Blind-Folded: Simple Power Analysis Attacks using Data with a Single Trace and no Training

Xunyue Hu, Quentin L. Meunier and Emmanuelle Encrenaz

Sorbonne Université, LIP6, CNRS, 4 Place Jussieu 75005 Paris
{xunyue.hu,quentin.meunier,emmanuelle.encrenaz}@lip6.fr

Abstract. Side-Channel Attacks target the recovery of key material in cryptographic implementations by measuring physical quantities such as power consumption during the execution of a program. Simple Power Attacks consist in deducing secret information from a trace using a single or a few samples, as opposed to differential attacks which require many traces. Software cryptographic implementations usually contain a data-independent execution path, but often do not consider variations in power consumption associated to data. In this work, we show that a technique commonly used to select a value from different possible values in a control-independant way leads to significant power differences depending on the value selected. This difference is actually so important that a single sample can be considered for attacking one condition, and no training on other traces is required. We exploit this finding to propose a single-trace attack without any knowledge gained on previous executions, using trace folding. We target the two modular exponentiation implementations in Libgcrypt, getting respectively 100% and 99.998% correct bits in average on 30 executions using 2,048-bit exponents. We also use this technique to attack the scalar multiplication in ECDSA, successfully recovering all secret nonces on 1,000 executions. Finally, the insights we gained from this work allow us to show that a proposed countermeasure from the literature for performing the safe loading of precomputed operands in the context of windowed implementations can be attacked as well.

Keywords: Simple Power Attack · Modular Exponentiation · ECDSA · Constant-Time Implementation · Side-Channel Attacks

1 Introduction

Side-Channel Attacks (SCA) target the recovery of key material in cryptographic implementations by measuring physical quantities such as power consumption during the execution of the program.

There are two major deterministic causes which influence the power consumption: the instruction executed by a program and the data manipulated by these instructions. Instructions have a major role in the power dissipated. In fact, looking at a power trace, it is possible to determine which path of instructions the program has followed, leading to the so called Simple Power Attacks (SPA). For this reason, sensitive programs such as cryptographic primitives are now always designed to have an instruction execution trace which is always the same, and in particular independent from the inputs.

Data, on the other hand, have a slighter effect on power consumption, and exploiting this power difference often requires to capture a lot of traces and perform differential attacks known as Differential Power Analysis attacks (DPA) such as the Correlation Power Analysis attack (CPA) [MOP08]. In this article, we show that it is possible to use data power consumption to perform SPA using single sample values, taking advantage of the

variation in the Hamming Weight of the data. These attacks allow to recover some secret values based on a single trace. This article makes three contributions:

- We identified that a common way of implementing constant time cryptographic algorithms leads to a significant weakness. More precisely, the masking technique used to assign one of two values to a variable in a control-independant way leads to significant power differences depending on the chosen value. We additionally identified an inherent weakness specific to elliptic curve cryptography, which is due to the point at infinity.

- We show on two widespread cryptographic implementations how to exploit this data-related power consumption to perform a SPA: on the modular exponentiation in `Libgcrypt` and on a secure elliptic curve implementation for performing a signature. The attack requires a single trace to recover directly the full secret, without any training or profiling phase. The only prerequisite is the identification of the loop performing the computation of the masking technique.

- Using the weaknesses identified in the proposed attacks, we show how to break a countermeasure proposed by Saito *et al.* [SIUH22] to safely load a precomputed operand in a windowed modular exponentiation.

We underline that the attack presented in this work is of practical relevance. Indeed, most if not all constant time implementations use a making technique to select one value to keep among two, and this countermeasure, while primarily targeting cache attacks, is commonly used against side-channels as well, in particular on embedded systems [GB23].

The rest of the article is organized as follows: section 2 describes some background related to modular exponentiation and its usage in RSA, and to ECDSA. These two algorithms are used in the following as attack targets. Section 3 presents some works related to power attacks on cryptographic implementations, and highlights differences of our work with previous works, especially regarding the attack hypotheses. Section 4 presents our attack on the modular exponentiation, and section 5 our attack on ECDSA. In section 7, we show that the proposed operand loading process from [SIUH22] does not prevent the exponent recovery. Finally, section 8 concludes and gives some perspectives for future works.

2 Background

2.1 Modular Exponentiation

2.1.1 Usage in RSA

Modular exponentiation is a critical operation in a cryptographic context, as it is often applied to secret data. More precisely, the RSA asymmetric encryption scheme uses modular exponentiation with the secret key as exponent in two contexts: for deciphering a message, and for signing a message.

In the textbook RSA scheme, a private key is made of values (p, q, d), a public key is (e, N): p and q are two large prime numbers and $N = pq$, $d = e^{-1} \mod (p-1)(q-1)$ where e is a small value and e has to be relatively prime with $(p-1)(q-1)$. Signature and verification of a message m are expressed as:

$$s = m^d \mod N$$
$$m = s^e \mod N$$

in which s is the signature. RSA is often implemented using the so-called CRT-RSA scheme for efficiency reasons. In this scheme, the owner of the private key computes:

$$\begin{aligned} d_p &= d \mod (p-1) \\ d_q &= d \mod (q-1) \\ q_p &= q^{-1} \mod p \end{aligned}$$

The signing operates as follows:

$$\begin{aligned} s_p &= m^{d_p} \mod p \\ s_q &= m^{d_q} \mod q \\ s &= (s_p - s_q).q_p.q + s_q \mod N \end{aligned}$$

We can notice that the secret exponent d is no longer used as exponent; however, recovering d_p and d_q from the corresponding exponentiations still allows to recover d: we can then compute $s'_p = m^{d_p} \mod N$, and since we have $s = m^d \mod N$, we can recover:

$$\begin{aligned} p &= \gcd(s - s'_p, N) \\ q &= N/p \\ d &= e^{-1} \mod (p-1)(q-1) \end{aligned}$$

Using the notation $a \equiv b \mod n$ for the congurence of a and b modulo n:

$$\begin{aligned} \text{Since} \quad & s = m^d \mod N \\ \text{we have} \quad & s \equiv m^d \mod p \\ \exists \text{ k such that} \quad & s \equiv m^{d_p + k(p-1)} \mod p \\ & \equiv m^{d_p}.(m^{(p-1)})^k \mod p \\ \text{Euler's Theorem} \quad & \equiv m^{d_p}.(1)^k \mod p \\ & \equiv m^{d_p} \mod p \\ \exists \text{ a such that} \quad & s = m^{d_p} + a.p \end{aligned} \tag{1}$$

$$\begin{aligned} \text{Similarly} \quad & s'_p = m^{d_p} \mod N \\ & s'_p \equiv m^{d_p} \mod p \\ \exists \text{ b such that} \quad & s'_p = m^{d_p} + b.p \end{aligned} \tag{2}$$

$$\begin{aligned} (1) - (2) \implies s - s'_p &= m^{d_p} + a.p - m^{d_p} - b.p \\ s - s'_p &= (a-b).p \end{aligned}$$

So $(s - s'_p)$ and N have a common factor p which is their greatest common divisor since $N = pq$ and p and q are prime numbers.

In `Libgcrypt`'s RSA implementation, a random protection has been added, known as blinded exponent: a random number r is generated for each calculation and the expression $s_p = m^{d_p + r(p-1)} \mod p$ is computed instead of $s_p = m^{d_p} \mod p$. This does not change the value of s_p, but the exponent of the operation is $d_p + r(p-1)$, which means that the "secret" value we recover is actually $d_p + r(p-1)$, and the same for d_q. However, as detailed in Vergnaud's article [Ver20], when we have the full value of $d_p + r(p-1)$, we can use RSA's own properties to find p, and the same for q. Even if we only find a part of $d_p + r(p-1)$, we can still find the complete key using lattice, as described in [MV19] (section 6).

2.1.2 Implementations

Traditional modular exponentiation implementations follow the algorithm shown in Algorithm 1. In this algorithm, the exponent is traversed bit by bit; when the bit is 0, the current result is squared, and when it is 1, the current result is squared then multiplied. Obviously, recovering the sequence of operations, e.g. via a cache attack [YF14, LGS+16] or power traces, allows to recover the exponent.

Algorithm 1 Traditional modular exponentiation. Bold variables indicate large integers.

Input: Base $\boldsymbol{c}$, exponent $\boldsymbol{d} = (\mathtt{d}_{\mathtt{k}-1}...\mathtt{d}_0)_2$, modulus $\boldsymbol{N}$
Output: $\boldsymbol{r} = \boldsymbol{c}^d \bmod \boldsymbol{N}$
function ModularExponentiation($\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{N}$)
  $\boldsymbol{r} \leftarrow 1$
  for $\mathtt{i}$ from $\mathtt{k}-1$ to 0 **do**
    $\boldsymbol{r} \leftarrow \boldsymbol{r} \times \boldsymbol{r} \bmod \boldsymbol{N}$ # Squaring
    if $\mathtt{d}_\mathtt{i} = 1$ **then**
      $\boldsymbol{r} \leftarrow \boldsymbol{r} \times \boldsymbol{c} \bmod \boldsymbol{N}$ # Multiplication
  return $\boldsymbol{r}$

To circumvent this problem, recent implementations always perform the multiplication and store the result in a different variable, choosing at the end of the iteration which result to keep.

Algorithm 2 Windowed modular exponentiation. Bold variables indicate large integers.

Input: Base $\boldsymbol{c}$, exponent $\boldsymbol{d} = (\mathtt{d}_{\mathtt{k}-1}...\mathtt{d}_0)_2$, modulus $\boldsymbol{N}$, window size $\mathtt{w}$
Output: $\boldsymbol{r} = \boldsymbol{c}^d \bmod \boldsymbol{N}$
function ModularExponentiation($\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{N}$, $\mathtt{w}$)
  $\boldsymbol{c}_0 \leftarrow 1$
  for $\mathtt{i}$ from 1 to $2^{\mathtt{w}-1}$ **do**
    $\boldsymbol{c}_\mathtt{i} \leftarrow \boldsymbol{c}_{\mathtt{i}-1} \times \boldsymbol{c} \bmod \boldsymbol{N}$ # Precomputing the $2^{\mathtt{w}-1}$ first powers in a table
  $\boldsymbol{r} \leftarrow 1$
  $\mathtt{z} \leftarrow \mathtt{k} - 1$
  while $\mathtt{z} \geq 0$ **do**
    $\mathtt{y} \leftarrow \max(\mathtt{z} - \mathtt{w} + 1, 0)$
    $\mathtt{u} \leftarrow (\mathtt{d}_\mathtt{z}...\mathtt{d}_\mathtt{y})_2$
    for $\mathtt{i}$ from 1 to $\mathtt{z} - \mathtt{y} + 1$ **do**
      $\boldsymbol{r} \leftarrow \boldsymbol{r} \times \boldsymbol{r} \bmod \boldsymbol{N}$ # Squaring
    $\boldsymbol{r} \leftarrow \boldsymbol{r} \times \boldsymbol{c}_\mathtt{u} \bmod \boldsymbol{N}$ # Multiplication
    $\mathtt{z} \leftarrow \mathtt{y} - 1$
  return $\boldsymbol{r}$

Another approach, which is computationally more efficient consists in precomputing $2^\mathtt{w}$ values of the base at the beginning, and then in traversing the exponent bits by groups of $\mathtt{w}$, as shown in Algorithm 2. We can see that in this version of the algorithm, the computations are independant from the exponent value. However, since the $\boldsymbol{c}_\mathtt{u}$ loaded depends on it, it is possible to use cache timing information to infer what the possible values for $\mathtt{u}$ are. To avoid this issue, some side-channel resistant implementations load all the elements $\boldsymbol{c}_\mathtt{u}$, keeping as result the result of the load corresponding to the correct window value.

However, as we will show in section 4, both the multiply always version and the windowed version making all loads are subject to SPA when keeping or discarding the result of a useful or useless operation.

2.2 ECDSA

2.2.1 Principle

Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] is a public-key digital signature algorithm that is a variant of DSA. It uses elliptic curve cryptography which relies on an elliptic curve defined over some finite field of integers $\mathbb{F}_p$, $Curve(a, b, p, G, n) = x^3 + ax + b \mod p$ where p is a large prime, G is a base point on the curve that can generate a subgroup of a large prime-order n.

The public parameters of ECDSA scheme include a description of $Curve(a, b, p, G, n)$. The private key d is a random positive integer less than n, the corresponding public key is set as the point $H = d * G$. The $*$ here is the multiplication between a scalar and a point of the curve, and $d * G$ is defined by doing d times the elliptic curve's addition of the point G.

A signature of a message m is $Sign(z, d) = (r, s)$, where $z = hash(m)$ for some hash function; $r = \text{xcoord}(P)$, where $P = k * G$ with k a random nonce greater than 0 and less than n generated for each signature; $s = k^{-1}(z + dr) \mod n$ where k^{-1} is the inverse of $k \mod n$. One can then verify the signature from the public key H by computing $u = s^{-1}z \mod n$, $v = s^{-1}r \mod n$, and checking whether $\text{xcoord}(u * G + v * H) = r$, meaning $s^{-1}z * G + s^{-1}r * H \mod n = k * G$.

We can see that once we find any nonce k, we can easily compute the private key $d = r^{-1}(sk - z) \mod n$.

2.2.2 Implementations

Traditional implementations of ECC's multiplication rely on traversing the bits of the scalar, either one by one or using a window of bits for indexing a table containing precomputed multiples of the base point.

Algorithm 3 Fixed-Window ECC Multiplication. Bold variables indicate large integers, and variables in capital indicate EC points.

Input: Base Point $\boldsymbol{G}$, scalar $\boldsymbol{k} = (\mathtt{k_{n-1}}...\mathtt{k_0})_2$, window size $\mathtt{w}$
Output: Point $\boldsymbol{P} = \boldsymbol{k} * \boldsymbol{G}$

function ECCMULTIPLICATION($\boldsymbol{G}$, $\boldsymbol{k}$, $\mathtt{w}$)
  $\boldsymbol{C}_0 \leftarrow O$ # O represents the point at infinity, neutral element for the addition
  for $\mathtt{i}$ from 1 to $2^{\mathtt{w}-1}$ **do**
    $\boldsymbol{C}_\mathtt{i} \leftarrow \boldsymbol{C}_{\mathtt{i}-1} + \boldsymbol{G}$ # Precomputing the first $2^{\mathtt{w}-1}$ multiples in a table
  $\boldsymbol{P} \leftarrow O$
  $\mathtt{z} \leftarrow \mathtt{n} - 1$
  while $\mathtt{z} \geq 0$ **do**
    $\mathtt{y} \leftarrow \max(\mathtt{z} - \mathtt{w} + 1, 0)$
    $\mathtt{u} \leftarrow (\mathtt{k_z}...\mathtt{k_y})_2$
    for $\mathtt{i}$ from 1 to $\mathtt{z} - \mathtt{y} + 1$ **do**
      $\boldsymbol{P} \leftarrow \boldsymbol{P} + \boldsymbol{P}$ # Computing $2^{\mathtt{z}-\mathtt{y}+1} * \boldsymbol{P}$
    $\boldsymbol{P} \leftarrow \boldsymbol{P} + \boldsymbol{C}_\mathtt{u}$ # Computing $(2^{\mathtt{z}-\mathtt{y}+1} + u) * \boldsymbol{P}$
    $\mathtt{z} \leftarrow \mathtt{y} - 1$
  return $\boldsymbol{P}$

Algorithm 3 shows the fixed-window ECC multiplication. As for the modular exponentiation, the lookup can be made constant time and indistiguishable from the loaded value by loading each element in the precomputed table and using a mask technique to only keep the correct value.

Yet again, as we will show in section 5, the recombination of all the loaded value can be efficiently exploited to recover the secret nonce in a single trace without prior training.

3 Related Works and Discussion

Several recent works have targeted the recovery of the secret exponent in RSA and the recovery of the secret nonce in ECDSA, often limited to the recovery of a part of the secret bits.

In 2017, Bernstein *et al.* [BBG+17] proposed an attack to recover the full RSA exponent of a non constant-time sliding window implementation of modular exponentiation, based on the fact non constant-time implementations seek to reduce the number of multiplication by adjusting the window depending on the location of the zeros. Following this work, constant-time sliding window exponentiation was proposed.

In 2019, Weissbart *et al.* [WPB19] presented an attack of ECDSA using different techniques of various accuracy for recovering the secret nonce. The full secret recovery is achieved with a Convolutional Neural Network (CNN), which must be trained with around 500 traces for the complete recovery. This work has been extended in 2020 [WCPB20] in which two different implementations of the scalar multiplication are studied: a baseline using power consumption, and a protected implementation using electromagnetic emission (EM). While the authors manage to break the baseline implementation with a single trace, they make use of a CNN trained with 6400 labeled traces, while not being able to explain the cause of the leakage.

In 2020, Lee *et al.* [LH20a] consider the detection of dummy operations as a multi-label classification problem and propose a deep learning method based on CNN to solve it, using power measures. They target an AES software implementation with dummy loads countermeasure, and show that their method performs well compared to their previously proposed method [LH20b]. However, their multi-label CNN is designed for attacking this specific AES implementation, and the applicability to other cryptographic implementations is unclear. Also using deep learning, Lei *et al.* presented a profiling attack based on VGGNet, a small and deep neural network, performed on a smart-card CRT-RSA implementation, using security countermeasures including masking and time jittering [LLQ+20]. An ad-hoc method is applied to extract sample points from traces to perform an effective deep learning profiling attack.

Finally, three works are closely related to the contributions we make in this article. First, Nascimento *et al.* [NCOS16] attack an embedded ECC implementation with power measures. However, there are several important differences between their work and ours: they target the bitwise and of the conditional copy, resulting in 4288 samples (64 iterations * 67 samples/cycle) for determining a single data value while we use only one sample, targeting the mask computation; they use a template attack requiring to build a model from reference executions while we have no training phase; their success rates are not as high the ones we obtain, showing the relevance of targeting the mask computation.

Second, Alam *et al.* describe an attack on ECDSA using a fixed window [AYW+21]. Their attack targets the conditional swap, a technique to conditionally swap two values based on the condition result, using many EM samples resulting from the 80 exclusive-or that the swap comprises. Then, it uses a classifier trained with known secret scalar values to conclude, while giving some incorrect bits. Our work however shows that a single power measurement is enough for determining a data value without error.

Third, Saito *et al.* propose a single-trace [SIUH22] attack on the GnuMP implementation of the exponentiation in RSA based on EM measures. They use a deep-learning technique to detect dummy loads, but the approach requires a significant training phase (more than 61 millions traces mentioned in the article), while not giving much insight on the reason of the leakage. A countermeasure is presented in the article, which we think does not prevent the exponent recovery. We will explain how to attack this countermeasure in section 7.

In contrary to most of these works which rely on profiling and learning techniques, the attack we present in this article does not need any specific training phase. The base idea is

very simple, and consists in comparing the power consumption of the two different possible values of the mask, which is almost always used in constant-time implementations. We show that this can be used to determine the mask value without the need for a pre-training step. As it turns out, the mask computation, which only depends on data, is sufficient to distinguish clearly the condition using a single power measure, even with a measure tool limited to a single sample per cycle.

4 Attacking RSA Modular Exponentiation with a SPA

This section presents our Simple Power Analysis attack on modular exponentiation. The attack targets the recovery of the exponent bits, either bit by bit for the traditional version, or slice by slice for the windowed version. Note that we are not interested here in the recovery of the exponent should any error occur, and such algorithms have already been proposed, e.g. in [SIUH22]. Besides, the attack has a very high precision and typically produces no error.

We target implementations of the `Libgcrypt` library (v.1.10.3 released on the 14$^{\text{th}}$ November 2023), compiled for Arm v7 and executed on an Arm Cortex-M4 integrated on a STM32F3 board. The board used is the ChipWhispererPro board from NewAE, measuring 4 samples per cycle for short traces, and one sample per cycle using streaming mode for long traces [OC14][1].

4.1 Characterisation: Conditional Assignment using Masking

A common way to perform a conditional assignment is to use the so-called "masking" technique, consisting in having two words: one containing only '0' bits, and one containing only '1' bits, depending on the condition. A way to achieve this is presented in Algorithm 4. It uses the fact that in this implementation, a large integer is represented as an array of limbs, each limb being encoded on a machine word – 32 bits in our case.

Algorithm 4 Control-independant conditional assignment using masking, used for example in `Libgcrypt`. Bold variables indicate large integers.

1: **function** SetCond($\boldsymbol{v}$, $\boldsymbol{u}$, `c`) ▷ if `c` $= 0$ $\boldsymbol{v}$ remains unchanged, else $\boldsymbol{v}$ gets $\boldsymbol{u}$
2: `msk0` $\leftarrow$ `wzero` $-$ `c` ▷ `wzero` is a word containing only '0' bits
3: `msk1` $\leftarrow$ `c` $-$ `wone` ▷ `wone` is a word containing the value 1 (0b0...01)
4: **for** `i` from 0 to `nlimbs` $- 1$ **do** ▷ `nlimbs` is the number of limbs
5: $\boldsymbol{v}_{\texttt{i}} \leftarrow$ (`msk0` & $\boldsymbol{u}_{\texttt{i}}$) | (`msk1` & $\boldsymbol{v}_{\texttt{i}}$) ▷ $\boldsymbol{v}_{\texttt{i}}$ and $\boldsymbol{u}_{\texttt{i}}$ are the `i`$^{\text{th}}$ limbs of $\boldsymbol{v}$ and $\boldsymbol{u}$

We first tried to determine whether the computation of `msk0` and `msk1` leads to significant power consumption differences depending on the value of `c`. For this characterisation, we used the implementation of the SetCond function of the library. Extracting only the masks computation lines 2 and 3 (and not the recombination line 4), we captured 1000 traces with a random value for the condition `c`.

The results of the captures are shown in Figure 1. On this figure, recorded traces with a condition value of 0 are displayed in blue, while recorded traces with a condition value of 1 are displayed in red. We can clearly see that there are some sample points for which the power consumption distributions for the two values of `c` are disjoint. Thus, measuring the power consumption at one of these this sample points only allows to determine the value of `c`, using a simple threshold to discriminate.

[1] All the captures and attack codes used in this work are available online at the following address: `https://largo.lip6.fr/blind-folded`

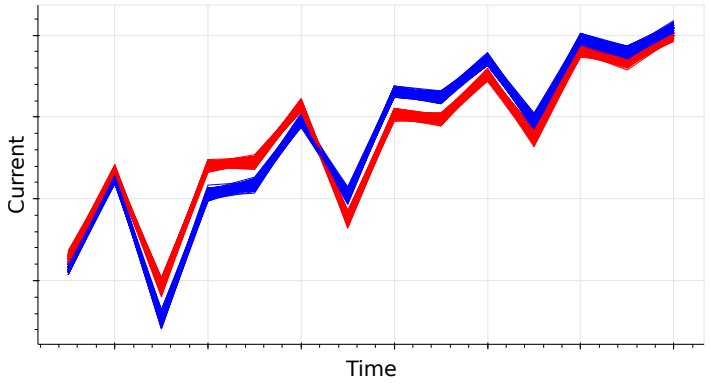


Figure 1: Power consumption associated to the masks' computation of the SetCond function in `Libgcrypt` with 4 samples per cycle. Blue curves correspond to a condition value of 0, while red curves correspond to a condition value of 1.

4.2 Attacking the Full Exponentiation

We used the two versions of the modular exponentation present in the `Libgcrypt` library: the traditional version based on the traversal of the exponent bits, and the second one using a sliding window. In the traditional version, for each bit of the exponent, the condition passed to the SetCond function is the bit value, as shown in Algorithm 5.

Algorithm 5 Simplified iteration of the main loop of the traditional exponentiation function in `Libgcrypt`. Bold variables indicate large integers.

▷ $\boldsymbol{rp}$ contains the current result
1: $\boldsymbol{xp} \leftarrow \boldsymbol{rp} \times \boldsymbol{rp}$ ▷ Squaring
2: $\boldsymbol{xp}, \boldsymbol{rp} \leftarrow \boldsymbol{rp}, \boldsymbol{xp}$ ▷ Swapping $\boldsymbol{rp}$ and $\boldsymbol{xp}$
3: $\boldsymbol{xp} \leftarrow \boldsymbol{rp} \times \boldsymbol{b}$ ▷ Multiplication
4: $\boldsymbol{rp} \leftarrow$ SetCond($\boldsymbol{rp}$, $\boldsymbol{xp}$, $\mathtt{e}_\mathtt{i}$) ▷ Keeping correct result depending on exponent bit

Algorithm 6 Simplified iteration of the main internal loop of the sliding window exponentiation function in `Libgcrypt`, loading all precomputed values. Bold variables indicate large integers.

1: $\boldsymbol{rp} \leftarrow$ `result`; ▷ $\boldsymbol{rp}$ contains the current result
2: **for** `i` from `j` to 0 **do** ▷ `j` is the number of squares
3: **for** `k` from 0 to $(2^{\mathtt{w}-1} - 1)$ **do** ▷ `w` $= 5$ is the window size
4: $\boldsymbol{u} \leftarrow$ `precomp`[`k`] ▷ `precomp`[`k`] contains $\boldsymbol{base}^{2\times\mathtt{k}+1}$ [pointer copy]
5: SetCond($\boldsymbol{w}$, $\boldsymbol{u}$, `k` $=$ `e0`) ▷ $\boldsymbol{w} \leftarrow$ `precomp`[`e0`] if `k` $=$ `e0`, `e0` being the slice value derived from the window value
6:
7: SetCond($\boldsymbol{w}$, $\boldsymbol{rp}$, `i` $\neq 0$) ▷ $\boldsymbol{w} \leftarrow \boldsymbol{rp}$ if `i` $\neq 0$, $\boldsymbol{w} \leftarrow$ `precomp`[`e0`] if `i` $= 0$
8: $\boldsymbol{xp} \leftarrow$ MulMod($\boldsymbol{rp}$, $\boldsymbol{w}$, $\boldsymbol{mp}$) ▷ Square if `i` $\neq 0$, Multiply if `i` $= 0$, $\boldsymbol{mp}$ is the modulus
9:
10: $\boldsymbol{rp} \leftarrow \boldsymbol{xp}$

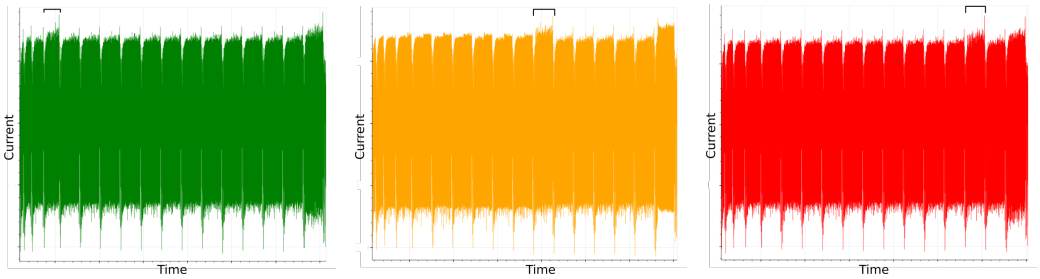
The implementation for the windowed version is shown in Algorithm 6. In order to minimize the number of multiplications, the considered slice value for the exponent is not directly the window value: the implementation first discards the trailing zeros and processes them in the next iteration. Then, the trailing 1 is removed for the index computation: the obtained slice value `e0` is thus comprised between 0 and 15, and is used for indexing

the `precomp` table which contains only odd powers between 1 and 31 (i.e. `precomp[e0]` = $\boldsymbol{base}^{2\times \texttt{e0}+1}$). In order to implement constant time, all elements in the table `precomp` are loaded using iteration variable `k`, and using as condition for the SetCond function that `k` equals `e0`.

As an example, if the window value is 10100_2, the two trailing zeros are first removed, giving 101_2. Then, the slice value `e0` is obtained by removing the trailing 1, i.e. `e0` = 10_2 = 2. The function thus copies `precomp`[2] = $\boldsymbol{base}^{2\times 2+1} = \boldsymbol{base}^{101_2}$ into $\boldsymbol{w}$, ignoring other loaded values.

Attacks on both the traditional and the windowed versions comprise the following parts: **1)** Locating the Regions of Interest (ROI) corresponding to where the SetCond function is called; **2)** Determining the Points of Interest (POI) from these ROI; **3)** Determining the operations from the power consumption at the POI; **4)** Reconstructing the key from the sequence of operations.

4.3 Locating the Regions of Interest



(a) SetCond function call with a slice value of 3 (b) SetCond function call with a slice value of 10 (c) SetCond function call with a slice value of 14

Figure 2: Power consumption for the windowed version where the function SetCond is called, for three different slice values

The ROI correspond to where the SetCond function is called. Examples of such ROI for the windowed version are shown in Figure 2. For this version, each ROI contains 17 sections, and each section contains a POI. The first 16 sections represent the calls to the SetCond function in the loop and the last one represents the call to the SetCond function outside of the loop (lines 3–7 in Algorithm 6). As for the traditional version, ROI contain only one section since they comprise a single call to the SetCond function.

Algorithm 7 Finding all ROI in the trace using a pattern

```
1: function FindROI(pattern, trace)
2:     roi ← [ ]
3:     for i from 0 to (nb_samples − len(pattern) − 1) do
4:         diff ← 0
5:         for j from 0 to (len(pattern) − 1) do
6:             diff ← diff + abs(pattern[j] − trace[i + j])
7:         if (diff < threshold) then
8:             roi.append(i)
9:     return roi
```

For both versions, the offsets between two consecutive ROI are not the same, as some operations depend on the data values. For the windowed version, it is due to the different number of limbs and the way `Libgcrypt` cuts the exponent into slices during the

computation; while for the traditional version, it is due to some possible required memory allocations depending on the already allocated variable sizes.

In order to locate the different ROI, we use a simple approach which consists in using one ROI as a power pattern for finding the others. Locating all ROI is then achieved by simply computing, for each subtrace of the same size, the sum of the absolute difference between the subtrace and the pattern, and keeping the ROI if this difference is below some threshold, as illustrated in Algorithm 7. The threshold value is typically obtained empirically: we can manually find a few ROI in the trace, compute this difference and define a threshold. This threshold needs to be larger than the maximum difference observed and smaller than the difference for the other non-ROI subtraces. Finding such a threshold is not complicated since the difference between the results of a ROI and a non-ROI subtrace is very significant. Besides, if we miss some ROI due to the threshold being too low, we can deduce the location of these ROI from the number of cycles between neighbouring ROI we found: if this number is significantly higher than usual, it means that some ROI have been missed, and we can then adapt the threshold consequently.

Finally, the attack time is almost exclusively the time spent locating the ROI and this time increases linearly with the pattern size. Therefore, using patterns of different sizes can result in different trade-offs in terms of attack time and accuracy, and using shorter patterns can greatly reduce the attack time at the price of missing some ROI.

4.4 Locating the Points of Interest

Focusing on Figure 2, we can observe that for the windowed version, the section for which the iteration number is equal to the slice value (condition `k` = `e0` in Algorithm 6) has a different power profile compared to the other, the top part being slightly higher. However, this difference, related to the reading of the precomputed elements, is harder to use than the difference related to the masks' computation that happens at the beginning of each section. This is why we decided to focus on the latter and to use these samples points as POI as they are the most obvious and stable ones.

Since we have only one trace, the approach we use is to "fold" the trace, by overlaying all the ROI found in it. By doing so, the sample points at which the masks' computation occur will exhibit two sets of traces clearly separated, depending on the condition value.

Figure 3 illustrates the overlayed ROI for both versions. In this figure, we notice that two possible areas can be chosen, corresponding to the computation of `msk0` and `msk1`, represented by squared dotted areas. In order to determine the threshold for each POI in the traditional version, we compute the greatest power difference for all ordered power measures at the corresponding sample, and take the middle of the two corresponding samples, as illustrated in Algorithm 8. As each ROI contains a single POI at the same position, locating all the POI is straightforward.

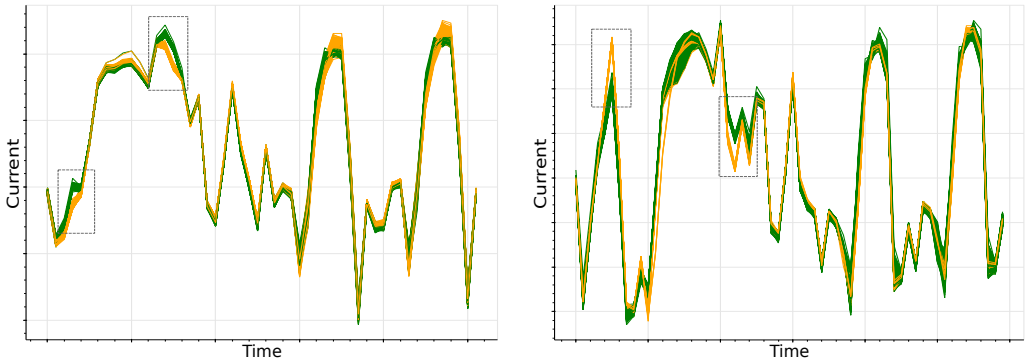
For the windowed version, overlaying the ROI yields 17 distinct POI, and a threshold can be computed for each. Actually, as long as the size of the ***base*** remains the same, no matter how the exponent changes, the location of the different POI inside the ROI will remain the same. In fact, the width of the sections inside a ROI first increase – as shown in Figure 2 – because when the result of the precomputed power is smaller than the modulus, `Libgcrypt` encodes it only on the required number of limbs. Therefore, the copy time changes according to this number. In our case, we sign a hash from the hash224 function, which is 224-bit long i.e. 7 limbs. Even when the condition is 0, if the value is not really copied, the number of iterations still depends on the number of limbs as shown lines 4–5 in Algorithm 4. The loaded precomputed results grows until reaching the modulus size, after what it remains the same size. Thus, the POI are always located at the same position in a ROI for a given ***base*** size, and their location can easily be found. Indeed, each section of a ROI has two parts: one containing the mask generation and one containing the reading of the limbs. In our case, we can observe that the part containing

the mask generation takes 89 cycles while the reading takes 13 cycles per limb. Using these values, we can easily calculate the location of all POI in each ROI as long as we know one POI location of one ROI from the trace.

Locating the POI automatically could alternatively be done by looking for the sample in each section of the ROI for which the greatest power difference for all ordered power measures is maximum. However, it still requires to identify the section, therefore bringing little advantage.

Algorithm 8 Finding the POI threshold

```
1: function FindPOIThresholds(pm_list)                    ▷ A list of power measures
2:     opm ← sort(pm_list)                                ▷ Ordered power measures
3:     power_diff ← [(opm[i + 1] - opm[i]) for i in range(len(opm) - 1)]
4:     idx ← argmax(power_diff)                           ▷ Index of the maximum value
5:     threshold ← opm[idx] + (power_diff[idx] / 2)
6:     return threshold
```



(a) Traditional version (b) Windowed version

Figure 3: Power consumption at the POI corresponding to the mask computation in the SetCond function for one trace and for both versions. For the windowed version, the displayed POI are those corresponding to the first of the 17 sections. Traces in orange correspond to a condition of 1, while traces in green correspond to a condition of 0. Dashed areas indicate where the masks are computed.

4.5 Determining the Operations from the POI

For the windowed version, each ROI contains 17 POI, corresponding to the 17 calls to the SetCond function, as shown in Algorithm 6. The last POI indicates if the operation is a Square ($\texttt{i} \neq 0$) or a Multiply ($\texttt{i} = 0$). Therefore, the recovery shown in Algorithm 9 starts by checking the power value at this POI in order to determine the operation (line 6). If the operation is a Multiply, we then check each of the first 16 POI, until we find the section for which the condition is true, meaning that the section number is equal to the exponent slice value (line 8). We then store this slice value and the corresponding number of preceding squares.

For the traditional version, the recovery of the sequence of operations is trivial, as each POI directly gives the operation (Square or Multiply) depending on the condition value, which is directly the exponent bit value.

Algorithm 9 Recovering the sequence of squares and multiplications from the POI

```
1: function RecoverOperations(poi, trace)          ▷ Returns the sequence of operations
2:     j ← 0
3:     e0_list ← [ ]                                          ▷ List of exponent slice values
4:     nb_squares ← [ ]   ▷ nb_squares[i] is the number of squares preceding the i^th multiply
5:     for i from 0 to len(poi) − 1 do
6:         if trace[poi[i][16]] > threshold_16 then                          ▷ Multiply detected
7:             for k from 0 to 15 do                                                ▷ precomp[k]
8:                 if trace[poi[i][k]] < threshold_k then                  ▷ k is equal to e0
9:                     e0_list.append(k)                          ▷ k contains the exponent slice value
10:                    nb_squares.append(j)                                        ▷ j squares
11:                    j ← 0
12:                    break
13:        else
14:            j ← j + 1
15:    return e0_list, nb_squares
```

4.6 Reconstructing the Secret Exponent

Algorithm 10 Exponent recovery from the sequence of operations

```
1: function RecoverExp(e0_list, nb_squares)
2:     exp ← [1]
3:     for i from 0 to (len(e0_list) - 1) do
4:         exp.extend([0] × nb_squares[i])      ▷ Concatenate nb_squares[i] '0' to the right
5:         b ← bin(2 × e0_list[i] + 1)                                  ▷ Binary representation
6:         for j from 0 to (len(b) - 1) do
7:             exp[len(exp) - 1 - j] ← b[len(b) - 1 - j]
8:     return exp
```

The reconstruction of the full secret exponent for the windowed version is shown in Algorithm 10, which traverses the list of exponent slice values and first concatenates the corresponding number of '0' bits (line 4), which is equal to the number of squares, and then adds the exponent value $2 \times \texttt{e0\_list}[\texttt{i}] + 1$ (lines 5–7). As an example for the first iteration, if $\texttt{e0\_list}[0] = 3$ and $\texttt{nb\_squares}[0] = 5$, we first have $\texttt{exp} = (1)_2$, then we concatenate 5 '0' bits to the right ($\texttt{exp} = (100000)_2$), finally the exponent value $(2 \times 3 + 1) = 7 = (111)_2$ is added ($\texttt{exp} = (100111)_2$).

4.7 Experimental Results

In this section, we show how using the power consumption of the mask computation, we can recover very accurately the full exponent with a single trace without any prior knowledge from other traces. For a given target, the only required preliminary step is to identify one ROI in the trace, which can be more or less time-consuming depending on the attack hypotheses – specifically whether the code can be modified or not. For these experiments, we thus suppose that for each trace, we have located manually one ROI, and then we use it as a pattern for the detection of all the other ROI. In practice however, the detection of this ROI of each trace was done using a common pattern in order to speedup the process, since the ROI shape does not change much between two different traces.

The attack achieved in this section comprises the following parts for each trace: **1)** Select one ROI as pattern; **2)** Locate all ROI using the pattern; **3)** Overlay all ROI found in the trace, in order to obtain the folded trace; **4)** For the traditional (resp. windowed) version, locate in the folded trace (resp. in each section of the folded trace), a sample

point for which all trace portions split very clearly into two sets, and use it as POI as shown in Figure 3; determine the POI thresholds; **5)** Recover the operations from the power consumption at the POI, and the exponent from the operations.

The attack presented only uses basic knowledge on the traces. However, by making a more detailed analysis, it is possible to improve the attack execution time. For the windowed version, using a 2,048-bit exponent, the full exponentiation takes around 530 millions cycles. Taking as pattern a complete ROI of 13,711 samples (large pattern), detecting all the ROI approximately takes two hours. As mentioned in section 4.3, we do not necessarily need to use the entire ROI as a pattern if we can find a part of the ROI which is sufficiently different from other parts of the trace. We have thus found two other good pattern choices: a 277-sample pattern corresponding to the first section of a ROI (medium pattern), and a 50-sample pattern corresponding to the mask computation part of the 16^{th} section (short pattern). However, the shorter the pattern, the more sensitive it is to variations in data. Therefore, we need two versions for the short and medium patterns, one for each condition value, for matching reliably both cases. In this setup, a ROI is detected when any pattern matches. On the other hand, as mentioned earlier, even though the number of cycles between two consecutive ROI vary, it is comprised in a given range. As a consequence, we can skip a certain number of samples after finding a ROI.

Table 1: Attack results for both the traditional and windowed versions, using 2,048-bit exponents

Version	Traces	Total bits	Pattern	Missed ROI	Time/ trace	Incorrect bits
Traditional	30	61,440	short	0	0.6 s	1
Windowed	30	61,440	large	0	5992 s	0
			medium	5	77 s	
			short	7	11 s	

Experiments were made on 30 traces using distinct 2048-bit exponents for each version. We used as POI the computation of `msk1` for both versions, as it seems a little more consistent in the last section of the windowed version. Thus in total, with patterns of different lengths, we obtain 4×30 independent ROI recoveries and 2×30 independent bits recoveries. The results are presented in Table 1. For the traditional version, no ROI is missed with the 30-sample pattern corresponding to the entire ROI, and all bit values except one are correctly found. For the windowed version, no ROI is missed with the large pattern, 5 ROI (in 5 different traces) are missed with the medium pattern pattern and 7 ROI (in 7 different traces) are missed with the short pattern. In any case, no incorrect bit value is recovered from a POI, showing the high accuracy of this attack. Besides, the selected medium and short patterns and the skip operation allow to greatly reduce the attack time for the windowed version, from almost two hours for the full pattern without the skip operation, to respectively 77s and 11s.

These results underly the fact that securing the exponentiation against side-channel attacks, even simple ones, is a challenging task. They suggest that it is hopeless to secure the exponentiation with techniques based on constant time or code tricks, but that countermeasures should instead focus on mathematical aspects, or include randomness.

5 Attacking ECDSA with a SPA

5.1 Characterisation and Setup

For ECDSA, we used the `bearssl` library [Por23], which includes some countermeasures against side-channel attacks. It is compiled with `gcc` 10.2.1 using optimisation level `O2`, and using the curve p256m31. The ECC multiplication in this library is constant time and uses a fixed 4-bit window over the scalar value, performing the loads of all the values in the table (external loop over `i`), and keeping only the correct value using a masking technique, as shown in Algorithm 11. Each value is a point of the curve (which is the multiplication of $\boldsymbol{G}$ by a scalar between 1 and 15), encoded as several 32-bit limbs in order to store the 256-bit coordinates: 9 limbs for the x coordinate and 9 limbs for the y coordinate[2]. The internal loop over `j` reads and masks these 18 limbs.

Algorithm 11 Pseudo-code of the LookupGwin function in the `bearssl` library

```
    ▷ idx contains the window value of the scalar bits
    ▷ Gwin[i][j] contains word j of (i+1)*G
    ▷ EQ(x, y) returns 1 if x = y, 0 otherwise
 1: function LookupGwin(idx)                          ▷ returns idx*G
 2:     P ← 0
 3:     for i from 0 to 14 do
 4:         mask ← -EQ(idx, i + 1)
 5:         for j from 0 to 17 do
 6:             P[j] ← mask & Gwin[i][j]
 7:     return P
```

We recorded traces consisting of the 64 iterations, each iteration processing 4 bits of the scalar value. Such a trace requires approximately 6,000,000 processor cycles to complete. As for RSA, we recorded the samples using the ChipWhisperer Pro device, with a STM32F3 target board comprising a Arm Cortex-M4. The number of cycles and samples in a trace imposes here again to use the capture device in streaming mode, which limits the number of samples to a single sample per cycle.

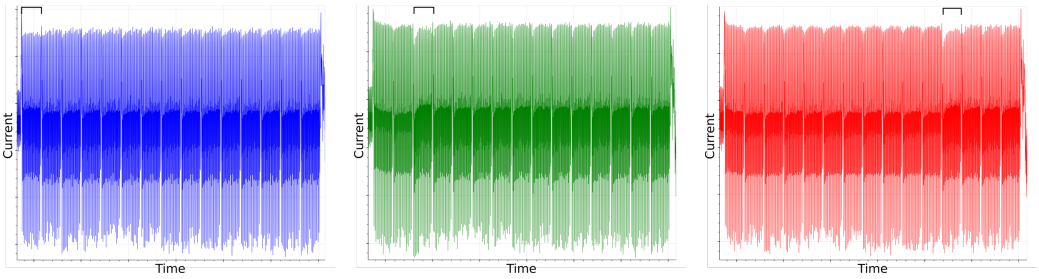
5.2 Attacking the Masked Loads

We first targeted the loads in the precomputed values table, and more specifically the computation of the `mask`. We manually located in the trace the power corresponding to the LookupGwin function calls, which constitutes our ROI. Since all the ROI are separated with a constant number of samples, the identification of the 64 ROI is straightforward. In Figure 4, we can clearly see that there are 15 sections on each subfigure (again separated by a fixed number of samples), while zooming in reveals that each section containing 18 smaller sections. These are exactly the `for i` and `for j` loops in LookupGwin.

We can already observe in Figure 4 that when the 4-bit value is equal to `i + 1` (`mask` = -1), the power consumption of the whole iteration is significantly different. This is because the program actually keeps the precomputed value only in this case, and thus the result of the bitwise AND operation will have a average Hamming Weight value of 15, instead of 0.

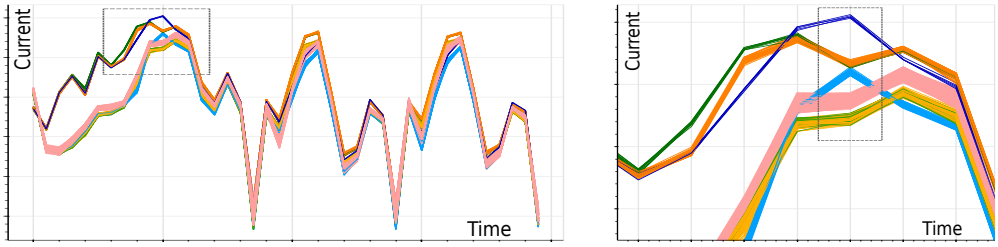
However, as in the case of the exponentiation, the power consumption difference resulting from mask computation is way clearer. By overlaying different ROI, we can observe a sample at the beginning of each section (`for i` iteration) whose value highly depends on the condition value. This sample in each section, that we use as POI, corresponds to the operation: `mask` ← -EQ(`idx`, `i` + 1).

[2] A point coordinate is encoded on 9 32-bit words, using 30 bits for the first 8 words, and 16 bits for the last



(a) Window value = 1 (b) Window value = 3 (c) Window value = 12

Figure 4: Power consumption of one LookupGwin function call for different window values.



(a) Mask computation and beginning of the copy (b) Focus on mask computation

Figure 5: Power consumption associated to the mask computation in ECDSA. The different colors indicate different sections and condition values.

Figure 5 illustrates the power consumption for all overlayed ROI, at the samples corresponding to the mask computation. In this figure, the top curves correspond to POI in the first section: the dark blue curves when the condition is true which means the window value is 1; the dark orange curves when the condition is false and the window value is different from 0; and the dark green curves when the condition is false and the window value is 0, meaning the condition will be false for all other sections. It is somewhat surprising to see that some sample points allow to discriminate very clearly between a 0 and a non-zero window value; yet regardless, we did not use this information in our attack, focusing only on the condition result at each iteration. The bottom curves correspond to sections 1 to 14 (from the second to the 15^{th}): the blue curves when the condition is true, i.e. when the iteration matches the window value, the pink curves when the condition is false but has already been true in one of the previous sections of current ROI, the orange curves when the condition is false and has not yet been true in one of the previous sections of current ROI, and the green curves (mixed with the orange ones) correspond to false conditions when the window value is 0, because no condition of any section in this ROI will be true when the value is 0.

From this figure, we can observe that the power consumption in the first section is significantly different from the other sections, for which the measures all coincide. Therefore, we decided to use two thresholds (determined with the technique described in Algorithm 8): one for the first section and one for the others. As the number of iterations is only 64, this allows to have a threshold even for window values which are never reached. As for the window value 1, if it is never reached, all curves at the POI will overlay and the assumption must be made that this value is never obtained.

Putting apart the first iteration, we thus decided to look whether one of the measure

Algorithm 12 ECDSA Digits recovery using the power measures

```
1: function BitsRecovery(trace, poi)
2:     cons1 = [ ], cons2 = [ ], secret = ''
3:     for j from 0 to 63 do
4:         cons1.append(trace[poi[j][0]])
5:         for i from 1 to 14 do
6:             cons2.append(trace[poi[j][i]])
7:     threshold2 = find_threshold(cons2)
8:     threshold1 = find_threshold(cons1)
9:     for j from 0 to 63 do
10:        if cons1[j] > threshold1 then
11:            secret += '1'
12:        else
13:            cons_j = cons2[j × 14 : (j + 1) × 14] ▷ Sublist with the POI of the j^th ROI only
14:            if max(cons_j) > threshold2 - ε then
15:                secret += str(cons_j.argmax(cons_j) + 2)   ▷ hex character encoding 4 bits
16:            else
17:                secret += '0'
18:    return secret
```

at the POI was greater than the threshold: if not, the value 0 is deduced, otherwise, the window value taken is the index corresponding to the maximum of the observed values (Algorithm 12). Finally, since the discrimination using the threshold must only be done between a 0 value and a non-0 value (sets of blue and light green/light orange curves), the threshold can optionally be adapted for the attack to be more robust. In practice, we used an ε value of 0.01, from the observations made in Figure 5.

We captured 1,000 traces and ran the attack on each of these traces: on the 1,000 $\times$ 64 window values, we recovered correctly all the scalar bits of the nonce `k`, using Algorithm 12. These results are summarized in table 2.

Table 2: Results for the ECDSA attack for 1,000 traces.

Nb. Traces	Nb. of digits	Nb. of secret nonces	Percentage of recovered digits	Percentage of recovered secret nonces
1,000	64,000	1,000	100%	100%

5.3 Attacking the 0 Values

As seen in section 2.2, the considered implementation iteratively loads precomputed values from a table. Each value consists of several limbs in order to reconstruct the 256-bit coordinates. Due to the nature of the algorithm, when the window value is 0, the result of its multiplication with the base point is the point at infinity, encoded as limbs with value 0 only. Thus, when loading and writing iteratively all of the words of the resulting point, the Hamming Weight of each word will be 0, while it will be 15 on average for other points. This difference in Hamming Weight is actually sufficient to identify when the point at infinity is used. This can actually lead to an attack, since the knowledge of the "zero" values in `k` on a few traces is sufficient to recover the secret [GRV17, HGS01]. Note that this attack is independent from the masking technique, and targets the copy of the resulting point at the end of the LookupGwin function.

Here again, by overlaying the ROI corresponding to the copy, two sets of traces appear clearly, as shown in Figure 6. This figure shows that there is a clear difference in power

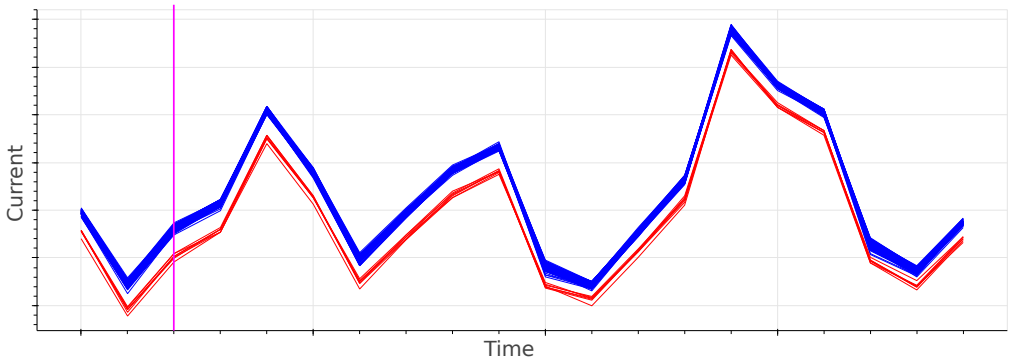


Figure 6: Power consumption for the reads and writes of machine words representing a point of the curve. Red curves indicate the point at infinity, while blue curves indicate another point. The purple line indicates the sample used for the attack.

Table 3: Results for the ECDSA attack consisting in determining the 0 values.

Number of traces	Number of digits	Number of '0' digits incorrectly found	Number of non-'0' digits incorrectly found	Attack Accuracy
1,000	64,000	1	0	99.998%

consumption between a window value of 0 (red curves) and a window value different from 0 (blue curves), due to the difference in Hamming Weights.

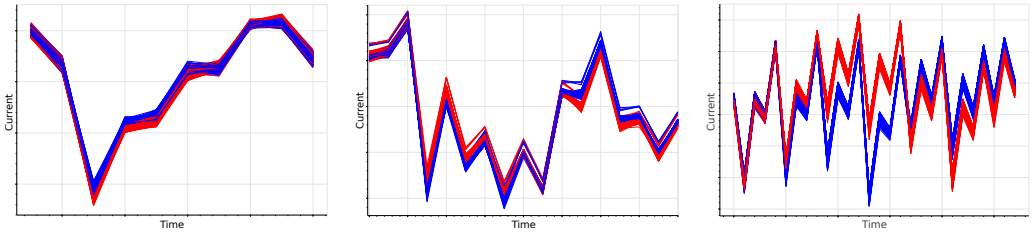
Once again, we achieved this attack without the knowledge of previous executions, and the thresholds were determined using only the trace itself. We captured 1000 traces and for each trace and each iteration, we guessed if the loaded point was the point at infinity or not based on the power consumption of the selected sample. Similarly to the previous attack, for traces containing no zeros, we used a criteria on the found maximum difference in the ordered power values, and concluded that all windowed values were different from 0. The results, summarized in table 3 show that all digits were correctly identified except a single digit.

6 Extensibility of the Attack and Limitations

6.1 Extensibility Regarding the Architecture

The two hypotheses required for the presented attacks to work are the following: 1) all the ROI should overlay; 2) the power consumption for the mask computation should be significantly different depending on the condition value. While we do not know if these hypotheses will hold for any architecture, it is likely that they should be true often: the first one because the compiled code should follow the source code structure, which does not contain any conditional branch inside the ROI; the second one because the mask computation will always produce a different number of bit flips on the data path depending on the condition value, these bit-flips constituting the most important part in power consumption.

In order to support this statement, we have analysed the power consumption of the SetCond function on different architectures: Arm Cortex-M0 (STM32F0), Arm Cortex-M3 (STM32F1), and XMEGA (ATMEGA128-16AUR). The results in Figure 7 show that



(a) Cortex-M0 core (b) Cortex-M3 core (c) XMEGA core

Figure 7: Power consumption of the masks' computation in the SetCond function for alternative architectures depending on the condition value (0 in blue, 1 in red).

there is a clear difference between the two condition values for the XMEGA and the Cortex-M3 targets – even if for the latter, some traces are shifted above the others. For the Cortex-M0, a single sample does not allow to discriminate clearly, but a difference is still visible. These results show that this attack can work on different architectures.

6.2 Extensibility Regarding the Library

In order to evaluate the impact of the proposed attack, we have analysed the source code of three other libraries, in addition to `libgcrypt` and `bearssl`, for the exponentiation used in RSA: `GMP`, `OpenSSL` and `mbedtls`. As it turns out, all three libraries use similar techniques with a mask computation for achieving constant time.

For `GMP`, the computation is made in a function called `mpn_sec_tabselect`, which loads a given precomputed element from an array. There are actually two versions of this function, which process in a different order the limbs and the elements in the array of precomputed elements, but make essentially the same treatment. A code fragment of this function is shown in listing 1, in which `k` is the iteration number, and `which` is the window value. The shown code is performed for each element in the array.

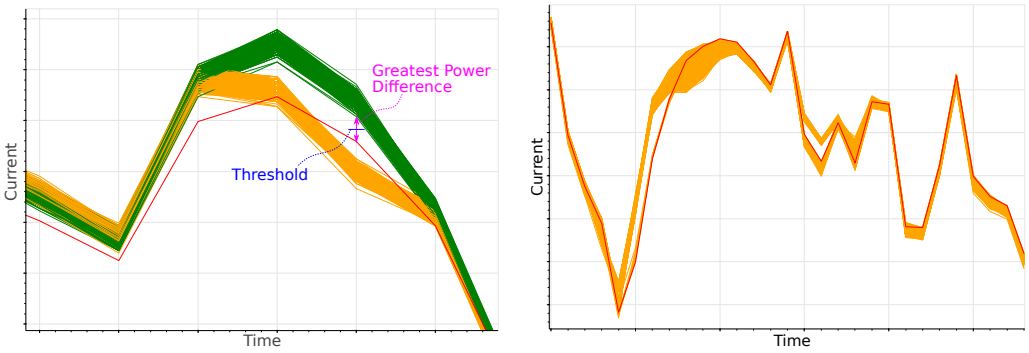
Listing 1: Mask computation and loading of the precomputed element in `GMP`

```
mask = -(mp_limb_t) ((-(unsigned long) (which ^ k)) >> (
    BITS_PER_ULONG - 1));
tp += n;
for (i = 0; i < n; i++)
    rp[i] = (rp[i] & mask) | (tp[i] & ~mask);
```

For `OpenSSL`, the function `MOD_EXP_CTIME_COPY_FROM_PREBUF` accumulates in a machine word the corresponding word from all precomputed elements after having masked them. Code listing 2 shows how this is performed, using the function `constant_time_eq` which computes the mask value based on the current iteration `j` and the window value `idx`.

Listing 2: Mask computation and loading of the precomputed element in `OpenSSL`

```
for (i = 0; i < top; i++, table += width) {
  BN_ULONG acc = 0;
  for (j = 0; j < xstride; j++) {
    acc |= ((table[j+0*xstride] & y0) | (table[j+1*xstride] & y1) |
            (table[j+2*xstride] & y2) | (table[j+3*xstride] & y3))
          & ((BN_ULONG) 0 - (constant_time_eq_int(j, idx) & 1));
  }
  b->d[i] = acc;
}
```



(a) Incorrect recovered bit from POI (in red) for the traditional version. The first 3 iterations have been omitted for clarity.

(b) Missed ROI (in red) compared to the other ROI for the 50-sample pattern and the windowed version.

Figure 8: Examples of errors encountered by the RSA attack.

Finally, in `mbedtls`, the function `mbedtls_ct_uint_eq` performs a constant time mask computation using the function `mbedtls_ct_uint_eq`, followed by a conditional assignment with the function `mbedtls_mpi_core_cond_assign`. This is illustrated in listing 3 in which `index` is the window value, `count` the number of precomputed elements, and `assign` the mask value.

Listing 3: Mask computation and loading of the precomputed element in `mbedtls`

```
for (size_t i = 0; i < count; i++, table += limbs) {
    mbedtls_ct_condition_t assign = mbedtls_ct_uint_eq(i, index);
    mbedtls_mpi_core_cond_assign(dest, table, limbs, assign);
}
```

In conclusion, while we do not know if this technique is used in every cryptographic library, it is used in the most common ones and thus the attack presented has a larger scope than the two attacked libraries.

6.3 Limitations and Error Analysis

For the RSA attack, one bit is incorrectly detected for the traditional version. Figure 8a shows this incorrect bit: green curves correspond to a condition value of 0 while orange curves a condition value of 1. The red curve is detected as a 1 instead of a 0, due to the threshold being above the red curve. We can notice that this error could be managed with a slightly more complex approach, considering for example two or more consecutive power values around the POI and compute their difference in order to deduce the shape of the correct curve. However, even with the simple threshold strategy, this is the single error on the 61,440 attacked bits for the traditional version, while no error occurs for the windowed version.

We also investigated the missed ROI for the windowed version and noticed that they all occurred at the end of the trace. For these ROI, a small part of the trace seems shifted, leading to a higher difference with the pattern, as illustrated in Figure 8b. While we did not manage to understand the cause of this difference, its impact on the attack remains limited and as we mentioned earlier, all the missed ROI are easy to locate.

7 Attacking a Countermeasure from the Literature

In their 2022 article, Saito *et al.* [SIUH22] propose the solution recalled in Algorithm 13 to safely load the window value, using two random masks. They conclude that it will mitigate the two vulnerabilities exploited in their proposed attack.

Algorithm 13 Proposed Operand Loading Process from [SIUH22]

Inputs: Precomputed table ($\boldsymbol{c}^0$, $\boldsymbol{c}^1$, ..., $\boldsymbol{c}^{2^{\mathtt{w}}-1}$), window size $\mathtt{w}$, window value $\mathtt{b}$
Output: Multiplication operand $\boldsymbol{s} = \boldsymbol{c}^{\mathtt{b}}$

1: **function** LoadOperand(($\boldsymbol{c}^0$, $\boldsymbol{c}^1$, ..., $\boldsymbol{c}^{2^{\mathtt{w}}-1}$), $\mathtt{w}$, $\mathtt{b}$)
2: $\quad \boldsymbol{r_{value}} \leftarrow \text{GenerateRandom}_l()$ ▷ Generate a l-bit random mask
3: $\quad \boldsymbol{s} \leftarrow \boldsymbol{c}^0 \oplus \boldsymbol{c}^1 \oplus ... \oplus \boldsymbol{c}^{2^{\mathtt{w}}-1} \oplus \boldsymbol{r_{value}}$
4: $\quad \mathtt{r}_{\mathtt{order}} \leftarrow \text{GenerateRandom}_w()$ ▷ Generate a w-bit random mask
5: $\quad \mathtt{b}_{\mathtt{masked}} \leftarrow \mathtt{b} \oplus \mathtt{r}_{\mathtt{order}}$
6: $\quad$ **for** $\mathtt{i}$ from 0 to $2^{\mathtt{w}-1}$ **do**
7: $\quad\quad \mathtt{mask} \leftarrow$ MakeBitMask($\mathtt{i}$, $\mathtt{b}_{\mathtt{masked}}$) ▷ MakeBitMask returns 1...1 if $\mathtt{i} = \mathtt{b}_{\mathtt{masked}}$, else 0
8: $\quad\quad \boldsymbol{s} \leftarrow \boldsymbol{s} \oplus (\boldsymbol{c}^{\mathtt{i} \oplus \mathtt{r}_{\mathtt{order}}} \,\&\, \neg\, \mathtt{mask} \mid \boldsymbol{r_{value}} \,\&\, \mathtt{mask})$
9: $\quad$ **return** $\boldsymbol{s}$

Using the experience acquired with our previous attacks, we show how the exponent value can still be recovered. The attack works by identifying two distincts iterations: the iteration $\mathtt{it}_0$ for which $\mathtt{i} = \mathtt{b}_{\mathtt{masked}}$, and the iteration $\mathtt{it}_1$ for which $\mathtt{i} \oplus \mathtt{r}_{\mathtt{order}} = 0$. We thus have $\mathtt{it}_0 = \mathtt{b}_{\mathtt{masked}}$ and $\mathtt{it}_1 = \mathtt{r}_{\mathtt{order}}$. Using the fact that $\mathtt{b}_{\mathtt{masked}} = \mathtt{b} \oplus \mathtt{r}_{\mathtt{order}}$, we have $\mathtt{b} = \mathtt{it}_0 \oplus \mathtt{it}_1$.

Identifying $\mathtt{i} = \mathtt{b}_{\mathtt{masked}}$. As we have seen in sections 4 and 5, determining the value of a condition when the latter is used to create a word with only zeros or ones can be done very accurately. Therefore, the MakeBitMask function is very likely to exhibit this iteration.

Identifying $\mathtt{i} \oplus \mathtt{r}_{\mathtt{order}} = 0$. In this iteration, the element loaded in the precomputed table is $\boldsymbol{c}^0 = 1$. As we have seen in section 4, in `Libgcrypt` the precomputed elements are encoded on the minimum required number of limbs. Hence in this case, it is sufficient to count the number of iterations during the copy: when a single limb is copied, it means that $\boldsymbol{c}^0$ is loaded. Yet, even if all precomputed elements are stored on the same number of limbs (e.g. 64 limbs for 2,048 bits), it is possible to identify precisely when words containing the value 0x0 are loaded as opposed to words containing roughly half of their bits set, as seen in section 5. In this case, we can then simply count the number of words with value 0 loaded during the copy: the maximum will be reached for $\boldsymbol{c}^0$.

As an alternative, the iteration for which $\mathtt{i} \oplus \mathtt{r}_{\mathtt{order}} = 1$ can also be identified, i.e. when $\boldsymbol{c}^1$ is loaded. Indeed, most implementations follow the PKCS #1 v1.5 standard [MKJR16], which specifies that the base must be padded with the byte value 0xFF to reach the modulus size. In that case, the power consumption associated to loading words with all bits set will easily be recognizable from the power consumption when loading pseudo-random words.

Either way, identifying any of these two iterations allows the recovery of $\mathtt{r}_{\mathtt{order}}$, and thus of $\mathtt{b}$, making the countermeasure inefficient.

8 Conclusion

In this article, we showed that power differences associated to variations in data values can be exploited for practical Simple Power Attacks. In particular, we showed that the masking technique often used in constant time implementations to keep one among two possible values leaks its condition in a way that a single sample allows to recover it without error. We illustrated this attacks on the two implementations of the modular exponentiation in `Libgcrypt` and on the scalar ECC multiplication of the `bearssl` library. We additionally

showed that the load of memory words containing either the value 0 or a random value can be clearly identified and also used as an attack vector. Finally, we explained how a proposed countermeasure from the literature for performing secure loads of precomputed values can be attacked. Future work includes the applicability of the proposed attack concepts to other cryptographic domains, as well as designing secure implementations taking into account the presented findings.

References

[AYW+21] Monjur Alam, Baki Yilmaz, Frank Werner, Niels Samwel, Alenka Zajic, Daniel Genkin, Yuval Yarom, and Milos Prvulovic. Nonce@Once: a single-trace EM side channel attack on several constant-time elliptic curve implementations in mobile platforms. In *IEEE European Symposium on Security and Privacy*, 2021.

[BBG+17] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 555–576. Springer, 2017.

[GB23] Utku Gulen and Selcuk Baktir. Side-channel resistant 2048-bit RSA implementation for wireless sensor networks and internet of things. *IEEE Access*, 2023.

[GRV17] Dahmun Goudarzi, Matthieu Rivain, and Damien Vergnaud. Lattice attacks against elliptic-curve signatures with blinded scalar multiplication. In *Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers 23*, pages 120–139. Springer, 2017.

[HGS01] Nick A Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23:283–290, 2001.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1:36–63, 2001.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[LH20a] JongHyeok Lee and Dong-Guk Han. DLDDO: deep learning to detect dummy operations. In *International Conference on Information Security Applications*, pages 73–85. Springer, 2020.

[LH20b] JongHyeok Lee and Dong-Guk Han. Security analysis on dummy based side-channel countermeasures—case study: AES with dummy and shuffling. *Applied Soft Computing*, 93:106352, 2020.

[LLQ+20] Qi Lei, Chao Li, Kexin Qiao, Zhe Ma, and Bo Yang. Vgg-based side channel attack on RSA implementation. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1157–1161. IEEE, 2020.

[MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[MV19] Thierry Mefenza and Damien Vergnaud. Cryptanalysis of server-aided RSA protocols with private-key splitting. *The Computer Journal*, 62(8):1194–1213, 2019.

[NCOS16] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. In *International Conference on Selected Areas in Cryptography*, pages 99–119. Springer, 2016.

[OC14] Colin O'Flynn and Zhizhang David Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In *COSADE*, 2014.

[Por23] Thomas Pornin. BearSSL implementation of the SSL/TLS protocol. https://bearssl.org/, 2023.

[SIUH22] Kotaro Saito, Akira Ito, Rei Ueno, and Naofumi Homma. One truth prevails: A deep-learning based single-trace power analysis on rsa–crt with windowed exponentiation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 490–526, 2022.

[Ver20] Damien Vergnaud. Comment on "efficient and secure outsourcing scheme for RSA decryption in internet of things". *IEEE Internet of Things Journal*, 7(11):11327–11329, 2020.

[WCPB20] Léo Weissbart, Łukasz Chmielewski, Stjepan Picek, and Lejla Batina. Systematic side-channel analysis of curve25519 with machine learning. *Journal of Hardware and Systems Security*, 4:314–328, 2020.

[WPB19] Leo Weissbart, Stjepan Picek, and Lejla Batina. One trace is all it takes: Machine learning-based side-channel attack on eddsa. In *Security, Privacy, and Applied Cryptography Engineering: 9th International Conference, SPACE 2019, Gandhinagar, India, December 3–7, 2019, Proceedings 9*, pages 86–105. Springer, 2019.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.