Return of ECC dummy point additions: Simple Power Analysis on efficient P-256 implementation

Andy Russon
andy.russon@orange.com

Orange

Abstract. This paper aims to show that the efficient implementation of elliptic curve P-256 (also known as `secp256r1`), present in several cryptographic libraries such as OpenSSL and its forks, is vulnerable to Simple Power Analysis (SPA).

This is made possible by the use of dummy point additions to make to the scalar multiplication constant-time with a regular behavior. However, when not done carefully, those can be revealed on a power trace and divulge the corresponding part of a secret scalar.

Combined with a lattice attack, it is then possible to recover a secret ECDSA signing key.

1 Introduction

Elliptic Curve Cryptography (ECC) is an ensemble of public-key cryptosystems that has become widely used for key-exchange or authentication. The growing interest makes ECC a target for side-channel attacks. Those rely on auxiliary data made available to an attacker with physical leakage of implementations. The most common is Simple Power Analysis (SPA) introduced by Kocher *et al.* [5] using the correlation between power consumption and the manipulated data to deduce the type of operation and eventually information on a secret key.

To prevent such attacks, several propositions have been given such as using algorithms with a regular behavior. For elliptic curves, it means a scalar multiplication algorithm that executes the same sequence of point doublings and point additions regardless of the secret bits manipulated. For example, this can be achieved with the introduction of dummy point additions [1].

Nonetheless, it was shown that many protection mechanisms might not prevent SPA. Indeed, the authors of [3] proposed to use special points with a coordinate equal to zero implying a lower power consumption due to low Hamming weight. With a chosen input, the special point appears during the calculation only under some assumption on secret bits which can be confirmed with the power trace. This attack is called Refined Power Analysis (RPA).

In SSTIC 2020 [8], we showed that the efficient P-256 curve implementation [4] (present in OpenSSL and its forks BoringSSL and LibreSSL) could be targeted with fault injections to detect the presence of dummy point additions. In this paper, we propose a different strategy based on RPA to attack this implementation, combined with a lattice attack to recover an ECDSA private key [6].

The outline is the following. In Sect. 2, we describe the model of the attack, present a simulation of power traces and the results. Sect. 3 explains why the implementation is vulnerable to RPA, and finally, we conclude in Sect. 4 with remarks on mitigations.

2 The attack

In this section, we present the target, model, and steps of the attack. Then we give the results of our simulation of power traces capture and ECDSA key recovery.

2.1 Target

The target is the implementation of the P-256 curve that has part of the code written in assembly [4] that we found present in:

- OpenSSL (introduced in version 1.0.2 for `x86_64`, and later for `x86`, `ARMv4`, `ARMv8`, `PPC64` and `SPARCv9`);
- BoringSSL (introduced in commit `1895493` (november 2015) for `x86_64`);
- LibreSSL (introduced in november 2016 in OpenBSD, but not present in the re-packaged version for portable use as of version 3.3.1).

The particularity of this implementation is that the point addition and big integer modular arithmetic are coded with efficient assembly code. For ECDSA signature generation, the scalar multiplication algorithm uses large precomputed tables to reduce considerably the number of point additions to execute. These choices make the implementation one of the fastest available.

This implementation is constant-time, and it is achieved with the introduction of dummy point additions when necessary. In [8], it is shown that those could be detected with fault injections. We introduce a non-invasive strategy in the following that also detects the presence of a dummy point addition.

2.2 Model and steps

The attack is carried out in two steps. The first consists of the capture of power consumption traces of many ECDSA signature generation with an unknown identical private key. Therefore physical access to the target device is necessary. The second step is the analysis of the traces to filter signatures to be used in the lattice attack. Thus it is necessary to collect the signatures and messages, alongside the public key (to match with private key candidates).

We are looking for traces for which the beginning of the scalar multiplication during a signature generation has a noticeable lower power consumption. This will indicate that a dummy point addition occurred and reveal that the secret randomly generated secret nonce has its lowest bits set to $0s$.

A summary of the attack:

1. Capture power consumption of an ECDSA signature generation;
2. Collect the signature and the corresponding signed message;
3. Keep the signature and message if there is a lower power consumption during the first point addition;
4. Repeat steps 1-3 until at least 37 signatures are collected;
5. Apply the lattice attack to attempt a recovery of the private key;
6. Go back to step 1 to add signatures in case the recovery failed.

Lattice attack. Nonce bias is a critical vulnerability in ECDSA. It can be rewritten as a problem of finding a short vector in a lattice, and efficient algorithms are available with the library `fplll` and its Python wrapper [2]. The stronger the bias, the fewer signatures we need for the attack to succeed. In our case, we identify signatures where the 7 least significant bits of the secret nonces are null, and the number of such signatures to recover a 256-bit private key is at least 37 in general.

2.3 Simulation

We experimented on OpenSSL version 1.1.1k to validate the model of the attack. The tools to reproduce the results of the simulation are available on GitHub.[1]

Power trace simulation. To simulate power traces, we used the TracerGrind tool of Side-Channel Marvels.[2] This tool is a Valgrind plugin that can record executed instructions, memory read (or written) of a running process.

To obtain a power trace, we used the tool to record the values read from memory during a signature generation with a command such as:

```
valgrind --tool=tracergrind --output=memread.trace --trace-instr=no
    --trace-memread=yes --trace-memwrite=no openssl dgst -sha256
    -sign privkey.pem -out sig.bin msg.txt
```

Listing 1. Command line to capture memory trace with the TracerGrind tool.

Then we applied the Hamming weight model to the values. An example of a simulated power trace is given in Fig. 1.

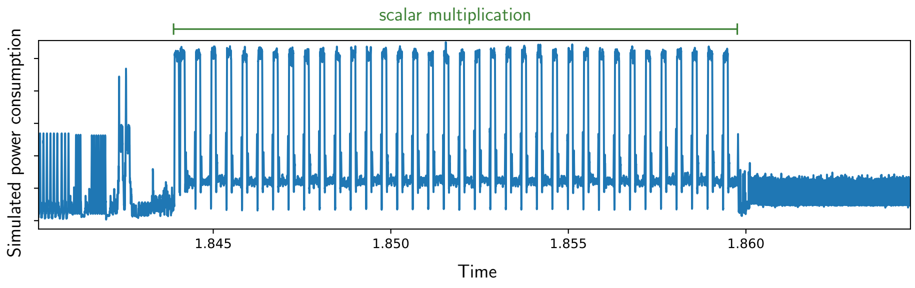

Fig. 1. Power trace simulation of a signature generation in OpenSSL with the efficient P-256 implementation in assembly.

Analysis. The simulation was run to get 6000 power traces. In this particular setup, the part related to the first point addition is easy to extract from the traces since it occurs at the same position. We give in Fig. 2 two traces representing the first 4 point additions of the signature generation.

1. https://github.com/orangecertcc/ecdummyrpa
2. https://github.com/SideChannelMarvels/Tracer

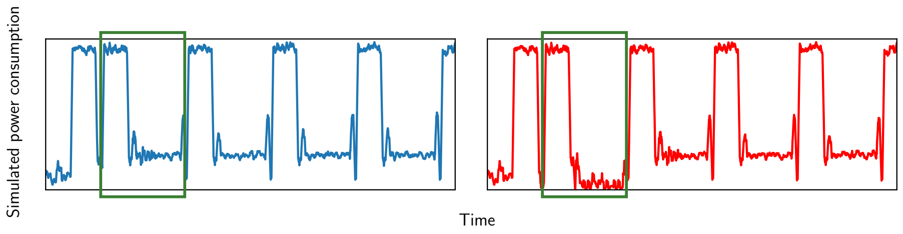

Fig. 2. Zoom on the beginning of power trace simulation of the scalar multiplication of two signature generation in OpenSSL with the efficient P-256 implementation in assembly (highlight: first point addition).

As can be seen, the first point addition on the second trace has a valley considerably lower than the others indicating a dummy point addition and a nonce that has its 7 least significant bits set to 0.

To distinguish the traces in two classes, we used the `GaussianMixture` class from the `Scikit-learn` machine learning Python library [7]. The tool was able to identify a cluster of 50 traces and another with the remaining traces. The ratio between the two cluster sizes is consistent with the expectation, since the probability that the 7 least significant bits of a nonce are simultaneously 0 is 1/128.

Private key recovery. We ran the lattice attack using the function `findkey` in the Python tool of our SSTIC 2020 paper.

```
key = findkey(secp256r1, pubkey, signatures, msb=False, 7)
```

Listing 2. Attempt to find the private key from a list of signatures.

The parameters of the function are

- `secp256r1`: the elliptic curve P-256 (predefined in the tool);
- `pubkey`: the public key point in the form (x, y);
- `signatures`: list of filtered signatures in the form (m, r, s) where m is the hashed message, and the couple (r, s) the signatures;
- `msb=False` and `l=7` to indicate that the 7 least significant bits of the nonces are $0s$.

We note that the first 38 signatures out of the 50 were enough to recover the private key.

3 Details of P-256 implementation

In this section, we describe how the dummy point addition is managed in the scalar multiplication algorithm and why the attack works.

3.1 Scalar multiplication algorithm

The implementation makes use of large precomputed tables to reduce the whole calculation as only 36 elliptic curve point additions (those can be counted in Fig. 1).

The secret scalar k of size at most 256 bits is rewritten as 37 windows K_0, . . . , K_{36}, each calculated from consecutive bits of k. An accumulator is initialized with a precomputed point P_0 selected in a table from the value K_0. Then, for each subsequent K_i a point addition occurs between the accumulator and a point P_i. Therefore the whole scalar multiplication is calculated as only 36 point additions:

$$(\cdots((P_0 + P_1) + P_2) + \cdots) + P_{36}.$$

The formulas for the point addition[3] have exceptions, and those are handled with dummy point additions to make the execution constant-time. Our interest is how the dummy point addition is done.

3.2 Dummy point addition

The point addition is executed as presented in Algorithm 1. It uses two different point representations:

— The first entry is the accumulator represented by three coordinates X_1, Y_1 and Z_1; when Z_1 is null, it is the infinity point (the null point of the elliptic curve), otherwise $(X_1/{Z_1}^2, Y_1/{Z_1}^3)$ is the affine representation;

— The second entry is a point from the precomputed table given by two coordinates x_2 and y_2 which is the affine representation; in this situation, the infinity point is represented by $(0, 0)$ (not a valid affine point).

When K_0 is null (this happens only when the 7 least significant bits of the scalar are 0s), the accumulator is initialized with $(X_1, Y_1, Z_1) = (0, 0, 0)$, then the boolean `in1infty` is true and all calculations are discarded in line 22. However, almost all operands in the point addition are null (see highlights in Algorithm 1). Each of these operations is composed of dozens of instructions to perform big integer modular arithmetic with machine words of zero Hamming weight for most of them.

The goal of Refined Power Analysis is to find such computation on a trace and explains why we observe lower valleys on the simulated power traces in the small cluster.

3. In the function `ecp_nistz256_point_add_affine`.

```
Require: $P_1 = (X_1, Y_1, Z_1)$,
    $P_2 = (x_2, y_2)$
Ensure: $P_1 + P_2 = (X_3, Y_3, Z_3)$
 1: in1infty ← $(Z_1 == 0)$
 2: in2infty ← $(x_2, y_2) == (0, 0)$
 3: $\mathbf{t_0} \leftarrow \mathbf{Z_1}^2$
 4: $\mathbf{t_1} \leftarrow x_2 \cdot \mathbf{t_0}$
 5: $\mathbf{t_2} \leftarrow \mathbf{t_1} - \mathbf{X_1}$
 6: $\mathbf{t_3} \leftarrow \mathbf{t_0} \cdot \mathbf{Z_1}$
 7: $\mathbf{Z_3} \leftarrow \mathbf{t_2} \cdot \mathbf{Z_1}$
 8: $\mathbf{t_3} \leftarrow \mathbf{t_3} \cdot y_2$
 9: $\mathbf{t_4} \leftarrow \mathbf{t_3} - \mathbf{Y_1}$
10: $\mathbf{t_5} \leftarrow \mathbf{t_2}^2$
11: $\mathbf{t_6} \leftarrow \mathbf{t_4}^2$
12: $\mathbf{t_7} \leftarrow \mathbf{t_5} \cdot \mathbf{t_2}$
13: $\mathbf{t_1} \leftarrow \mathbf{X_1} \cdot \mathbf{t_5}$
14: $\mathbf{t_5} \leftarrow 2 \cdot \mathbf{t_1}$
15: $\mathbf{X_3} \leftarrow \mathbf{t_6} - \mathbf{t_5}$
16: $\mathbf{X_3} \leftarrow \mathbf{X_3} - \mathbf{t_7}$
17: $\mathbf{t_2} \leftarrow \mathbf{t_1} - \mathbf{X_3}$
18: $\mathbf{t_3} \leftarrow \mathbf{Y_1} \cdot \mathbf{t_7}$
19: $\mathbf{t_2} \leftarrow \mathbf{t_2} \cdot \mathbf{t_4}$
20: $\mathbf{Y_3} \leftarrow \mathbf{t_2} - \mathbf{t_3}$
21: if in1infty then
22:     $(X_3, Y_3, Z_3) \leftarrow (x_2, y_2, 1)$
23: if in2infty then
24:     $(X_3, Y_3, Z_3) \leftarrow (X_1, Y_1, Z_1)$
25: return $(X_3, Y_3, Z_3)$
```

Algorithm 1. Mixed point addition with projective Jacobian coordinates (highlights: zero values when $P_1 = (0, 0, 0)$).

Remarks. When K_1 is null, it means $(x_2, y_2) = (0, 0)$, and the point addition is also dummy. However, only four operations involve a zero operand (two multiplications and two subtractions) out of the 18 big integer arithmetic operations. So we can expect that this case is harder to distinguish from the normal case and cannot be misinterpreted as the case $K_0 = 0$.

A final remark is that for the accumulator to be $(0, 0, 0)$ during the processing of window K_i, it would be necessary that all the previous windows be null which happens for very few scalars: K_0 is null if the 7 least significant bits are 0s, but for K_1 to also be null would need 7 more bits set to 0s, and so on. Hence why we focus only on the first point addition.

4 Mitigations and conclusion

In this paper, we have shown that the efficient implementation of the popular elliptic curve P-256 in OpenSSL and its forks is vulnerable to Refined Power Analysis (RPA), a particular power analysis attack that relies on the correlation between a power trace and zero operands.

We have simulated power traces for a few thousand ECDSA signature generations under an identical private key, and have shown that RPA reveals when the 7 least significant bits of the secret nonce are 0s due to the use of a dummy point addition. A lattice attack is successful in these conditions to reconstruct the private key.

One possibility to mitigate the attack would be to perform the dummy point addition with nonzero coordinates. The proposition in our SSTIC 2020 paper also mitigates this attack as it avoids the introduction of dummy point additions.

References

1. Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 292–302, 1999.
2. The FPyLLL development team. fpylll, lattice reduction for Python, Version: 0.5.5. Available at `https://github.com/fplll/fplll`, 2021.
3. Louis Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 199–210. Springer, 2003.
4. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptogr. Eng.*, 5(2):141–151, 2015.
5. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
6. Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.
7. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
8. Andy Russon. Exploiting dummy codes in Elliptic Curve Cryptography implementations. *SSTIC*, 2020.

A Appendix

Algorithm 2 presents how a window of consecutive bits of the scalar k is encoded as a relative integer (one sign bit and the absolute value), and Algorithm 3 is the complete scalar multiplication algorithm.

Every window is calculated from 8 consecutive bits of the scalar k at most (including one bit from the previous window), and is encoded as a null window in two cases: the bits are all $0s$ or all $1s$. For the first window, since there is no previous window, a bit 0 is added, so it is encoded as a null window if and only if the 7 least significant bits are $0s$.

```
**Require:** $d$ with $0 \leq d < 256$
**Ensure:** encoding of $d$
1: **if** $d \geq 128$ **then**
2:     $d \leftarrow 255 - d$
3:     $\mathrm{s} \leftarrow 1$
4: **else**
5:     $\mathrm{s} \leftarrow 0$
   **return** s, $\lfloor (d+1)/2 \rfloor$
```

Algorithm 2. Window encoding for scalar multiplication algorithm in the efficient P-256 implementation.

```
**Require:** $k = (k_{255}, \ldots, k_0)_2$, $P$
**Ensure:** $[k]P$
   **Precomputation phase (offline)**
1: **for** $i \leftarrow 0$ **to** 36 **do**
2:     **for** $j \leftarrow 0$ **to** 64 **do**
3:         $\mathrm{Tab}[i][j] = [j2^{7i}]P$
   **Evaluation phase**
4: $s_0, K_0 \leftarrow \mathrm{Encoding}((k_6, \ldots, k_0, 0)_2)$
5: $R \leftarrow (-1)^{s_0}\mathrm{Tab}[0][K_0]$
6: **for** $i \leftarrow 1$ **to** 36 **do**
7:     $s_i, K_i \leftarrow \mathrm{Encoding}((k_{7i+6}, \ldots, k_{7i}, k_{7i-1})_2)$
8:     $R \leftarrow R + (-1)^{s_i}\mathrm{Tab}[i][K_i]$
   **return** $R$
```

Algorithm 3. Single scalar multiplication with the generator of the efficient P-256 implementation.