Exhaustive single bit fault analysis. A use case against Mbedtls and OpenSSL's protection on ARM and Intel CPU

Sébastien Carré, Matthieu Desjardins, Adrien Facon, Sylvain Guilley

Version of Record: https://www.sciencedirect.com/science/article/pii/S0141933118305143
Manuscript_af0ae5eecfda7608a400f65d82488a2b

Exhaustive single bit fault analysis. A use case against Mbedtls and OpenSSL's Protection on ARM and Intel CPU

Carre Sebastien[a], Sylvain Guilley[a], Adrien Facon[b]

[a] Telecom Paristech, 75013 Paris, France
[b] ens, France

Abstract. Faults in software implementations target both data and instructions at different locations. Bellcore attack is a well-known fault attack that is able to break CRT-RSA. In response, cryptographic libraries such as OpenSSL are designed with protection. In this paper, new faults locations are shown on OpenSSL implementation of the CRT-RSA signature running on Intel and ARM processors. Among those faults, one restores the Bellcore attack on the OpenSSL library despite a protection and another is a complete new fault that exploits the OpenSSL protection to get RSA private key. Quite surprisingly, one of the exhibited faults is, ironically enough, made possible because of the existence of such protection. Mbedtls library is also analysed in this paper. A way to find all exploitable faults on monobit flip fault model is also detailed.

1 Introduction

Fault attacks against cryptographic implementations consist in triggering anomalies during the execution of a program in order to retrieve sensitive data such as cryptographic keys. Dan Boneh *et al.* [6] demonstrated the feasibility of such attack on RSA signature that use the Chinese Remainder Theorem (CRT). They have shown that faulting a computation of the signature modulo p or q allows an attacker to retrieve the RSA private key. This attack, known as Bellcore attack, is not possible if a fault occurs on both computations of signatures modulo q and modulo p. Christian Aumller *et al.* [2] have shown that the fault can also occur after the signatures modulo p or q are computed, namely when they are used in the recombination stage.

1.1 Contributions

This paper investigates OpenSSL and Mbedtls reliability against mono-bit faults. This paper first investigates it in the rowhammer attack context, where the fault model is to flip bits from 1 to 0, on Intel processors. This is then extends to 0 to 1 fault model. The analysis is also extended to ARM processors. The cryptography operation that this paper targets is RSA signature performed with both padding and message blinding.

Our fault research method consists in an exhaustive analysis of code mutation, that is a very low-level methodology which does not require the attacker to analyze the code. This simple methodology revealed powerful, since two new faults location was found on OpenSSL compiled for Intel processors and more than ten new faults location on OpenSSL compiled for ARM processors that lead to retrieve the RSA private key by modifying only one instruction per fault.

The first fault on OpenSSL consists in an offset modification coded in a memory read instruction. This fault spreads to multiple locations of CRT-RSA algorithm. More specially both signatures modulo p and modulo q as well as Garner's CRT recombination [12] are faulted. In those conditions, the Bellcore attack should not work. However, the fault still allows to recover the private RSA key. Interestingly, that is made possible thanks to the OpenSSL protection against fault attacks on CRT. This is a paradoxical situation where a defensive feature is jeopardized to become an attack vector. This first fault consists in modifying an offset in a read memory instruction which results in passing an incorrect argument to a given function.

The second fault on OpenSSL is a way to restore the Bellcore attack despite the OpenSSL protection against it. This fault is performed on Garner's CRT recombination. It mainly consists in a confusion between the addresses of two registers which causes data substitution, also known as splicing. This allows to exploit the processor calling convention to permanently modify the function so that it takes different arguments. During this attack, the protection is simply skipped, hence it is ineffective to detect the injected fault.

None of those faults directly modify the value of a register, and despite one could achieve those faults by this way, such scenario requires to know the addresses from the victim, which is an impractical assumption when ASLR protection is activated. The way the faults are performed is not affected by such protection.

The two last faults are possible on both Intel and ARM processors. However, more faults was found on OpenSSL compiled for ARM processors. The faults can be classified as deterministic faults, that always work independently of the message that have to be signed, and probabilistic faults that occurs only with specific inputs. Those specific inputs are finely detailed in this paper and the probability for which those faults works is drift. A probabilistic fault is then more difficult to detect since it behaves like a random functional error.

Namely, six deterministic exploitable faults and five probabilistic exploitable faults were found on OpenSSL compiled for ARM. On Intel, all faults are deterministic.

This paper is the extension of Carré et al. paper [9] that only focuses on Intel processor. From [9], our paper is extended as follows. Firstly, ARM processors is take into account in the addition of Intel processors. Secondly, Mbedtls is analysed with the same methodology used by Carré et al. [9]. Thirdly, the number of exploitable faults is extended and probabilistic faults are introduced. Fourthly, the fault model that flips a 1 to a 0 is extended to a fault model that also flips 0 to 1. Finally, practical considerations about Rowhammer attack is discussed.

1.2 Outline

This paper is organized as follows. Section 2 gives prerequisites to understand how the faults that are described in this paper work, including x86_64 and ARM calling convention. Section 3 gives background about RSA signatures, including CRT-RSA and its implementation in the OpenSSL cryptographic library. More specifically, this section gives the details of the register usage while calling the two functions responsible for the fault attacks that are described in this paper. The Bellcore attack is also reminded in this section. Section 4 gives the methodology used in this paper to find the faults before explaining how they lead to a key recovery by Bellcore attack for Intel processors in Sec. 5 and for ARM processors in Sec. 6. Finally, the section 7 discusses several generalizations and mitigations of the described attacks. The conclusion is then given in section 8.

2 Prerequisite

2.1 Fault attacks

Faults can be injected using multiple techniques such as laser irradiation, penetration by an electromagnetic pulse, or by tampering with the clock of the targeted electronic device. Most of these techniques require a specific equipment and cannot be perpetrated remotely. On an electronic device, several fault locations are possible. For example, Ingrid Verbauwhede *et al.* [22] give a classification model of fault attacks in which registers can be used as a place to fault. Hagai Bar-El *et al.* [3] also point out registers as a location to make a fault.

Other injections like the rowhammer attack do not rely on sophisticated equipment. This attack is a Dynamic Random Access Memory (DRAM) specific fault injection, introduced by Yoongu Kim *et al.* [16], that allows an attacker to directly change the values stored in DRAM. The high density of capacitors in such memory makes it possible to discharge one of these capacitors by accessing another one. This allows an attacker to flip a bit from 1 to 0 and thereby gives the opportunity to modify either data or instructions of a computer software despite protections such as $W \oplus X$.

Practical exploitation scenarios have been accounted by Mark Seaborn *et al.* [21] who show a privilege escalation by modifying one bit in a page table using native code.

Despite the rowhammer attack originally had strong constraints such as the need to know physical addresses and its mapping with physical memory structure, various techniques were developed to improve the feasibility of this attack. For example Sarani Bhattacharya and Debdeep Mukhopadhyay [4] use timing analysis on the DRAM row buffer to find an interesting location to make a fault.

Daniel Gruss *et al.* [15] show the possibility to use this attack remotely, using high precision timers available on JavaScript. Pierre Carru [10] shows that even in a trusted execution environment using TrustZone, a cryptosystem can be broken by rowhammer. More general faults on cryptographic applications like the one

exposed by Éric Brier *et al.* [7] on CRT-RSA are also exposed to rowhammer attack on the public elements of the cryptosystem, such as its modulus.

Our paper reports that a naïve bit-reset analysis of a cryptographic library suffices to find relevant fault locations. However, more sophisticated tools would greatly help to better identify more numerous weaknesses. For example, Marie-Laure Potet *et al.* [20] use symbolic and concolic execution to find exploitable faults that rely on modifying the control flow of a software. Still in an evaluation context, formal verification can be used. For instance Lucien Goubet *et al.* [14] have submitted a tool that helps to formally evaluate fault attack countermeasures thanks to an SMT solver. Thomas Given-Wilson *et al.* [13] developed a formal way to find fault injection vulnerabilities using a model.

Despite our results are not specific to the rowhammer attack and can be generalized to other means of fault injection, this paper first restrict to the context of rowhammer faults. More specifically, this paper first focus on finding mono-bit faults that carry out a bit flip from 1 to 0 in the `.text` section of the cryptographic library, and then investigate on bit flips from 0 to 1. Indeed, the code of a shared library can be repetitively accessed in read mode by the attacker, which creates errors for both the attacker and any victim linked to this shared code.

2.2 Calling convention

Nowadays processors use multiple registers. Some of these registers can be use for general purpose, others have specific purpose like containing the pointer of next instruction to be executed. Some of these registers are given in Table 1 and 2.

Table 1. General purpose registers

Intel x86_64	**ARM**
`rax`, `rbx`, `rcx`, `rdx` `rsi`, `rdi`, `r8 to r15`	`R0 to R10`

Table 2. Specific purpose registers

Intel x86_64	ARM64	Description
`rip`	`PC`	instruction pointer
`eflags`	`CPSR`	flags
`rsp`	`SP`	Stack pointer
`rbp`	`FP`	Stack base
`-`	`LR`	Return address

Moreover, some of those registers are mobilized to pass parameters upon a function call. The purpose of those registers is explained in Tab. 3.

Table 3. Calling convention

x86_64	ARM64	Purpose
`rax`	`R0`	return integer value
`rdi`	`R1`	first integer parameter
`rsi`	`R2`	second integer parameter
`rdx`	`R3`	third integer parameter
`rcx`	`R4`	fourth integer parameter
`r8`	`R5`	fifth integer parameter
`r9`	`R6`	sixth integer parameter

2.3 Instruction structure

In order to be able to understand the fault that are described in this paper, the structure of some instructions has to be formulate. For Intel processor, only a `mov` instruction is interesting. This instruction have the following structure

`mov OFF(%REG_BASE),%REG_DEST`,

in which `%REG_DEST` is the register where the read data will be stored to and where `OFF(%REG_BASE)` is the AT&T syntax that indicates that the data to be read is located at address `%REG_BASE+OFF`, where `REG_BASE` is a register that contains a memory address and `OFF` is an offset value. For example a local variable of a function can be read related to the stack frame register `rbp` as follows:

`mov -0x14(%rbp),%rax`.

On ARM processor, this paper focus on five classes of instructions that are briefly describe here.

Load and store instructions
For our purpose, those instructions have the following format:

`op{cond} Rt, [Rn, #off]`

where

- `op` is either `STR` for a store to memory instruction, or `LDR` for a load from memory instruction.
- `cond` is an optional condition code such as `vs` that performs the operation only if the overflow flag is set.
- `Rt` is a register used to load or store the wanted data.
- `Rn` contains a base memory address on which the processor adds the offset `#off` to locate the address to store or load.

For example, some load and store instructions that are used in this document are:

```
  STR R0,[FP, #-24]
  LDR R0,[FP, #-24]
LDRVS R1,[FP, #-28]
```

Stack instructions
Stack instructions are structured as follows:

`op` {*list*}

where

- `op` is either `PUSH` or POP to write to or read from the stack respectively.
- *list* is the list of registers to push on the stack or to retrieve data from it.

For example, the following instruction push and pop registers on the stack.

```
PUSH {R4, R11, LR}
 POP {R11,PC}
```

Test instruction
The test instruction performs a logical AND operation, discards the result and set the needed flags. The format of this instruction is

```
TST Rn, Operand2
```

where

- `Rn` is the register that contains the first operand.
- `Operand2` is a flexible operand, meaning it can be either a constant or a register with optional shift.

For example the following instruction does a test on the values contained in the register `FP` and the register `IP` shifted to the left by the value contained in the register `R0`.

```
TST FP, IP, LSL R0
```

Branch instruction
Multiple branch instructions can be used for ARM processors. For our purposes, this paper is only interested by the `BL` (branch with link) instruction that jump to a specific address and stores the return address in register `RL`. This instruction has the following structure:

```
BL Operand
```

where:

– `Operand` is the relative address, from the current instruction address, to jump.

For example, the following instruction jumps to relative address `0x9f788`.

```
BL 9f788
```

Arithmetic instructions
Arithmetic instructions have the following structure.

```
op Rd, Rn, Operand2
```

where

– `op` is an arithmetic operation. For our purposes, this paper only limits to `ADD` or `SUB` operations that perform addition and subtraction respectively.
– `Rd` register contains the result of the operation.
– `Rn` and `Operand2` are the two operands of the operation. `Operand2` can be a register or a value.

The following instructions show two examples of addition and subtraction operations:

```
add r11, sp, #8
sub sp, r11, #4
```

3 RSA

RSA is an asymmetric cryptosystem that can be used to sign or encrypt messages. In this paper, only RSA signature is considered. This cryptosystem uses a private key (p, q, d, d_p, d_q, i_q) to sign a message M and a public key (e, n) to verify this signature S. Algorithm 1 gives the details about this key generation.

Output: Public key (e, n), private key (p, q, d, d_p, d_q, i_q)

1 (A.1) Choose two prime numbers p and q large enough and not too close one from each other (see PKCS#1).
2 (A.2) Compute $n = pq$.
3 (A.3) Compute $\phi(n) = (p-1)(q-1)$.
4 (A.4) Choose e such as $\gcd(e, \phi(n)) = 1$.
5 (A.5) Compute $d = e^{-1} \bmod \phi(n)$.
6 (A.6) Compute $(d_p, d_q, i_q) = (d \bmod (p-1), d \bmod (q-1), i_q = q^{-1} \bmod p)$.
7 **return** public key (e, n) and private key (p, q, d, d_p, d_q, i_q).

Algorithm 1: RSA key generation.

Knowing at least one of the prime number p or q, an attacker can recover the whole private key.

The signature of a message M is performed by computing the modular exponentiation $S = M^d \bmod n$ whereas the verification of a signature is performed by computing the modular exponentiation $S^e \bmod n$ and by checking whether the result matches the message M. Note that a hash of the message M is often used instead of the message itself.

Notations: For reader convenience, the notations used in this paper are gather here-after:

- n: RSA modulus as described in Alg. 1.
- d_p, d_q: The secret exponents modulo $p-1$ and $q-1$ as described in Alg. 1.
- M: The message to be signed.
- M_p, M_q: The messages to be signed reduced modulo p and q respectively as described in Alg. 2.
- $S = M^d \bmod n$: Signature of the message M.
- S_p, S_q: Signatures modulo p and q respectively as described in Alg. 2.
- S_e: $S^e \bmod n$. Without fault, this is equal to M.

3.1 CRT-RSA

The modular exponentiation $S = M^d \bmod n$ to sign a message M can be computed using the Chinese Remainder Theorem (CRT) that splits this exponentiation into two modular exponentiations $S_p = M_p^{d_p} \bmod p$ and $S_q = M_q^{d_q} \bmod q$. The final signature S is then given by the Garner's recombination $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \bmod p)$. In the rest of the paper the term CRT-RSA is used to refer an RSA signature with the CRT. The whole algorithm is given in steps (B.1) to (B.5) in Alg. 2. The steps (B.6) and (B.7) of this algorithm check whether CRT-RSA signature is correctly computed. If it is not the case, the signature is computed without CRT, which does not expose p or q to Bellcore attack. Performing a fault on the conditional branch (B.7) of Alg. 2 is a way to bypass this protection, but still, **two** faults are needed (one to corrupt (B.2) or (B.4), and one to skip (B.7)). In this paper, only **single** bit fault model is considered.

3.2 Padding of messages to be signed by RSA

RSA as a simple modular exponentiation, also called RSA textbook, is not secure enough since if the message to sign is too small, it is possible to retrieve the original message by computing a root. To avoid this, `padding` is used. There exist multiple kinds of padding. The most common ones for signature are PKCS #1 types 1 and 2. Both are similar and ensure that the length of the message is large enough. PKCS #1 type 1 is a *deterministic* padding, whereas PKCS #1 type 2 is *probabilistic*. Note that a proven probabilistic signature scheme (PSS) for RSA signature is often recommended.

In an encryption context, PKCS OAEP should be used. This kind of padding does not only concatenate bytes, but ensures that the padded message looks like a completely random one.

Input : Message M, private key (p, q, d, d_p, d_q, i_q)
Output: Signature $S = M^d \mod n$

(B.1) $M_p = M \mod p$.
(B.2) $S_p = M_p^{d_p} \mod p$. `/* Signature mod p */`
(B.3) $M_q = M \mod q$.
(B.4) $S_q = M_q^{d_q} \mod q$. `/* Signature mod q */`
(B.5) $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$. `/* Garner recombination mod n */`
(B.6) $S_e = S^e \mod n$.
(B.7) **if** $(S_e - M) \neq 0$ **then** `/* verification */`
  (B.8) $S = M^d \mod n$ `/* RSA textbook */`
return S.

Algorithm 2: CRT-RSA (B.1∼5) with verification (B.6∼8).

3.3 Message blinding

The RSA textbook signature is vulnerable to timing side channel attack as described by Paul C. Kocher [17]. One way to prevent this attack is to use the blind signature. For RSA, message blinding consists in replacing M by $M \cdot r^e \mod n$ where r is a random value. The signature is then performed on the blinded message as shown in Alg. 3.

Input : Message M, private key (p, q, d, d_p, d_q, i_q)
Output: Signature $S = M^d \mod n$

(C.1) Choose r such that $\gcd(r, n) = 1$.
(C.2) Blind the message: $M_b = M \cdot r^e \mod n$.
(C.3) Sign the blinded message: $S_b = M_b^d \mod n$.
(C.4) Unblind: $S = S_b \cdot r^{-1} \mod n$.
return S.

Algorithm 3: RSA on a blinded message.

Note that RSA textbook is vulnerable to RSA blinded attack that makes possible to decipher a message if same keys are used in both signature and encryption schemes. Due to that attack, most implementations blind a hashed and padded message instead of the message itself. Moreover, it is highly recommended not to use the same keys for both signature and encryption schemes.

3.4 Bellcore attack

The Bellcore attack is an CRT-RSA specific fault attack introduced by Dan Boneh *et al.* [6]. The authors show that if the computation of S_p is corrupted but S_q is not, it is possible to recover the prime number q and subsequently the

entire private key. Indeed, in such condition, one gets a wrong partial signature S'_p and then a wrong signature S' of the message M. The following is then true:

$$\begin{cases} S' - S = S'_p - S_p \neq 0 \bmod p, \\ S' - S = S_q - S_q = 0 \bmod q. \end{cases}$$

Thus $S' - S$ is a multiple of q but is not a multiple a p. As $n = pq$, the secret prime q can be computed using the greatest common divisor (GCD):

$$q = \gcd(S' - S, n). \tag{1}$$

Bellcore attack with padding and blinding:

The Bellcore attack on CRT-RSA signature (Eqn. (1)) requires both a correct and a faulty output computation from the same given input to the modular exponentiation.

If an attacker can directly control the inputs of the modular exponentiation, the Bellcore attack always works. However, a more likely scenario is the one where the attacker simply provides a message to a signing oracle. In this condition, if the padding scheme is a non deterministic one, then even with two identical messages, the input of the modular exponentiation will be different and the Bellcore attack does not work except if the attacker finds a way to either disable or make the padding deterministic. This may be performed with another fault. OpenSSL nevertheless uses the PKCS #1 type 1 padding that is deterministic.

Moreover, message blinding does not preclude Bellcore attack from working. The Bellcore attack requires to have two signatures S_1, S_2 of the same message M, namely a correct one (S_1) and a faulted one (say faulted signature S'_2 of S_2). The correct signature is computed with blinding message Mr_1^e to get the signature $S_1 = (Mr_1^e)^d r_1^{-1} \bmod n = M^d \bmod n$. Clearly, the blinding factor r_1 cancels out in the final signature S_1. The second signature is a faulted version S'_2 of signature $S_2 = (Mr_2^e)^d r_2^{-1} \bmod n$, meaning that $S'_2 \neq S_2$ and, since $S_2 = M^d \bmod n = S_1$ it results in $S'_2 \neq S_1$. Thus, if the fault only occurs on the computation of the signature modulo p, the following is true:

$$\begin{cases} S'_2 = S_1 \bmod q, \\ S'_2 \neq S_1 \bmod p. \end{cases} \iff \begin{cases} S'_2 - S_1 = 0 \bmod q, \\ S'_2 - S_1 \neq 0 \bmod p. \end{cases}$$

Thus, even with blinding, $q = \gcd(S'_2 - S_1, n)$. Note that this is true for any r_1 and r_2. For convenience, and since $S_2 = S_1$, S design both the correct signature and S' the faulted one.

3.5 OpenSSL implementation of RSA signature

The version 1.1.0f of OpenSSL is considered in this paper. At the time of submission of this paper, the latest stable version of OpenSSL is 1.1.0h. This version

has no difference with version 1.1.0f about the way an RSA signature is generated. OpenSSL is compiled with default configuration using the OpenSSL script `./config shared -d`. By default the code is compiled with `-O3` gcc optimization class. Note that some of the described faults can disappear while others can appear whether one changes option of the compilation.

number representation in OpenSSL. OpenSSL represents a big number with the following C structure.

```
struct bignum_st {
    BN_ULONG *d;
    int top;
    int dmax;
    int neg;
    int flags;
};
```

Listing 1.1. Structure that represents numbers in OpenSSL

Here, the important fields of this structure is given for the purposes of this paper.

- `d` points to a memory location that contains the raw binary form of the number.
- `top` contains the size of the number. If it is zero, the number is considered to be equal to zero.
- `neg` is used to specify whether the number is negative or not.

Note for ARM processors. Since some OpenSSL variables have the same names as ARM registers the rest of this paper precises at each time if it deals with variables or registers. Moreover, registers are written with uppercase and variables with lowercase.

RSA signature. The main function that computes RSA signature in OpenSSL is `rsa_ossl_private_encrypt`. It is used to pad the message and blind the padded message before performing the modular exponentiation by calling `rsa_ossl_mod_exp`. This function implements Alg. 2. It computes the modular exponentiation using CRT and, as a protection against the Bellcore attack, it checks whether the computation is performed correctly. Otherwise, the modular exponentiation is computed without CRT, as it is resilient to the Bellcore attack.

One of our fault focuses on the function used to reduce the message M modulo p and q. To do this reduction, `rsa_ossl_mod_exp` calls the function `BN_mod`.

An other fault focuses on the subtraction operation used both in Garner's recombination and in the conditional branch used for the Bellcore fault attack protection in step (B.7) of Alg. 2. To perform this subtraction,`rsa_ossl_mod_exp` call the function `BN_sub`.

The way the parameters of those functions are retrieved and passed by `rsa_ossl_mod_exp` is the core of the discovered faults in this paper. The end

of this section explains how those parameters are handled. For simplicity, the notation used are the same defined in section 3 instead of OpenSSL variable names.

`BN_mod`. This function takes 4 arguments:

- The first one is the variable where the reduction result will be stored.
- The second one is the variable to be reduced. In our case it is the message M to sign.
- The third one is the modulus. In the case of CRT-RSA it is either p or q.
- The fourth one is a context that is out of the scope of this paper.

`BN_mod` is actually a macro to `BN_div` that computes an Euclidean division. This latter function takes five arguments:

- The first one is the variable where the quotient is stored. In our case, it is not used.
- The second one is the variable where the reduction result takes place.
- The third one is the variable to be reduced. In our case it is the message M to sign.
- The fourth one is the modulus. In our case it is either p or q.
- The fifth one is a context (`ctx` irrelevant for this paper).

Hence the reduction of M modulo p is computed using

$$\texttt{BN\_mod}(M_p,\ M,\ p,\ \texttt{ctx}),$$

that is actually

$$\texttt{BN\_div}(\texttt{NULL},\ M_p,\ M,\ p,\ \texttt{ctx}).$$

A symmetric operation is performed to reduce M modulo q.

The function `rsa_ossl_mod_exp` passes parameters to the function `BN_div` by using x86_64 or ARM calling conventions explained in Sec. 2.2. For x86_64 processor, this is performed by the assembly instruction sequence illustrated in Fig. 1. Similar instructions are used for ARM processor.

`BN_sub`. This function takes 3 arguments:

- The first one is the result of the subtraction.
- The second one is the variable to subtract from.
- The third one is the variable to subtract to.

`BN_sub` is called twice by the function `rsa_ossl_mod_exp`. The first one is called for the Garner's recombination ((B.5) in Alg. 2) when computing $S_p - S_q$ by doing `BN_sub(`S_p`,`S_p`,`S_q`)`. The second one is performed to check whether

```
mov     0x40(%rax),%rcx   ; Getting variable q stored at address 0x40(%rax).
mov    -0x80(%rbp),%rsi   ; Getting variable ctx stored at address -0x80(%rbp) into rsi register temporally.
mov    -0x28(%rbp),%rdx   ; Getting variable M stored at address -0x28(%rbp) into rdx register
                          ; used as the third argument of the function.
mov    -0x50(%rbp),%rax   ; Getting variable M_q stored at address -0x50(%rbp) into rax register temporally.
mov    %rsi,%r8           ; Copying value of register rsi into r8 register used as a fifth argument of
                          ; the function.
mov    %rax,%rsi          ; Copying value of register rax into rsi register used as a second argument of
                          ; the function.
mov    $0x0,%rdi          ; Setting register rdi, used as a first argument of the function, to 0 .
callq  88790 <BN_div@plt> ; Calling function BN_div.
```

Note: The message M is actually represented by two variables stored at `-0x28(%rbp)` and `-0x70(%rbp)`. They only differ by a flag (related to constant time operations) that is irrelevant in this paper. Since those two variables contain a pointer to the same number, they can be considered as identical.

Fig. 1. Passing parameters to `BN_div` to compute $M_q = M \bmod q$.

```
mov    -0x48(%rbp),%rdx   ; Getting variable S_q stored in -0x48(%rbp) into rdx register used as
                          ; the third argument of the function.
mov    -0x68(%rbp),%rcx   ; Getting variable S_p stored in -0x68(%rbp) into rcx register temporally.
mov    -0x68(%rbp),%rax   ; Getting variable S_p stored in -0x68(%rbp) into rax register temporally.
mov    %rcx,%rsi          ; Copying value of register rcx into rsi register used as the second argument of
                          ; the function.
mov    %rax,%rdi          ; Copying value of register rax into rdi register used as the first argument of
                          ; the function.
callq  89620 <BN_sub@plt> ; Calling function BN_sub.
```

Fig. 2. Passing parameters to `BN_sub` in the context of (B.5) in Alg. 2.

CRT-RSA is computed correctly by computing $S_e - M$ ((B.7) in Alg. 2) and check whether the result is 0. The subtraction is computed by calling `BN_sub(`S_e`,`S_e`,`M`)`.

Figure 2 gives the assembly instruction sequence, for Intel x86_64 processors, used for the first call to `BN_sub`. The second call is performed in the same way. A similar code is used for ARM processors.

In order to retrieve those parameters the function `BN_sub` reads the content of registers `rdi`, `rsi`, `rdx` for Intel processors(`R0`, `R1` and `R2` for ARM processors) and stores them in its stack frame that contains function local variables. This is performed by the following three assembly instructions for Intel processors

```
mov %rdi,-0x28(%rbp),
mov %rsi,-0x30(%rbp),
mov %rdx,-0x38(%rbp),
```

and the following three for ARM processors:

```
STR R0, [FP, #-24]
STR R1, [FP, #-28]
STR R2, [FP, #-32]
```

Internally `BN_sub` checks if $a < b$ by using `BN_ucmp` function. If it is the case, it actually computes $r = b - a$ and set a negative flag to the result variable `r`. If it is not, it computes $r = a - b$ and ensures that the negative flag of the result variable `r` is not set as explain in the listing 1.2. Note that the two differences $r = b - a$ and $r = a - b$ are performed using the BN_usub function.

```
if (BN_ucmp(a, b) < 0) {
  if (!BN_usub(r, b, a))
          return 0;
      r->neg = 1;
  } else {
      if (!BN_usub(r, a, b))
          return 0;
      r->neg = 0;
}
```

Listing 1.2. Internal of BN_sub function

4 Faults research automation

Despite this paper aim is not to release a fault analysis tool, this section gives the methodology used to find the exploitable faults described in the next section. In this aim, a signature oracle and a control process are required.

On the one hand, the signature oracle process aim is to sign a message with a faulted OpenSSL cryptographic library.

On the other side, the control process is responsible for creating a fault. More specifically, the fault consists in flipping a bit from 1 to 0 (or from 0 to 1 depending of the model) in OpenSSL cryptographic library `.text` section, and run the signature oracle process using `fork()` and `execv()` POSIX functions available on GNU Linux Operating System. The fault is simulated by modifying directly the bits in the OpenSSL file on the hard disk. This operation is repeated for each bit equal to 1 in the OpenSSL shared library. Since there are 16398352 bits equals to 1 (or 0 depending of the model) in the version of OpenSSL that is studied in this paper, this exhaustive methodology can take a lot of time. To reduce this time, the analysis is directly performed in memory in order to prevent disk accesses and disable any core dump generated by the operating system.

One other aim of the control process is to handle errors raised by the signing oracle process due to a fault. Indeed, some faults will result in a crash of the signature oracle padding process that can be due to multiple reasons. For example the fault can transform an instruction into a non-existing one that results in an illegal instruction signal.

Some faults can also result in an infinite loop in the signature oracle process. To prevent it, the control process implements a watchdog that forces the signature oracle process to terminate after a certain amount of time.

Finally, if the signature oracle process terminates without error, the control process checks whether the fault is exploitable to recover a cryptographic key. In our case, this check is performed by computing a GCD, as per Eqn. (1).

The 2D color map represented in Fig. 3 gives a view of faults results for OpenSSL compiled on Intel processors. This figure has to be read from left to right and from top to bottom. Each pixel of this figure represents one bit of the library that, if this bit is equal to 1, has been flipped to 0. The bits equal to 0 simply remain unchanged. The white color corresponds to bit flips for which $\gcd(S' - S, n) \in \{p, q\}$. This matches the two faults. The size of those points in the figure was intentionally increased for a better visualization. The black color represents the faults that have no effect during the computation of an RSA signature. Other colors are related to different errors or watchdog timeout. A similar method is used for the 0 to 1 model where the bits equal to 0 are flipped.

5 Fault on CRT-RSA implementation of OpenSSL compiled on Intel

5.1 Fault in `BN_mod` call

In this section details are given on the first fault that both restore the Bellcore attack and break the OpenSSL protection intended to thwart it. Interestingly, this fault works because of this protection at lines (B.7) and (B.8) of Alg. 2. This fault occurs at bit offset 3866 from the beginning of the function `rsa_ossl_mod_exp` that calls `BN_mod`.

As illustrated by the red color and with the lightning symbol in step (D.1) in Alg. 4, the fault occurs during storing the result of the reduction of the message

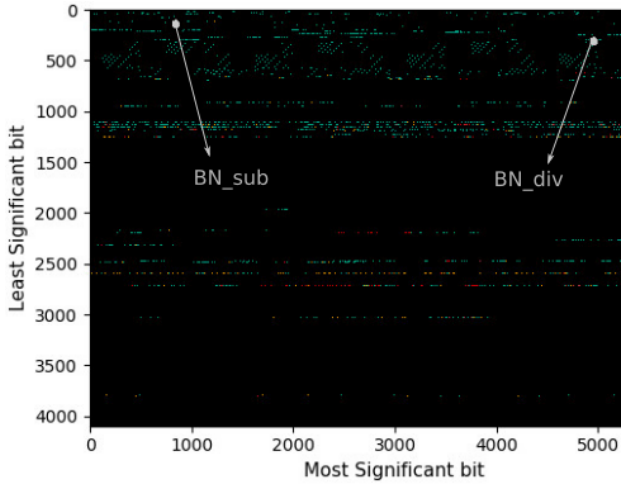

Fig. 3. Cartography of faults. The white color represents the locations of the exploitable faults. Other colors are a classification of errors including unexpected exit values, Linux signals SIGILL, SIGSEGV and watchdog timeout.

M modulo q in the variable M instead of M_q. This induces a faulted $M' = M \bmod q$ and a propagation of the fault on other variables. This is represented in orange in Alg. 4. Indeed, since M_q does not contain the result of the reduction, one can consider this variable to be faulted with $M'_q \neq M \bmod q$. This in turn leads to a fault on S_q denoted S'_q. The reduction of the message modulo p in step (D.3) in Alg. 4 is also badly computed because the reduction is performed from the faulted message M' instead of the good one. From this last reduction it results an incorrect signature S'_p and leads to an incorrect final signature S'. Since $S'_q \neq S_q$ and $S'_p \neq S_p$ are both incorrect, if the algorithm would return directly after the step (D.6) on the Alg. 4, just before the verification, the Bellcore attack would not work.

However, OpenSSL verifies whether the CRT-RSA is computed correctly by checking whether $(S^e - M) = 0$ that becomes $(S'^e - M') = 0$ with the fault.

Due to this last fault, the following is true:

$$\begin{cases} S'^e = ({S'_q}^e \bmod q) = (M'_q \bmod q). \\ M' = M \bmod q. \end{cases}$$

Moreover, one can remark the following implication

$$S'^e - M' = 0 \implies M'_q - (M \bmod q) = 0.$$

This last equation can be written

$$M'_q - (M \bmod q) \neq 0 \implies S'^e - M' \neq 0.$$

Since $M'_q \neq M \bmod q$, the condition $S'^e - M'$ in step (D.7) in Alg. 4 is always true and OpenSSL then computes $M'^d \bmod n$. Since $M' = M \bmod q$,

$$\begin{cases} S' - S = ((M \bmod q)^d \bmod n) - (M^d \bmod n), \\ S' - S = 0 \bmod q. \end{cases}$$

It results that $q = \gcd(S' - S, n)$.

Note that this is possible since $M \neq (M \bmod q) \bmod p$. Indeed, if this was not true, $S' - S$ would also be equal to 0 modulo p and the computation of the greater common divisor would not yield either q or p.

Input : Message M, key (p, q, d, d_p, d_q, i_q)
Output: Signature $M^d \bmod n$

1 (D.1) ~~M_q~~ $M' = M \bmod q$ [M_q is replaced by M]
2 [$M_q \neq M \bmod q$]
3 (D.2) $S'_q = M'^{d_q}_q \bmod q$
4 (D.3) $M'_p = M' \bmod p$
5 (D.4) $S'_p = M^{d_p}_p \bmod q$
6 (D.5) $S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q)) \bmod p)$
7 (D.6) $S'_e = S'^e \bmod n$
8 (D.7) **if** $S'_e - M' \neq 0$ *[True condition]* **then**
9 (D.8) $S' = M'^d \bmod n$
10 [$S' - S = (M \bmod q)^d \bmod n - M^d \bmod n$]
11 [$S' - S = 0 \bmod q$]
12 [$q = \gcd(S' - S, n)$, as per Eqn. (1)]
13 **return** S'

Algorithm 4: CRT-RSA (D.1∼5) with verification (D.6∼8). Red color and lightning indicate the location of the fault. The orange color indicates the variables that are affected by the fault. In blue, some lines are commented.

As explained in Sec. 3.5 the modular reduction is computed using the `BN_div` function. Without the fault, this function is called by

`BN_div(NULL,`M_q`,`M`,`q`,ctx).`

The fault then changes this last call by

`BN_div(NULL,`M`,`M`,`q`,ctx).`

According to Fig. 1, the second parameter is passed from `rsa_ossl_mod_exp` to `BN_div` by using the following assembly instructions.

```
mov -0x50(%rbp),%rax,
mov %rax,%rsi.
```

Note that `-0x50(%rbp)` and `-0x70(%rbp)` are respectively the address of the variable M_q and M. Thus the fault changes the previous assembly instructions to

```
mov -0x70(%rbp),%rax,
mov %rax,%rsi.
```

In a C level, since the variable that contains the message M is located at offset -0x70 from the address pointed by %rbp register and since M_q is located at offset -0x50 from the address pointed by this same register, the faulted replaced $M_q = M \bmod q$ by $M = M \bmod q$. All then behave as the message to sign is $M \bmod q$ instead of M pointed by %rbp register that makes the attacks work as described above.

5.2 Fault in `BN_sub` function

In this section the second fault is explained. This fault is also used to both restore the Bellcore attack and break the OpenSSL protection against this attack. This fault occurs at bit offset 147 from the beginning of the function `BN_sub`.

As illustrated in step (E.5) and (E.7) of Alg. 5, the fault takes place at the two locations where a subtraction operation occurs.

Input : Message M, key (p, q, d, d_p, d_q, i_q)
Output: Signature $M^d \mod n$

1 (E.1) $M_q = M \bmod q$
2 (E.2) $S_q = M_q^{d_q} \bmod q$
3 (E.3) $M_p = M \bmod p$
4 (E.4) $S_p = M_p^{d_p} \mod q$
5 (E.5) ↯ $S' = S_q + q \cdot (i_q \cdot (S_p - S_p \cancel{S_q}) \mod p)$
6 $[S' = S_q]$
7 (E.6) $S'_e = S'^e \bmod n$
8 (E.7) **if** ↯$(S'_e - S'_e \cancel{M}) \neq 0$ *[Always False]* **then**
9 (E.8) $S = M^d \bmod n$ [Never reached]
10 **return** S'
11 $[S' - S = S_q - S]$
12 $[S' - S = 0 \bmod q]$
13 $[q = \gcd(S' - S, n),$ as per Eqn. (1)$]$

Algorithm 5: CRT-RSA (E.1 to E.5) with verification (E.6 and E.7).Red color and lightning indicate the location of the fault. The orange color indicates the variables that are affected by the fault. In blue, some lines are commented.

More specifically, the fault works by transforming this operation to the null function(i.e., the function always returning 0) by replacing the second operand of the subtraction by the first one.

The first call of the subtraction operation is in the Garner's recombination in which the subtraction $S_p - S_q$ is faulted to $S_p - S_p = 0$ that leads to a faulted signature $S' = S_q$. The attacker can therefore compute $q = \gcd(S' - S, n)$.

The second one takes place in the OpenSSL protection against CRT-RSA fault attacks. Indeed, OpenSSL checks whether $S^e - M \neq 0$ in order to verify if the CRT-RSA is badly computed. Due to the fault, this subtraction is replaced by $S'^e - S'^e$ which is always equal to zero. By this way, OpenSSL fails to detect the fault.

As explained in Sec. 3.5, OpenSSL uses the `BN_sub` function in order to make the subtraction operation. For the first call, the fault consists in modifying the function

$$\texttt{BN\_sub}(S_p, S_p, S_q),$$

to

$$\texttt{BN\_sub}(S_p, S_p, S_p).$$

As described in 3.5, the third argument of `BN_sub` is retrieved by this function by getting the value of `rdx` register. Moreover, looking at Fig 2, one can note that the register `rsi` used for passing the second parameter to `BN_sub` has the same value than the register `rax`.

In assembly level the fault is then performed by modifying a register in the instruction that read the third argument:

```
mov %rdx,-0x38(%rbp),
```

to

```
mov %rax,-0x38(%rbp).
```

Note that one can directly modify the register `rdx` to `rsi`. However, this requires more than one bit flip which is out of topic of this paper.

Since the fault is performed inside the `BN_sub` function, and not in the caller function, the fault also applies in the same way to the subtraction that takes place in the OpenSSL protection (lines (E.7) and (E.8) of Alg. 5).

5.3 Overview of complete attack

A complete attack scenario is described below:

- **Step 1: Attack preparation phase**. The adversary characterizes which bits lead to exploitable effects if they are reset. This *identification phase* (in Common Criteria wording) is the main contribution of this paper.
- **Step 2: Correct signatures collection**. The attacker collects signatures for the RSA keys he intends to break next, in **step 5**.

– **Step 3: RowHammer**. The attacker installs its rowhammer code and runs it on the addresses identifies in **step 1**. The code is in assembly language; pseudo-code for this hammering is given in Alg. 6. Notice that this step is probabilistic, since it may fail. In this case, OpenSSL server will malfunction and the system administrator will have to restart it, and the attacker restarts **step 3**, until success (as instructions are 4-bytes wide, few attempts are needed).
– **Step 4: Attack**. If Alg. 6 returns success, then the attacker has successfully installed an APT (*Advanced Persistent Threat*), that is a permanent fault in the code (until restart of OpenSSL). Thus the attacker quickly launches signatures, collects there erroneous results, and executes the GCD attack of Eqn. (1) using genuine signatures collected at **step 2**.

Input : Address add where to reset one bit
Input : Genuine code instr at address add at DRAM row r
Input : Intended mutated code malicious_instr at add (see **step 1**)
Output: Status success or failure of hammering attempt

```
1 while True do
2     if *add = instr then
3         for 10,000 times do
4             Read DRAM row r-1
5             Read DRAM row r+1
6     else if *add = malicious_instr then
7         return success
```

Algorithm 6: Complete RSA key extraction from OpenSSL.

Note that the rowhammer attack requires cache eviction to force the CPU to read data in DRAM instead of the cache.

6 Fault on CRT-RSA implementation of OpenSSL compiled for ARM processors

The original aim of our work was to see whether OpenSSL cryptographic library was correctly protected against fault in a monobit erase model. For that, Intel processors were first investigated. Even whether the exploitable faults described in the previous section can be explained in the `C` language and can then be performed on any processors, the exploitability of these fault on our monobit erase model highly depend on assembly instructions. On ARM processor, one can always do the same fault but not not necessary by flipping only one bit.

The same research is done on ARM processor to check whether new exploitable faults can be found in our monobit erase model.

More specifically, Raspberry Pi 3 with Raspbian operating system were used for its ARM cortex-A53 processor. Interestingly, more faults on ARM processors were found compared to Intel processor. Here, those faults are described.

6.1 Fault in `BN_sub` call

One deterministic fault. From our analysis on ARM processors, the same fault that the one found at subsection 5.2 was found. Since the instruction are different, the only difference from 5.2 is that the fault modify

```
STR R2, [FP, #-32] (instead of mov %rdx,-0x38(%rbp))
```

to

```
STR R0, [FP, #-32] (instead of mov %rax,-0x38(%rbp))
```

Five probabilistic faults. In this section five probabilistic faults are described. At first, four similar faults are explained

Four probabilistic faults that affect a register

The `BN_sub` function computes the difference $r = b - a$ by first computing the absolute value of it. Thus, according if $a < b$ is true or not, it computes $r = b - a$ or $r = a - b$. Indeed, as shown in listing 1.2, the comparison is performed by using the function `BN_ucmp(a,b)` that lead the register `R1` to contain the value of variable `b`. Then, according to the listing 1.2, `BN_usub` is used to compute $r = b - a$ if $a < b$ or $r = a - b$ if not. In the second case, this last function uses the registers `R0`, `R1` and `R2` to pass parameters `r`, `a`, `b` respectively:

```
LDR R2, [FP, #-32]
LDR R1, [FP, #-28]
LDR R0, [FP, #-24]
```

The aim of each of the four faults is to change only one bit in the last assembly instruction `LDR R0, [FP, #-24]` to one of the following one:

- `LDRVS R1, [FP, #-28]`
- `TST FP, IP, LSL R0`
- `STR R1, [FP, #-28]`
- `LDR R0, [FP, #-28]`

All of those instructions keep the register `R1` unchanged. Since R1 contains the variable `b` because of the `BN_ucmp` function, the fault then transforms the call `BN_usub(r, a, b)` to `BN_usub(r, b, b)` which is equal to 0 and have then the same effect of the deterministic fault.

Those faults are probabilistic since the fault occurs at the `BN_usub` function that performs the subtraction $r = a - b$ where $a > b$ and not at the one that computes $r = b - a$.

Lemma 1. *Without blinding, the probability that the fault is exploitable is* $\frac{1}{2}(1-\frac{q+1}{2p})$.

Proof. Since the `BN_sub` function is called to compute the difference $S_p - S_q$ in the Garner's recombination and the difference $S^e \bmod N - M$ in the protection, the fault works when $S_p > S_q$ and $S^e \bmod N > M$.

The probability that each of the probabilistic faults work should be

$$Pr(success) = Pr(S_p > S_q \; and \; S^e \bmod N > M)$$

Since the two conditions are independent, it results that:

$$Pr(success) = Pr(S_p > S_q) \times Pr(S^e \bmod N > M)$$

Note that $Pr(S^e \bmod N > M) = \frac{1}{2}$ since $S^e \bmod N$ can be viewed as an RSA encryption and then result in an uniformly random value lower than N. Moreover, $Pr(S_p > S_q)$ can be computed as following

$$\begin{aligned} Pr(S_p > S_q) &= Pr(\{S_q = 0 \; and \; S_p > 0\}) && (2)\\ &+ Pr(\{S_q = 1 \; and \; S_p > 1\}) && (3)\\ &+ ... && (4)\\ &+ Pr(\{S_q = q-1 \; and \; S_p > q-1\}) && (5) \end{aligned}$$

$$\begin{aligned} Pr(S_p > S_q) &= \frac{p-1}{pq} + \frac{p-2}{pq} + ... + \frac{p-q}{pq} && (6)\\ &= 1 - \frac{q+1}{2p} && (7) \end{aligned}$$

The fault should then be exploitable with a probability of $\frac{1}{2}(1-\frac{q+1}{2p})$.

□

Lemma 2. *With blinding, the probability that the fault is exploitable is* $\frac{1}{4}(1-\frac{q+1}{2p})$.

Proof. This can be explained because of the blinding. Indeed, the signature is computed over the blinded message $M_b = Mr^e \bmod N$ and the final signature is returned after reverting this blinding. Then, $S = Sr^{-1} \bmod N$.

r^{-1} is computed using the extended Euclid algorithm that can give a negative number. In such case, OpenSSL returns the positive representation by computing the difference between the modulus N and r^{-1} computed by the extended Euclid algorithm. To do that, OpenSSL call `BN_sub(`r^{-1}`,`N`,`r^{-1}`)`. Since, $N > r^{-1}$, this function returns 0 because of the fault and the final signature is then equal to 0, that is not exploitable. Since r^{-1} is negative with a probability of $\frac{1}{2}$, this explains why our experimentation show that these faults appear with a 2 times lower probability than $\frac{1}{2}(1-\frac{q+1}{2p})$.

Note that r^{-1} is also negative with probability $\frac{1}{2}$ on the previous deterministic fault. However, this last fault transforms `BN_sub(r,a,b)` to `BN_sub(r,a,r)` that is equal to zero if the first and the second argument are the same. Thus, the call `BN_sub(`r^{-1}`,`N`,`r^{-1}`)` is not affected and this fault remains deterministic. □

Notice that in Lemma 1 (resp. Lemma 2) the probability of fault is about 1/4 (resp. 1/8), because in RSA, $p, q \gg 1$ and $p/q \approx 1$.

If the probabilistic faults does not permit to retrieve the private key, one can try again with the same message to sign since the message blinding will change the message passed to the modular exponentiation. Without message blinding, an attacker can still change the message.

One probabilistic fault in BN_sub result sign

Depending whether $a < b$ the result $r = a - b$ can obviously be negative. The sign of this result is defined by the `neg` variable of the OpenSSL structure that represents a big number as shown in listing 1.1.

If the condition $a < b$ is false, the result of the subtraction is positive and the variable `neg` is set to 0. If $a < b$ then the result is negative and the variable `neg` is set to 1.

This section focus on a fault that exploits the `neg` variable when it is set to 0 by the C instruction `r->neg = 0` in `BN_usub` function as shown in listing 1.2.

In assembly level, setting `r->neg` to 0 is performed with the following assembly instruction

```
STR R2, [R3, #12]
```

where the register R2 contains the value 0 and where the register `R3` contains the address of the variable `r`. Note that each element of the structure in listing 1.1 is four byte length. The location of the different fields of the variable `r` can be easily determined as follow:

`[r3,#0]` is the field `d` of the variable `r`
`[r3,#4]` is the field `top` of the variable `r`
`[r3,#8]` is the field `dmax` of the variable `r`
`[r3,#12]` is the field `neg` of the variable `r`
`[r3,#16]` is the field `flags` of the variable `r`.

The fault consists in flipping one bit in the last previous assembly instruction in order to transform the instruction

```
STR R2, [R3, #12] to STR R2, [R3, #4]
```

At C level this fault acts as changing `r->neg = 0` to `r->top = 0` in listing 1.2.

Since `r->top = 0` is one case in which OpenSSL considers the number `r` to be equal to zero, the BN_sub function returns 0. The effect of this fault is then the same of the previous faults.

This fault is also probabilistic since it targets the case where the condition $a < b$ is not true.

6.2 Fault in `rsa_ossl_mod_exp`

Three deterministic faults on register. In our analysis on OpenSSL library compiled for ARM processor, three deterministic faults were found. These faults

are quite similar that the one described in subsection 5.1. In this section, the differences between this latter subsection and the three deterministic faults on OpenSSL compiled for ARM are detailed.

As explained in 5.1, `BN_div` function is called to reduce the message modulo `q`. The two first parameters of this function are passed from `rsa_ossl_mod_exp` by using the following assembly instructions.

```
LDR R1, [FP, #-20]
MOV R0, #0
```

Note that `[FP, #-20]` is the address of the variable M_q.

The fault flips only one bit over one of the previous assembly instructions to change register `R1` to `R0`. The previous instructions are then replaced by the following ones.

```
LDR R0, [FP, #-20]
MOV R0, #0
```

Note that other instructions such as `LDRVS R1, [FP, #-20]`, that is a conditional memory read that is performed if the overflow flag is not set, or `TST FP, R4, LSL R0` can also be obtained by flipping only one bit instead of `LDR R0, [FP, #-20]`

The fault does not change the register `R1` which then still contains the variable I(that represents the message) because of the `BN_with_flags` function as explained previously. In C level, the fault changes `BN_div(NULL,r1,c,rsa->q,ctx)` to `BN_div(NULL,I,c,rsa->q,ctx)` resulting in a faulted message that leads to a leak of a prime factor by exploiting the protection.

Two faults on execution flow. This section explains another single bit fault that bypasses OpenSSL protection by modifying the normal instruction flow of the RSA signature algorithm.

Inside the function `rsa_ossl_mod_exp`, the signatures S_p and S_q are stored in function variables `m1` and `r0` respectively. Moreover, `r0` is also used as the output variable of the function `rsa_ossl_mod_exp` that computes $S = M^d \bmod N$ using CRT before checking the correctness of the signature. Thus, if the algorithm has returned just after the computation of S_p and S_q, after (B.4) in Alg. 2, before the Garner's recombination and before OpenSSL protection, a wrong signature that is equal to S_q is gotten

The fault then consists in modifying the execution flow to return immediately before the Garner's recombination and before the protection by flipping only one bit.

Since `rsa_ossl_mod_exp` is called by `rsa_ossl_private_encrypt`, the ARM register `LR` contains the return value to the `rsa_ossl_private_encrypt` function where `rsa_ossl_mod_exp` has been called.

The first assembly instruction of the function `rsa_ossl_mod_exp` (`PUSH R4, R11, LR`) pushes registers `LR,R11 and R4`,in this order, on the stack. Note that

the stack grows in the opposite way than the memory addresses. Since all registers are 4 bytes size, pushing a register on the stack also decrements the stack pointer `SP` from 4. Moreover, the second assembly instruction of the function `rsa_ossl_mod_exp` (`ADD R11, SP, #8`) leads the registers `R11` to point to the stack location that contains the `LR` register as illustrated in figure 4. By this way, the register `R11` point to a memory location that contains this return value to `rsa_ossl_private_encrypt`.

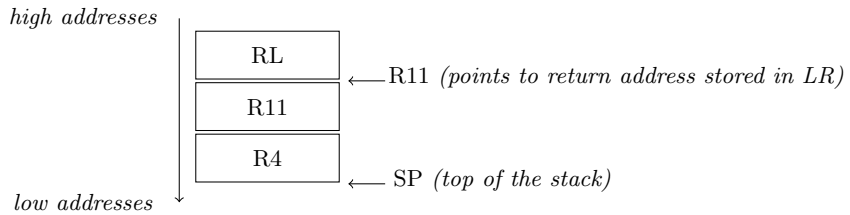

Fig. 4. Memory state at the beginning of `rsa_ossl_mod_exp` function

Note that a `BN_free` function is called just before the Garner's recombination in OpenSSL `rsa_ossl_mod_exp` function. In assembly level, this is performed with the following assembly instruction.

```
BL 9f788 <BN_free>
```

The fault consists in a single bit fault that changes the previous assembly instruction to

```
BL 9f778 <BN_clear_free+0xb0>.
```

This new instruction causes the program to jump to a special location in function `BN_clear_free` that contains the following assembly instructions:

```
SUB SP, R11, #4
POP R11,PC
```

Jumping this way is very similar to ROP attacks.

The first assembly instruction moves the stack pointer as shown in figure 5. This instruction simply modifies the stack pointer. By this way, the second stack element is then the return address to the `rsa_ossl_private_encrypt` function that is copied in the program counter register `PC` by the second assembly instruction.

By this way, the Garner's recombination and the protection are never executed and the signature is equal to the signature modulo p that leaks the factor p.

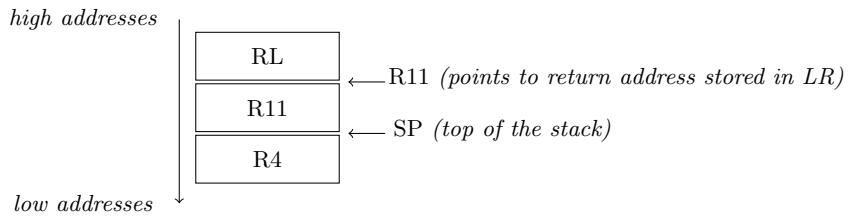

Fig. 5. Memory state at the beginning of `rsa_ossl_mod_exp` function

Similar fault can be done at byte 1761048 bit 5 of OpenSSL binary library file.

7 Discussion

In this section, variants of the described attacks are discussed. Precisely, section 7.1 investigates what kind of additional attacks can be performed when considering bit sets in addition to bit resets. Another cryptographic library, namely Mbedtls, is also analysed in section 7.2, and the analyse studies the impact of compilation options that is discussed in section 7.3. Practical considerations concerning specifically the rowhammer attack are tackled next in section 7.4. Eventually, practical fixes to the broken protection are analyzed in section 7.5.

7.1 Monobit set model - Intel side

In the previous sections, results using monobit erase model were shown. An attacker can obviously use a monobit set model that consists in flipping a bit from 0 to 1. The analysis founds faults of "bit set" type allowing for a Bellcore attack only in `BN_sub` function, that computes a difference $r = b - a$, in such model. Contrary to the "bit reset" fault discussed in Sec. 5.2, the new fault does not always mutate `BN_sub` to the constant function returning 0. Note that this last function internally calls `BN_usub`. This function computes either $b-a$ or $a-b$ depending whether b is larger or smaller than a. Note that, in the call of BN_sub function, the first and the second parameters are identical. Indeed, OpenSSL call `BN_sub(a,a,b)` instead of `BN_sub(r,a,b)`. Then, internally, `BN_sub(a,a,b)` calls either `BN_usub(a,a,b)` or `BN_usub(a,b,a)` according whether $a > b$ or not.

This section details the faults on `BN_usub` for monobit set model on Intel CPUs. First, note that the `BN_usub` function parameters are passed in two steps. Firstly, the arguments are copied on registers `rdx`, `rcx` and `rax`. Secondly, in order to follow the calling convention, registers `rcx` and `rax` are copied in registers `rsi` and `rdi` respectively.

Upon each of the two steps, two faults were found in monobit set model. The first one consists in setting a bit in the first instruction in order to modify the

instruction `mov -0x38(%rbp),%rdx` to `mov -0x28(%rbp),%rdx`. By this way, the first parameter and the last one turn out to become identical. In the case where $a > b$, the first and the second parameters are identical and then this fault have the function to return 0, irrespective of the input parameters.

The second one consists in preventing from copying the second argument of `BN_usub` to `rsi` register by replacing the instruction `mov %rcx,%rsi` by `mov %rcx,%r14`. By this way, the register `rsi` contains the value of the second argument of function called in the previous function. This previous function is `BN_cmp(a,b)`. Then, in the case where $a > b$, the fault actually transforms the function to `BN_usub(a,b,b)` that always returns 0. Therefore, the same attack as discussed in Sec. 5.2 is possible in this case, which occurs with probability $\frac{1}{4}(1 - \frac{q+1}{2p})$ as described with the lemmai 1.

7.2 Mbedtls - Intel side

Exploitable fault in erase and set monobit fault model in Mbedtls libraries on Intel CPUs were also looked for. In this respect, Mbedtls has been compiled using the default configuration at the exception that it was configured to build a shared library instead of a static one. This has no influences on the generated code but it allows us to speed up our research. Indeed, a static libraries require a recompilation at each bit modification. Testing Mbedtls with monobit erase model and monobit set model takes us less than one day. Contrary to OpenSSL, no exploitable faults were found in our model is possible. Since our methodology tries to find monobit fault in an exhaustive way, Mbedtls can then be considered as correctly protected against monobit fault attack. Compared to OpenSSL, this can be explained by two reasons.

Firstly, the fault on OpenSSL `BN_sub` function is possible since the first two parameters are the same as described in the previous sections. Indeed, `BN_sub(`S_p`,`S_p`,`S_q`)` is modified to `BN_sub(`S_p`,`S_p`,`S_p`)` by copying the first argument to the third one. Because the second argument and the third one are identical, it behaves like the third argument to be equal to the first one. Mbedtls does not overwrite parameters and is then not vulnerable to this kind of fault.

Secondly, Mbedtls does not try to correct a badly computed signature. In OpenSSL, this correction is, indirectly, responsible for the fault described in the previous section.

Note that even if Mbedtls is protected against monobit faults on RSA signature using PKCS1.5 padding, a multibit model fault is still possible but difficult. Indeed one can, for example, uses

7.3 Influence of the compilation optimization

Our experiments were performed on the OpenSSL version compiled by ourselves. More precisely the optimization level -O3 of gcc compiler is used. That is the default option in OpenSSL configuration scripts. Since each fault consists in flipping bit in an instruction, one can wonder if compilation optimization level can influence the faults that can be found. However, other optimization levels

were tested and the same faults were found. This is due to the necessity of the compiler to respect the calling convention. Furthermore, the two functions `BN_mod` and `BN_sub` are very frequent, hence are not inlined.

7.4 Practical considerations of Rowhammer attack

Difficulties to be overcome. This paper focus on the simulation of rowhammer attack in order to find vulnerabilities that can be exploited later on. Our methodology can actually be applied to any kind of fault injection techniques (see examples in Tab. 4). One can subsequently focus on practically injecting the fault that triggers the vulnerabilities identified. This section details practical considerations to perform a rowhammer attack in practice.

Table 4. FLASH fault perturbation means explained in the literature

Fault injection mean	Reference
RowHammer (transient fault)	[8]
Wear-out (permanent fault) owing to excessive usage	[5]
Laser fault injection	[11]
X-ray fault injection	[1]

First of all, one has to know the mapping between a virtual address and a DRAM structure that includes the DRAM channel, DRAM rank, DRAM bank and DRAM row. This translation is performed by the MMU (Memory Management Unit) that first translates the virtual to physical addresses and then to the DRAM structure. Before the version 4.0 of Linux kernel, it was possible for an attacker to know the mapping between virtual and physical addresses because the Operating System knew this translation and exposed it through an API accessible at user level. This is no longer possible as a protection against memory attacks. Moreover, the translation from the physical address to the DRAM structure is often only performed by MMU and the operating system has no information about it. Peter Pessl *et al.* [19] provides a response to this first issue. They exploit the fact that memory element that are contained in the same channel, rank and bank share the same DRAM row buffer. By using timing attack, they succeed in getting to know the mapping between virtual addresses and DRAM structure. Moreover, they performed physical probes to reverse the way the MMU translates a physical address to the DRAM structure.

Second, since the DRAM is targeted, the cache must be bypassed. On Intel processors, `clflush` addresses this issue. One can also use a cache set eviction strategy. Incidentally, this was used by Lipp *et al.* [18] to perform cache timing attacks on ARM CPUs.

Third, some DRAM capacitor are not vulnerable to Rowhammer. Since this paper focused on monobit faults, an attacker has to align a vulnerable cell with

the bit that he wants to flip. Since the minimal size of memory one can allocate is the page size, it is not always possible to align a DRAM cell with a desired specific bit of the targeted library. Moreover, in user space, the attacker cannot control where the page is allocated in DRAM. However one can increase the probability to make this alignment. Indeed, one can align a huge amount of memory page in DRAM area where capacitor are not sensitive to Rowhammer attack. This way, when the victim tries to load the target library it will be mapped in a sensitive DRAM area. Note that this requires the attacker to be able to allocate memory before the target application loads the cryptographic library.

Solutions to overcome identified problems. The following methodology can then be applied to perform practical rowhammer attack.

First, the attacker has to find a way to perform rowhammer attack. Since rowhammer attack consists in accessing adjacent rows of the target one, an attacker needs to know how to access those rows. Two strategies can be used. The first one consists in allocating a big chunk of memory and containing the target bit. The attacker then accesses two random elements of this allocated memory and checks whether any bit flip is observed. The second strategy consists in finding exactly what virtual addresses are associated with target's adjacent rows. This can be performed as Peter Pessl *et al.* [19] did (recall previous explanation).

Second, an attacker has to find vulnerable cells. One can simply try a rowhammer attack on each DRAM cell to identify them.

Third, one has to find vulnerable bits in the targeted cryptographic library. Those are researched as explained in Sec. 5 and 6.

Finally, one has to align at least one vulnerable cell with one vulnerable bit. This means one has to map the library in a very precise place in DRAM memory. One way to do so is to allocate all the memory and unmap memory pages that contain vulnerable cells. Since the minimal memory allocation is the page, that is usually 4096 bytes, this method does not ensure that one vulnerable bit will be align with one vulnerable cell. However, the most vulnerable bits are in a page, the larger is the probability to have a good alignment. This is particularly true for some cryptosystems like AES where faulting a bit in a large T-box tables could be exploitable using Zhang et al. attack [23]. This assumes that the library will be loaded after the attacker's memory allocation.

7.5 Mitigation

Even though fault attacks are usually complex to thwart in practice, mitigations still exist.

First, data scrubbing can be applied to check whether some instructions in memory are corrupted. This solution consists in a parallel task that periodically looks for errors by comparing potential faulted instructions with a clean version of them. However, this incurs memory and time consumption overheads.

Alternatively, since the faults described in this paper alter target function parameters, using inline functions could be used to mitigate the faults.

Using PSS padding should also be considered to mitigate our faults.

Finally, since the fault on `BN_div` is due to the attempt of OpenSSL to correct it, a recommendation could be to return an error when CRT-RSA was badly computed instead of trying to correct it.

Considering all those mitigations, a patch to OpenSSL developers were proposed.

To mitigate the fault on `BN_sub()` function, `BN_sub(vrfy,vrfy,I)` can be replaced by `BN_cmp(vrfy,I)`. The fault is then well detected since `BN_cmp()` does not use `BN_sub()`. Note that this modification requires the message I to be reduced modulo N. This is done before calling `BN_cmp()` by using `BN_mod()` function.

To mitigate the fault on `BN_div()` function, an error can be returned when a signature has been badly computed instead of trying to correct it. Note that checking whether the corrected signature is well computed is not necessarily efficient if one compares $S^e \bmod N$ with the message since the message is faulted and it will match with the faulted signature. Sensitive variables included badly computed signature and modified message are also cleared. Note that clearing the message require to have a copy of it (that is fault sensitive) since the message in `rsa_ossl_mod_exp()` is declared with `const` flag. With those mitigations, no exploitable monobit fault on RSA signature was found.

8 Conclusion and perspectives

In this paper, multiple new independent fault locations were identified on OpenSSL that restore a Bellcore attack despite the protection.

The two notable results of this paper are that one fault is possible since OpenSSL protection tries to correct a faulted RSA-CRT signature and another fault leads to bypass this protection. Both of those faults tamper with with x86_64 and ARM calling convention that specifies how parameters are given to a function.

In this paper, fault model of our original paper was extended to include monobit set fault model, that consist in flipping bits from 0 to 1, in addition to monobit erase fault model that consists in flipping bits from 0 to 1.

Faults research on OpenSSL and Mbedtls compiled for Intel and ARM processors were performed in this paper.

Other protections against CRT-RSA such as Shamir's protection can be applied in those libraries. Even if these protections can be efficient, their implementation still can be vulnerable and needs to be evaluated thoroughly.

Conflict of interest

The authors declare that they have no conflict of interest with any of the associate editors of the journal.

References

1. Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-Luc Rainard, and Rémi Tucoulou. Nanofocused X-Ray Beam to Reprogram Secure Circuits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2017.
2. Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: concrete results and practical countermeasures. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 260–275, 2002.
3. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370 –382, February 2006.
4. Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *LNCS*, pages 602–624. Springer, 2016.
5. Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In Randal C. Burns and Kimberly Keeton, editors, *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 115–128. USENIX, 2010.
6. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.
7. Éric Brier, David Naccache, Phong Q. Nguyen, and Mehdi Tibouchi. Modulus fault attacks against RSA-CRT signatures. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 192–206, 2011.
8. Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, pages 49–60, 2017.
9. Sébastien Carré, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. Openssl bellcore's protection helps fault attack. In *21st Euromicro Conference on Digital System Design, DSD 2018, Prague, Czech Republic, August 29-31, 2018*, pages 500–507, 2018.
10. Pierre Carru. Attack TrustZone with Rowhammer. `http://www.eshard.com/wp-content/plugins/email-before-download/download.php?dl=9465aa084ff0f070a3acedb56bcb34f5`, 04 2017.
11. Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. *IACR Cryptology ePrint Archive (accepted at HOST 2109 conference*, 2018:1042, 2018.
12. Harvey L. Garner. Number systems and arithmetic. *Advances in Computers*, 6:131–194, 1965.

13. Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*, pages 293–300. IEEE, 2017.
14. Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. Efficient Design and Evaluation of Countermeasures against Fault Attacks Using Formal Verification. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *LNCS*, pages 177–192. Springer, 2015.
15. Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, pages 300–321, 2016.
16. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372, 2014.
17. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
18. 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, Austin, TX, 2016. USENIX Association.
19. Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, 2016. USENIX Association.
20. Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 213–222. IEEE Computer Society, 2014.
21. Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`. Published: March 9, 2015 (also published at `blackhat` 2015).
22. Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. The fault attack jungle - a classification model to guide you. In *Proceedings of the 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '11, pages 3–8, Washington, DC, USA, 2011. IEEE Computer Society.
23. Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent Fault Analysis on Block Ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):150–172, 2018.


Sebastien CARRE is a PhD student at Telecom Paristech. He works on side channel attacks related to timing attacks