Exploiting dummy codes in Elliptic Curve Cryptography implementations

Andy Russon
andy.russon@orange.com

Orange

Abstract. With the growing interest of Elliptic Curve Cryptography, in particular in the context of IoT devices, security about both passive and active attacks are relevant. The use of dummy operations is one countermeasure to protect an implementation against some passive side-channel attacks. However, a specific case of fault attacks known as C safe-errors can reveal those dummy operations, and as a consequence the secret data related to it. Even if only a few bits are revealed, this can be enough to break the public-key signature scheme ECDSA.

In this paper, we show how to carry out such an attack on several implementations in libraries such as OpenSSL and its forks. We give an example with the assembly optimized implementation of the P-256 curve, and scripts to help reproduce the attack.

1 Introduction

Elliptic Curve Cryptography is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. One of its advantages against RSA based cryptography is the small size of its parameters, keys, and also signatures for the Elliptic Curve Digital Signature Agreement (ECDSA). As such, it is convenient for improving the efficiency of communications, and there has been a growing interest in implementing it in low-cost embedding systems such as smart cards, but can also be found everywhere, such as IoT devices, PCs, servers, as it is used in mutual authentication and key derivation protocols such as TLS, SSH, Bitcoin or Signal.

An elliptic curve can be considered, roughly, as a set of points that have an addition operation with a null point like zero. A private key k is an integer and its matching public key is the point $kP = P + P + \ldots + P$ where P is a point of an elliptic curve. This latter operation, called scalar multiplication, is critical and must be properly protected. If the implementation of this operation is too naive, an attacker can find the private key through passive attacks, consisting of obtaining traces of execution by analyzing timing, power consumption, electromagnetic

emanations, etc. For example, if the implementation of the *double-and-add* algorithm is not protected, a single trace of power consumption during its execution can be used by the attacker to distinguish at each step if the bit of the secret key is 0 (one point doubling) or 1 (one point doubling and one point addition).

There are various countermeasures against these attacks. One of them is to use an algorithm that executes the same operations for each bit of the key, such as the *double-and-add-always* algorithm. It performs a point doubling and a point addition for each secret bit by introducing a dummy point addition when the key bit is 0 to achieve the regularity.

However, these routines are not necessarily protected against active attacks, such as the use of fault injections to disrupt the execution of a cryptographic calculation (with various means such as clock glitches, voltage spikes, laser injection, electromagnetic pulses, etc). Fault attacks can be exploited in several ways. Usually, the fault injection causes the cryptographic algorithm attacked to output an erroneous result, which is then used to deduce the secret key. There is another class of fault attacks, called C safe-errors [16], consisting only of looking if the fault injection had an effect or not on the output. Indeed, a fault injection is induced on an alleged dummy operation, and the result is correct only if the operation was actually dummy. Consequently, secret bits related to this operation can be deduced. For instance, a fault on the point addition in the *double-and-add-always* algorithm reveals that the secret bit is 0 if the output is correct, or 1 otherwise.

In ECDSA the value k is an ephemeral key called nonce, and is unique to each signature. It has been shown that partial knowledge of this value for several signatures is sufficient to retrieve the private key of the signer [12]. C safe-error attacks can be used to recover a few bits per nonce to attack ECDSA. This has been done in [2] where it is applied on the countermeasures of [3, 15] that both introduce dummy operations to mask the difference between a point doubling and a point addition. In [1], it is shown that such an attack can also be applied when the *Montgomery ladder* algorithm [11] is used for scalar multiplication, since the last point additions can become dummies when the least significant bits of k are $0s$.

In this paper, we carry out C safe-error attacks on ECDSA signatures, but applied to implementations using a dummy point addition in windowing methods for the scalar multiplication. These can be seen as a generalization of the *double-and-add-always* algorithm, and can be found in several cryptographic libraries. In particular, we show that the optimized implementation of the P-256 curve with part of the code written

in assembly is vulnerable. It is present in OpenSSL (since version 1.0.2), and its forks LibreSSL and BoringSSL. For the latter, we also show that the default algorithm used with other elliptic curves is vulnerable. As a consequence, other libraries that rely on them for their cryptographic operations are vulnerable, such as Fizz (Facebook), S2n (Amazon), and Erlang/OTP. Given the nature of the algorithms and the model of the attack, it is practical. A proof of concept with its source code is made available.

The outline of the paper is as follows. In section 2 we present the vulnerable implementations and the attack, followed by an example using our tool. Then, section 3 describes the algorithms used in the vulnerable implementations, the reasons the attack works, and characteristics of the P-256 implementation we used to illustrate the attack. We finally propose mitigations in section 4, and the appendices contain technical details, including a proof of one of the proposed mitigations.

2 The attack in practice

In this section, we first give a list of vulnerable implementations in cryptographic libraries, then the model and the steps of the attack. We end with a presentation of how to use our tool that performs the mathematical aspects of the attack, with practical results on OpenSSL.

2.1 Vulnerable implementations in libraries

Several libraries rely on the introduction of dummy point additions to make the scalar multiplication constant-time and with a regular behavior.

This is the case of a specific implementation of the P-256 curve, with part of the code written in assembly for software optimization [5]. It is present in the following libraries:

— OpenSSL: introduced in version 1.0.2 for `x86_64`, and later for `x86`, `ARMv4`, `ARMv8`, `PPC64` and `SPARCv9`. It is the default implementation for this curve as long as the option `no-asm` (that disables assembly optimization) is not specified at compilation [9].
— BoringSSL: introduced in commit `1895493` (november 2015), but only for `x86_64` [7].
— LibreSSL: introduced in november 2016 in OpenBSD, but is not present in the re-packaged version for portable use (as of version 3.0.2) [8].

Additionally, the default algorithm in BoringSSL uses dummy point additions, and it concerns the curves P-384, P-521 (and P-224 depending of the compilation options).

In the previously cited implementations the dummy point addition is related to consecutive bits of the secret scalar, which is important in the attack. There are other cases where the algorithm uses dummy point additions, but the corresponding bits are not consecutive, and this would require several fault injections per execution as was done in [2]. This is outside the scope of this article.

We note that the current threat model of OpenSSL [1] does not include protection against physical attacks, in particular physical fault injection. Thus, attacks such as the one presented in this article are not considered vulnerabilities for OpenSSL.

2.2 Model and steps of the attack

In order to carry out the attack, an attacker must be able to make a fault on an instruction in a specifc set of potential dummy operations during an ECDSA signature calculation. He must be able to repeat this attack several times when a same private key is used. Finally, the result values must be retrieved by the attacker.

The location of the fault is important, but does not need to be completely precise, since the potential dummy operations are composed of many instructions that perform calculations on large integers. Moreover, the exact nature of the fault is irrelevant, therefore any random transient fault inducing a computational error will work.

The public key, signatures and signed messages are public, and we assume these can be acquired by the attacker.

We give below the main steps of the attack:

1. Make a fault on one of the alleged dummy instructions, during the generation of an ECDSA signature;
2. Collect the signature and the corresponding signed message, then check it with the public key of the signer and keep it if it is valid, to be used in step 4;
3. Repeat steps 1-2 until the number of valid signatures reaches a minimal value (a few dozens, and depends on the scalar multiplication algorithm's characteristic, see section 3);

1. `https://www.openssl.org/policies/secpolicy.html`

4. Use our tool presented in 2.3 to attempt a recovery of the private key from the valid signatures collected in step 2. If the private key was not recovered, go to step 1 to add valid signatures.

As said above, it is important that the fault is effectively induced in one of the targeted instructions that does not impact the computation. A fault made on an instruction other than these can have an impact on step 4 of the attack. Indeed, the algorithm used to discover the secret key would fail with a wrong signature taken into account.

Example of scenario. A possible scenario for this attack would be a device (IoT, smartphone, etc) that performs ECDSA signatures using the optimized implementation of the P-256 curve on OpenSSL, with a private key physically hard-coded. The targeted instructions for fault injection are those that compute the x-coordinate of the output in the last point addition performed by the scalar multiplication algorithm as is explained in section 3, and in particular in 3.2 for this implementation.

2.3 Simulation of the attack and tools

We provide a Python script [2] that contains the mathematical tools to perform the attack, and then we give an application to OpenSSL followed by results.

Tools for the attack. The script `ec.py` is written in Python 3 and its only requirement is to install the external dependency `fpylll`.

The main tool is a function to perform step 4 of the attack to recover the private key. The command is `findkey(curve, pubkey_point, signatures, msb, l)` that returns the value of the private key, or -1 otherwise. The arguments are:

— `curve`: predefined values are `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1`, and `secp521r1`. These objects are instances of a Python class `Curve` implemented in the script in order to perform elliptic curve operations, and is necessary to check if one of the candidates for the private key matches the public key. Other elliptic curves can be used by giving their explicit parameters.

— `pubkey_point`: the public key point of the signer, given as two integers representing its coordinates.

2. `https://github.com/orangecertcc/ecdummy`

— `signatures`: list of valid signatures, where each signature is given as three integers corresponding to the hash of the signed message, and the two components of the signature.
— `msb` and `l`: a boolean and an integer whose values depend on the characteristic of the scalar multiplication algorithm (to indicate if the `l` most significant bits (`msb=True`) or `l` least significant bits (`msb=False`) of the nonces used to generate the signatures in the list are set to 0, see 3.1).

Finally, we also provide a function to check if a signature is valid, with the command `check_signature(curve, pubkey_point, signature)`.

Application to OpenSSL. With the scenario given in 2.2, the attacker retrieves the public key point of the signer stored in the file `publickey.pem`, and store it as the variable `pubkey_point` in listing 1.

```
text = open('publickey.pem', 'r').read().split('\n')
pubkey_bytes = base64.b64decode(text[1] + text[2])[27:]
pubkey_point = int.from_bytes(pubkey_bytes[:32], 'big'), int.
    from_bytes(pubkey_bytes[32:], 'big')
```

Listing 1. Converting a public key of the curve P-256 as two integers from a PEM file.

Then, for each signature generation the attacker makes a fault during the execution in one of the determined instructions, and retrieves the signature and the signed message in the files `sig.bin` and `message.txt`. The signature is checked in listing 2, and the valid signatures are kept in the list `valid_signatures`.

```
m = int.from_bytes(sha256(open('message.txt', 'rb').read()).digest()
    , 'big')
raw_sig = open('sig.bin', 'rb').read()
rlen = raw[3]
r = int.from_bytes(raw[4:4+rlen], 'big')
s = int.from_bytes(raw[6+rlen:], 'big')
valid = check_signatures(secp256r1, pubkey_point, (m,r,s))
if valid:
    valid_signatures.append((m,r,s))
```

Listing 2. Converting the signature and signed message as integers, and verification with the public key point.

Finally, in listing 3, an attempt to recover the private key can be made. According to the implementation characteristics given in 3.2 and to table 2, the parameters `msb` and `l` in the function `findkey` must be set to `True` and 5, and the number of valid signatures should be greater than 52.

```
key = findkey(secp256r1, pubkey_point, valid_signatures, True, 5)
if key != -1:
    print('The private key is {:064x}'.format(key))
```

Listing 3. Attempt to find the private key from a list of valid signatures.

Results on OpenSSL. The previous example has been tested. First, a script `openssl_p256_attack_simulation.py` is used to run an OpenSSL binary to simulate the fault injection during the execution. This is done by modifying systematically the output of one of the instructions that could be dummy in the last point addition in the code of OpenSSL. Then, a script `p256_privatekey_recovery.py` uses the tools of `ec.py` to verify the signatures and recover the private key.

We give in listing 4 the output of one simulation, where the private key was recovered from 54 valid signatures.

```
$ ./openssl_p256_attack_simulation.py ./openssl_altered privkey.pem
    SSTIC 2200
Signatures and messages will be stored in the directory SSTIC
Generating 2200 signatures with fault in last point addition...
  ... done
$
$ ./p256_privatekey_recovery.py publickey.pem SSTIC
Nb valid signatures:  1 /   51
Nb valid signatures:  2 /   70
Nb valid signatures:  3 /   91
(...)
Nb valid signatures: 51 / 1836
Nb valid signatures: 52 / 1838
Recovering the key, attempt 1 with 52 signatures...
Nb valid signatures: 53 / 1880
Recovering the key, attempt 2 with 53 signatures...
Nb valid signatures: 54 / 1885
Recovering the key, attempt 3 with 54 signatures...
SUCCESS!
The private key is:
    ba2c97646898ee0cf8ab9673eb2656de76c2ef674454b3609323f767f9c8759d
Nb signatures valid: 54
Nb signatures total: 1885
```

Listing 4. Output of our running example on the altered OpenSSL binary.

To give an idea of the number of valid signatures needed in average, and the number of signature generations attacked, we ran 100 tests and give the results in table 1. In the majority of cases, the last point addition is not dummy, so the number of signature generations to attack is far greater.

	min	average	max
Number of valid signatures	52	55	58
Number of signatures attacked	1274	1764	2382

Table 1. Number of valid and total number of signatures attacked to recover the private key out of 100 tests on the assembly optimized implementation of the P-256 curve in OpenSSL.

3 Technical details

In this section, we first give a general description of the algorithms used in the vulnerable implementations, and the reasons why the attack works. Then, we present the characteristics of the assembly optimized implementation of the P-256 curve based on the code from OpenSSL.

3.1 Why the attack works

We give a description of the windowing methods for scalar multiplication, where the fault has to be injected during the execution, and why the attack works.

Scalar multiplication with windowing methods. Windowing methods process several bits of the scalar at a time. Those are used when storage is available in order to have precomputed values and decrease the number of point additions. They consist of three phases:

1. Precomputation: precomputed points are stored in a table (this phase can be offline);
2. Encoding: the scalar k is split into windows d_0, d_1, ..., where each d_i comes from several bits of the scalar;
3. Evaluation: the core of the computation of kP: at each iteration of the loop of the algorithm, a point addition with a precomputed point that depends on a value d_i occurs.

The important parts for the attack are the encoding phase, and the addition with the precomputed point in the evaluation phase. We target algorithms that meet the following two conditions:

— the values d_i are computed from consecutive bits of the scalar and can be equal to zero;
— the point addition in the loop is dummy when either of the points is the null point $\mathcal{O}$ of the curve.

The last case is the consequence that commonly used point addition formulas do not handle the null point $\mathcal{O}$ (they are said to be incomplete). Therefore a dummy point addition is introduced instead, as in the *double-and-add-always* algorithm. This occurs when a value d_i is equal to zero, hence the first condition.

We give in algorithm 1 a 2^w-ary windowing method [4, section 2.2] as an example where each d_i is composed of w consecutive bits of the scalar.

Require: $k = (k_{t-1}, \ldots, k_0)$, P, w
Ensure: kP

Precomputation phase
1: **for** $i \leftarrow 0$ **to** $2^w - 1$ **do**
2: $\quad \text{Tab}[i] \leftarrow iP$

Encoding phase
3: $m \leftarrow \lceil t/w \rceil$
4: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
5: $\quad d_i \leftarrow (k_{iw+(w-1)}, \ldots, k_{iw+1}, k_{iw})_2$

Evaluation phase
6: $R \leftarrow \text{Tab}[d_{m-1}]$
7: **for** $i \leftarrow m - 2$ **down to** 0 **do**
8: $\quad R \leftarrow 2^w R$
9: $\quad R \leftarrow R + \text{Tab}[d_i]$
return R

Algorithm 1. 2^w-ary windowing scalar multiplication algorithm.

Target of the fault injection. Our C safe-error attack consists of targeting the last point addition that occurs in the evaluation phase of the algorithm. It is important to know what are the corresponding bits of the scalar k, which are the number l of bits, and if they are related to the most or to the least significant bits. For instance, in algorithm 1, the last addition corresponds to the w least significant bits of the scalar. These two characteristics are needed for step 4 of the attack described in section 2.

We note in particular that the instructions to target in the last point addition should be related to the calculation of the x-coordinate of the output, since only this coordinate is used for the generation of an ECDSA signature (see appendix A). Then, a valid signature reveals that it was in fact a dummy point addition.

Why it reveals knowledge of the nonce. To understand what is obtained about the nonce from a valid signature, we have to look back at the algorithm in the last iteration of the loop in the evaluation phase. The dummy point addition means one of the entries is $\mathcal{O}$. The active point R depends on all the windows except the last one, and can be $\mathcal{O}$ if all of them are null. On the other hand, the precomputed point depends only on one window, therefore it is much more likely that $\mathcal{O}$ is this point, from which we deduce the value of the l bits of the nonce corresponding to this window.

Keeping only the valid signatures and the corresponding messages, then we can apply the technique described in [12] and given in appendix C to recover the private key since we know that the valid signatures give us a partial knowledge of the nonces.

Table 2 gives an estimate of the minimum number of valid signatures needed based on experiments for several elliptic curve sizes. The number of total signatures can be estimated by multiplying with 2^l.

Elliptic curve size	224 bits				256 bits				384 bits		
l	4	5	6	7	4	5	6	7	5	6	7
Minium number of valid signatures	56	45	37	31	65	52	43	36	91	65	56

Table 2. Estimation of the minimum number of valid signatures needed where l is the number of bits known from the nonce in each signature.

3.2 The assembly optimized implementation of the P-256 curve in OpenSSL

We present the scalar multiplication algorithm and how the point addition is handled in this implementation, to show how it relates to the description in 3.1.

Scalar multiplication algorithm. The scalar multiplication used in ECDSA signature generation for this implementation is a variant of the windowing method presented in 3.1. We give in algorithms 2 and 3 respectively the encoding of each window, and the scalar multiplication. The important elements to notice for the attack is that the window that corresponds to the last addition is null only if the 5 most significant bits

of the scalar are 0s. Therefore, the values `msb` and `l` in our tool `findkey` must be set to `True` and `5`.

Require: d with $0 \leq d < 2^8$
Ensure: encoding of d
1: **if** $d \geq 2^7$ **then**
2: $d \leftarrow 2^8 - 1 - d$
3: s $\leftarrow 1$
4: **else**
5: s $\leftarrow 0$
return s, $\lfloor (d+1)/2 \rfloor$

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

Require: $k = (k_{255}, \ldots, k_0)_2$, P
Ensure: kG

Precomputation phase (offline)
1: **for** $i \leftarrow 0$ **to** 36 **do**
2: **for** $j \leftarrow 0$ **to** 64 **do**
3: Tab$[i][j] = j2^{7i}P$

Encoding phase
4: **for** $i \leftarrow 0$ **to** 36 **do**
5: $s_i, d_i \leftarrow \text{Encoding}(k_{7i+6}, \ldots, k_{7i}, k_{7i-1})$

Evaluation phase
6: $R \leftarrow (-1)^{s_0}\text{Tab}[0][d_0]$
7: **for** $i \leftarrow 1$ **to** 36 **do**
8: $R \leftarrow R + (-1)^{s_i}\text{Tab}[i][d_i]$
return R

Algorithm 3. Single scalar multiplication with the generator in P-256 implementation in OpenSSL.

To provide the rationale, we first notice that the processing order of the windows makes the last addition related to the most significant bits. Second, we give some remarks about the encoding phase:

— Each window is computed from 8 consecutive bits of the scalar (including a common bit with a previous window, or bit 0 for the first window);
— The window is null in two cases: when the 8 selected bits are all 0s or all 1s. Indeed, in algorithm 2 if $d \geq 2^7$, then the encoding will be $\lfloor (2^8 - d)/2 \rfloor$ which is zero only if $d = 255 = (11111111)_2$,

and if $d < 2^7$, the encoding is $\lfloor (d+1)/2 \rfloor$ which is zero only if $d = (00000000)_2$.

The last window is composed of only 5 bits of the scalar and is padded with 0 bits. According to the previous remark, it is encoded as zero only if those 5 bits are $0s$.

Then, this implementation meets the first condition given in 3.1. In particular, we have $l = 5$, and a 256-bit curve.

Point addition. The point addition used in line 8 of algorithm 3 is implemented in the function `ecp_nistz256_point_add_affine`. We give in algorithm 4 the arithmetic instructions of the formulas and in listing 5 part of the `x86_64` assembly code (instructions are similar for other architectures).

Require: $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2)$, $P_1 \neq \mathcal{O}$, $P_2 \neq \mathcal{O}$, $P_1 \neq P_2$
Ensure: $P_1 + P_2 = (x_3, y_3, z_3)$

1: $\mathbf{t_0} \leftarrow \mathbf{z_1^2}$
2: $\mathbf{t_1} \leftarrow \mathbf{x_2} \cdot \mathbf{t_0}$
3: $\mathbf{t_2} \leftarrow \mathbf{t_1} - \mathbf{x_1}$
4: $\mathbf{t_3} \leftarrow \mathbf{t_0} \cdot \mathbf{z_1}$
5: $\mathbf{z_3} \leftarrow \mathbf{t_2} \cdot \mathbf{z_1}$
6: $\mathbf{t_3} \leftarrow \mathbf{t_3} \cdot \mathbf{y_2}$
7: $\mathbf{t_4} \leftarrow \mathbf{t_3} - \mathbf{y_1}$
8: $\mathbf{t_5} \leftarrow \mathbf{t_2^2}$
9: $\mathbf{t_6} \leftarrow \mathbf{t_4^2}$
10: $\mathbf{t_7} \leftarrow \mathbf{t_5} \cdot \mathbf{t_2}$
11: $\mathbf{t_1} \leftarrow \mathbf{x_1} \cdot \mathbf{t_5}$
12: $\mathbf{t_5} \leftarrow \mathbf{2} \cdot \mathbf{t_1}$
13: $\mathbf{x_3} \leftarrow \mathbf{t_6} - \mathbf{t_5}$
14: $\mathbf{x_3} \leftarrow \mathbf{x_3} - \mathbf{t_7}$
15: $t_2 \leftarrow t_1 - x_3$
16: $t_3 \leftarrow y_1 \cdot t_7$
17: $t_2 \leftarrow t_2 \cdot t_4$
18: $y_3 \leftarrow t_2 - t_3$

Algorithm 4. Arithmetic instructions of point addition, in the field of the curve, in line 8 of algorithm 3 (highlights: instructions to target for fault injection in the last addition).

The entries are two points P_1 and P_2, and those formulas are not compatible with the point $\mathcal{O}$. Two values are created to serve as booleans to indicate if one of the two points is $\mathcal{O}$ (from line 5 to line 17 of listing 5). The instructions of the point addition formulas are executed regardless of these values. Then, if $P_1 = \mathcal{O}$ (respectively $P_2 = \mathcal{O}$), the coordinates of the resulting point are replaced with those of P_2 (resp. P_1). As a consequence the previous calculations are ignored.

Therefore, if one of the inputs is $\mathcal{O}$, the point addition is dummy. The second condition given in 3.1 is met, and makes the implementation vulnerable to the attack.

```
leaq    64-0(%rsi),%rsi
leaq    32(%rsp),%rdi
call    __ecp_nistz256_sqr_montq

pcmpeqq %xmm4,%xmm5
pshufd  $0xb1,%xmm3,%xmm4
movq    0(%rbx),%rax
movq    %r12,%r9
por     %xmm3,%xmm4
pshufd  $0,%xmm5,%xmm5
pshufd  $0x1e,%xmm4,%xmm3
movq    %r13,%r10
por     %xmm3,%xmm4
pxor    %xmm3,%xmm3
movq    %r14,%r11
pcmpeqd %xmm3,%xmm4
pshufd  $0,%xmm4,%xmm4

leaq    32-0(%rsp),%rsi
movq    %r15,%r12
leaq    0(%rsp),%rdi
call    __ecp_nistz256_mul_montq
leaq    320(%rsp),%rbx
leaq    64(%rsp),%rdi
call    __ecp_nistz256_sub_fromq
```

Listing 5. Excerpt from the x86_64 assembly code generated by the perl script `https://github.com/openssl/openssl/blob/master/crypto/ec/asm/ecp_nistz256-x86_64.pl` in OpenSSL 1.1.1d (first three instructions of algorithm 4).

4 Proposal of mitigations

We propose in this section mitigations against the attack. They consist mainly of using a different encoding of the scalar to avoid the introduction of dummy point additions.

The first assumption made in 3.1 is the use of an encoding that can generate null windows so the point $\mathcal{O}$ can appear in the addition. Encodings that avoid null windows are given in [6, 10, 13], but we warn the reader that for some of those propositions, the two points in the last addition may be equal in rare cases and could be incompatible with the formulas. The use of complete formulas from [14] for the last addition only can take care of it.

A particular case is the odd-signed-comb method implemented in Mbed TLS (as of version 2.16.5), based on a modification of [6]. It avoids all special cases of the point addition formulas if the curve cardinality q satisfies $q \equiv 1 \bmod 4$ (contrary to what is claimed in the source code of this

library, the doubling case is possible for curves satisfying $q \equiv 3 \bmod 4$), and we give a proof of it in appendix D.

This algorithm is well suited when storage of precomputed values is possible, and could replace the scalar multiplication algorithms used for key and signature generation in OpenSSL and its forks for the curves P-224, P-256 and P-521. It retains a similar efficiency, and without the need of bitwise masking technique or branch conditions to manage the special cases of the point addition formulas.

Remark. The second assumption in 3.1 is the use of point addition formulas incompatible with the point $\mathcal{O}$. It might be tempting to use complete formulas [14] to managed this special case. However, it can be shown that there are still dummy instructions when one of the inputs is $\mathcal{O}$.

5 Conclusion

In this paper, we have shown that implementations of Elliptic Curve Cryptography in some libraries such as OpenSSL, BoringSSL or LibreSSSL, are vulnerable to C safe-error attacks. As a result, several bits of secret nonces during ECDSA signature generations can be obtained, leading to the recovery of the private key.

We proposed mitigations that can prevent this attack while retaining other characteristics of the original algorithms and formulas, such as efficiency and protection against passive attacks.

A ECDSA

In this appendix, we recall briefly how a signature is generated in ECDSA. Given a base point G of prime order q on an elliptic curve, a private key α in $[1, q-1]$ and a hashing function H (which outputs a t-bit integer), signing a message m is done by generating a nonce $k \in [1, q-1]$ and computing

$$\begin{cases} r = x(kG) & \bmod q, \\ s = k^{-1}(\mathrm{H}(m) + \alpha r) & \bmod q. \end{cases}$$

The pair (r, s) forms the signature of the message m. The verification process consists in computing the point $P = \mathrm{H}(m)s^{-1}G + rs^{-1}Q$ where $Q = \alpha G$ is the public key of the signer. Then if $x(P) = r \bmod q$, the signature is valid.

B Mixed point addition

We give more details on the point addition formulas notably used in the implementation given in section 3.2.

B.1 Definition

A mixed point addition is when P_1 and P_2 have a different representation. We present the case when P_1 is in projective Jacobian coordinates, represented by x_1, y_1 and z_1 whose affine coordinates are ${x_1}/{{z_1}^2}$ and ${y_1}/{{z_1}^3}$, and the point P_2 by x_2 and y_2 which are its affine coordinates.

The resulting point of the addition is given in Jacobian coordinates by the formulas

$$\begin{cases} x_3 = (y_2 z_1^3 - y_1)^2 - (x_2 z_1^2 - x_1)^2 (x_2 z_1^2 + x_1), \\ y_3 = (y_2 z_1^3 - y_1)(x_1 (x_2 z_1^2 - x_1)^2 - x_3) - y_1 (x_2 z_1^2 - x_1)^2, \\ z_3 = (x_2 z_1^2 - x_1) z_1. \end{cases}$$

Those formulas are not compatible in these situations:
— $P_1 = P_2$: doubling formulas must be used instead;
— P_1 or P_2 is the point $\mathcal{O}$, and in this case the shortcut $P_1 + \mathcal{O} = P_1$ is used.

B.2 Handling of the special cases by OpenSSL

In the specific implementation of the curve P-256 described in section 3.2 used in particular for signature generation, the special cases are managed as follows:
— $P_1 = P_2$: this case is not managed, but it cannot happen;
— $P_1 = \mathcal{O}$: in this case $\mathcal{O}$ is represented by having $z_1 = 0$, and a bitwise masking technique replaces the resulting point with P_2;
— $P_2 = \mathcal{O}$: in this case $\mathcal{O}$ is represented by having $x_2 = y_2 = 0$, and a bitwise masking technique replaces the resulting point with P_1.

C Lattice techniques

In this appendix, we explain the technique in [12] that recovers the private key from ECDSA signatures with partial knowledge of the nonces.

C.1 Description

We note $\lfloor \cdot \rfloor_q$ the reduction modulo q in the range $[0, q-1]$ and $|\cdot|_q$ the absolute value of the reduction modulo q in the range $[-q/2, q/2]$.

Suppose we have the following system of linear equations in variables α, x_i for $1 \leq i \leq n$:

$$a_i x_i + b_i \alpha \equiv c_i \mod q.$$

With n equations and $n+1$ unknowns, we cannot solve this system. Now suppose we know the l_i most significant bits of x_i, meaning we know x'_i such that $|x_i - x'_i| < 2^{t-l_i}$ and by centering around 0 we have $|x_i - x'_i - 2^{t-l_i-1}| < 2^{t-l_i-1}$.

We pose $u_i = \lfloor -a_i^{-1} b_i \rfloor_q$ and $v_i = \lfloor x'_i - a_i^{-1} c_i \rfloor_q + 2^{t-l_i-1}$. Then we have the inequality

$$|\alpha u_i - v_i|_q < 2^{t-l_i-1},$$

since $\lfloor \alpha u_i - v_i \rfloor_q = \lfloor x_i - x'_i - 2^{t-l_i-1} \rfloor_q$. It means that some multiple of u_i is very close to v_i modulo q. This can be transformed as an instance of a shortest vector problem by constructing a lattice generated by this integer matrix:

$$L = \begin{bmatrix} 2^{l_1+1}q & & & & & \\ & 2^{l_2+1}q & & & & \\ & & \ddots & & & \\ & & & 2^{l_n+1}q & & \\ 2^{l_1+1}u_1 & 2^{l_2+1}u_2 & \cdots & 2^{l_n+1}u_n & 1 & 0 \\ 2^{l_1+1}v_1 & 2^{l_2+1}v_2 & \cdots & 2^{l_n+1}v_n & 0 & q \end{bmatrix}.$$

Given the two vectors

$$U = (2^{l_1+1}u_1, \ldots, 2^{l_n+1}u_n, 1, 0)$$
$$V = (2^{l_1+1}v_1, \ldots, 2^{l_n+1}v_n, 0, q),$$

then there exist integers λ_i such that the vector $\alpha U - V + \sum_{i=1}^{n} \lambda_i L_i$ (where L_i is the i-th line of the matrix L) is a short vector of the lattice. By applying a reduction algorithm such as LLL or BKZ, we can hope one of the vectors of the reduced basis is this short vector which contains the secret value α in its penultimate coordinate by construction.

In the case we know the least significant bits of x_i noted x'_i, we have $u_i = \lfloor -(a_i 2^{l_i})^{-1} b_i \rfloor_q$ and $v_i = \lfloor x'_i 2^{-l_i} - (a_i 2^{l_i})^{-1} c_i \rfloor_q + q/2^{l_i+1}$.

C.2 Application to ECDSA

Given n ECDSA signatures (r_i, s_i) with their corresponding messages m_i, if the l most significant bits of the nonces are $0s$, the values u_i and v_i are

$$u_i = \lfloor r_i s_i^{-1} \rfloor_q$$
$$v_i = \lfloor -s_i^{-1} m_i \rfloor_q + 2^{t-l-1},$$

and if the l least significant bits of the nonces are $0s$, the values are

$$u_i = \lfloor -(s_i 2^l)^{-1} r_i \rfloor_q$$
$$v_i = \lfloor (s_i 2^l)^{-1} m_i \rfloor_q + q/2^{l+1}.$$

D Odd-signed comb scalar multiplication algorithm

In this appendix, we give a description of the encoding and scalar multiplication used for short Weierstrass curves in Mbed TLS, and a proof that all exceptional cases of the point addition formulas cannot happen and do not require a special treatment.

D.1 Odd-signed encoding

For a window size w, we note $m = \lceil t/w \rceil$ and for a w-bit integer $d = (d_{w-1}, \ldots, d_0)_2$ we note $[d] = d_0 + d_1 2^m + \ldots + d_{w-1} 2^{m(w-1)}$. The scalar $k = \sum_{i=0}^{t-1} k_i 2^i$ can be rewritten as

$$k = \sum_{i=0}^{m-1} [d_i] 2^i,$$

where $d_i = (k_{i+m(w-1)}, k_{i+m(w-2)}, \ldots, k_{i+m}, k_i)_2$ is composed of w bits of k all separated from a same distance m and $[d_i]$ is called a comb.

We suppose the scalar k is odd, then $[d_0]$ is odd. We apply the following algorithm to encode the other windows as odd values. Suppose every comb $[d_j]$ is odd for $0 \leq j \leq i-1$. If the comb $[d_i]$ is even, then we add the bits representing $[d_{i-1}]$ to the ones representing $[d_i]$ bit-by-bit and for every w bits of the comb, we save the eventual carry, and $[d_{i-1}]$ is changed to $-[d_{i-1}]$. The carry is then added to the next comb. The operation means that $[d_{i-1}] + 2[d_i]$ is changed to $-[d_{i-1}] + 2([d_{i-1}] + [d_i])$ in the expression of k, which does not change its value.

The carry propagates until it reaches a new comb $[d_m]$ that is positive. Denoting $[d'_i]$ the value of the new comb windows and s_i a bit indicator for

the sign (0 for positive and 1 for negative), the scalar is then encoded as

$$k = \sum_{i=0}^{m} (-1)^{s_i} [d'_i] 2^i,$$

where d'_i is odd for all $0 \leq i \leq m$.

D.2 Comb method with odd-signed representation

The scalar multiplication is presented in algorithm 5. One drawback is that it works only for an odd scalar k, but if k is even then $q - k$ is odd, so this case can still be handled by carefully implementing a branchless selection between k and $q - k$.

Require: $k = (k_{t-1}, \ldots, k_0)_2$, P, w
Ensure: kP

Precomputation phase
1: **for** $i \leftarrow 0$ **to** $2^{w-1} - 1$ **do**
2: $\quad$ Tab$[i] \leftarrow [2i + 1]P$

Encoding phase
3: **if** k is even **then**
4: $\quad k' \leftarrow q - k$
5: **else**
6: $\quad k' \leftarrow k$
7: $(s_0, d'_0), \ldots, (s_m, d'_m) \leftarrow$ Encoding(k')

Evaluation phase
8: $R \leftarrow$ Tab$[(d'_m - 1)/2]$
9: **for** $i \leftarrow m - 1$ **down to** 0 **do**
10: $\quad R \leftarrow 2R$
11: $\quad R \leftarrow R + (-1)^{s_i}$ Tab$[(d'_i - 1)/2]$
12: **if** k is even **then**
13: $\quad y(R) \leftarrow -y(R)$
return R

Algorithm 5. Single scalar multiplication with odd-signed comb method.

D.3 Proof that there is no exception

We suppose the formulas for point addition are the same as those in appendix B and we prove here that the special cases can never happen when the curve order q satisfies $q \equiv 1 \bmod 4$ and $m \geq 2w + 5$. This last condition is satisfied for 256-bit curves when $w \leq 10$.

For ease of notation, we note $C_i = (-1)^{s_i}[d'_i]$ and $k = M_0 + 2^j M_1$ where $M_0 = \sum_{i=0}^{j-1} C_i 2^i$ and $M_1 = \sum_{i=j}^{m} C_i 2^{i-j}$. Also, we note the two bounds that will be useful in the proof:

$$1 \leq |C_i| < 2^{(w-1)m+1}, \quad \text{and} \quad 2^{(m-1)w} < q < 2^{mw}.$$

Apparition of the null point in a loop. Suppose that $j \geq 1$ and the result of the addition in the loop is the point $\mathcal{O}$, meaning the relation $R = -C_j P$, from which we get the relation

$$M_1 \equiv 0 \mod q,$$

and we get the bound $|M_1| < 2^{mw-j+2}$. If $j \geq w+2$, the bound becomes $|M_1| < q$, so $M_1 = 0$ which is impossible since M_1 is odd. Now we suppose $j < w+2$. We have $k \equiv M_0 \bmod q$ and the bound $|M_0| < 2^{(w-1)m+w+3}$. Since we supposed $m \geq 2w+5$, we have $|M_0| < q$. Then either $k = M_0$ which implies $M_1 = 0$, or $k = M_0 + q$ which implies q is even, both are impossible.

Then the result of the addition is proved to never be the null point $\mathcal{O}$, except in the last loop which happens when the scalar is q. In this case, the formulas compute a correct representation of this point in Jacobian coordinates.

Doubling case in the last iteration of the loop. Before the addition in the last loop, we have $R = \sum_{i=1}^{m} C_i 2^i P$ and cannot be the null point $\mathcal{O}$ as proved above. Then the only exception would be if we have $R = C_0 P$, it means that $k \equiv 2C_0 \bmod q$.

Since m is large enough, we have $|2C_0| < q$, then either $k = 2C_0$ or $k = q + 2C_0$. The first case is impossible because k is odd. In the second case, C_0 is negative, it means C_1 is even according to the encoding. So the second least significant bit of k is 0. Then $k \equiv 1 \bmod 4$ from which we get that $q \equiv 3 \bmod 4$.

If a curve order satisfies this condition, it is possible that there is a scalar that produces the doubling exception in the last loop. In particular it happens for curve P-384 in the implementation of this algorithm in Mbed TLS. But this is not possible when $q \equiv 1 \bmod 4$, which is the case for curves P-224, P-256 and P-521.

Doubling case in a previous iteration of the loop. Suppose that for $j \geq 1$, the doubling case happens in the loop so we have the equality

$R = C_j P$, from which we get the relation

$$M_1 \equiv 2C_j \mod q.$$

We get a large bound $|M_1 - 2C_j| < 2^{mw-j+2}$ using the bounds on C_i and q. If $j \geq w+2$, then we have $|M_1 - 2C_j| < q$, so $M_1 = 2C_j$ which is impossible due to parity. Now we suppose $j < w+2$. We have $k \equiv M_0 + 2^{j+1}C_j \mod q$, and the bound $|M_0 + 2^{j+1}C_j| < 2^{(w-1)m+w+5}$. Since we supposed $m \geq 2w+5$, we have $|M_0 + 2^{j+1}C_j| < q$. Then either $k = M_0 + 2^{j+1}C_j$ which implies the impossible equality $M_1 = 2C_j$, or $k = M_0 + 2^{j+1}C_j + q$ which implies that q is even, but q is odd.

Other remarks. The doubling in line 10 of algorithm 5 can be removed at the cost of having a precomputed table for each comb as has been done in algorithm 3 for the implementation of curve P-256 in OpenSSL.

The algorithm is initialized by taking a precomputed point in the table which cannot be $\mathcal{O}$, then its coordinates can be easily randomized to add protection against Differential Power Analysis.

References

1. Jeremy Dubeuf, David Hely, and Vincent Beroulle. Enhanced elliptic curve scalar multiplication secure against side channel attacks and safe errors. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 65–82, Cham, 2017. Springer International Publishing.
2. Pierre-Alain Fouque, Sylvain Guilley, Cédric Murdica, and David Naccache. *Safe-Errors on SPA Protected Implementations with the Atomicity Technique*, pages 479–493. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
3. Christophe Giraud and Vincent Verneuil. Atomicity Improvement for Elliptic Curve Scalar Multiplication. In D. Gollmann and J.-L. Lanet, editors, *CARDIS 2010*, volume 6035 of *LNCS*, pages 80–101, Passau, Germany, April 2010. Sprinter.
4. Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.
5. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering*, 5(2):141–151, Jun 2015.
6. Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. A comb method to render ECC resistant against side channel attacks. Cryptology ePrint Archive, Report 2004/342, 2004. `https://eprint.iacr.org/2004/342`.
7. P-256 implementation in BoringSSL. `https://boringssl.googlesource.com/boringssl/+/refs/heads/master/crypto/fipsmodule/ec/p256-x86_64.c`. Accessed: 2020-05-01.
8. P-256 implementation in LibreSSL. `https://github.com/libressl-portable/openbsd/blob/master/src/lib/libcrypto/ec/ecp_nistz256.c`. Accessed: 2020-05-01.

9. P-256 implementation in OpenSSL. `https://github.com/openssl/openssl/blob/master/crypto/ec/ecp_nistz256.c`. Accessed: 2020-05-01.
10. Bodo Möller. Securing elliptic curve point multiplication against side-channel attacks. In George I. Davida and Yair Frankel, editors, *Information Security*, pages 324–334, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
11. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
12. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, Sep 2003.
13. Katsuyuki Okeya and Tsuyoshi Takagi. The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 328–343, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
14. Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 403–428, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
15. Franck Rondepierre. Revisiting atomic patterns for scalar multiplications on elliptic curves. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications*, pages 171–186, Cham, 2014. Springer International Publishing.
16. Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and Sangjae Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In Kwangjo Kim, editor, *Information Security and Cryptology — ICISC 2001*, pages 414–427, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.