Practical timing and SEMA on embedded OpenSSL's ECDSA

Julien Eynard[1], Guenaël Renault[1], Franck Rondepierre[1], Adrian Thillard[2]

[1] Harware Security Lab, ANSSI
[2] Ledger Donjon

Abstract

Timing attacks are a class of side-channel attacks allowing an adversary to recover some sensitive data by observing the execution time of some underlying algorithm. Several cryptographic libraries have been shown to be vulnerable against these attacks, oftentimes allowing practical key recoveries. While recent papers showed that these threats are well-known amongst developers, these libraries are often left unpatched, due to the perceived burden of implementing efficient countermeasures. Instead, many libraries chose to modify their threat model and to not consider attacks where the adversary have local access to the target anymore.

In this paper, we show how to implement, on a real world device, a recent timing attack described by Weiser et al. at Usenix20, targeting OpenSSL's ECDSA. We expand their discovery and demonstrate that this attack applies to a bigger set of curves than claimed in the original paper. After characterising the weakness against timing, we show that the perceived safety that can be provided by a practical resistance against those attacks can easily be shattered using slightly costlier attacks such as Simple Electro-Magnetic Analysis. Our work hence highlight that secure embedded purposes require a very careful choice of side-channel resistant library.

1 Cryptographic libraries and timing attacks

Side-channel attacks exploit various physical leakages related to variables being manipulated by a device. An attacker can leverage the observation of these leakages to recover underlying secret values. These attacks have first been described in the literature by Kocher [7], and have since be applied on dozens of embedded targets. Several of these attacks have been illustrated at SSTIC on eg. smart cards or smartphones [2, 10, 13].

Timing attacks, as described in 1996 by Paul Kocher in his seminal work [7] allow to recover information about secret data by measuring the execution time of the underlying algorithm. They have been shown to be particularly

powerful against various cryptographic libraries, and caused the publication of many CVEs[1]

Recently, [5] conducted a survey on cryptographic libraries developers. The survey showed that, even though most developers are aware of these kind of attacks, the perceived cost of implementing efficient countermeasures against them is too high. Instead, several libraries have decided to exclude "hardware" side-channel attacks from their threat model. This is in particular the case of the widely used library OpenSSL. While numerous side-channel attacks have been published and exploited against this library, its threat model has for a while excluded so-called "hardware" side-channel and it was updated in May 2019 to exclude "same physical system side-channel" (eg. prime+probe attacks).

Before this change, from 2003 to 2018, 10 CVEs were published mentioning timing attacks on OpenSSL. In this work, we want to stress that this library still suffers from timing flaws that could be exploited. In particular, we want to highlight the practical damage that can be induced by an attacker that is supposed out of the scope of this model. At Usenix20, Weiser et al. [14] described a class of side-channel vulnerabilities present in many ECDSA implementations. OpenSSL decided not to fix their big number library, which was responsible for this weaknesses.

In this section, we shortly describe the OpenSSL's ECDSA vulnerability, and how it can theoretically be exploited. Furthermore, we show that the results of Weiser et al. can be extended to more curves than originally identified.

1.1 ECDSA

Elliptic Curve Digital Signature Scheme (ECDSA) is a signature scheme based on the hardness of computing a discrete logarithm over elliptic curves. The scheme requires a given curve $\mathcal{E}$, a generator point G of order n, and a hash function H. Besides, the signer generates a key pair (d, P), where $d < n$ is a private key, and $P = [d]G$ is the associated public key.

In order to sign a message m, the signer performs the following operations:

1. draw a random nonce k such that $0 < k < n$
2. compute $r = x([k]G)$, where $x(\cdot)$ returns the abscissa of a point
3. compute $s = k^{-1}(H(m) + rd) \mod n$

The signature is defined as the couple (r, s). It is well known that the nonce k shall remain secret, the private key d can be trivially recovered since:

$$d = r^{-1}(sk - H(m)) \mod n$$

[1]For example, the first timing-related CVE on OpenSSL was CVE-2003-0078, and was exploitable through Vaudenay's padding oracle attack.

1.2 OpenSSL's implementation

As suggested in appendix A.3.1 of the FIPS publication on Digital Signature (FIPS 186-5) OpenSSL draws a random nonce k' in the interval $[0, 2^{\log(n-1)+64}]$ and then compute $k = k' \mod n$. This modular reduction is computed through a euclidean division.

Modular reduction `BN_mod`, in a straightforward call to `BN_div` (code link)

The following snippet illustrates a simplified version of the div function [2]

```
1  div(BIGNUM *dv, BIGNUM *rm, const BIGNUM *num, const BIGNUM *
       divisor, BN_CTX *ctx)
2  {
3      int ret;
4      [...]
5      ret = bn_div_fixed_top(dv, rm, num, divisor, ctx);
6
7      if (ret) {
8          if (dv != NULL)
9              bn_correct_top(dv);
10         if (rm != NULL)
11             bn_correct_top(rm);
12     }
13     return ret;
14 }
```

Note that line 11 calls `bn_correct_top`, which "corrects" the number of words needed to store the remainder, that is, it gets rid of the zero most significants words. Consequently, at the end of step 1, the random nonce k is coded on the exact number of words that it requires. It follows that any observation of the number of words used to store k would reveal information on its length.

Furthermore, OpenSSL's big number inversion is dependent to the manipulated data. The inversion is computed as an exponentiation $k^{-1} = k^{n-2} \mod n$, since the order n is a prime value. The exponentiation is performed thanks to Montgomery products denoted as $\star$: $a \star b = a \cdot b \cdot R^{-1} \mod n$, with $R = 2^u \mod n$ for an appropriate integer u. The first operation consists in computing $k \star R^2$, where $R^2 \mod n$ is a precomputed value with same length as n. This multiplication involves the following snippet (code link). Part of its code is reproduced hereafter [3]:

```
1  int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM
       *b, BN_MONT_CTX *mont, BN_CTX *ctx)
2  {
3      BIGNUM *tmp;
4      int ret = 0;
5      int num = mont->N.top;
6
```

[2] It was pointed in a 2018 issue (https://github.com/openssl/openssl/issues/6367) that `BN_div` is not time-constant. However, it is unclear how this issue could be straightforwardly exploited.

[3] Some parts of this code are conditioned to the set of compilation flags `OPENSSL_BN_ASM_MONT` and `MONT_WORD`. These flags are very often present by default.

```
7     if (num > 1 && a->top == num && b->top == num) {
8         if (bn_wexpand(r, num) == NULL)
9             return 0;
10        if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num)
      ) {
11            r->neg = a->neg ^ b->neg;
12            r->top = num;
13            r->flags |= BN_FLG_FIXED_TOP;
14            return 1;
15        }
16    }
17    [...]
18    if (a == b) {
19        if (!bn_sqr_fixed_top(tmp, a, ctx))
20            goto err;
21    } else {
22        if (!bn_mul_fixed_top(tmp, a, b, ctx))
23            goto err;
24    }
25    /* reduce from aRR to aR */
26    if (!bn_from_montgomery_word(r, tmp, mont))
27        goto err;
28    ret = 1;
29 err:
30    BN_CTX_end(ctx);
31    return ret;
32 }
```

The execution flow of this function depends on its inputs: if the condition at line 7 is satisfied, then the code between lines 8 to 15 will be executed (and the rest will most likely not be), otherwise, the code between lines 18 to 31 will be executed instead.

This part of the code will provide us the required observability of the bias. Indeed, considering the inputs $a = k$ and $b = R^2$, the condition at line 7 simply checks that the number of machine words used to store the curve order n is strictly larger than one, and that the same number of words are used to store a and b. Therefore, whenever k is stored on the same number of words as n, the condition will be satisfied, and, whenever it is not, the condition will not be satisfied. The amount of instructions and different functions in each part of the code will lead to an observable behaviour by a side-channel attacker, hence allowing him to recover this information about the size of k.

1.3 Hidden Number Problem and LLL

The Hidden Number Problem (HNP) is a mathematical problem introduced by Boneh and Venkatesan [1]. In the ECDSA context, the problem asks for the recovery of the secret key x from the knowledge of some information on the nonces k. We provide the interested reader with an explanation of the HNP in a python notebook in the supplementary material of this paper. This notebook also proposes a hands-on resolution of a practical instance of this problem.

1.4 Finding vulnerable ECDSA curves

We showed in the previous section that an attacker is able to distinguish whether the nonce k and the curve order n are stored on the same number of words. In order to be exploitable, this observation needs to allow the recovery of different nonce sizes a significant number of times. This simple observation implies that not all curves are vulnerable against this attack *in practice*.

As a simple counter-example, let us consider the standard curve `secp256k1`. The order n of this curve is $\frac{256}{w}$-word long on any w-bit architecture. n is in fact close to $2^{256} - 2^{128}$, and hence, on the vast majority of architectures, the binary expression of its most significant word is fully composed of ones. The probability for any uniformly generated nonce between 0 and n to require less than $\frac{256}{w}$ words is then close to 2^{-w}. Since a practical attack requires several dozens of such nonces, the amount of total signatures to collect is prohibitive on most architectures, eg. at least several dozens or hundreds of billions observations in the case of a 32-bit architecture, and significantly worse on a 64-bit one.

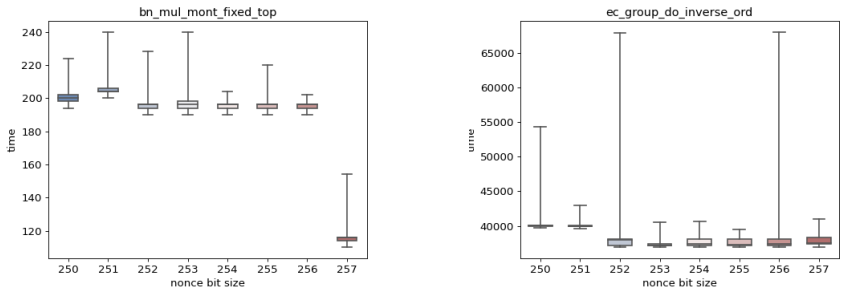
On the contrary, let us consider the standard curve `secp521r1`, whose order's most significant 64 bits are `0x000001FF`. Considering a 64-bit or a 32-bit architecture, any uniformly generated nonce between 0 and this order has a probability of 2^{-9} to have a null most significative word. This event is much more likely to be observed, and the total number of required signatures to perform an attack can then be estimated around tens of thousands, which is much more practical.

Using this method, Weiser et al. [14] found 32 standardized curves implemented in OpenSSL that are vulnerable against this attack on 32-bit architectures (3 of them stay vulnerable on 64-bit architectures). By slightly extending their study, we are able to discover twelve more theoretically-vulnerable curves, among which 3 of them are vulnerable in practice. Simply put, Weiser et al. looked for curve orders implying a biased distribution of the nonces, producing most of the time *full-size* nonces, and sometimes *short* nonces. We extend this search and observe that some curve orders may imply a converse distribution, ie., a production of *short* nonces most of the time, and *full-size* nonces sometimes. This is for example the case for the standardized curve `c2pnb272w1`, which order's 64 most significant bits are `0x000100FA`: a nonce generated uniformly between 0 and this order has a probability close to $1 - 2^{-8}$ to have null 32 most significant bits, and therefore only a 2^{-8} probability to require as many words as the curve order on a 32-bit architecture.

The complete list of vulnerable curves can be found in the full version of this paper.

2 Timing attack on some bits of the exponentiated message

In their paper, Weiser et al.[14] claim that the identified vulnerability is exploitable with a SGX-enclave setup. We will demonstrate that this attack can



(a) Execution time (in CPU ticks) of `bn_mul_mont_fixed_top`.

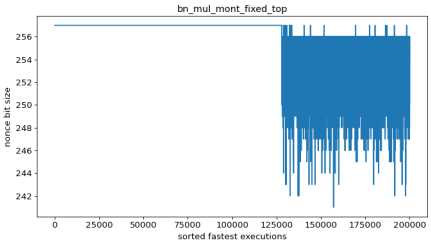
(b) Execution time (in CPU ticks) of `ec_group_do_inverse_ord`

Figure 1: Means and variance of the execution time (in CPU ticks) of two different functions depending on the size of their inputs. (10K samples per size)

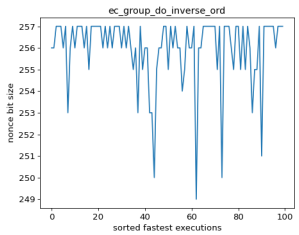
in fact also be mounted in a practical embedded context. To this end, we embedded the OpenSSL library (version 1.1.1k) on a Raspberry Pi 4 board. Our first goal is to show that the difference of execution path between the two possible branches actually translates into an observable timing difference.

We choose to illustrate this difference at the lower level, by measuring the execution time of the function `bn_mul_mont_fixed_top`. We start by setting ourselves in an ideal setup, and measure the ticks between the start and end of this function, for different input sizes around 256 bits. The results of this experiment can be seen in Figure 1 (a). One can clearly observe a huge timing difference on this function, which validates the presence of the detected bias. This difference can be used by an attacker to detect with a near-perfect confidence the word-length of the input. In order to confirm that this bias can be exploited when timing this function, we drew 2^{25} random nonces smaller than the order, and ran the function. We observed (see Fig. 2 (a)) that the fastest $\sim 2^{17}$ executions were associated with a maximal nonce size. It is by far sufficient to mount and succeed the attack.

We exhibit in Figure 1 (b) the same experiment on the higher level function `ec_group_do_inverse_ord`. This time, no timing difference can be observed by the attacker. This is easily explained by the noise induced by a longer execution time, and the possibility of other timing biases occuring during the process. Once again we ran 2^{25} executions with random nonces. As it can be seen on Figure 2 (b), it is not possible to select the maximal nonce size based on the execution time. This negative result may lead a designer under the false impression that the bias is too hard to exploit in practice, and hence that the vulnerability is residual. We will show in the next session that the bias is in fact still observable by a slightly stronger adversary, and we demonstrate how to recover the secret key of the whole ECDSA signature.



(a) Zoom on 200,000 fastest executions of `bn_mul_mont_fixed_top`



(b) Zoom on 100 fastest executions of `ec_group_do_inverse_ord`

Figure 2: Nonce size (y-axis) for the fastest of 2^{25} execution times of (a) `bn_mul_mont_fixed_top`, (b) `ec_group_do_inverse_ord`.

3 Practical SEMA attack on ECDSA

In this section, we fill the gap from theory to practice, by performing a practical, end-to-end implementation of this attack. We now consider the highest possible function call, and setup a practical attack on the whole ECDSA. To do so, we target our Raspberry Pi setup and call the ECDSA signature through the command line. We choose to perform our experiments using `secp521r1` for illustration purposes, since this is the curve relying on operations on the biggest numbers.

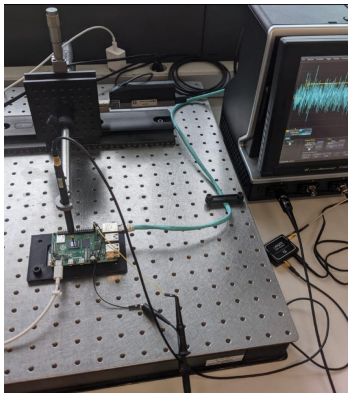


Figure 3: SEMA Acquisition bench.

We now leverage our physical access to the device by observing its electro-magnetic emanations while performing the signatures. This approach is commonly known as Simple Electro-Magnetic Analysis (SEMA), and relies on the electronical hypothesis that different manipulated data and operations will induce different EM behaviour. This side-channel has been often used in practice to break cryptographic implementations (eg. [11]).

To this end, we use an EM observation probe Langer RF-U 2,5-2 in order to measure the signal, as well as a Lecroy scope capturing 1G samples by second. Figure 3 illustrates our setup bench. We then record the electromagnetic emanations of the device during the execution of the ECDSA signature, for different nonces. The knowledge of the secret key allows us to recover these nonces and separate the curves.

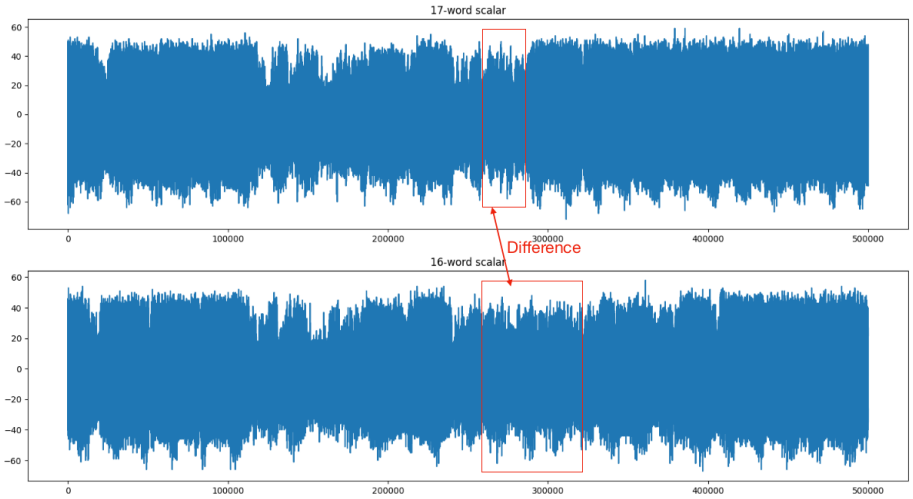


Figure 4: Electromagnetic emanations of the Raspberry Pi during the execution of the ECDSA signature on secp521r1, for different nonces lengths. Top: 17-word long nonce, bottom: 16-word long nonce. An easily observable difference can be identified by an adversary.

This experiment clearly exhibits an important difference between the behaviour of long and short nonces, as can be seen in Figure 4. Indeed, while the first half of the window is similar in the two contexts, a divergence occurs starting near time sample 250000, revealing the underlying different paths taken by the executed code.

We indeed tested this approach on an attacker scenario, with a supposed unknown key. To this end, we collected 51200 traces, which we automatically processed to find 104 of them containing the distinguishing pattern. By running an LLL approach on the 104 corresponding signatures, we were able to solve the hidden number problem and recover the private key.

4 Conclusion

We showed how an identified timing flaw can translate into a practical attack in an embedded setting. We extended the results of Weiser et al. by discovering several more vulnerable curves in OpenSSL and validated the observability of the bias in timing at the lower level. Finally, we used SEMA to actually perform an end to end exploitation of the vulnerability. While we stress that OpenSSL is now considering these attacks as out of scope, - despite developers being most of the time aware of the threats of timing attacks-, we firmly believe that our work strongly highlights the potential drawbacks of this decision, and warns users of cryptographic libraries to carefully take into account these vulnerabilities.

References

[1] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 1996.

[2] Christophe Devine, Manuel San Pedro, and Adrian Thillard. A practical guide to differential power analysis of usim cards. SSTIC, 2018.

[3] Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. Attacking (EC)DSA given only an implicit hint. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, volume 7707 of *Lecture Notes in Computer Science*, pages 252–274. Springer, 2012.

[4] Martin Hlavác and Tomás Rosa. Extended hidden number problem and its cryptanalytic applications. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *Lecture Notes in Computer Science*, pages 114–133. Springer, 2006.

[5] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ¨ They are not that hard to mitigate¨ : What cryptographic library developers think about timing attacks. *IACR Cryptol. ePrint Arch.*, page 1650, 2021.

[6] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sýs. Minerva: The curse of ECDSA nonces systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):281–308, 2020.

[7] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[8] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[9] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptol.*, 15(3):151–176, 2002.

[10] Manuel San Pedro, Victor Servant, and Charles Guillemet. Side-channel assessment of open source hardware wallets. Cryptology ePrint Archive, Report 2019/401, 2019. `https://ia.cr/2019/401`.

[11] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 231–248. USENIX Association, 2021.

[12] Keegan Ryan. Return of the hidden number problem. A widespread and novel key extraction attack on ECDSA and DSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):146–168, 2019.

[13] Aurélien Vasselle. Side-channel attack on mobile firmware encryption. SSTIC, 2019.

[14] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1767–1784. USENIX Association, 2020.