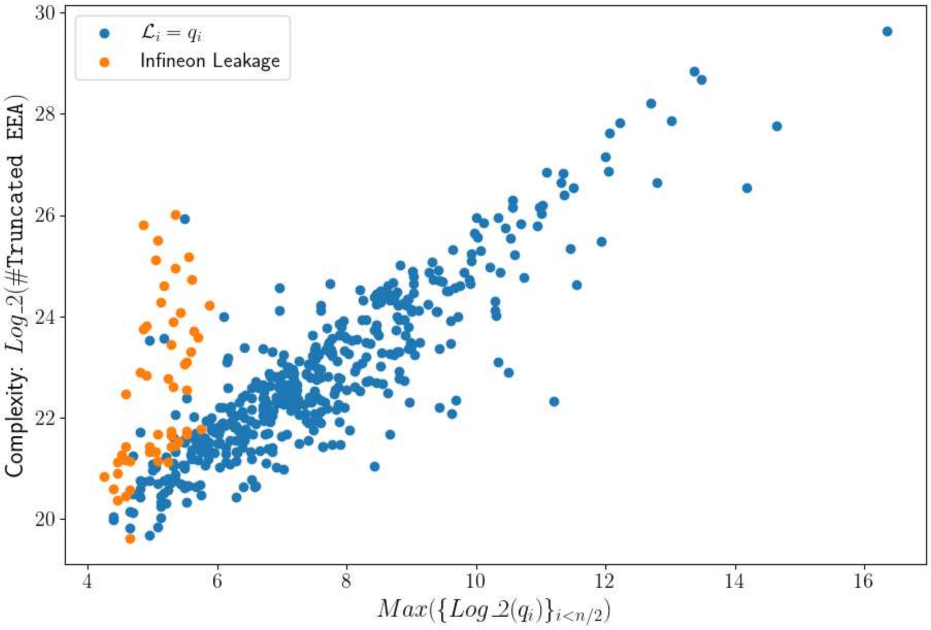


Figure 5.1: Simulation of the Generic Attack – Comparison Between $\mathcal{L}_i = q_i$ and Infineon Leakage

These positive simulation results call for practical validation. Before going there, let us briefly explain how to find the real nonce k when the blinded nonce k' is recovered by the side-channel attack.

5.2 From Blinded Nonce to ECDSA Long Term Private Key

Since the multiplicative mask is 32-bit long, one could simply brute-force it: for each mask candidate $\hat{m}$ compute $[k'/m]G_x$ and compare it to the first part of the ECDSA signature r. They would match for the correct candidate.

There is a much more efficient way to find the mask:

- From the signature (r, s), infer the two points $A_{(x,y)}$ and $B_{(x,y)}$ on the curve such that $A_x = B_x = r$ (and, in fact $A = -B$).
- From k' compute $Q_{(x,y)} = [k']G_{(x,y)} = [m][k]G_{(x,y)}$
- finally we have $[m]A = Q$ or $[m]B = Q$

Hence, finding m corresponds to solve the discreet logarithm on the elliptic curve. The Pollard's kangaroo algorithm [26] will find the mask in $O(2^{\frac{len(m)}{2}})$. Since there is a bit of uncertainty on the point (A or B), the Pollard's kangaroo algorithm might need to be run twice.

For 32-bit odd masks (as in Infineon implementation), this computation is not necessary (since the naive brute-force approach would work) but it tells us that the mask should not just be doubled for security guaranty. Ideally, the mask should be as long as the nonce; that way it would be as hard to find k from k' than to find k directly from r.

Finally, from the value of k it is immediate to find the value of the ECDSA long term private key d as per equation 1.1.

Now that we have a complete key-recovery attack on the paper, let us test it in practice.

5.3 Application to `Feitian A22 JavaCard`

First, we apply the attack to an ECDSA signature execution on the `Feitian A22 JavaCard` acquisition campaign. We selected the 10^{th} ECDSA execution for two reasons:

- we brute-forced the multiplicative mask for this ECDSA execution in Chapter 2, Section 2.3.2. Hence, if the attack does not succeed, we can easily check were the error comes from;
- among the 10 first traces (for which we have the mask), the 10^{th} has one single iteration with $\Delta_{r_0,r_1} > 5$ and it appears in the second half of the `EEA` sequence. From our simulations, this trace should be the easiest to attack.

5.3.1 Leakage Extraction

To extract the leakage information about each and every iteration of the `EEA` side-channel execution trace, we first created a dictionary of leakage traces. From the ECDSA signature verification campaign, where we know the `EEA` inputs (see Chapter 2, Section 2.2), we can list and classify all iterations (and then side-channel subtraces) according to their leakage value (following the encoding described in Section 5.1).

Since we do not have any iteration with $\Delta_{r_0,r_1} > 5$ in our target EEA execution (at least not in the first half of the EEA), we simply build this dictionary for all iterations such that $\Delta_{r_0,r_1} \leq 5$ (for even and odd iterations separately since we have seen that these two cases behave differently).

At the end of this *profiling* step, we end up with 179 sets for every leakage information that an iteration can provide (when $\Delta_{r_0,r_1} \leq 5$). We then select a unique reference subtrace for each of these sets. The number of different sets, with respect to the value of Δ_{r_0,r_1} is given in Table 5.1, we clearly see that odd iterations give away more information as they are divided in more sets than even iterations.

Δ_{r_0,r_1}	odd	even
0	1	1
1	5	2
2	12	6
3	22	9
4	34	15
5	51	21

Table 5.1: Number of Different Sets of Iterations *w.r.t.* Δ_{r_0,r_1}

For the *attack* phase, we first regrouped the subtraces that looked similar (this was done using an smoothing process in order remove small jitters, it is presented in the attack on YubiKey 5Ci, Section 5.4). Then, by hand (there is a total of 156 iterations in the target EEA execution), we looked for wrongly regrouped subtraces. Finally, for each group (again by hand), we tried to match the group with a reference set (by visual comparison with the reference subtraces obtained in the profiling step). This by-hand step took hours of work and would necessitate to be automatized if this attack were to be taken outside a laboratory. For our purpose (demonstrate that the attack is possible) it was the (painful but) shortest path.

5.3.2 Attack Results

The attack succeeded in about 2^{22} calls to the truncated EEA algorithm (see Algorithms 2 and 3). Since the multiplicative mask was already known to us, we could easily check that the recovered blinded nonce was correct.

5.4 Application to YubiKey 5Ci

Now that the attack has been successfully tested on the training device (the Feitian A22 JavaCard), we can finally go back to our primary target, the YubiKey 5Ci. On which an unknown long term private key has been generated and where we can run ECDSA signatures with chosen messages. By opposition to the Feitian A22 JavaCard, we have no access to the private key value and we cannot run ECDSA signature verifications (and then *profile* the iterations).

5.4.1 Side-Channel Acquisitions

The acquisition setup is presented in Chapter 1, Section 1.5.2, the first acquired trace of the full ECDSA is displayed in Figure 1.5. For the attack, we restrict our acquisition to the end of the ECDSA signature algorithm, the details are given in the following table 5.2, one of the acquired trace is displayed in Figure 5.2 after a slight re-centering around the masked nonce modular inversion.

operation	ECDSA signature
equipment	LeCroy WavePro 254HD (12-bit), Langer ICR HH 500-06
inputs	Messages are random, Key is constant (unknown)
number of operations	200
length	5ms
sampling rate	10GSa/s
samples per trace	50MSamples (16-bit)
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-40, 40]$mV
file size	20GB
acquisition time	about 5 minutes

Table 5.2: Acquisition parameters on `YubiKey 5Ci` – ECDSA Signature

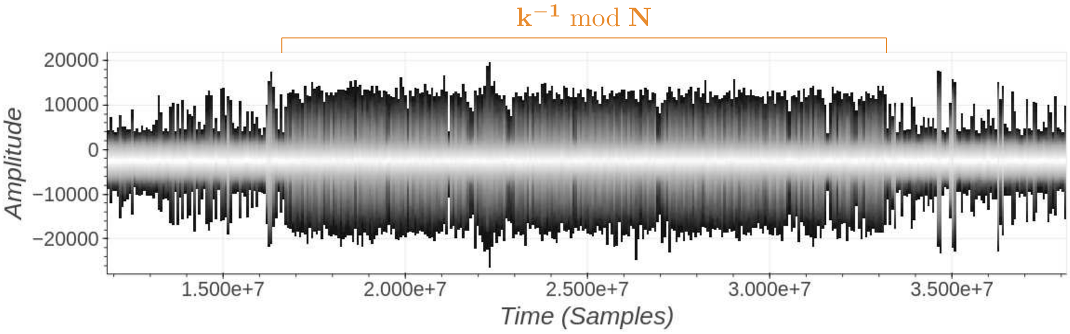


Figure 5.2: `YubiKey 5Ci` – ECDSA Signature – k^{-1} mod N Computation

After detecting and removing the interrupts (as for the `Feitian A22 JavaCard` traces, see Chapter 2, Section 2.1.2), we get the cleaned trace as depicted in Figure 5.3 first subfigure. When zooming into the trace, we recognize the different iterations of the `EEA`. If the EM signal is a bit different, we get very similar patterns as for the `Feitian A22 JavaCard` for the odd and even iterations. The implementation of the `EEA` seems to be exactly the same (recall that the Infineon EC cryptolib is not the same version on the `YubiKey` and then the implementation might have been updated).

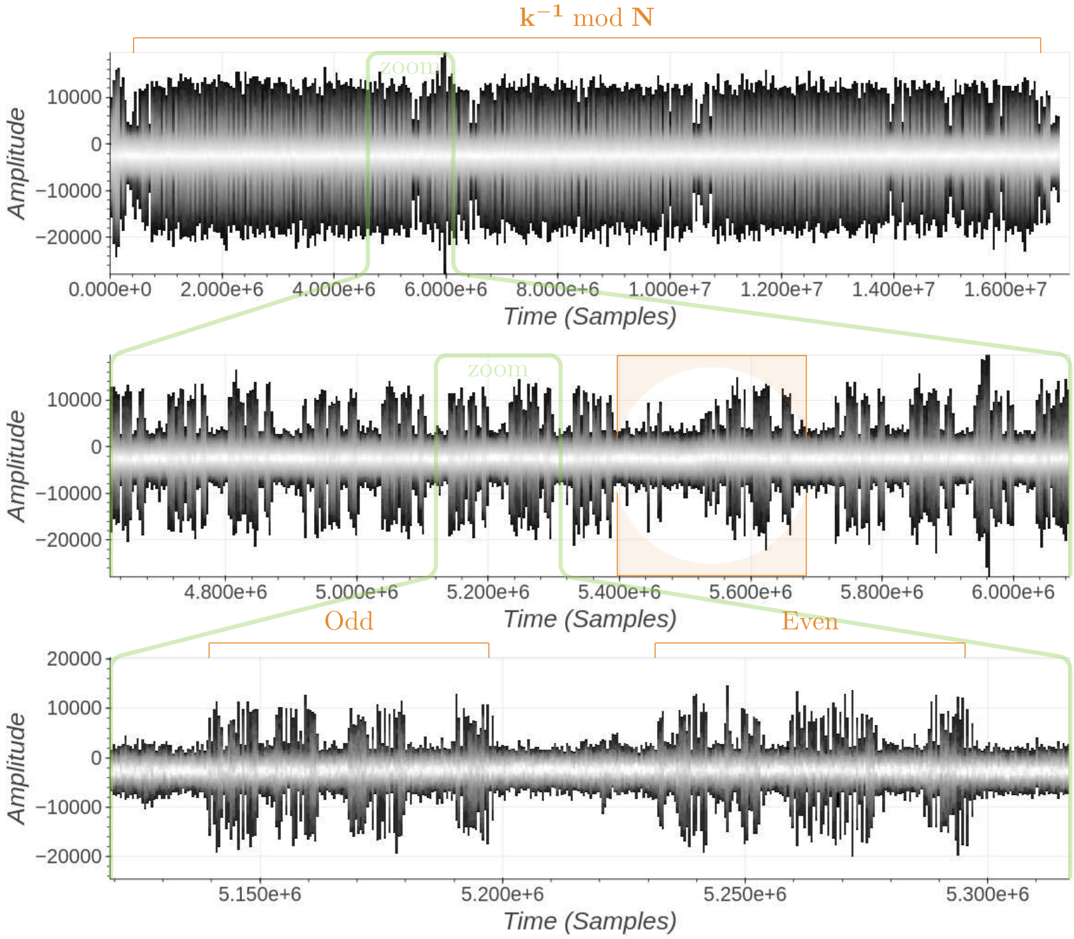


Figure 5.3: `YubiKey 5Ci` – ECDSA Signature – Cleaned k^{-1} mod N Computation (Top) - Zoom on Several Iterations (Middle) - Zoom on Two Consecutive Iterations (Bottom)

Since the iterations subtraces are similar, we decide to use the dictionary of subtraces built in the previous section and use it directly to attack the `YubiKey 5Ci` traces. Since this dictionary has been built for $\Delta_{r_0,r_1} \leq 5$ iterations, our first step is to select `EEA` side-channel execution traces where the first half of the iterations are all small iterations (as illustrated on the middle subfigure of Figure 5.3 the large iterations appear and are quite easy to spot).

5.4.2 Leakage Extraction

So our first step is to develop a detection algorithm (adapted to the `YubiKey 5Ci` EM signal) to soundly detect all iterations of the side-channel traces and identify the traces without large iterations (*i.e.* $\Delta_{r_0,r_1} > 5$) for the first half of the iterations. Over the 200 acquired traces, 6 are identified. For each of the 6 selected traces, the iteration detection algorithm is applied and the result is illustrated on Figure 5.4.

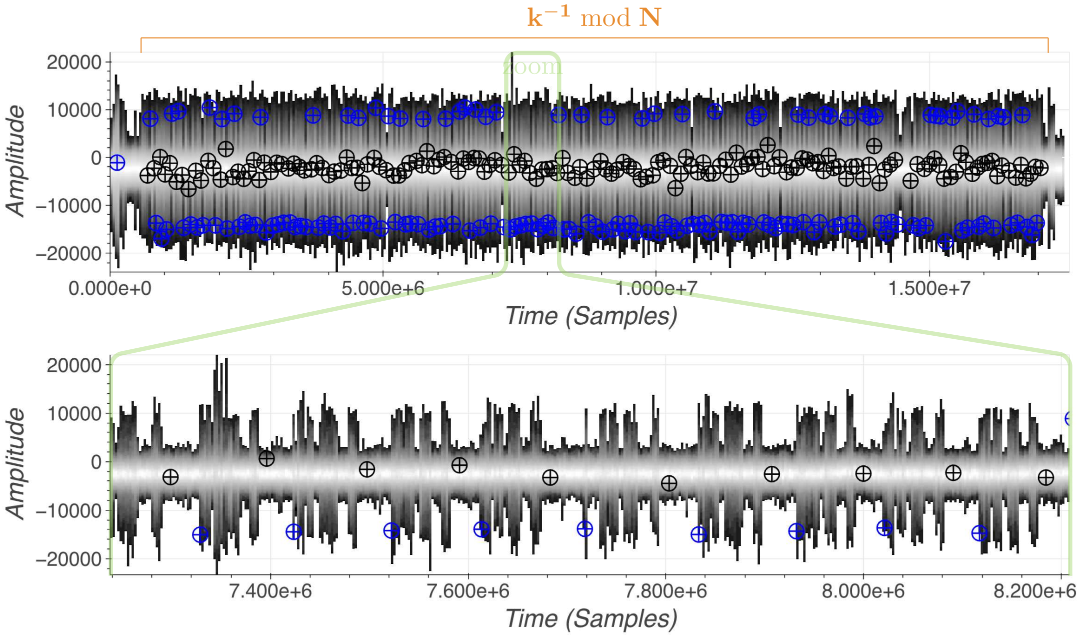


Figure 5.4: `YubiKey 5Ci` – ECDSA Signature – k^{-1} mod N Computation – Iterations Detection (Top) - Zoom on Few Iterations (Bottom)

Thanks to the iteration detection, we can split the side-channel trace into subtraces corresponding to single iterations. Figure 5.5 shows a single, resynchronized, iteration. As identified on the figure, the Δ_{r_0,r_1} timing leakage is a bit different in shape than in the `Feitian A22 JavaCard` traces, but it is still visible and quite easy to extract. For the rest of the leakage (the Infineon euclidean division leakages), we proceed as for the `Feitian A22 JavaCard` traces.

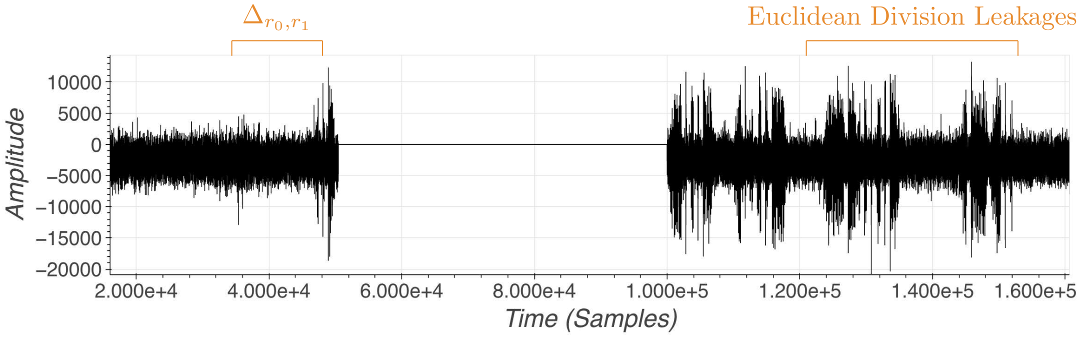


Figure 5.5: `YubiKey 5Ci` – ECDSA Signature – k^{-1} mod N Computation – Odd Iteration Subtrace

The subtraces are smoothed with an averaging rolling window of 1500 samples. The result of this step is depicted on Figure 5.6 for two different odd iterations. The smoothing of the traces allows to easily regroup iterations with similar signal even in the presence of a small jitter.

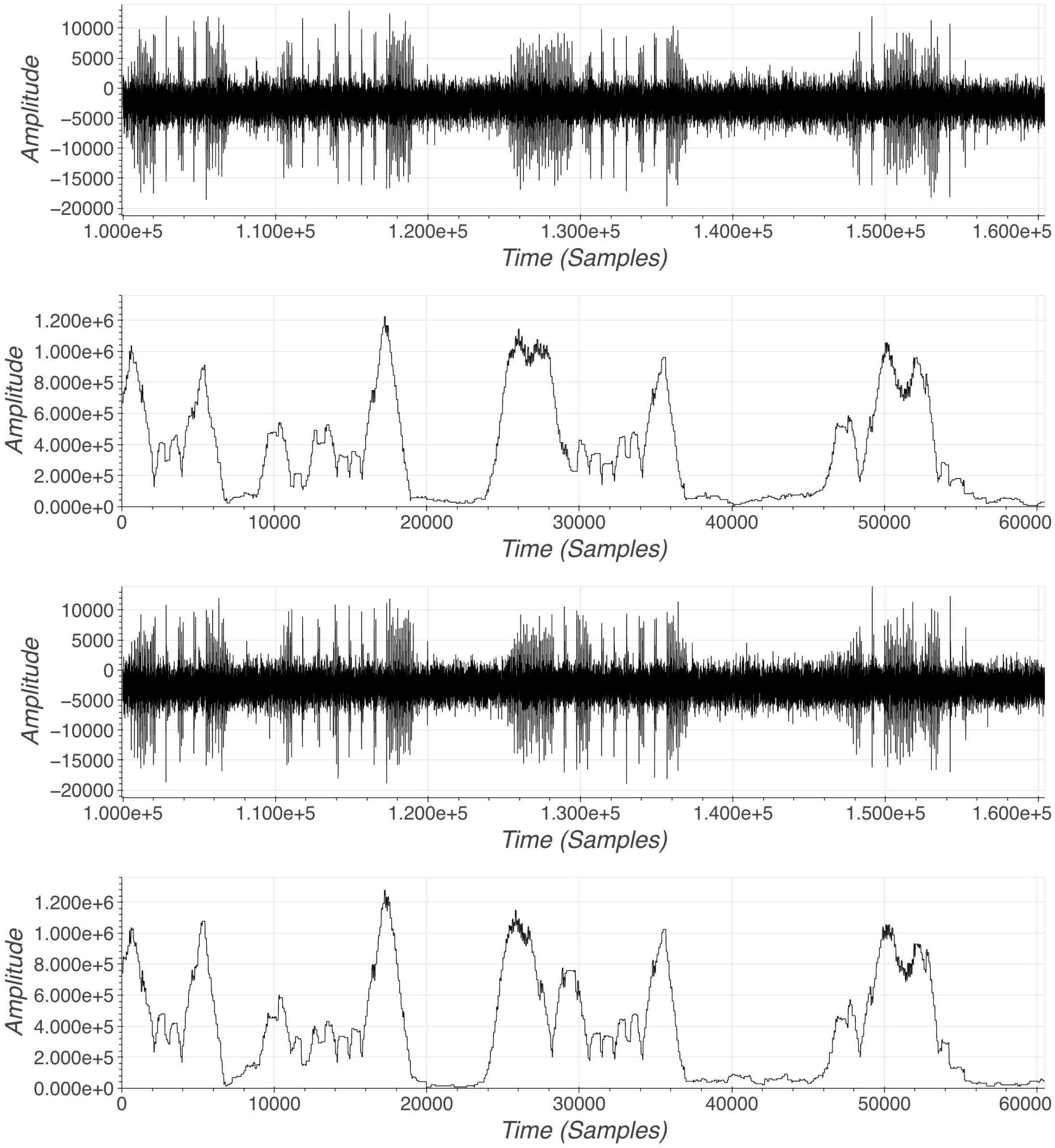


Figure 5.6: `YubiKey 5Ci` – ECDSA Signature – k^{-1} mod N Computation – Single Iteration – 2 Different Odd Iterations (First and Third) - After Averaging Rolling Window (Second and Last)

Once the (odd and even) iterations subtraces are automatically regrouped for the 6 selected `EEA` executions, each group is *by hand* matched with a profile group. Again, this step should be automatized, as it is a long effort and subject to errors. This step took a bit less than 24 hours.

5.4.3 Attack Results

Finally, the attack was run over the 6 EEA executions with the extracted leakages (and erasures in case of unreadable subtraces). Five out of the six attacks were successful with complexity below 2^{28} calls to the truncated EEA. For these five successful attacks, a single blinded nonce was found and the Pollard's kangaroo algorithm finished the work by finding the nonce from which we deduced the long term private key (as explained in Section 5.2).

From the private key, we could roll back to the EEA execution where the attack did not work. We computed the nonce, brute-forced the multiplicative mask and simulated the leaked information from the blinded nonce. It appeared that two different errors of leakage extraction were made in the first half of the EEA iterations. This is due to our error prone step of leakage extraction, we strongly believe that with some engineer work, this step could be greatly improved in time and reliability.

5.5 Conclusions

The attack was demonstrated on two different devices, one of which the long term private key was unknown to us. The overall attack time can be divided in two steps:

- the online phase, where the device must be in the adversary hands. In this phase, the adversary needs to open the device to access the Infineon secure element, put an EM probe and run several ECDSA signatures. Then the device must be re-packaged and quietly returned to the legitimate user. Since the acquisition phase takes few minutes, we estimate to less than an hour the whole process (assuming one can find a reliable way to open and close the device, we did not work on this part);
- the offline phase took about 24 hours in total for us. The vast majority of this time is spent in a by-hand process that should be automatized. The rest is the iteration detection and splitting of the traces (few seconds), the attack process on the extracted leakages (few minutes) and the discreet logarithm computation (few seconds). With enough engineering work, this offline step should take less than an hour.

Finally, for the attack on the YubiKey 5Ci, we used 200 ECDSA signatures to produce 5 successful attacks. Hence, in average, our attack cost about 40 ECDSA signatures. The main limitations to reduce this number are (1) the learning phase must be extended to the $\Delta_{r_0,r_1} > 5$ iterations, there does not seem to be any reason for this to be a problem (but we did not do it, so we never know) and (2) the attack process cost will increase with large Δ_{r_0,r_1} iterations. This would impact the offline phase of the attack, and certainly be acceptable for most of the EEA executions [2].

[2] EEA executions with many large Δ_{r_0,r_1} iterations do not appear very often

Chapter 6

Beyond SLE78

6.1 Infineon Security Microcontrollers

It is not that easy to get a proper picture of the Infineon security microcontrollers families [1] (and this is also true for the other big secure element manufacturers). The CC certificates publication of these chips is certainly the best way to keep track with the different chips hardware and embedded firmware versions.

One thing that helps with Infineon security microcontrollers is that they are all certified by the German certification body (BSI [2]) except for two of them certified by the Netherlands certification body (NSCIB [3]). All of these CC evaluations were performed by the same ITSEF, TÜVIT [4] which happens to be accredited for CC hardware certifications by both BSI and NSCIB.

All of these CC certificates are public and come along with public security target and certification report documents that contain valuable information about the chip and firmware versions. Furthermore, the BSI has a quite nice database search engine [5] that helps in finding the different documents. We also must mention the great SEC-CERTS initiative [13, 12] that ended up in a powerful CC documents search engine [6].

Thanks to these certifications, we were able to identify 4 large families of Infineon security microcontrollers. Figure 6.1 illustrates these 4 families along with the number of CC certifications (`AVA_VAN_4` for TPMs and `AVA_VAN_5` for security microcontrollers with optional cryptographic library [7]). These 4 families distinguish from each other by the technological node of the chips: from 90nm for the `SLE78` to 28nm for the most recent family. While the two oldest families are based on proprietary 16-bit microcontrollers from Infineon, the two most recent ones are ARM-based 32-bit microcontrollers: the first one is based on an ARM SecureCore SC300 [8] while

[1] https://www.infineon.com/cms/en/product/security-smart-card-solutions/

[2] https://www.bsi.bund.de/

[3] https://trustcb.com/common-criteria/nscib/

[4] https://www.tuvit.de/en/home/

[5] https://www.bsi.bund.de/SiteGlobals/Forms/Suche/EN/Expertensuche_Formular.html

[6] www.sec-certs.org

[7] We limit ourselves to the lowest level of certifications that includes the vulnerable Infineon cryptolib. JavaCard OSes and Applets (banking, passport, etc.) will be certified on top of IC certifications and will rely on them for security guaranties using the handful composite certification process of CC. The TPM is a special case in CC certification, they usually are certified as a single block including both IC and top firmware levels.

[8] https://developer.arm.com/Processors/SecurCore%20SC300

the second one is certainly based on the more recent ARM Cortex-M35P [9].

Figure 6.1 gives, for each family of security microcontrollers, the number of identified IC models along with the number of CC certificates and CC maintenance certificates. We also show the date of the first and last certificate that was published. Note that for the SLE78 family, a certificate was produced on July the 3rd 2024 (*i.e.* during the EUCLEAK responsible disclosure period) where all cryptolibs are removed from the scope [10]. Moreover, in the ARM SC300 family, a certificate was produced even more recently: on August the 30th 2024 (few days prior to the initial publication of this report). Surprisingly, the certificate relates to vulnerable cryptolibs versions [11]. NSCIB, the certification body that issued this certificate assures that the certificate takes into account the EUCLEAK attack by updating the user guidance of the cryptolibs. The exact same strategy was taken for three other post-EUCLEAK-publication certificates for the 16-bit (65 nm) family [12] [13] and the Armv8-M (28 nm) family [14]. This is a pity that these guides are not public... The detailed information about each IC model and certificate is provided in Annex B.

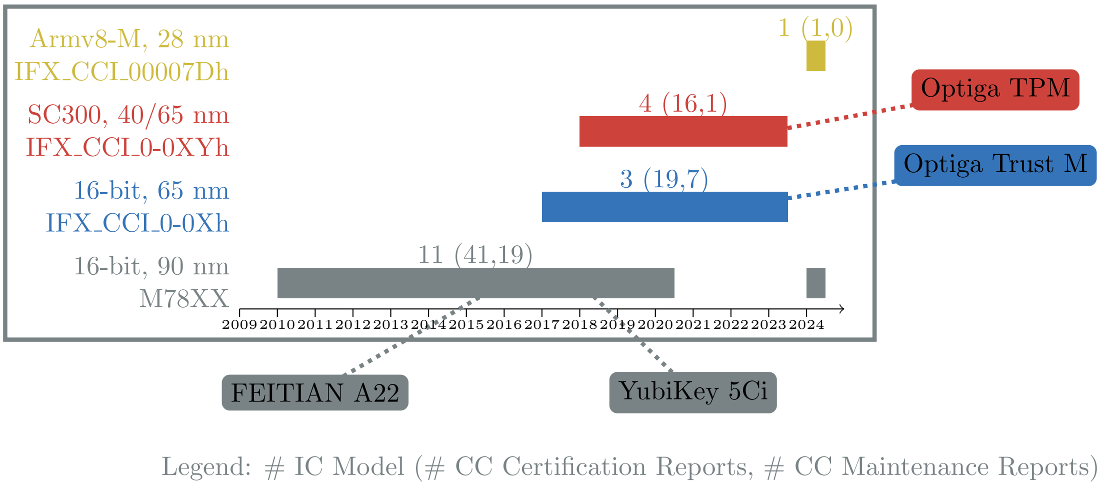


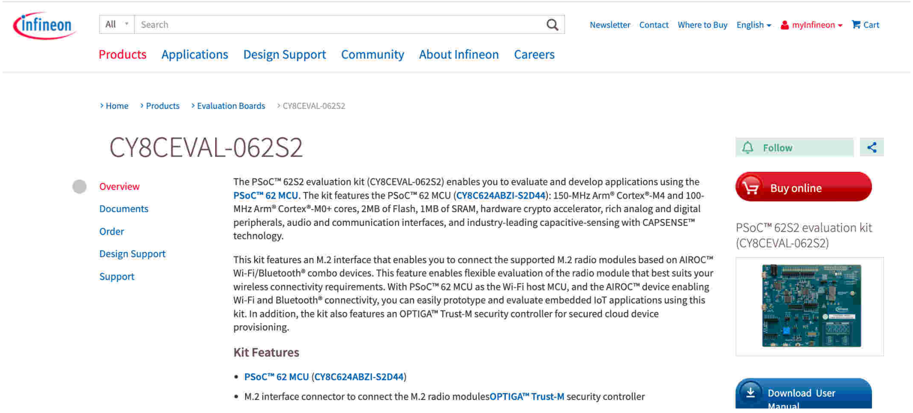
Figure 6.1: Infineon Security Microcontroller Families

In Figure 6.1, we identified the two targets of our attack as being part of the SLE78 family (the oldest one), the YubiKey being slightly more recent than the Feitian A22 JavaCard. We could identify two other products: the Optiga Trust M and the Optiga TPM chips that are part of more recent families of Infineon security microcontrollers. The good thing about these products is that development boards to play with them are available. The chips are not completely open but in both cases one can send commands that will trigger an ECDSA signature / verification computation. Let us have a look at these products.

[9] https://developer.arm.com/Processors/Cortex-M35P
[10] See https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0891
[11] See https://www.trustcb.com/download/nscib-cc-2200060-02-cr/
[12] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1110.html
[13] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1079.html
[14] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1229.html

6.2 Infineon `Optiga Trust M`

In order to run cryptographic commands on the `Optiga Trust M` chip, we acquired a PSoC$^{\text{TM}}$ 62S2 evaluation kit (CY8CEVAL-062S2) (see Figure 6.2). Source code and documentation is freely available [15], we strictly followed Infineon examples to get started with `Optiga Trust M` cryptographic commands [16]. We could easily develop a simple code that sends ECDSA signature and ECDSA signature verification commands to the `Optiga Trust M` and could even control GPIOs that will tell our oscilloscope that the ECDSA operation is starting.



CY8CEVAL-062S2

The PSoC™ 62S2 evaluation kit (CY8CEVAL-062S2) enables you to evaluate and develop applications using the **PSoC™ 62 MCU**. The kit features the PSoC™ 62 MCU (**CY8C624ABZI-S2D44**): 150-MHz Arm® Cortex®-M4 and 100-MHz Arm® Cortex®-M0+ cores, 2MB of Flash, 1MB of SRAM, hardware crypto accelerator, rich analog and digital peripherals, audio and communication interfaces, and industry-leading capacitive-sensing with CAPSENSE™ technology.

This kit features an M.2 interface that enables you to connect the supported M.2 radio modules based on AIROC™ Wi-Fi/Bluetooth® combo devices. This feature enables flexible evaluation of the radio module that best suits your wireless connectivity requirements. With PSoC™ 62 MCU as the Wi-Fi host MCU, and the AIROC™ device enabling Wi-Fi and Bluetooth® connectivity, you can easily prototype and evaluate embedded IoT applications using this kit. In addition, the kit also features an OPTIGA™ Trust-M security controller for secured cloud device provisioning.

PSoC™ 62S2 evaluation kit (CY8CEVAL-062S2)

Kit Features

- **PSoC™ 62 MCU (CY8C624ABZI-S2D44)**
- M.2 interface connector to connect the M.2 radio modules**OPTIGA™ Trust-M** security controller

Figure 6.2: `Optiga Trust M` – PSoC$^{\text{TM}}$ 62S2 evaluation kit (CY8CEVAL-062S2)

6.2.1 Side-Channel Acquisitions

Figure 6.3 shows the side-channel setup, the green wire is there to catch the GPIO signal that will trig the acquisition right before the ECDSA operation. Figure 6.4 displays the EM probe position we used over the Infineon chip.

[15] https://github.com/Infineon/optiga-trust-m

[16] https://github.com/Infineon/mtb-example-optiga-crypto

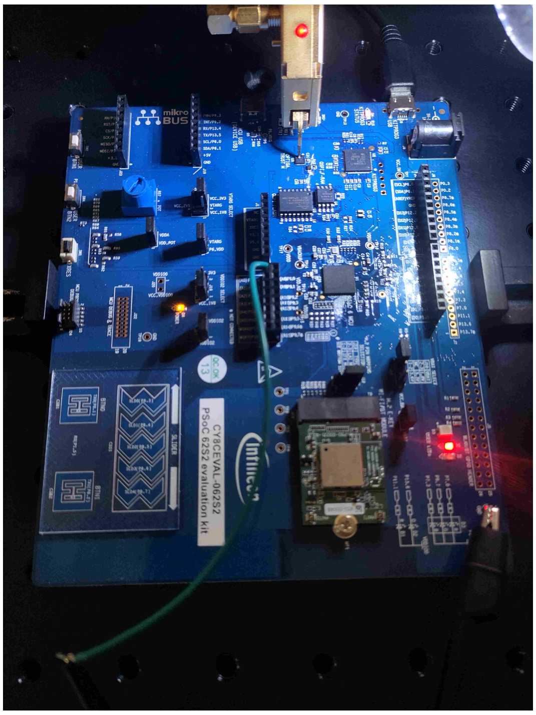


Figure 6.3: `Optiga Trust M` – Acquisition Setup

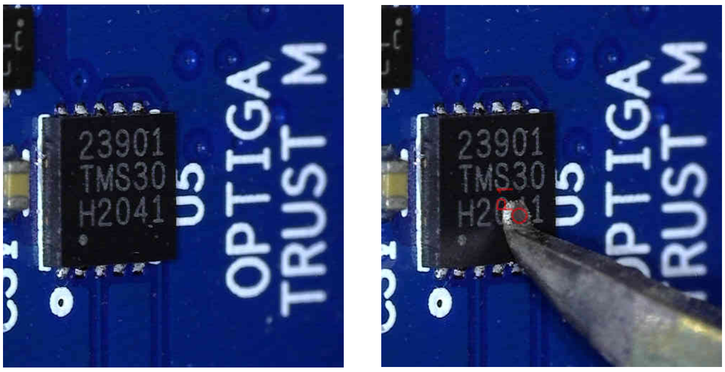


Figure 6.4: `Optiga Trust M` – EM Probe Position

For the record, the acquisition details for the ECDSA signature operations are given in Table 6.1. We did not change the parameters for few ECDSA signature verification operations.

operation	ECDSA signature
equipment	LeCroy WavePro 254HD (12-bit), Langer ICR HH 500-06
inputs	Messages are random, Key is constant (randomly chosen)
number of operations	10
length	200ms
sampling rate	1GSa/s
samples per trace	200MSamples (16-bit)
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-200, 200]$mV
file size	4GB
acquisition time	about 1 minute

Table 6.1: Acquisition parameters on `Optiga Trust M` – ECDSA Signature

Figure 6.5 (resp. Figure 6.6) displays a full side-channel ECDSA signature (resp. signature verification) execution trace. These execution traces are very similar to what was observed on the `SLE78` experiments and finding the nonce modular inversion was quite straightforward (as illustrated on the figures).

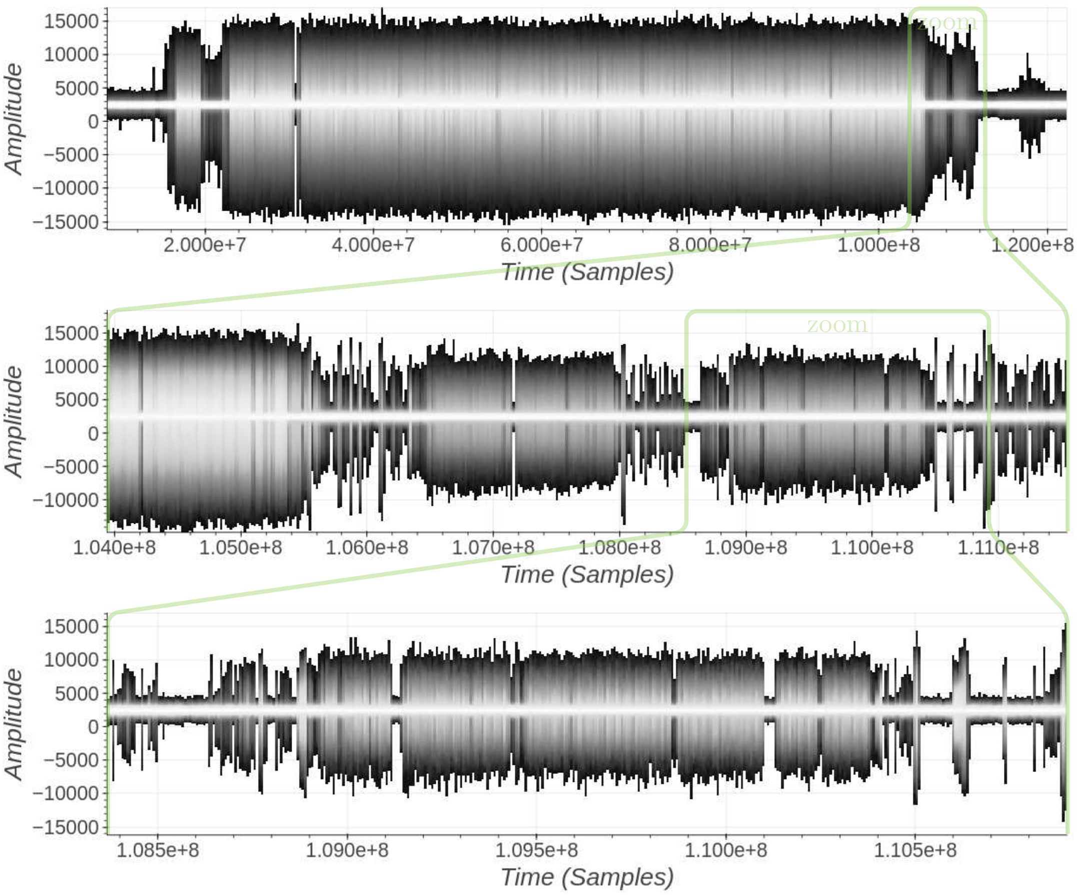


Figure 6.5: `Optiga Trust M` – ECDSA Signature – Full Trace (Top) - Zoom in Computation End (Middle) - $k^{-1} \bmod N$ Computation (Bottom)

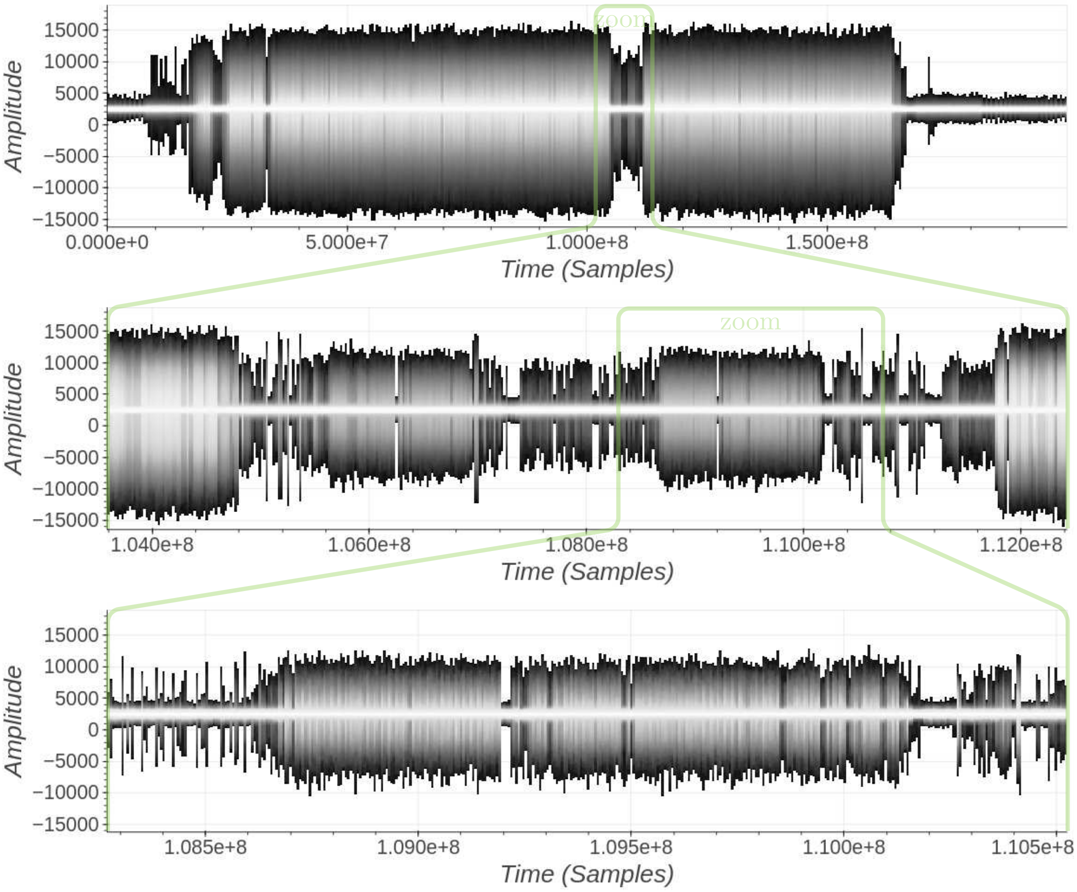


Figure 6.6: `Optiga Trust M` – ECDSA Signature Verification – Full Trace (Top) - Zoom before Double Scalar Mult. (Middle) - s^{-1} mod N Computation (Bottom)

These observations tend to show that Infineon modular inversion implementation did not change in these more recent chips. We will go one step further to completely validate this assumption.

6.2.2 Leakage Observation

In order to acquire more executions and more precisely, we reduced the acquisition time span by selecting the area before the Double Scalar Multiplication in ECDSA signature verification operation. We could then reduce the y-axis dynamic and increase the sampling rate and the number of acquired ECDSA signature verification executions. Details are given in Table 6.2. An example of the acquired traces is displayed in Figure 6.7.

operation	ECDSA signature verification
equipment	LeCroy WavePro 254HD (12-bit), Langer ICR HH 500-06
inputs	Messages are random, Signature generated from a constant Key
number of operations	200
length	10ms
sampling rate	5GSa/s
samples per trace	50MSamples (16-bit)
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-80, 80]$mV
file size	15GB
acquisition time	about 9 minute

Table 6.2: Acquisition parameters on `Optiga Trust M` – ECDSA Signature Verification

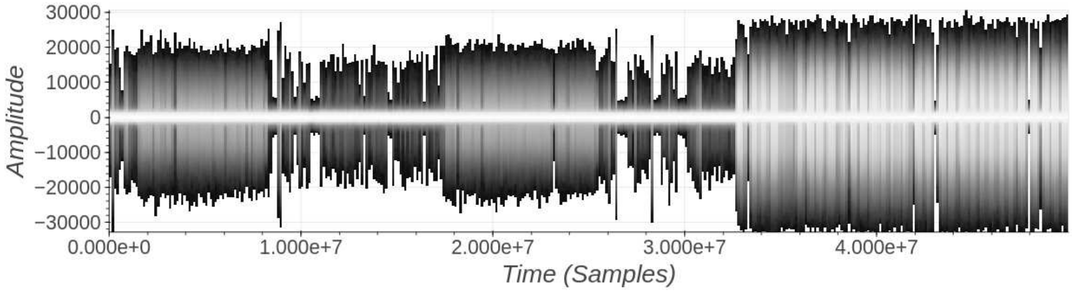


Figure 6.7: `Optiga Trust M` – ECDSA Signature Verification – Full Trace

With this new acquisition, we first cleaned the interrupts and detected each iteration of the `EEA` execution as this was done before on `Feitian A22 JavaCard` and `YubiKey 5Ci` traces. Figure 6.8 shows a cleaned trace (first subfigure) and an iteration subtrace (second subfigure).

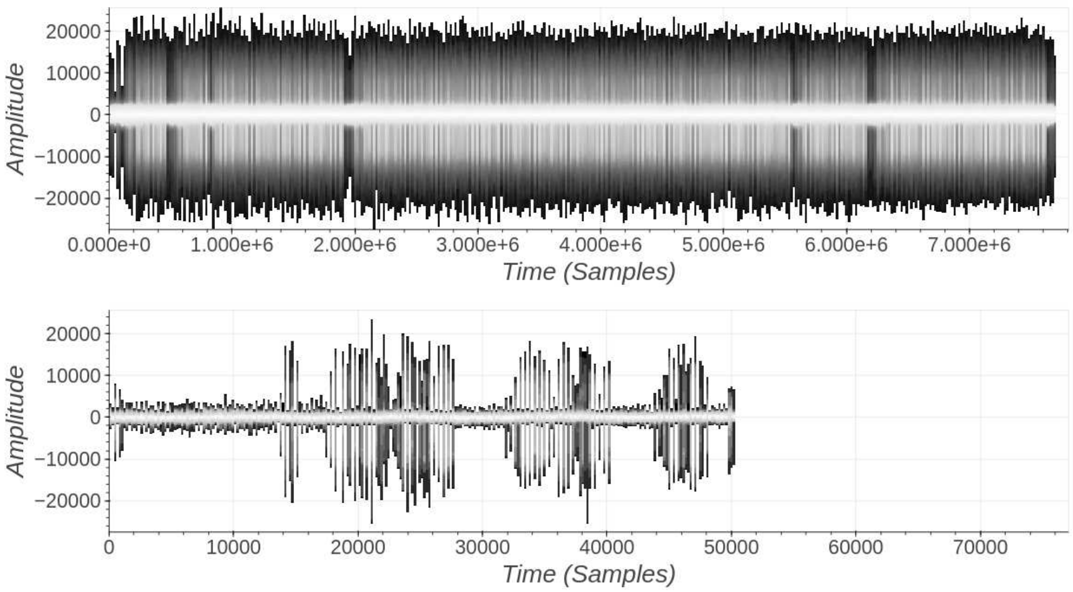


Figure 6.8: `Optiga Trust M` – ECDSA Signature Verification – Cleaned s^{-1} mod N Computation (Top) - Single Odd Iteration after Splitting (Bottom)

Since we are working on the modular inversion of the known s (the second part of the ECDSA signature), and that Infineon did not add side-channel countermeasures to this operation, we can predict the Infineon leakages (see Chapter 4, Section 4.3) from the value of s. We could then validate that the subtraces still show the same leakages as before. Figure 6.9 illustrates this verification: 100 odd iterations subtraces are superposed, the color of the subtrace is dictated by the values of Δ_{r_0,r_1}, one can clearly see that the subtraces are naturally grouped by color.

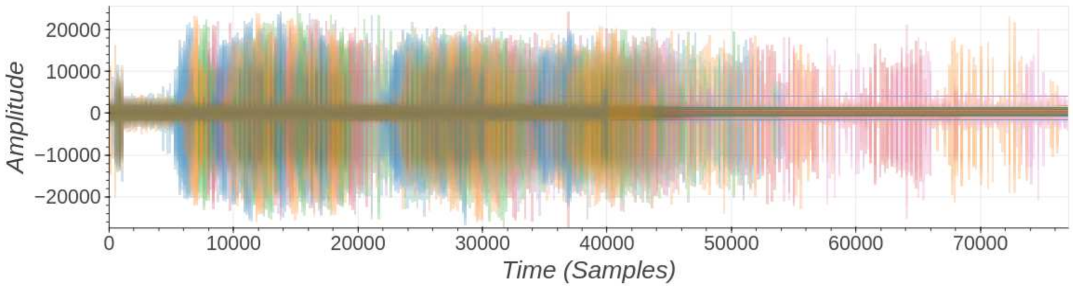


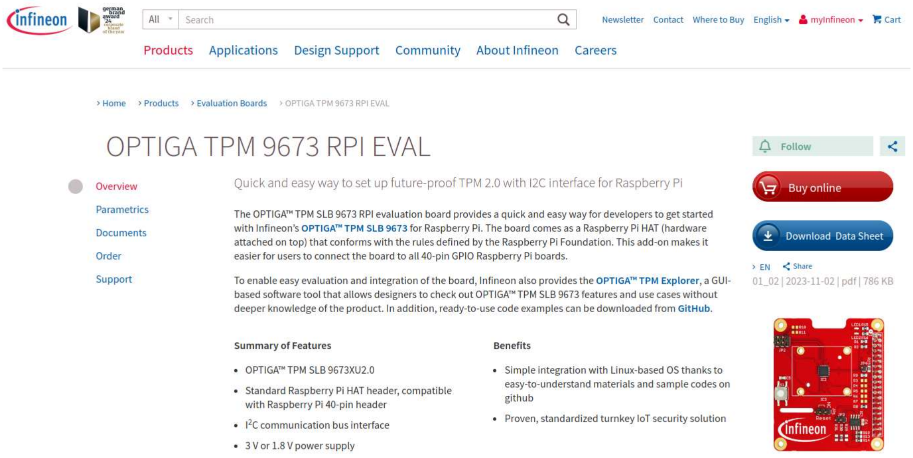
Figure 6.9: `Optiga Trust M` – ECDSA Signature Verification – 100 superposed Odd Iterations)

Hence, we validated that the implementation did not change. However, since we did not do the attack completely on the ECDSA signature operation, we cannot fully conclude. For instance, Infineon might have simply increased the size of the multiplicative mask used to protect the nonce modular inversion.

After this test we contacted Infineon and the other stakeholders to start the responsible disclosure. We explained our experiments on `Optiga Trust M` and that we supposed that the mask length was still 32 bits. Infineon neither confirmed nor denied and went on with implementing a countermeasure: increase the size of the multiplicative mask up to the size of the nonce. We can then safely conclude that the `Optiga Trust M` is vulnerable to the EUCLEAK attack.

6.3 Infineon Optiga TPM

It is only after the start of the responsible disclosure that we remarked that the `Optiga TPM` was available (and part of a family that we did not analyze yet). So we ordered on Farnell the OPTIGA TPM 9673 RPI evaluation kit on May the 15th, 2024.



OPTIGA TPM 9673 RPI EVAL

Quick and easy way to set up future-proof TPM 2.0 with I2C interface for Raspberry Pi

The OPTIGA™ TPM SLB 9673 RPI evaluation board provides a quick and easy way for developers to get started with Infineon's **OPTIGA™ TPM SLB 9673** for Raspberry Pi. The board comes as a Raspberry Pi HAT (hardware attached on top) that conforms with the rules defined by the Raspberry Pi Foundation. This add-on makes it easier for users to connect the board to all 40-pin GPIO Raspberry Pi boards.

To enable easy evaluation and integration of the board, Infineon also provides the **OPTIGA™ TPM Explorer**, a GUI-based software tool that allows designers to check out OPTIGA™ TPM SLB 9673 features and use cases without deeper knowledge of the product. In addition, ready-to-use code examples can be downloaded from **GitHub**.

Summary of Features

- OPTIGA™ TPM SLB 9673XU2.0
- Standard Raspberry Pi HAT header, compatible with Raspberry Pi 40-pin header
- I^{2}C communication bus interface
- 3 V or 1.8 V power supply

Benefits

- Simple integration with Linux-based OS thanks to easy-to-understand materials and sample codes on github
- Proven, standardized turnkey IoT security solution

Figure 6.10: `Optiga TPM` – OPTIGA TPM 9673 RPI EVAL

We followed Infineon examples to get started with TPM cryptographic commands [17] and easily developed a simple code to send ECDSA signature and signature verification commands to the TPM.

6.3.1 Side-Channel Acquisitions

Figure 6.11 shows the side-channel setup with the EM probe position we used over the Infineon chip.

[17] `https://github.com/Infineon/optiga-tpm`

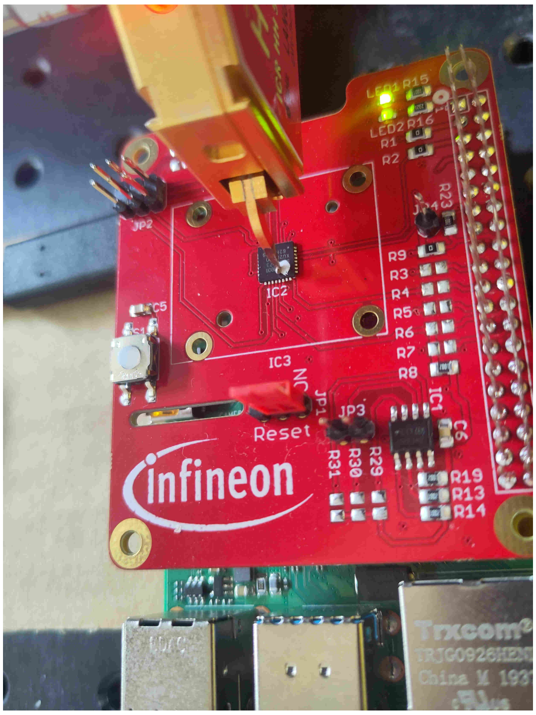


Figure 6.11: `Optiga TPM` – Acquisition Setup

First we acquired few side-channel traces of the whole TPM ECDSA signature verification command with low sampling rate because of the command length (about 1 second). The resulting traces have 1.25G samples and, at first sight, it is not so easy to find the actual ECDSA computations (see first subfigure of Figure 6.12). Indeed, on the TPM board we do not have a handy GPIO signal telling us when the ECDSA operation is starting.

Nevertheless, by changing the elliptic curve size (to 384 bits), the cryptographic operation stands out (as it changes significantly its execution time while the rest of the command does not). We hence could isolate the ECDSA signature verification operation (as shown in the second subfigure of Figure 6.12)

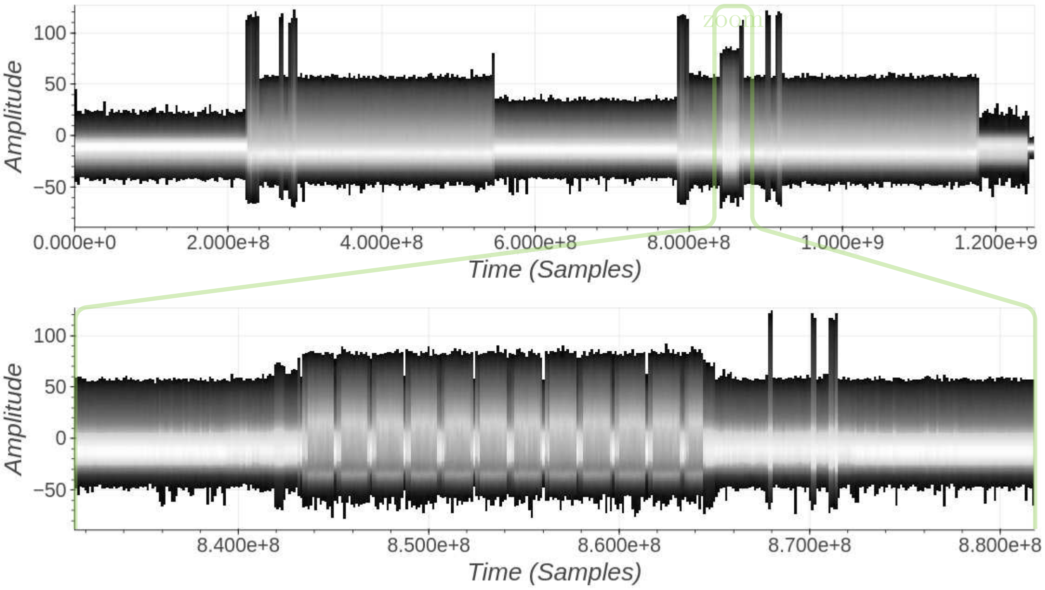


Figure 6.12: `Optiga TPM` – ECDSA Signature Verification – Full Trace (Top) - Zoom on ECDSA Operation (Bottom)

Interestingly the ECDSA signature verification on the `Optiga TPM` does not match the execution we have seen on `SLE78` or `Optiga Trust M`. It seems that the first part of the computation is missing and only the double scalar multiplication is left. This might mean that the cryptographic library has been updated.

In order to improve our understanding of the traces, we acquired a new set of traces, around the double scalar multiplication. For the record, the acquisition details for the ECDSA signature verification operation are given in Table 6.3.

operation	ECDSA signature verification
equipment	PicoScope 6424E, Langer ICR HH 500-06
inputs	Messages are random, Signature generated from a constant Key
number of operations	200
length	200ms
sampling rate	5GSa/s
samples per trace	1GSamples
channel(s)	EM activity
channel(s) parameters	DC 50ohms, $[-20, 20]$mV
file size	200GB
acquisition time	about 20 minutes

Table 6.3: Acquisition parameters on `Optiga TPM` – ECDSA Signature Verification

The acquired traces are displayed in Figure 6.13, one can see that the double scalar multiplication is preceded by a pattern that looks very much like the modular inversion of s (and indeed this operation needs to be executed prior to the double scalar multiplication). When zooming into the modular inversion trace, one can guess the `EEA` iterations (see last subfigure) and a quick count of these iterations shows that their number matches what is expected from the `EEA` iterations. The signal is weaker than before and to better see the iterations, the signal on the last subfigure went through a bandpass filter (few MHz around 200MHz).

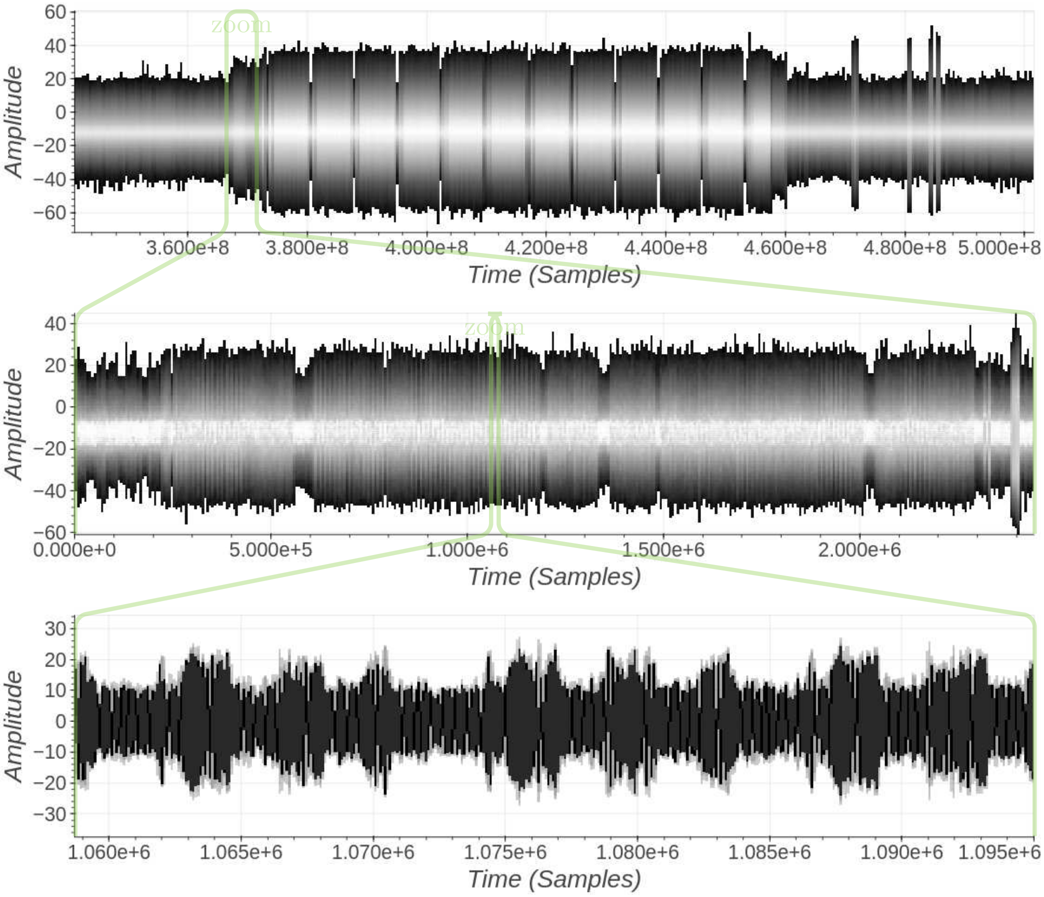


Figure 6.13: `Optiga TPM` – ECDSA Signature Verification – Full Computation (Top) - Zoom on s^{-1} mod N Computation (Middle) - Zoom on Few Iterations (Bottom)

From this observation, one cannot really prove that the sensitive leakage still exists and is exploitable. However, we have good reasons to believe that the implementation of the modular inversion did not change. At the end of May 2024, and then during the responsible disclosure, we described our experiments on the `Optiga TPM` to Infineon. We hence considered that, without new inputs, all Infineon security microcontrollers and all Infineon EC cryptolib versions are susceptible to the EUCLEAK attack. Infineon neither confirmed nor denied.

Chapter 7

Conclusions

To conclude, we present the impact analysis of the EUCLEAK attack, then the attack mitigations and avenues of research that follow this work. Finally, the project timeline is also provided.

First, let us summarize what are the basic requirements for the EUCLEAK attack to be feasible in practice:

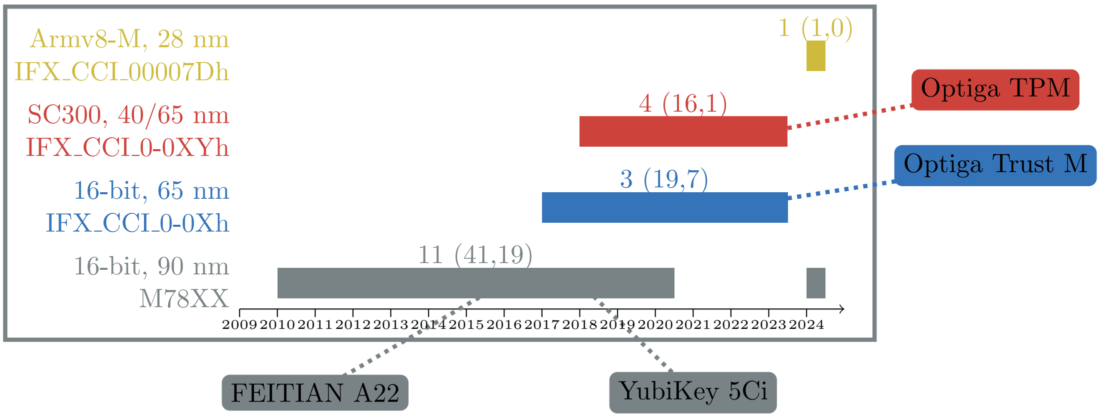
- A secure system using an Infineon security microcontroller that embeds the Infineon cryptographic library and runs cryptographic operations involving the modular inversion of a secret (*e.g.* ECDSA).
- The attacker should have physical access to the device. This is the *online phase* of the attack. Concretely, the attacker needs:
  - to open the device in order to access to the Infineon security microcontroller chip package. Depending on the difficulty of this opening procedure, the online phase can take more or less time;
  - an electromagnetic probe, an oscilloscope and a computer to capture the electromagnetic side-channel signal of the chip during the cryptographic computations, by putting the electromagnetic probe just above the chip package during the executions (few minutes). Note that the side-channel setup can be mobile (e.g. by using a PicoScope [25] and a laptop which fit in a case or a backpack).
- Later, the offline phase will take one hour to one day for the attacker to retrieve the secret from the captured side-channel traces.

Note that the side-channel capture must be reproduced for every different secret stored in the chip (for instance in FIDO, for every web service protected by FIDO, a dedicated key pair is associated – one must then capture a set of side-channel acquisitions for each different targeted secret – the device however has to be opened only once). The offline phase also has to be reproduced for every secret.

7.1 Impact on Infineon Security Microcontrollers

We suspect that all Infineon security microcontrollers embedding Infineon cryptolib [1] are affected by the attack. We reproduce below Figure 6.1: our understanding of Infineon security microcontrollers families, it was built upon the public documents produced by the Common Criteria (CC for short) certification process, details are given in Annex B.

Figure 6.1 reads as follows: the 65nm family of Infineon security microcontrollers (in blue) possesses three different IC models (or configurations), they went through 19 CC certifications and 7 CC maintenances from 2017 (first certificate) to 2023 (last certificate at the time of writing this report).



Legend: # IC Model (# CC Certification Reports, # CC Maintenance Reports)

Figure 6.1: Infineon Security Microcontroller Families (repeated from page 65)

The four identified products on the figure (`Feitian A22 JavaCard`, `YubiKey 5Ci`, `Optiga Trust M` and `Optiga TPM`) were analyzed in our work. The EUCLEAK attack was fully demonstrated on `Feitian A22 JavaCard` and `YubiKey 5Ci` (see Chapter 5). Strong arguments that the attack is also applicable to `Optiga Trust M` and `Optiga TPM` were presented (see Chapter 6).

Infineon did not clearly confirm nor deny our suspicion but went on to develop a patch for their cryptolib (see the mitigation in Section 7.4). To our knowledge, at the time of writing this report, the patched cryptolib did not yet pass a CC certification. Anyhow, in the vast majority of cases, the security microcontrollers cryptolib cannot be upgraded on the field, so the vulnerable devices will stay that way until device roll-out.

7.2 Confirmed Vulnerable End-User Products

So far, here is the list of products where the impact is confirmed.

[1] Remark that this is not always the case, *e.g.* large smartcard manufacturers, like Thales-DIS, IDEMIA or G+D, usually develop their own cryptolibs instead of using the ones from the secure element manufacturer.

7.2.1 YubiKey 5 Series

All `YubiKey 5 Series` with firmware version below 5.7 are vulnerable to the attack. `YubiKey` firmware is not upgradable [2], only the security keys shipped with firmware 5.7 or later are resistant. EUCLEAK vulnerability impacts FIDO/FIDO2 and most certainly PGP and PIV (when ECDSA is used).

7.2.2 Feitian A22 JavaCard

Feitian claims that the `Feitian A22 JavaCard` analyzed in this work does not exist anymore, the new versions as well as all other Feitian products based on an Infineon security microcontroller embed a cryptolib developed by Feitian. We could partially verify this claim by looking at the side-channel traces of a Feitian K44 device running ECDSA (see slides 23 to 27 of [18] for more details about Feitian FIDO hardware tokens), it is indeed quite different from Infineon implementation.

7.2.3 Infineon TPMs

All Infineon TPMs from the SLB96xx version (see the TPMs listed in Annex B) are impacted by the attack.

7.3 Potentially Vulnerable Products

Secure elements, and at a good place Infineon's, are involved in many large secure systems. Building a comprehensive list of the impacted systems is clearly out of NinjaLab reach.

Any system whose security relies on ECDSA (or the modular inversion of a secret) that is run on an Infineon security microcontroller (at least starting with the first version of the `SLE78`), embedding Infineon cryptographic library (any version until the patch is out and shipped) might be at risk.

We have already discussed **FIDO/FIDO2** hardware tokens and **TPMs**, mentioned **PIV** and **PGP**, many more products can be considered:

- **Crypto-Currency Hardware Wallets**:
  They naturally rely on ECDSA (for signing transactions). Several embed an Infineon chip with Infineon cryptolib [3].
- **ID cards/Passports/Health cards**:
  In this area, Infineon security microcontrollers are omnipresent. for obvious reason, it is quite complicated for small independent research companies like NinjaLab to attack governmental-critical products, even for demonstration purposes.
  Nevertheless, recent electronic passports [11] support either the Active Authentication protocol (that involves ECDSA computation) or the more recent PACE-CAM protocol (that involves the modular inversion of the long term secret) for anti-cloning and therefore might be subject to the EUCLEAK attack.

[2] See https://support.yubico.com/hc/en-us/articles/360013708760-YubiKey-Firmware-is-Not-Upgradable

[3] See for instance https://bitcointalk.org/index.php?topic=5304483.0 for a list of the secure elements used in different crypto-currency hardware wallets.

For instance, from the BSI CC certification database, several passport applet developers directly use Infineon cryptolib when running on an Infineon security microcontroller (MaskTech, Eviden, Universal Information Technology LLC, cryptovision GmbH).

- **Matter**:
  The Matter standard for IoT devices relies on ECDSA [4], Infineon products are involved in this initiative [5].

- **V2X**:
  Cars are also a good example where secure elements show themselves very useful and where Infineon is present [6]. The associated protocols naturally involve ECDSA [7].

7.4 Attack Mitigations

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

7.4.1 Hardening the Infineon Cryptographic Library

The simplest mitigation to the EUCLEAK attack at the cryptolib level would be to improve the Infineon nonce blinding countermeasure by increasing the size of the multiplicative mask to the size of the elliptic curve.

According to Infineon, this is the mitigation they have chosen to develop and they already have a working implementation. They offered to send us an `Optiga TPM` with the patched version of the EC library, but we declined. Indeed, it would be quite hard to validate that they did what they say without access to the source code of the library. We then let the CC ITSEF check the new countermeasure implementation as they do have access to the source code during the CC evaluation.

Another mitigation would be to change the modular inversion algorithm and, *e.g.*, switch to a modular exponentiation instead of the `EEA`.

7.4.2 High-Level Mitigations

Let us emphasize here that *we strongly encourage to continue to use an EUCLEAK vulnerable product rather than switching to a solution that does not involve a secure element.* So in cases where a vulnerable product has to be used (*e.g.* it cannot be patched and roll-out is not coming fast enough), some temporary mitigations might exist:

- **Avoid ECDSA**: Many protocols/applications offer the possibility to choose the cryptographic primitive from a list. This is for instance the case for FIDO2, PIV or PGP.

- **Defense in Depth**: When this is possible activate all defense layers, *e.g.* enforce the use of a PIN (or any biometrics) to access to the device.

[4] See https://csa-iot.org/wp-content/uploads/2022/03/Matter_Security_and_Privacy_WP_March-2022.pdf

[5] https://www.nagra.com/kudelski-iot-partners-infineon-enhance-smart-home-device-security-new-matter-certified-solution

[6] https://globalplatform.org/wp-content/uploads/2023/12/5.-Antoaneta-Kondeva_eSE_in_Automotive_GP_automotive.pdf

[7] See *e.g.* https://www.commoncriteriaportal.org/files/ppfiles/pp0114b_pdf.pdf

- **Protocol Specific Mitigations**: For FIDO devices, a mitigation exists that can be implemented to reduce the attack threat without changing the device. As explained in section 8.1 of [5], a counter *may* be used as a signal for detecting cloned FIDO device. Thus if a web service protected with FIDO receives a cryptographically correct authentication message, but with a counter value smaller or equal to the previous counter value recorded, it means that a clone of the FIDO device has been created and used. Then the web service 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, the account will be locked by the next access from the legitimate user.

7.5 Avenues of Research

Many different directions of research could be investigated, this report leaves open the following questions:

- Sensitive modular inversions are not exclusively found in the ECDSA scheme. It would be interesting to look at the RSA (and especially the key generation) and study the application of EUCLEAK there.
- The proposed attack process should be studied in more depth, starting with a theoretical analysis of its performances. More generally, the study of the minimal quantity of information leaked necessary for attacking the `EEA` would be quite interesting.
- On a more practical side, pushing the attack effectiveness to make it a proper single-trace attack would also be interesting. As well as working on the side-channel acquisition: capturing the signal a bit farther from the chip or capturing the signal on a *contactless-only* product (like a passport) would both be interesting directions.
- If the side-channel analysis of the `EEA` is not quite common in the literature (we did not find any reference), there are few papers on the side-channel analysis of the Binary `EEA` (certainly because the algorithm is more often used in practice), see *e.g.* [9, 2, 1]. It might be interesting to confront this literature with the generic attack approach proposed in our work.
- In [1], one can read *"LibreSSL uses [the EEA] as a side-channel hardened modular inversion function, even though the execution flow of this algorithm also depends on its inputs. No evidence of a side-channel attack on this algorithm variant has been published yet, so its resilience against SCA remains as an open problem. Therefore, to the best of our knowledge, this algorithm is considered safe in this context."* We believe that this assumption should be re-considered in light of the EUCLEAK attack.

7.6 Project Timeline

The project spreads over 2 years, with large pauses and small, often intense, work slots. The project timeline is then a bit fuzzy. Most of what is presented in Chapter 2 – the first reverse engineering phase – was done at the beginning of 2022.

Then the project was paused until spring 2023 where the generic attack (or at least its first version) was developed and tested with the negative conclusion of Chapter 3. The project was then definitely closed during the summer 2023.

The project reopened itself early 2024, as definitely-closed projects sometimes do, and the longest part of the project started, we tried to describe it in Chapter 4. After that, everything went much faster:

- March 21st, 2024:
  - Attack validated on `Feitian A22 JavaCard`.
- April 5th, 2024:
  - Attack validated on `YubiKey 5Ci`.
- April 18th, 2024:
  - Leakage Validation on `Optiga Trust M`.
- April 19th, 2024:
  - Contact Infineon, Yubico, Feitian , CERT-FR (ANSSI), CERT-BUND (BSI), with a short technical description of our work and our coordinated responsible disclosure plan (expected to end at the beginning of September 2024).
  - Acknowledgment of reception from Yubico Security team and ANSSI CERT-FR.
- April 22nd to 24th, 2024:
  - Acknowledgment of reception from BSI CERT-BUND, Infineon and Feitian
  - Feitian explains that the `Feitian A22 JavaCard` has been updated years ago and none of their products is impacted.
- April 25th, 2024:
  - Meeting with Yubico security team, they acknowledge that all `YubiKey 5 Series` are impacted by the attack. Yubico explains that they are actually in the middle of switching from Infineon cryptolib to their own.
- May 2nd, 2024:
  - At the request of Infineon, NinjaLab sends more technical details of the attack.
- May 6th, 2024:
  - `YubiKey` firmware 5.7 update [8], Yubico switches to its own cryptolib.
- May 15th, 2024:
  - `Optiga TPM` evaluation kit ordered on Farnell.
- May 30th, 2024:
  - Notification to Infineon that `Optiga TPM` is also impacted.

[8] https://yubi.co/yubikey-5-7-authenticator-7-blog

- June 24th, 2024:
  - Meeting with ANSSI to present our work and discuss technical details.
- July 18th, 2024:
  - Meeting with BSI to discuss technical details.
- July 26th, 2024:
  - Infineon shares with us the news that they have a patch working; they offer that NinjaLab checks the leakage of a `Optiga TPM` with the patched library, we decline the offer.
- August 27th, 2024:
  - CVE ID requested.
- September 3rd, 2024:
  - publication of this report on the NinjaLab website.
- September 4th, 2024:
  - CVE Assignment Team from MITRE dismissed our CVE request and went on with CVE-2024-45678 [9]. They refuse to acknowledge that the vulnerabilty impacts other products than `YubiKey 5 Series`. Seems like Infineon's strategy of staying completely silent since the EUCLEAK publication is working...
- September 9th, 2024:
  - The BSI produces a short public post (in German) about EUCLEAK [10].
- August 30th, September 5th, 30th and October 8th , 2024:
  - The NCISB and BSI issue `AVA_VAN_5` certificates to replace some of the vulnerable ones [11] [12] [13] [14]. These certificates still involve the vulnerable ECC cryptolib versions, just the user guidance documents are updated. These guides are not public so we still do not know exactly what are the vulnerable hardware and software versions.

Acknowledgements

I would like to thank Victor Lomné for his help and support throughout this project, notably unearthing Infineon `Optiga Trust M` and `Optiga TPM` dev kits and the careful proof-reading of this report. Also Laurent Imbert and Cyril Bouvier for insightful discussions about the Extended Euclidean Algorithm. Finally, special thanks to Marine for keeping our little monsters busy, I owe you one! Ready for your HDR yet?

[9] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2024-45678

[10] https://www.bsi.bund.de/DE/Service-Navi/Presse/Alle-Meldungen-News/Meldungen/EUCLEAK_Seitenkanalangriff_240909.html

[11] https://www.trustcb.com/download/nscib-cc-2200060-02-cert/

[12] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1229.html

[13] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1110.html

[14] https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1079.html

Bibliography

[1] A. C. Aldaya, C. P. García, and B. B. Brumley. From A to Z: Projective coordinates leakage in the wild. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):428–453, 2020.

[2] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019.

[3] 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.

[4] Dino-Lite. Dino-Lite digital microscope AM4113TL. https://www.dino-lite.eu/en/am4113tl, 2019. [online; accessed 2-September-2024].

[5] 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].

[6] W. Fischer. Device and method for determining an inverse of a value related to a modulus, U.S. Patent 08290151B2, Oct. 2012.

[7] W. Fischer. Device and method for determining an inverse of a value related to a modulus, U.S. Patent 10318245B2, Jun. 2019.

[8] D. Freedman, R. Pisani, and R. Purves. Statistics. *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[9] C. P. García and B. B. Brumley. Constant-Time Callees with Variable-Time Callers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 83–98, Vancouver, BC, Aug. 2017. USENIX Association.

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

[11] International Civil Aviation Organization (ICAO). Machine readable travel documents. part 11: Security mechanisms for MRTDs. Technical report Doc 9303. https://www.icao.int/publications/Documents/9303_p11_cons_en.pdf, Eighth Edition, 2021. [online; accessed 2-September-2024].

[12] A. Janovsky, Ł. Chmielewski, P. Svenda, J. Jancar, and V. Matyas. Chain of Trust: Unraveling References Among Common Criteria Certified Products. In N. Pitropakis, S. Katsikas, S. Furnell, and K. Markantonakis, editors, *ICT Systems Security and Privacy Protection*, pages 191–205, Cham, 2024. Springer Nature Switzerland.

[13] A. Janovsky, J. Jancar, P. Svenda, Łukasz Chmielewski, J. Michalik, and V. Matyas. sec-certs: Examining the security certification practice for better vulnerability mitigation. *Computers & Security*, 143:103895, 2024.

[14] T. Jebelean. A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers. *Journal of Symbolic Computation*, 19(1):145–157, 1995.

[15] 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 2-September-2024].

[16] LeCroy. WavePro HD Oscilloscope datasheet. https://cdn.teledynelecroy.com/files/pdf/waveprohd-datasheet.pdf, 2023. [online; accessed 2-September-2024].

[17] Ledger Donjon. Ledger Scaffold Documentation. https://donjonscaffold.readthedocs.io/en/latest/index.html, 2020. [online; accessed 2-September-2024].

[18] V. Lomné. An Overview of the Security of Some Hardware FIDO(2) Tokens – Hardwear.io Netherlands 2022 conference. https://hardwear.io/netherlands-2022/presentation/security-of-Hardware-FIDO(2)-tokens.pdf, Oct. 2022.

[19] V. Lomné and T. Roche. A Side Journey to Titan. https://ninjalab.io/wp-content/uploads/2022/05/a_side_journey_to_titan.pdf, Feb. 2021.

[20] Martin Paljak. Ant JavaCard Project Github repository. https://github.com/martinpaljak/ant-javacard. [online; accessed 2-September-2024].

[21] 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.

[22] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1631–1648. ACM, 2017.

[23] 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 2-September-2024].

[24] Oracle. JavaCard Connected Platform Specifications 2.2.2. https://www.oracle.com/java/technologies/java-card/platform-specification-v222.html. [online; accessed 2-September-2024].

[25] Pico Technology. PicoScope 6000E Series datasheet. https://www.picotech.com/download/manuals/picoscope-6000e-series-data-sheet.pdf, 2019. [online; accessed 2-September-2024].

[26] B. J. M. Pollard. Monte Carlo methods for index computation (). *Mathematics of Computation*, 32:918–924, 1978.

[27] T. Roche, V. Lomné, C. Mutschler, and L. Imbert. A Side Journey To Titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248. USENIX Association, Aug. 2021.

[28] Thorlabs. Manual 3-axes Stage PT3/M. https://www.thorlabs.com/thorproduct.cfm?partnumber=PT3/M#ad-image-0, 2019. [online; accessed 2-September-2024].

Appendix A

YubiKey 5C Case Opening

The EUCLEAK attack relies on timing leakages and then does not need ultra-high quality side-channel traces. In fact, during this work, we never needed to open a chip package to put our EM probe very close to the die [1].

Nevertheless, capturing the EM signal with a small EM probe would not work if this probe is too far from the chip. We hence have to open the YubiKey plastic case to access its logic board. We bought two YubiKey 5C (see Figure A.1) and tried two different case openings.

Figure A.1: Brand New YubiKey 5C

First by cutting the edge of the product until the plastic case can be opened in half, the result is shown in Figure A.2. The process takes few minutes, the device is still working. The second approach takes even less time, we just dig a hole at the right place, this takes few seconds, the result is shown on Figure A.3.

[1] For instance, in our previous work on Google Titan Security Key [19], the side-channel acquisition needed much more care.

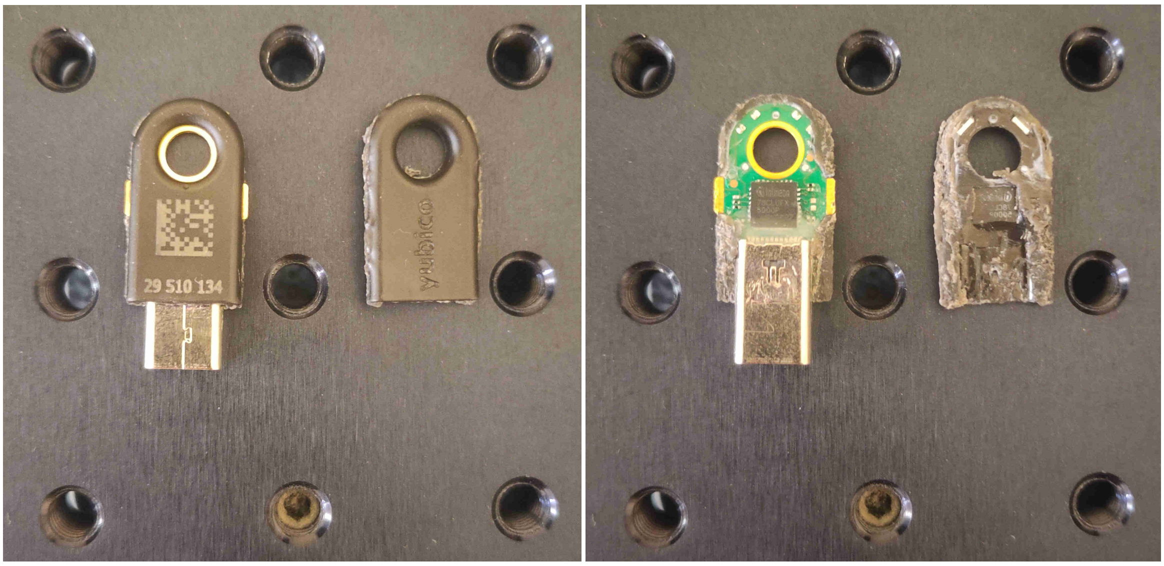


Figure A.2: `YubiKey 5C` – First Opening

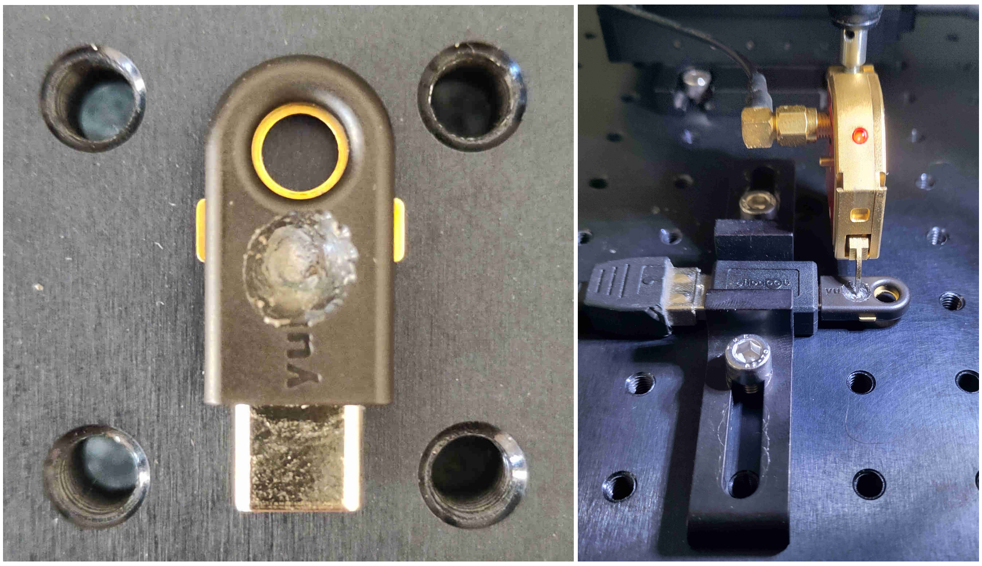


Figure A.3: `YubiKey 5C` – Second Opening

In both cases however, the device needs to be re-packaged if the adversary wants to give it back to legitimate user without him noticing. We did not study further this issue.

Appendix B

Infineon Security Microcontrollers

Name	Certificate Reference	Dates	#Certificates/ #Maintenances	Library Version	Link
M7820 A11	BSI-DSZ-CC-0640	2010-2010	1/2	EC v1.1.18	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0640.html
	BSI-DSZ-CC-0728	2011-2011	1/1	EC v1.02.008	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0728.html
	BSI-DSZ-CC-0758	2012-2012	1/1	EC v1.02.013	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0758.html
	BSI-DSZ-CC-0829	2012-2017	2/3	EC v1.02.013	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0829_V2.html
M7801 A12	BSI-DSZ-CC-0606	2010-2010	1/0	EC v1.1.18	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0606.html
	BSI-DSZ-CC-0727	2011-2011	1/0	EC v1.02.008	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0727.html
M7820 M11	BSI-DSZ-CC-0695	2011-2013	1/2	EC v1.02.008	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0695.html
M7892 A21	BSI-DSZ-CC-0813	2012-2012	1/0	EC v1.02.008	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0813.html
M7892 B11	BSI-DSZ-CC-0782	2012-2020	5/1	EC v1.02.013 or v2.07.003	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0782_0782V2_0782V3_0782V4_0782V5.html
M7893 B11	BSI-DSZ-CC-0879	2014-2022	5/1	EC v1.03.006	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0879_0879V2_0879V3_0879V4_0879V5.html
M7892 D11 and G12	BSI-DSZ-CC-0891	2015-2024	7/1	EC v1.02.013 or v2.03.008	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0891
M7892 P11	BSI-DSZ-CC-1105	2020-2020	1/0	EC v2.03.008 or v2.07.003	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1105.html
SLB96xx (TPM)	BSI-DSZ-CC-0844	2014-2017	2/1	v4.43.0257.00, v4.43.0258.00 and v4.43.0259.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0844_0844V2.html
SLB9665 (TPM)	BSI-DSZ-CC-0965-2015	2015-2015	1/0	v5.51.2098.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0965.html
	BSI-DSZ-CC-1020	2016-2017	2/1	v5.62.3126.00, v5.62.3127.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1020_1020V2.html
	BSI-DSZ-CC-1056-2018	2018-2018	1/0	v5.63.3144.00, v5.63.3149.00, v5.63.3353.00, v5.63.3355.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1056.html
SLB9670 (TPM)	BSI-DSZ-CC-0958	2015-2017	2/2	v6.43.0243.00, v6.43.0244.00, v6.43.0245.00 and v6.43.0246.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0958_0958V2.html
	BSI-DSZ-CC-0998-2016	2016-2016	1/1	v7.40.2098.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0998.html
	BSI-DSZ-CC-1021	2016-2017	2/1	v7.62.3126.00, v7.62.3127.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1021_1021V2.html
	BSI-DSZ-CC-1057-2018	2018-2018	1/0	v7.63.3144.00, v7.63.3149.00, v7.63.3353.00, v7.63.3355.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1057.html
	BSI-DSZ-CC-1058-2018	2018-2018	1/0	v7.83.3358.00, v7.83.3360.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1058.html
	BSI-DSZ-CC-1086-2018	2018-2018	1/1	v7.85.4555.00, v7.85.4567.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1086_1086ma01.html
	BSI-DSZ-CC-1100-2018	2018-2018	1/0	v13.11.4555.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1100.html

Table B.1: Infineon Security Microcontrollers – M78XX (SLE78) – 16-bit, 90 nm

Name	Certificate Reference	Dates	#Certificates/ #Maintenances	Library Version	Link
IFX_CCI_000003h	BSI-DSZ-CC-1110	2019-2023	6/1	EC V2.06.003 or V2.07.003 or V2.08.007	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1110.html
	BSI-DSZ-CC-0945	2017-2018	3/3	EC V2.06.003 or V2.07.003	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0945_0945V2_0945V3.html
IFX_CCI_00000Fh	BSI-DSZ-CC-1079	2018-2023	4/1	EC v2.07.003, v2.08.007, or v3.33.003	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1079.html
IFX_CCI_00007h	BSI-DSZ-CC-0961	2017-2022	6/2	EC V2.06.003, V2.07.003, V2.08.007	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/0961_0961V2_0961V3_0961V4_0961V5_0961V6.html

Table B.2: Infineon Security Microcontrollers – IFX_CCI_0-0Xh – 16-bit, 65 nm

Name	Certificate Reference	Dates	#Certificates/ #Maintenances	Library Version	Link
IFX_CCI_000011h	BSI-DSZ-CC-1025	2018-2023	5/0	EC v2.08.006 or v3.03.003 or v3.04.001	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1025.html
IFX_CCI_001Fh (SLC37)	BSI-DSZ-CC-1102	2019-2020	1/1	NA	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1102.html
IFX_CCI_00003Fh (SLC37)	NSCIB-CC-2200060-01-CR	2023-2023	1/0	EC v3.03.003 or v3.04.001	https://www.trustcb.com/download/nscib-cc-2200060-01-cr/
	NSCIB-CC-2200060-02-CR	2024-2024	1/0	EC v3.03.003 or v3.04.001 or v3.05.002	https://www.trustcb.com/download/nscib-cc-2200060-02-cr/
	NSCIB-CC-0173264-CR3	2021-2021	1/0	EC v3.03.003	https://www.tuv-nederland.nl/assets/files/cerfiticaten/2021/12/cc-21-0173264-cr3-1.0.pdf
SLB9672/SLB9673 (TPM)	BSI-DSZ-CC-1113	2021-2023	5/0	v15.20.15686.00, v15.21.16430.00, v15.22.16832.00 und v15.23.17664.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1113.html
	BSI-DSZ-CC-1178	2021-2023	4/0	v16.10.16488.00, v16.12.16858.00 and v26.10.16688.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1178_1178V2_1178V3.html
	BSI-DSZ-CC-1179	2021-2023	4/0	v17.10.16488.00, v17.12.16858.00, v17.13.17733.00, v27.10.16688.00 and v27.13.17770.00	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1179_1179V2_1179V3.html

Table B.3: Infineon Security Microcontrollers – IFX_CCI_0-0XYh – ARM SC300, 40/65 nm

Name	Certificate Reference	Dates	#Certificates/ #Maintenances	Library Version	Link
IFX_CCI_00007Dh	BSI-DSZ-CC-1229	2024-2024	1/0	CryptoSuite v4.06.002	https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1229.html

Table B.4: Infineon Security Microcontrollers – IFX_CCI_00007Dh – Armv8-M, 28 nm