Side-Channel assessment of Open Source Hardware Wallets

Manuel San Pedro, Victor Servant, and Charles Guillemet
`manuel.sanpedro@ledger.fr`, `victor.servant@ledger.fr`,
`charles@ledger.fr`

Ledger Donjon

Abstract. Side-channel attacks rely on the fact that the physical behavior of a device depends on the data it manipulates. We show in this paper how to use this class of attacks to break the security of some crypto-currencies hardware wallets when the attacker is given physical access to them. We mounted two profiled side-channel attacks: the first one extracts the user PIN used through the verification function, and the second one extracts the private signing key from the ECDSA scalar multiplication using a single signature. The results of our study were responsibly disclosed to the manufacturer who patched the PIN vulnerability through a firmware upgrade.

1 Introduction

The paper is organized as follows: section 1 briefly presents the target of our evaluation, a hardware wallet (section 1.1) and introduces side-channels (section 1.2). At section 2, the setup used to mount our attacks is described. Section 3 presents our side-channel attack on the PIN authentication mechanism of the *Trezor One* device, and section 4 presents the side-channel Analysis of the scalar multiplication implemented in `trezor-crypto` library. We also present at section 5 our emulator tool, `Rainbow` [1], which could have been used to identify from the code only the presented side-channel vulnerabilities.

1.1 Blockchain and Hardware Wallets

Crypto-currencies use blockchain technology which is secure by design. Blockchain technology pushes the security problem to the user who has the sole responsibility of keeping his funds safe. Owning cryptocurrencies only means knowing the private key to which the funds correspond. Spending cryptocurrencies (making a transaction) means proving the knowledge of the private key by computing a digital signature.

Wallets are means to store and use these private keys. There are different kinds of wallets such as:

— Software wallets: online, mobile, desktop,... They are cheap and convenient while present significant risks in terms of security.
— Paper wallets: These wallets are very cheap, the security of these rely on the physical management of the private keys, while they are not very convenient when it comes to performing a transaction.
— Hardware wallets: They present the best trade-of between convenience and security.

Hardware wallets have been designed to prevent the access to the private keys they protect, because they never leave the device. This is called the principle of isolation, also known as *cold storage*. The private keys are stored and used inside the device, they are never *hot* (online), avoiding their exposition to the internet or to the computer to which it is connected.

1.2 Side-channel Analysis

Side-channel analysis relies on the fact *that the physical behavior of a device depends on the data it manipulates*. An attacker able to measure the physical behavior can characterize this dependency in order to retrieve information on sensitive data.

Side-channel attacks can leverage several physical behaviors (the so-called side-channels):
— Execution time (see [13]),
— Power consumption of the device: can be measured using a shunt resistor and a current probe plugged to an oscilloscope (see [14]),
— Electromagnetic emanation of the device: can be measured using an EM probe and an amplifier plugged to an oscilloscope (see [8]).

The physical leakages (called side-channel traces) are recorded using a digital oscilloscope and a statistical post-processing is applied to extract information about sensitive data.

Side-channel attacks can be divided in two categories: *profiled* and *non-profiled* attacks .

Profiled attacks can be applied when an attacker has access to an open device, on which she is able to characterize the physical behavior (also called *leakage*) of a sensitive value she targets. This characterization is called the *learning* phase, and results in a database describing the physical behavior of the sensitive values on the target. Once the *learning* or *profiling* is done, the attacker can then use this database on a whole new device, with an unknown sensitive value, that will be retrieved in a certain number of attempts (*i.e.* traces). Classical state-of-the art profiled attacks are Template Attacks (cf. [7]), Machine Learning-based attacks (cf. [11]) and more recently Deep-Learning based Attacks (cf. [16]).

Non-profiled attacks use the same mechanism but without prior leakage characterization. That means the attacker does not need an open device on which she knows or controls the sensitive values. In this situation the attacker needs to induce the leakage model herself. Classical leakage models exist in order to accomplish this but *non-profiled* attacks are less efficient.

The context of an open source code running on a general purpose microcontroller unit such as the Trezor One lends itself perfectly to profiled side-channel attacks.

One important point to mention is that side-channel attacks make use of a *Divide & Conquer strategy*: the secret value is often recovered chunk-wise, given the implementation does not (and a vast majority of time, can not) use the whole cryptographic secret in a single cycle.

2 Target and setup

Several targets have been considered during this study. The Ledger Nano S, Keepkey and Trezor are the main hardware wallets on the market. During this paper, we will focus on two paramount security mechanisms within a Hardware Wallet:

— the PIN authentication: breaking the PIN would allow an attacker to empty all accounts.

— the scalar multiplication : used within elliptic curve signature, it is used to sign every transaction on the blockchain.

2.1 Trezor One hardware wallet

Trezor is an Open Source Hardware Wallet created by a Czech company called SatoshiLabs. Trezor has developed 2 different products: the *Trezor One* and the *Trezor model T*. *Trezor One*s is the star product of the company. As described at figure 1, the *Trezor One*s device is built around a STM32F205RE MCU. The PCB also contains:

— Two buttons used to get user inputs.

— A small screen to display information to the user

— An 8-MHz external crystal

The STM32 is a very popular chip family, which includes several variants, depending on the targeted application: Low Power, DSP, High performances,... (see [2]). This chip however does not implement hardware

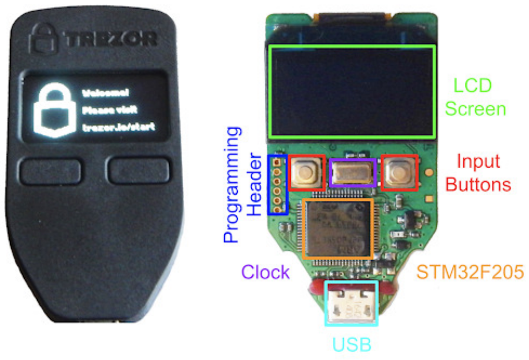


Fig. 1. Trezor PCB description, from [15]

security countermeasures. The core of the STM32F205RE is an ARM 32-bits Cortex-M3 which runs up to 120MHz. As the chip doesn't embed any cryptographic accelerator, all long-integer arithmetic operations are performed by the CPU.

Although the STM32 MCU is not designed for security, the Trezor device is used for a security application: it is a Hardware Wallet. Its purpose is to:

— generate a BIP32 seed, which will be used to derive public/private keys

— store public and private keys for receiving or sending cryptocurrencies.

— perform cryptocurrency transactions

From the manufacturer website, we noticed the security of the device relies on a few different items such as:

— Secure PIN authentication function

— Confidentiality of the data stored inside the device

These security claims can be challenged (and have been challenged) using various attack vectors: software attacks, fault attacks, side channel attacks. This article focuses only on side-channels.

2.2 Our setup

The figure 2 summarizes the setup used to mount our attack: a computer requests specific operations to a *Trezor One*s, while a digital oscilloscope measures its power consumption throught a resistor plugged onto the device.

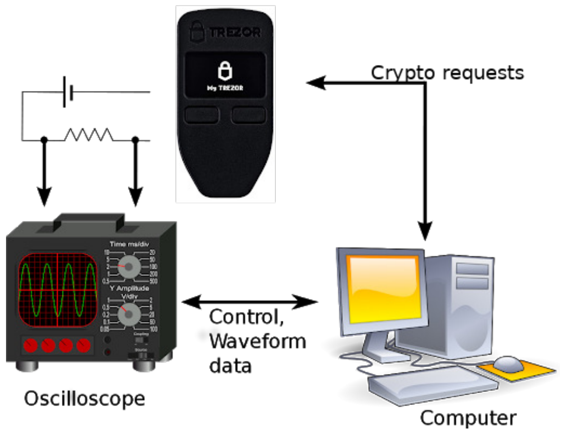


Fig. 2. Schematic representation of our Side-channel setup

On the Trezor One side:

In order to measure real time power consumption, a 5Ω resistor is inserted in the VCC line of the device to measure real time power consumption (see figure 3)

A slightly modified version of the firmware available at [4] was loaded onto our Trezor One device, which allows easy characterization for the profiling phase of the attacks.

— NVM writes have been disabled: since the characterization needs hundred of thousands executions of an operation

— A GPIO pin (see black wire on the left at figure 3) is used as a trigger mechanism: the GPIO is pulled up at the beginning of the operation, and pulled down at the end, hence framing the targeted operation.

— Various Trezor security mechanisms were disabled: such as the Pin Try Counter, the increasing timer between wrong PIN requests, the `pinMatrix` randomization.

This modified firmware was used only on the device used for the characterization of our attacks. The firmware used to actually pass our attack was not modified.

On the digital oscilloscope side:

We use a Tektronix MSO54 at Ledger's Donjon. The sampling rate for both our attacks is set to $3\text{GSamples} \cdot s^{-1}$, with a 500MHz bandwith. The sampling rate might appear high (the attacked MCU runs at 120MHz),

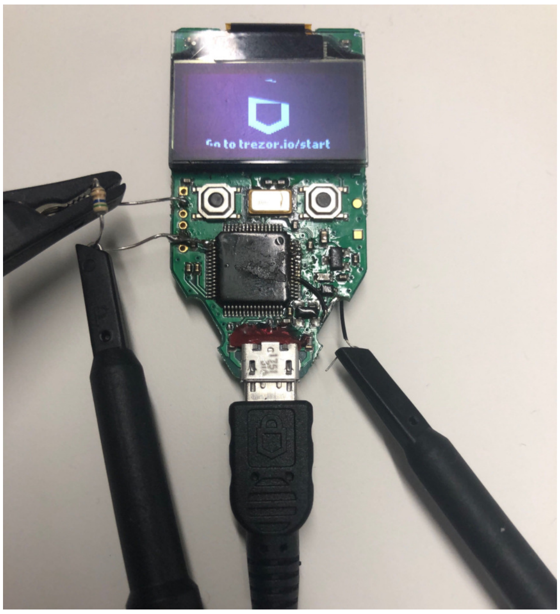


Fig. 3. Trezor One device prepared for side-channel: the probe to the left measures the power thanks to a resistor, and the probe to the right is plugged to a GPIO for triggering the scope.

the attack works probably well using a cheaper scope with a much lower sampling rate.

On the computer side:

We use `python-trezor`, the Python library and commandline client `trezorctl` for communicating with Trezor Hardware Wallet. Beside that, all our scripts use `lascar` to manage the setup: `lascar` is the open source side-channel library developed at Ledger Donjon (see [18]). The script is in charge of the following:

- — request the Trezor device to perform an operation with specific inputs
- — acquire power traces from the oscilloscope
- — store the side-channel data
- — process the data/perform the attack

3 Breaking PIN authentication

This section describes the steps we took to mount a profiled side-channel attack leading to a Trezor user PIN recovery.

First at subsection 3.1 we will present the targeted function: `storage_containsPin`. Then subsection 3.2 will describe what we call a *leakage characterization*. The subsection 3.3 describes the matching phase. At subsection 3.4, we summarize how we actually applied and optimized the so-called *profiled* side-channel attack. We finally present an optimization strategy at subsection 3.5.

3.1 Targeted function

As most hardware wallets, *Trezor One* offers a PIN authentification mechanism prior to almost all operations, including transactions (*i.e.* accessing private keys).

Searching in their firmware code (up to version 1.8) leads us to the `storage_containsPin` function, which implements the user-PIN verification and whose source code is shown in listing 1.

Every time a user inputs a PIN, it passes through this function, and is compared to `storageRom->pin`, a N-digit `char` array. This is the value that we are targeting.

```c
/* Check whether pin matches storage.  The pin must be
 * a null-terminated string with at most 9 characters.
 */
bool storage_containsPin(const char *pin)
{
    /* The execution time of the following code only depends on the
     * (public) input.  This avoids timing attacks.
     */
    char diff = 0;
    uint32_t i = 0;

    while (pin[i]) {
        diff |= storageRom->pin[i] - pin[i];
        i++;
    }

    diff |= storageRom->pin[i];
    return diff == 0;
}
```

Listing 1. The source code of the user-PIN verification function, from [4].

From the code presented in listing 1, we can see that the function has been designed to resist timing side-channel attacks, but *time* is not the problem.

We also noticed that `storageRom->pin` digits are processed one after other and the comparison with `pin` is done in a deterministic way.

Observing the power consumption of the `storageRom->pin` function allows to implement a *Divide & Conquer strategy*: instead of brute-forcing a N-digit PIN (9^N possible values), we will attack each PIN digit independently, leading to N side-channel attacks, each one of them on a single digit ($N \times 9$ possible values).

From a side-channel perspective, there are several *sensitive values* in this function which depend on the secret `storageRom->pin`: the value at each step (digit) of the `while` loop of:

$$f_i(\texttt{storageRom->pin}, \texttt{pin}) = \texttt{storageRom->pin}[i] - \texttt{pin}[i], \text{ for } 0 \leq i < N$$

Looking at the code shown in listing 1, we can deduce from the typing used that f_i can output only 18 differents values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 248, 249, 250, 251, 252, 253, 254, 255$. The f_i functions handle both the secret and the input value, allowing a side-channel attacker to induce differentiability. These N *sensitive values* will be characterized and used for the profiled attack in the next subsection.

3.2 Leakage characterization

From the setup presented in section 2.2, we acquire side-channel traces resulting from executions of the `storage_containsPin` function. Figure 4 displays several power traces. From now, for the sake of clarity, we will only work with 4-digit PIN ($N = 4$): since all digits are processed independently from one another, the attack can be extended to any number of digits.

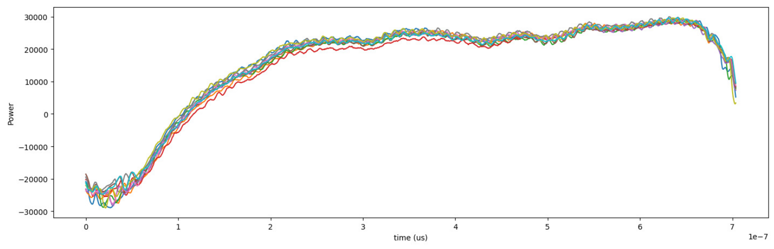


Fig. 4. 10 power traces of the PIN verification function.

In order to perform our profiled attack, we acquired 150000 such traces, where we set random values both for `storageRom->pin` and `pin`. This set of power traces (with the corresponding `storageRom->pin`/`pin` values) will be our *profiling set*.

From this *profiling set*, and the 4 *sensitive values* f_i defined at section 3.1, we'll use a statistical tool dedicated to leakage detection (a *distinguisher*) to measure the dependency between our side channel traces (the power traces) and the processed data (the values of `storageRom->pin/pin`). We chose the Normalized Inter-Class Variance (NICV, see [6]) in order to do so.

For each *sensitive value* f_i, NICV consists in our case in partitioning the traces into 18 classes (1 class for each possible output for f_i). The mean of each batch is computed and the variance of those 9 means is compared to the variance of all traces.

$$NICV(traces, f_i) = \frac{Var[E[traces|f_i]]}{Var(traces)}$$

A NICV close to 0 means that our partitioning (*i.e. sensitive value* f_i) failed to explain the variance. A NICV of 1 means that our partitioning perfectly explained the variance.

The four NICV (one per f_i) are computed on the *profiling set*, and the results are displayed on figure 5.

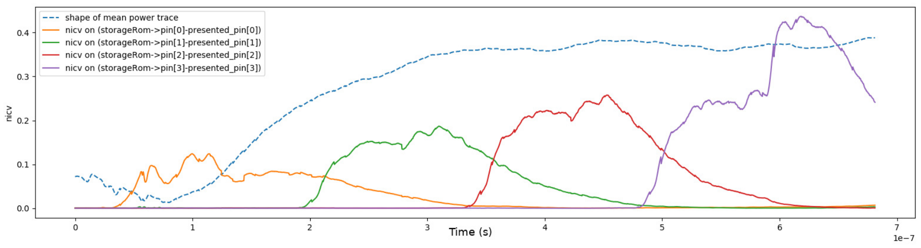


Fig. 5. NICV curves for our 4 *sensitive values* f_i.

As we can see, each NICV appears to *peak* one after the other, following the comparison order as expected. One can also notice that the larger i, the higher the NICV of f_i (this particular effect will not be explained nor used).

These NICV curves attest the strong dependencies between our traces and our *sensitive values* and conclude our *leakage characterization.*

3.3 Profiled attack

As explained in section 2.2, *profiled* side-channel attacks are in two phases:

— the learning phase: where we use an open device A to learn how it behaves

— the matching phase: where we use what we have learned on a new device B whose secret PIN is unknown.

Learning phase: on an open device

From the *profiling set* used in the previous paragraph, the next step is to *profile* each of the *sensitive values.* Our side-channel data is basically an instance of Machine Learning classification (see [9], [12]):

Based on a training set of data containing observation whose categories are known, a classification problem consists in identifying to which category a new observation belongs.

In our case:

— *an observation* is a power traces from a PIN login attempt on the open device A
— *the training set of data* are the power traces from the *profiling set*
— *the categories* are the values returned by f_i at each trace
— *the new observation* is a power trace from a PIN attempt on the device B for which we don't know the secret PIN.

For each digit i ($0 \leq i < 4$), we build $\mathsf{Classifier}_i$, by feeding it with the power traces from the *profiling set* L, labeled with the value of $f_i(\texttt{storageRom->pin\_j}, \texttt{pin}_j)$. At the end of this learning phase, we get a statistical classifier: a decision function that is designed to predict the value of `storageRom->pin`[i]. From a new power trace l, for which we only know the value of `pin` (but not `storageRom->pin`), we get:

$$\mathsf{Classifier}_i(l) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

Moreover, the information brought by these probabilities can be accumulated by using multiple power traces captured during PIN attempts on the same target device B.

Let $\mathrm{L} = (l_j)_{0 \leq j < m}$ be a set of m such power traces. Then we use all the power traces to retrieve `storageRom->pin`:

$$\mathsf{Classifier}_i(\mathrm{L}) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

There exists a lot of different statistical classifiers (LDA, QDA, SVM, AdaBoost, neural networks), which led to the same results for our attack. The classifiers we build for each digit in our attack are all Linear Discriminant Analysis classifiers.

Linear Discriminant Analysis is a method used in statistics, pattern recognition and machine learning to find a linear combination of features that characterizes or separates different classes of objects or events. The resulting combination may be used as a linear classifier. In our case, the so-called classes are the values of our f_i.

Now that the learning phase is done, we have 4 classifier functions, each of them in charge of retrieving a PIN digit from a power trace. Their efficiency will be tested in the next subsection, the matching phase.

Matching phase: retrieving the first PIN digit on a device B

This subsection presents the results of the attack on a new device B. The matching phase consists in applying the previously built statistical classifiers on new power traces, acquired from a new *Trezor One* device, with unknown `storageRom->pin`, and random known values for the *presented* `pin`.

We will first describe how a single attack is mounted. Then we will present the results we got from multiple attacks launched on this new device.

From a new *Trezor One* device on which the PIN is unknown, we acquire 15 power traces resulting from a PIN authentication, with a fixed 4-digit `storageRom->pin` and a random 4-digit *presented* `pin`. The Max PIN Tries on a *Trezor* device is 15. Beyond this value the device wipes its data, which means the attack has to succeed within 15 traces.

In this example, we only show the attack on `storageRom->pin`[0]: the resulting traces are passed one-by-one through $\mathsf{Classifier}_0$. With each new trace, $\mathsf{Classifier}_0$ returns a log-probability (*i.e.* a score) for each possible value of `storageRom->pin`[0]. The digit with the highest score is returned by the classifier as the most likely value for `storageRom->pin`[0].

Figure 6 shows the progression of the 9 scores for each possible value of the digit 0. As we can see, from the 6^{th} trace, the value reaching the best score is also the value of the solution (digit0 == 1, plotted with red $\times$). This means that $\mathsf{Classifier}_0$ needed only 6 power traces to return the correct solution for digit0.

Matching phase: generalization on all digits and average results

To demonstrate the effectiveness of our attack, we acquired 300 sets of 15 power traces, each set sharing the same 4-digit `storageRom->pin`. Then we applied on each of these sets, on each of the PIN digits, the attack illustrated at figure 6.

As a metric for our study, we monitored the progression of the rank of the correct solution along the 15 traces. Figure 7 displays the mean ranks given by the 300 attacks on each one of the 4 digits. What these 4

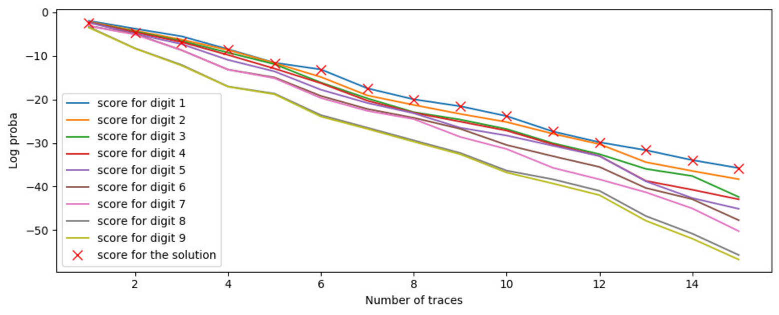


Fig. 6. Matching phase of a single classifier on a set of 15 traces: here we use $\mathsf{Classifier}_0$. The 9 curves represent the progression of the score for each possible digit of the PIN. The score for the solution (correct PIN digit) is plotted in red with $\times$.

curves show is that the attack is a success: after 10 traces (PIN attempts), we **always** get `storageRom->pin[i]` at rank 1 for all digits i ($0 < i <= N = 4$).

3.4 Summing up

In order to actually mount the attack, the attacker has to guess the correct value of the PIN and to input it on the device B. To do so, he first performs the learning phase using his own device A. Then, he can for instance try 10 random PINs on the device B, gather the power consumption measurements of this device during the PIN verification and apply the matching phase on these 10 tries. The matching phase will provide him the most likely value for each digit which trivially gives the most likely value for the whole PIN. The success rate of our matching with 10 traces is 100%, which means the attacker will input the correct value of the PIN for the 11^{th} try and unlock the device.

The device also implements an exponential waiting time. An incorrect PIN will trigger a waiting time that is twice that of the previous attempt, starting from 1 sec. In this setup, the attacker would need 511 seconds to guess the correct value of the PIN, and then wait 512 additional seconds to input it (around 17 min).

3.5 Improving the attack

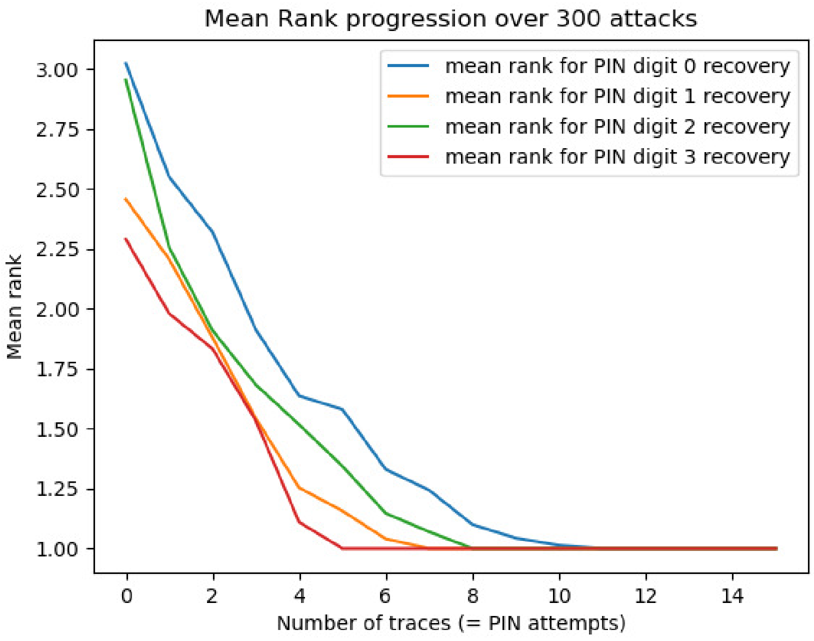


Fig. 7. Mean Rank progression of the attacks on the 4 PIN digit.

A possible improvement of the attack consists in choosing the PIN presented to maximize the information gathered with each trace during the matching phase. Indeed, the performance of the matching phase depends on the value of the correct PIN but also on the value of the input PIN.

The figure 8 shows a strong bias in the matching performance. Several remarks can be made:

— The matching performance depends on the value of the correct PIN and also on the value of the presented PIN.
— When the presented digit is correct the matching works very well ($\approx 100\%$).
— The matching performance also depends on the position of the digit PIN. This can mainly be explained by the measurement itself.

If the target is not directly the result of the subtraction but instead its generated carry, the performance of the matching is close to 100% with only one trace, cf Figure 9. On the other hand, it only indicates if the value correct PIN is greater than the presented one. Indeed the code implements `storageRom->pin - pin[i]` subtraction. Looking at the generated binary, we noticed this subtraction is integer promoted

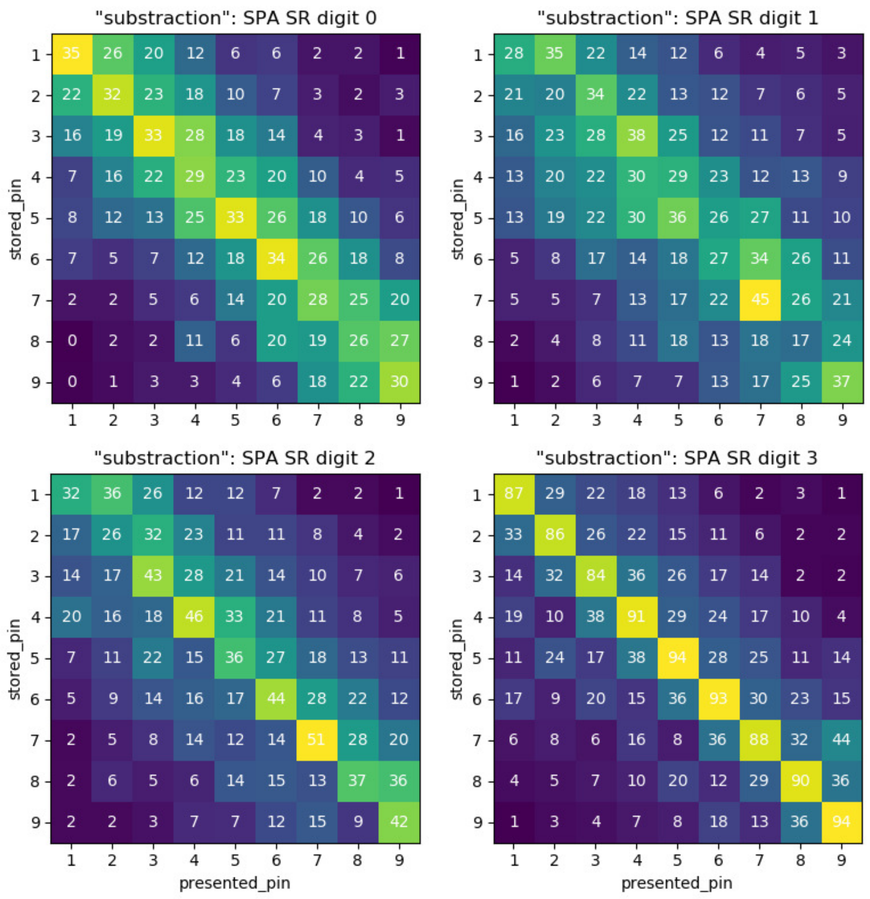


Fig. 8. Matching performance as a function of presented digit and correct digit

(see listing 2, that means this subtraction produces values of the form `0xFFFF FFxx` when the input PIN digit is larger than the store PIN digit, and `0x0000 00xx` values when it is not, which induces a large difference in power consumption. These integers are finally cast to unsigned bytes at the end of the comparison loop iteration. This explains the difference in matching performance. It's possible to use this fact to implement a dichotomy in the chosen PIN strategy, however it yields slightly poorer results compared to the strategy presented hereafter.

```
    ; r1 = stored_pin
    ; r3 = input_pin
    ; r6 = input_pin
    subs      r1, r1, r6    ; 32 bits subtraction
```

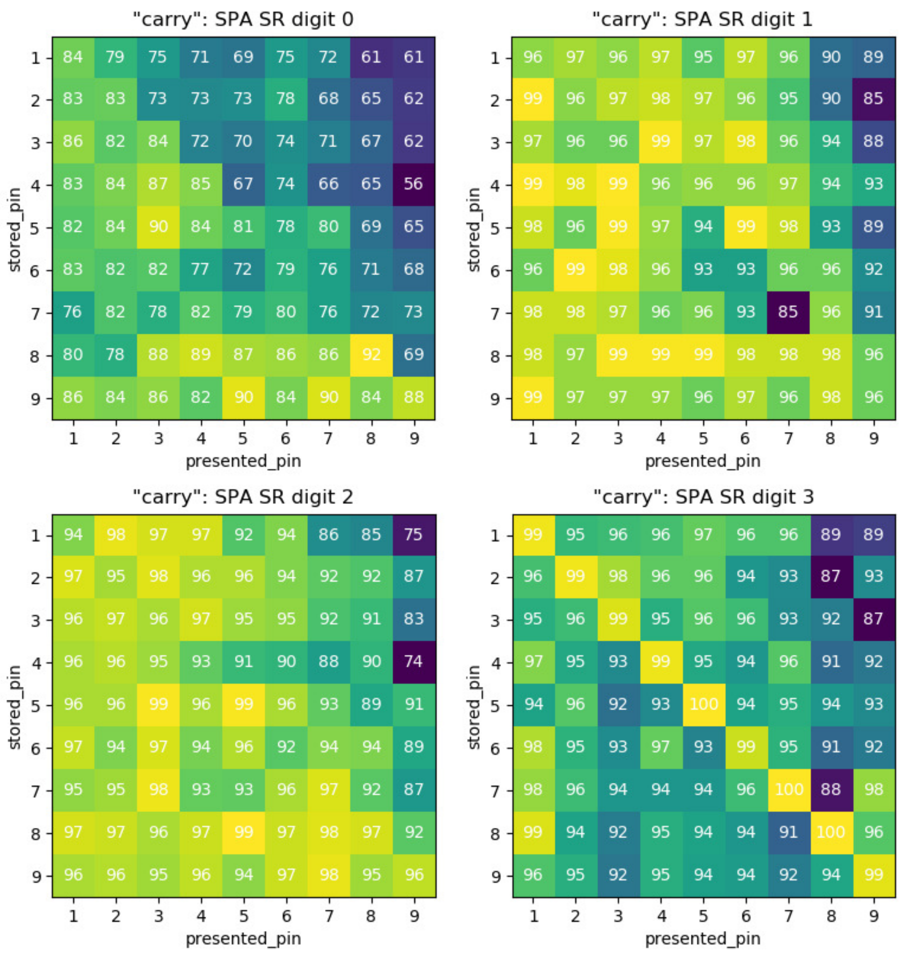


Fig. 9. Matching performance targeting the carry of the subtraction

```
    orrs        r3, r1          ; 32 bits OR
    uxtb        r3, r3          ; cast to byte
```

Listing 2. PIN digit subtraction-comparison

Taking these observation into account, it is then possible to improve the attack performance using two distinct techniques:

— Perform the learning step according to the presented PIN digit
— Implement a chosen PIN strategy

Learn the correct PIN value knowing the presented PIN

As the performance of the matching depends both on the correct PIN value and on the presented PIN value, one can take advantage of this. The principle of this optimization is the following: instead of profiling the result of the subtraction $storageRom- > pin[i] - pin[i]$, we'll profile the value of the correct PIN value knowing the value of the presented PIN value. Considering the first digit, 9 profiling phases are performed. For each i, k, we build $\mathsf{Classifier}_{i,k}$, by feeding it with the power traces from the *profiling set* l_j, labeled with the value of $f_{i,k}((\texttt{storageRom->pin\_j}, \texttt{k}_j)$. At the end of this learning phase, we get k statistical classifiers for each digit i: decision functions that are designed to predict the value of `storageRom->pin`[i] depending on the value of `pin`[i]=k. From a new power trace l, for which we only know the value of `pin` but not `storageRom->pin`, we get:

$$\mathsf{Classifier}_{i,k}(l) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

These classifiers are more accurate since they take into account more precisely the value of the input PIN. Furthermore, the classification is mapped to 9 distinct values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ instead of the possible 18 subtraction values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 248, 249, 250, 251, 252, 253, 254, 255$.

Match with a chosen PIN strategy

Using these efficient classifiers, we can now implement a chosen PIN strategy to attack the PIN verify function.

1. Input 5555 as PIN (on average, this is the value which gives the most information). If this is the correct value of PIN, the attack is successful. Otherwise the corresponding trace $l0$ is retrieved and matched with $\mathsf{Classifier}_{i,5}(l0)$ which will give probabilities for each possible digit value.
2. Input the most likely and not yet presented PIN value. If this is the correct value of the PIN digit, the attack is successful. Otherwise the corresponding trace l_j is retrieved and matched with $\mathsf{Classifier}_{i,k}(l_j)$ which will give probabilities for each possible digit value and go to 2.

This pseudo algorithm gives an efficient way to retrieve the correct value of PIN and log in the device.

On a set of 300 traces, 4.8 tries are necessary to log into the device. This corresponds to ≈ 10 sec to break the security of the device due to the implemented waiting time.

4 Breaking Scalar Multiplication

In this section, we will describe how to mount a side-channel attack on the scalar multiplication from `trezor-crypto`, the open-source cryptographic library developed by Trezor (see [3]), used on its device, but also used by other hardware-wallets such as Keepkey and Archos Safe-T. The attacks presented below, as for the PIN comparison, are performed on a *Trezor One*.

The scalar multiplication is used on elliptic curve operations. Let $\mathbb{C}$ be an elliptic curve, $P \in \mathbb{C}$ a point on this curve, and $k \in \mathbb{N}$ a positive integer. The scalar multiplication computes the point $G = [k]P$, which also lies on the curve.

The reason we are evaluating the scalar multiplication implementation is that it is used with sensitive parameters:

— During an elliptic curve public key derivation: the scalar used is the value of the *private key*

— During an ECDSA signature: the scalar used is a nonce whose value can lead to the disclosure of the *private key* from the signature

The scalar multiplication implemented within `trezor-crypto` as the `point_multiply` function has already been the target of a side-channel attack (see [10]). Following this attack the code has been patched, and the comments in the code now mention a side-channel protected implementation. We will however show that this countermeasure does not protect from our attack, since we show how to retrieve a scalar using a single `point_multiply` execution and an oscilloscope.

During all this study, we will only work on the `secp256k1` elliptic curve: meaning our scalar will be at most 256 bits long.

In a very similar order as section 3, section 4.1 will present the targeted function, and the *sensitive values* we picked to reconstruct a scalar. Section 4.2 will describe the *leakage characterization*, and section 4.3 and 4.4 will describe the two kinds of side-channel attacks we used to recover a scalar: a *timing* attack and a *profiled* side-channel attack.

4.1 Targeted function

The targeted function, `point_multiply` is part of `trezor-crypto` library. As it is mentioned in the comments, it implements an optimized 4-NAF (Non-Adjacent Form) algorithm for scalar multiplication described in [17].

```
// simplified pseudo-code of point_multiply():
```

```
// gives an idea of what the code is actually doing.

point_multiply(bignum256 k, curve_point p)
{
    a = k + 2 ^ 256 (mod curve->order) //a is odd

    pmult = [P, 3 P, 5P, ..., 15P]; //precomputation:

    Q = P;

    a = [a[0], ... a[63]] // a is split into 64 nibbles

    for(i = 62 ; i >= 0 ; i--) {

        nsign,sign, bits = f(a[i], a[i+1])
        """
        based on a[i] and a[i+1],
        three values are computed at each step:
        - sign: 1 bit
        - nsign: 1 bit
        - bits: 4 bits
        """

        Q = 16 P;
        conditional_negate( sign ^ nsign , Q.z, prime);
        point_jacobian_add(pmult[ bits >> 1 ], Q);
    }
    return Q
}
```

Listing 3. The pseudo code of the targeted function: Elliptic Curve scalar multiplication

Listing 3 is the pseudo-code we wrote describing what is done by the `point_multiply` function. For the whole source code, see the github repository (cf. [3]).

During a scalar multiplication $[k]P$:

— k is transformed into a: $a = k + 2^{256}$mod order,

— $[3]P, [5P], \ldots, [15]P$ are precomputed and stored into `pmult`,

— The main loop is executed, on each nibble of a

In this main loop we see that:

— `sign`, `nsign` and `bits` are derived from a,

— `conditional_negate` is called with `sign ^ nsign` (1 bit per loop step),

— one of the precomputed `pmult` points is manipulated depending on `bits >> 1` (3 bits per loop step).

Knowing the 64 values of `(sign, nsign, bits>>1)` allows to reconstruct a and therefore, k.

We define the sensitive values this way:

$$f_i(k) = \text{sign}_i \oplus \text{nsign}_i, \quad 0 \leq i < 64$$

$$g_i(k) = \text{bits}_i >> 1, \quad 0 \leq i < 64$$

Still following a *Divide & Conquer strategy*, these are the 2×64 values that we will retrieve through independent side-channel attacks, in order to reconstruct the scalar k.

4.2 Leakage characterization

The same setup as the PIN attack is used for power acquisition. We acquire side-channel power traces during several executions of the `point_multiply` function. Figure 10 shows a single power trace acquired from the beginning of the `point_multiply` function. The red rectangles frame the successive steps of the algorithm described in listing 3:

— The construction of a from k at the first red rectangle
— The 7 steps of the *pmult* computation at the second red rectangle
— The 14 first steps out of the 64 in the main loop at the last red rectangle

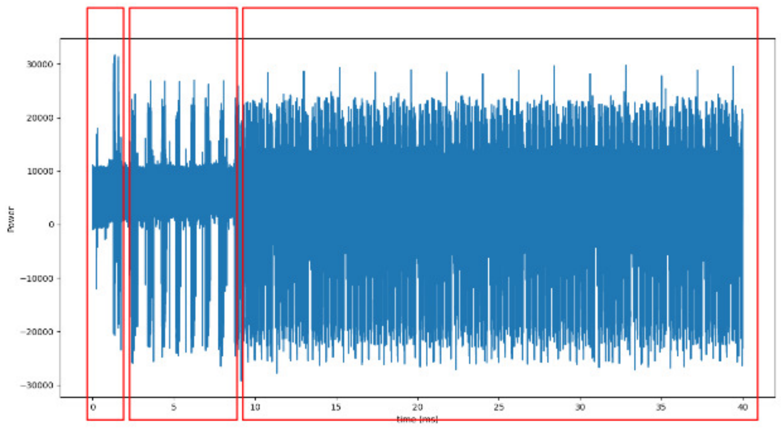


Fig. 10. The power trace from the beginning of a scalar multiplication. The red rectangles show the 14 first steps of the algorithm.

An additional steps has to be performed on the power traces in order to pass the attack: *leakage synchronization*. It consists in modifying each power traces in order to make them look alike. Figure 11 shows several raw power traces plotted together on the top part. The traces may look alike, but a *jitter* is present and has to be corrected. Several distinct patterns are used to modify each power trace to correct this *jitter*. The ouput of this *leakage synchronization* is shown on the bottom plot in figure 11.

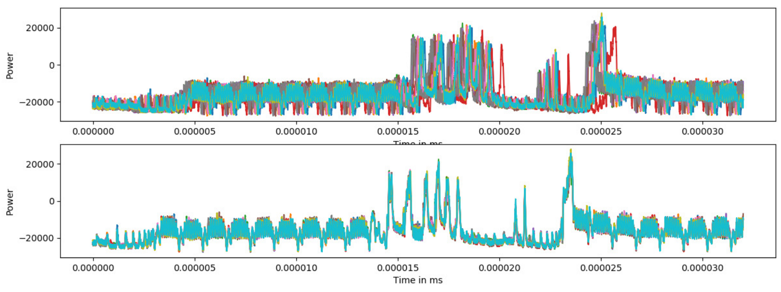


Fig. 11. 10 power traces zoomed in on a loop step: before (up) and after (down) *leakage synchronization*: the *jitter* has disappeared.

We build our *profiling set* by acquiring 150000 power traces captured during the `point_multiply` exectution on a *Trezor One* device, with known random scalars k. The *leakage synchronization* processing is then applied on thoses power traces.

Just as in section 3.2, we compute NICV on the *profiling set* with our *sensitive values* identified at section 4.1. Figure 12 displays the NICV curves for our sensitive values. We observe a strong dependency between our power leakage and our *sensitive values*.

Regarding the f_i, we get scores close to 1, which will be investigated and explained in the next section. Regarding the g_i, a *profiled* side-channel attack will be mounted in section 4.4 (in a similar fashion to the PIN attack in section 3.3).

4.3 Retrieving $\mathbf{sign}_i \oplus \mathbf{nsign}_i$ with a timing attack

From the middle plot of figure 12, the values of the NICV on the f_i *sensitive values* are suspiciously high. They are indeed induced by a timing leakage we found on the power traces. Figure 13 shows 10 power traces zoomed in on a portion of a single step of the loop. The traces can clearly be split into two classes. A closer look shows that the separation into those two classes is exactly explained by the value of $f_i(k) = \mathrm{sign}_i \oplus \mathrm{nsign}_i$:

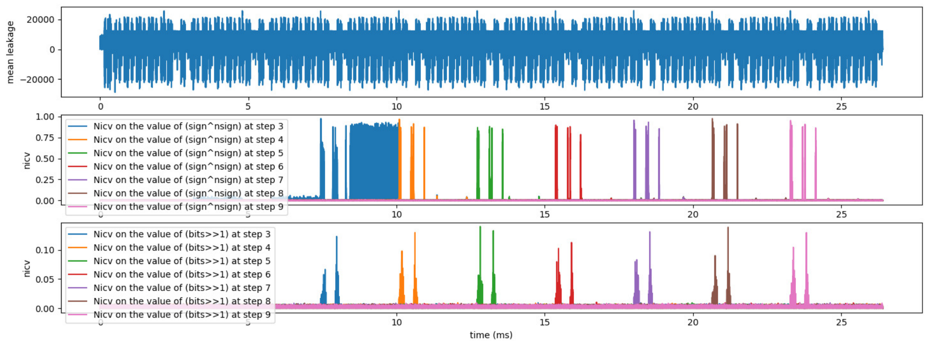


Fig. 12. The top figure represents the mean power trace zoomed in on the 10 first steps on the main loop. The middle figure shows the NICV curves corresponding to the *sensitive values* $f_3, \ldots, f_9$ (dealing with $\text{sign}_i \oplus \text{nsign}_i$). The bottom figure shows the NICV curves corresponding to the *sensitive values* $g_3, \ldots, g_9$ (dealing with $\text{bits}_i >> 1$).

the first class occurs when $f_i(k) = 0$, and the second class occurs when $f_i(k) = 1$.

Hence we have a visual distinguisher which allows us, with a 100% success rate, to extract the value of the $\text{sign}_i \oplus \text{nsign}_i$ in a single trace.

This timing leakage occurs during the `conditional_negate` function call, whose execution flow is conditioned by the value of $\text{sign}_i \oplus \text{nsign}_i$ (see listing 3).

4.4 Retrieving ($\text{bits}_i >> 1$) with a profiled attack

Now we have to retrieve the 64 successive values of ($\text{bits}_i >> 1$). In order to do so, we will mount a *profiled* side-channel attack, very closely resembling the one described at section 3.3.

The learning phase consists in using the *profiling set* with the *sensitive values* g_i. The end result is a set of $\mathsf{Classifier}_i$ such that:

$$\mathsf{Classifier}_i(l) = \mathsf{Proba}[(\text{bits}_i >> 1) = x] \text{ , for } 0 \leq x < 8$$

Once again, on a new device, running the `point_multiply` function with unknown scalar k, we acquired a few power traces and input them to the $\mathsf{Classifier}_i$. Figure 14 shows the progression of the 8 possible values that ($\text{bits}_i >> 1$) could take. This attack works with only a single trace.

This attack has been repeated on different steps of the loop, on 300 set of traces. In more than 99% of the cases, the attack works with a single power trace. In the remaining cases, 2 power traces are needed.

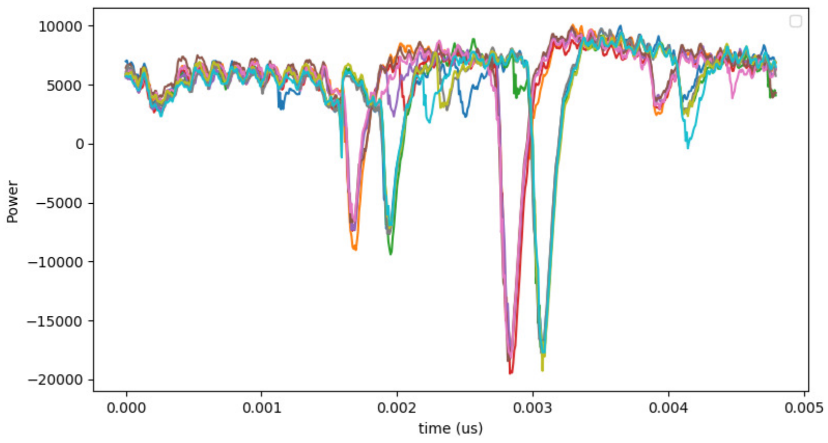


Fig. 13. 10 power traces zoomed on a portion of a single loop step exhibiting two different behaviors.

4.5 Summing up

We just demonstrated how to reconstruct a scalar k used on a *Trezor One* device in a single execution of the `point_multiply` function from its corresponding power trace, despite the constant-time execution. Recovering the whole k means that we have been able to mount 128 independent side-channel attacks:

— 64 *timing* attacks to recover the 64 values of the $\text{sign}_i \oplus \text{nsign}_i$, by using a single trace

— 64 *profiled* attacks to recover the 64 values of the $(\text{bits}_i >> 1)$, by using a single trace

From all those recovered intermediate values, the scalar k can be reconstructed.

5 Replaying attacks without the hardware

We have developed a tool on top of Unicorn [5], a generic CPU emulator, in order to easily trace execution of code snippets and, among other uses, simulate side-channel traces in a purely software way. This tool is called `Rainbow` [1] and is open sourced on GitHub. In its examples, one can find a function that emulates the Trezor PIN comparison directly from the ELF binary file that can be compiled from `trezor-mcu` sources [4].

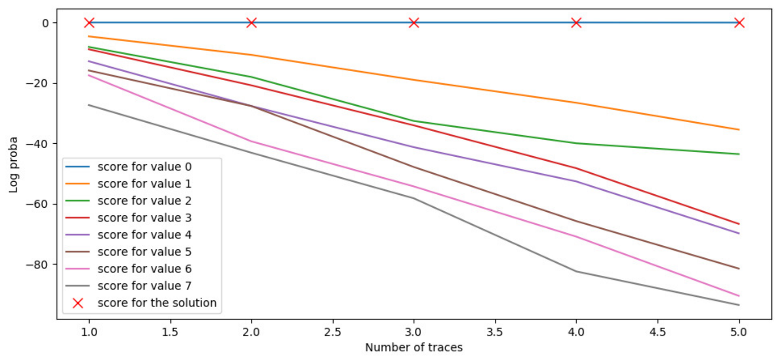


Fig. 14. Matching phase of a single classifier on a set of 5 traces: here we use $\mathsf{Classifier}_4$. The 8 curves represent the progression of the score for each possible value of $(\text{bits}_i >> 1)$. The score for the solution is plotted in red with $\times$.

Along with a dedicated viewer, we can display the sequence of instructions on the left-hand side and the corresponding traces on the right-hand side.

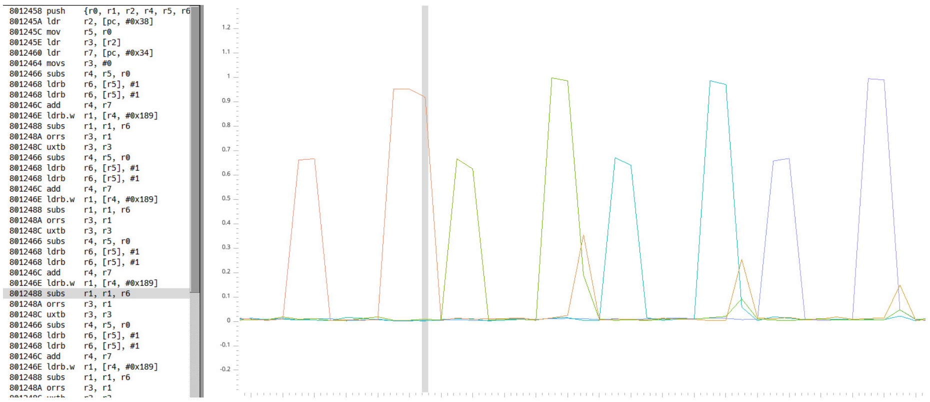
For the two attacks in this article, we are most interested in regenerating the NICVs, which help identify the presence of a vulnerability. We can directly compute those from the simulated traces, without the need for an oscilloscope or even the target device.

For the PIN attack from section 3, this view of the NICV alongside the instructions does not reveal any surprising leakage compared to our initial source code analysis. Nonetheless it would be helpful in identifying a conceptual flaw in terms of side-channel leakage in such an implementation, and for automated leakage assessment purposes.

A longer and heavier operation such as the full scalar multiplication can also be emulated without too much trouble, as shown by a portion of the execution trace in 16.

6 Conclusion

In this paper, we presented two side-channel attacks targeting the open source hardware wallet *Trezor One*. The first attack is targeting the `trezor-mcu` firmware code allowing to retrieve the user PIN, and the second attack targets the scalar multiplication implemented by `trezor-crypto` library, allowing to retrieve information on the scalar used.



```
8012458 push   {r0, r1, r2, r4, r5, r6
801245A ldr    r2, [pc, #0x38]
801245C mov    r5, r0
801245E ldr    r3, [r2]
8012460 ldr    r7, [pc, #0x34]
8012464 movs   r3, #0
8012466 subs   r4, r5, r0
8012468 ldrb   r6, [r5], #1
8012468 ldrb   r6, [r5], #1
801246C add    r4, r7
801246E ldrb.w r1, [r4, #0x189]
8012488 subs   r1, r1, r6
801248A orrs   r3, r1
801248C uxtb   r3, r3
8012466 subs   r4, r5, r0
8012468 ldrb   r6, [r5], #1
8012468 ldrb   r6, [r5], #1
801246C add    r4, r7
801246E ldrb.w r1, [r4, #0x189]
8012488 subs   r1, r1, r6
801248A orrs   r3, r1
801248C uxtb   r3, r3
8012466 subs   r4, r5, r0
8012468 ldrb   r6, [r5], #1
8012468 ldrb   r6, [r5], #1
801246C add    r4, r7
801246E ldrb.w r1, [r4, #0x189]
8012488 subs   r1, r1, r6
801248A orrs   r3, r1
801248C uxtb   r3, r3
8012466 subs   r4, r5, r0
8012468 ldrb   r6, [r5], #1
8012468 ldrb   r6, [r5], #1
801246C add    r4, r7
801246E ldrb.w r1, [r4, #0x189]
8012488 subs   r1, r1, r6
801248A orrs   r3, r1
```

Fig. 15. NICV from simulated curves for the PIN attack, to be compared with figure 5.

The open access to the code and the devices facilitates the use of the more powerful class of side-channel attacks that are *profiled attacks*. Using machine learning techniques on modified version of the targeted firmware on an *open device* allowed us to extract the user PIN from a *Trezor One* device in 5 attempts (below the limit of 15 attempts normally enforced by the device) and extract the nonce of an ECDSA signature in a single execution, which leads to complete recovery of the user's private key. Several other wallets were affected, being based on the same code as the *Trezor One*. The PIN vulnerability was patched by Trezor following our responsible disclosure. The nonce extraction was not, however it has no immediate impact on the user (since knowledge of the PIN is required to mount that attack).

References

1. Rainbow - unicorn-based tracer. `https://github.com/Ledger-Donjon/rainbow`.
2. Stm32 portfolio. `https://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html`.
3. trezor-crypto: Heavily optimized cryptography algorithms for embedded devices. `https://github.com/trezor/trezor-crypto`.
4. trezor-mcu: Trezor one bootloader and firmware. `https://github.com/trezor/trezor-mcu`.
5. Unicorn engine - multiarchitecture emulator. `http://www.unicorn-engine.org/`.
6. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Nicv: Normalized inter-class variance for detection of side-channel leakage. Cryptology ePrint Archive, Report 2013/717, 2013. `https://eprint.iacr.org/2013/717`.

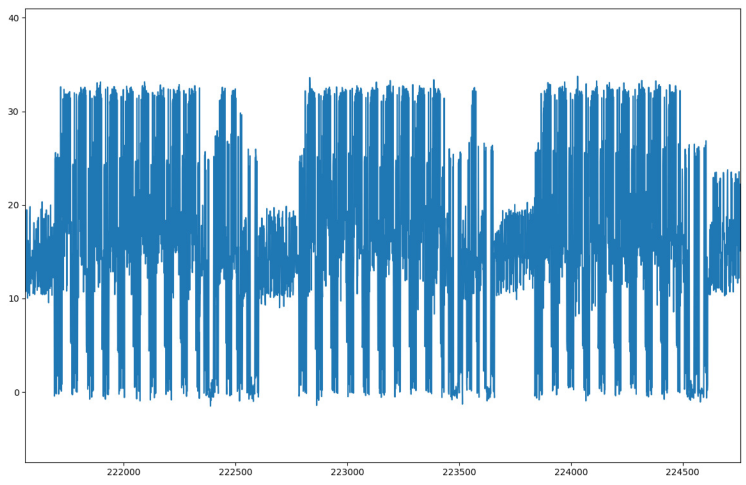


Fig. 16. Emulated scalar multiplication from the Trezor firmware (with artificial noise).

7. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
8. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 251–261, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
9. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.
10. Jochen Hoenicke. Extracting the private key from a trezor. `https://jochen-hoenicke.de/crypto/trezor-power-analysis/`.
11. Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293, Oct 2011.
12. Peter Karsmakers, Benedikt Gierlichs, Kristiaan Pelckmans, Katrien De Cock, Johan Suykens, Bart Preneel, and Bart De Moor. Side channel attacks on cryptographic devices as a classification problem. 03 2019.
13. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

14. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.
15. Karl Kreder. Hardware wallet vulnerabilities. `https://blog.gridplus.io/hardware-wallet-vulnerabilities-f20688361b88`, 2017.
16. Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. *IACR Cryptology ePrint Archive*, 2016:921, 2016.
17. Katsuyuki Okeya and Tsuyoshi Takagi. The width-w naf method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In *CT-RSA*, 2003.
18. M. San Pedro, V. Servant, and C. Guillemet. Lascar: Ledger's advanced side channel analysis repository. `https://github.com/Ledger-Donjon/lascar`, 2018.