SECURE-IC

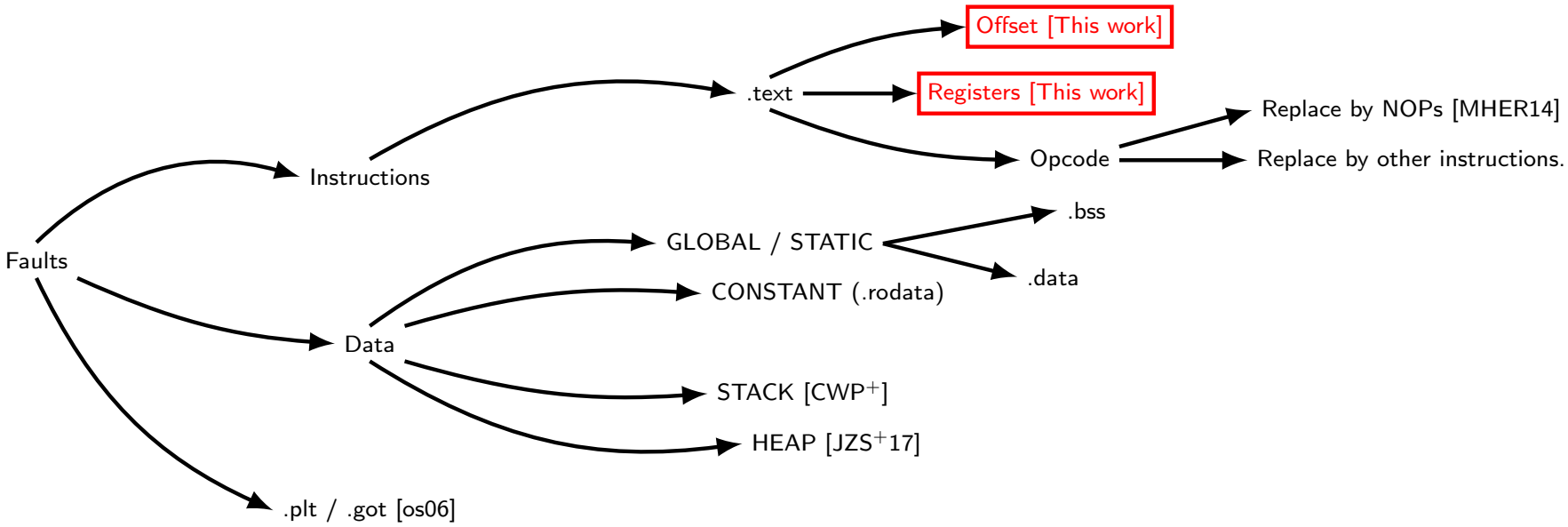
THE SECURITY SCIENCE COMPANY

Using fault attack to break RSA protection on OpenSSL implementation.

Sébastien Carré, Secure-IC and TELECOM-ParisTech

⟨ sebastien.carre@secure-ic.com ⟩

Aims

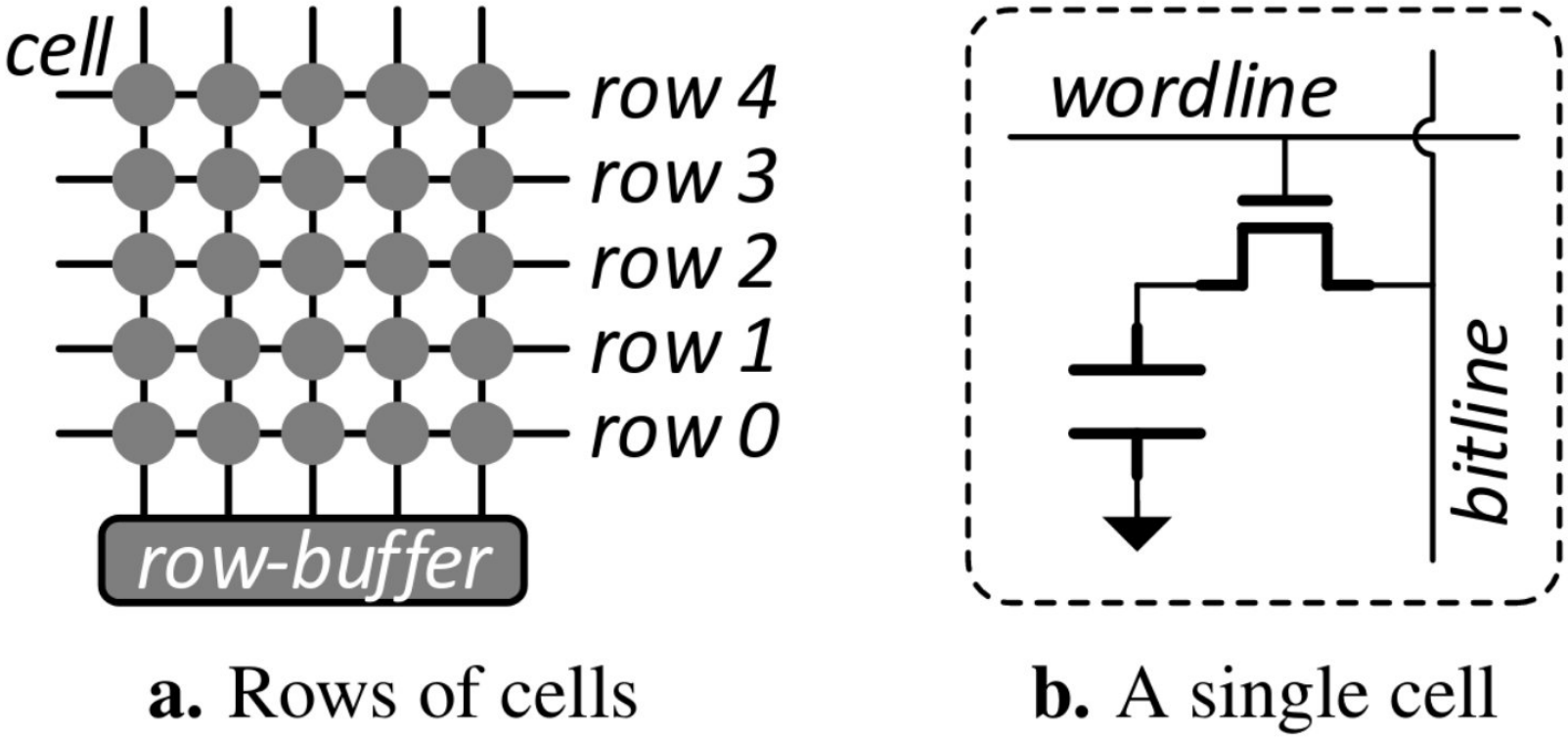


- Two fault locations.
- Each one breaks RSA and its protection in a monobit erase model.
- One of them exists because of the protection.

Outline

1. Background
   - Rowhammer attack
   - ISA and calling convention on x86_64
   - Bellcore attack
2. Faults
   - How do we find the two faults
   - OpenSSL implementation
   - Faults description

Rowhammer



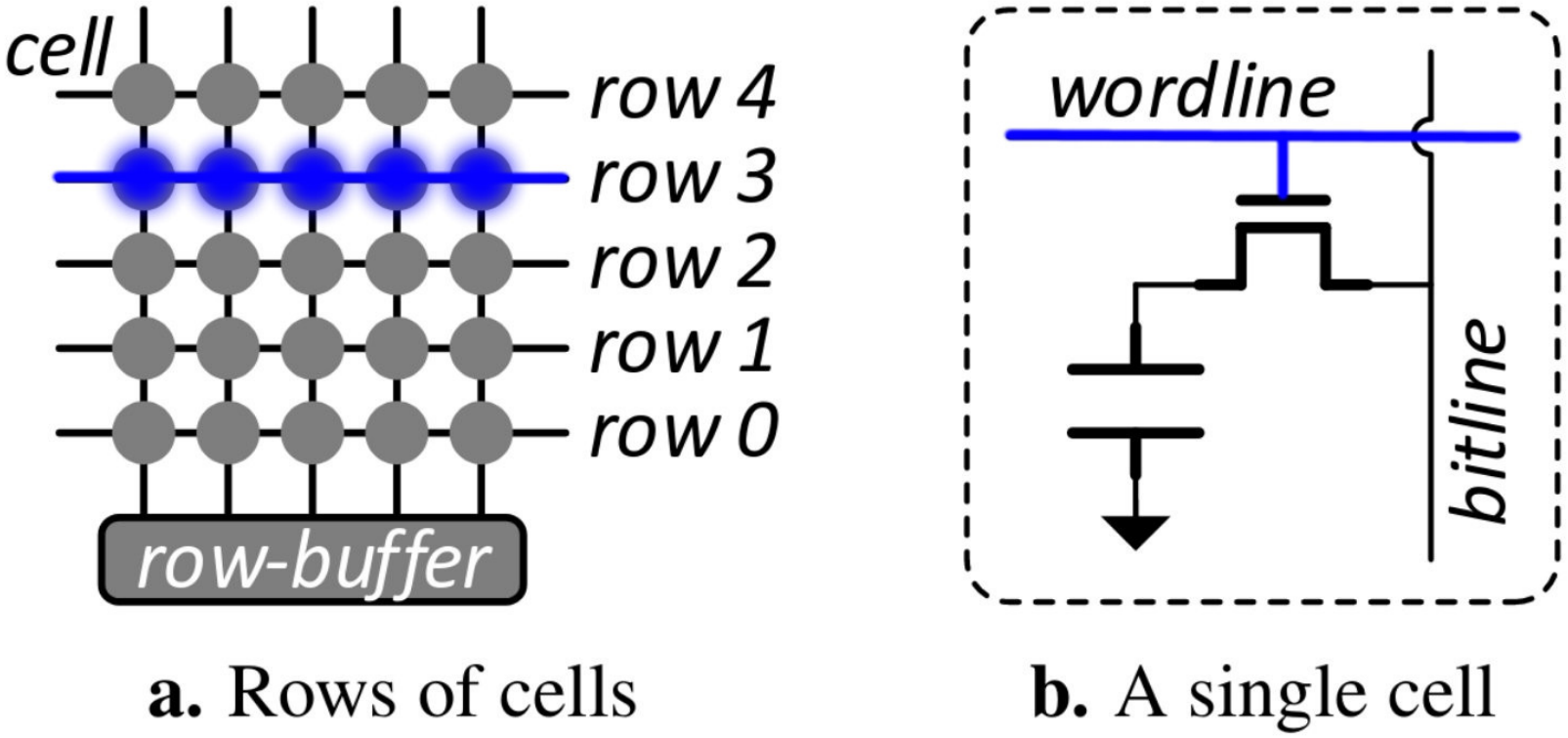
a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

(Kim *et al.*)

Memory refresh is needed periodically.

Rowhammer



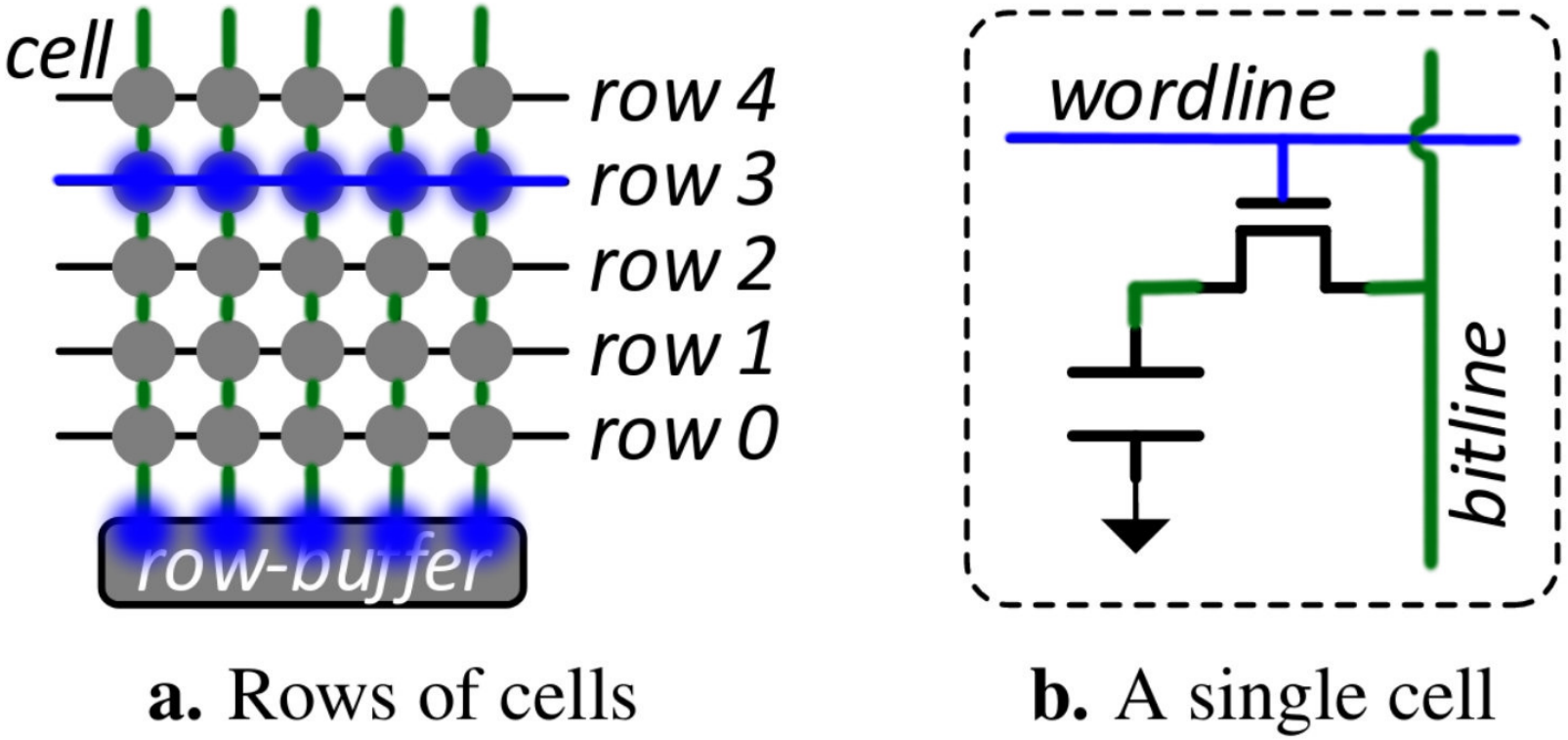
a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

(Kim *et al.*)

Memory refresh is needed periodically.

Rowhammer



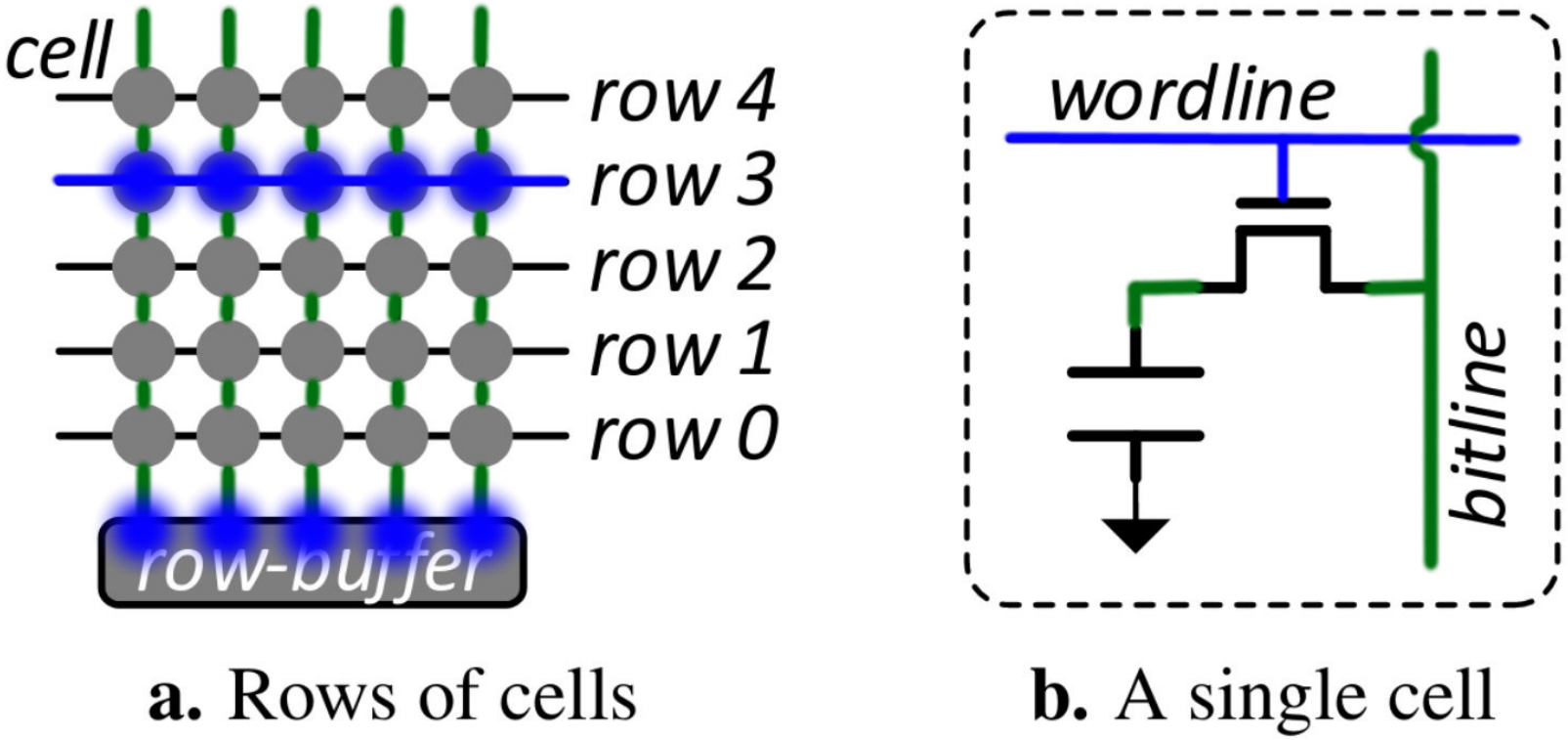
a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

(Kim *et al.*)

Memory refresh is needed periodically.

Rowhammer



a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

(Kim *et al.*)

Memory refresh is needed periodically.

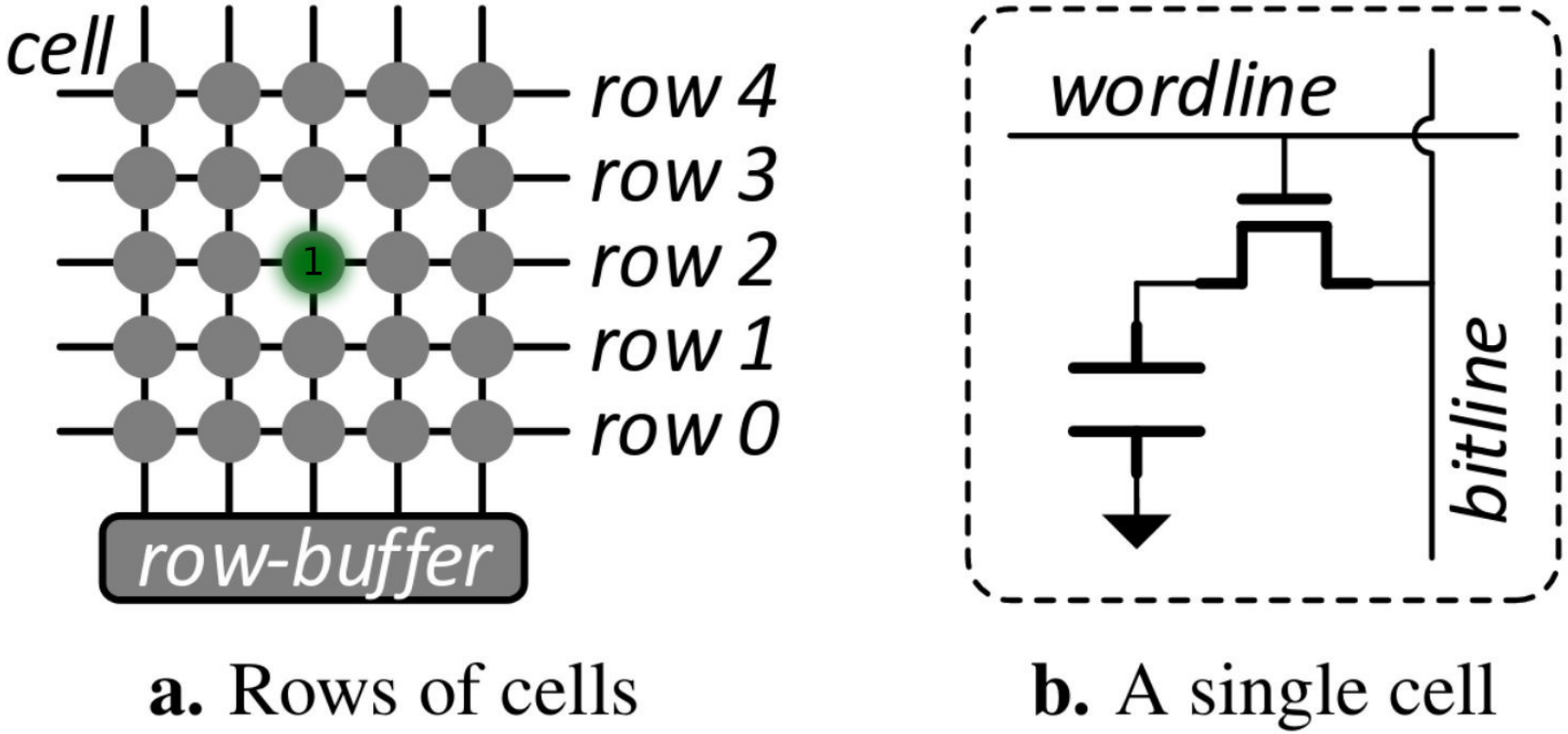
Rowhammer

- Nowadays, DRAMs have a high density of capacitors.
- Discharging adjacent row capacitors.

Only bit erase from 1 to 0.

Rowhammer

- Discharging adjacent row capacitors.



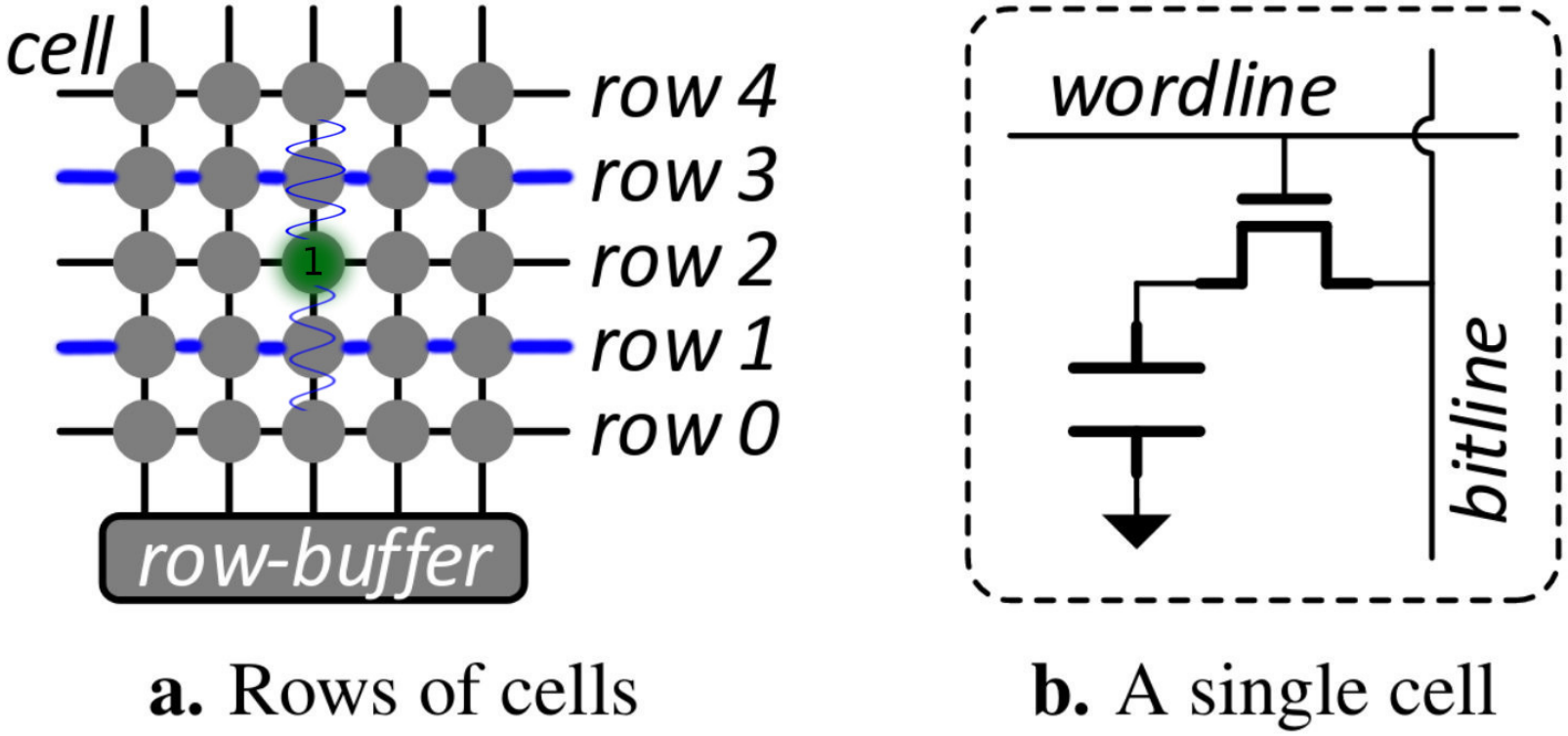
a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

Only bit erase from 1 to 0.

Rowhammer

- Discharging adjacent row capacitors.



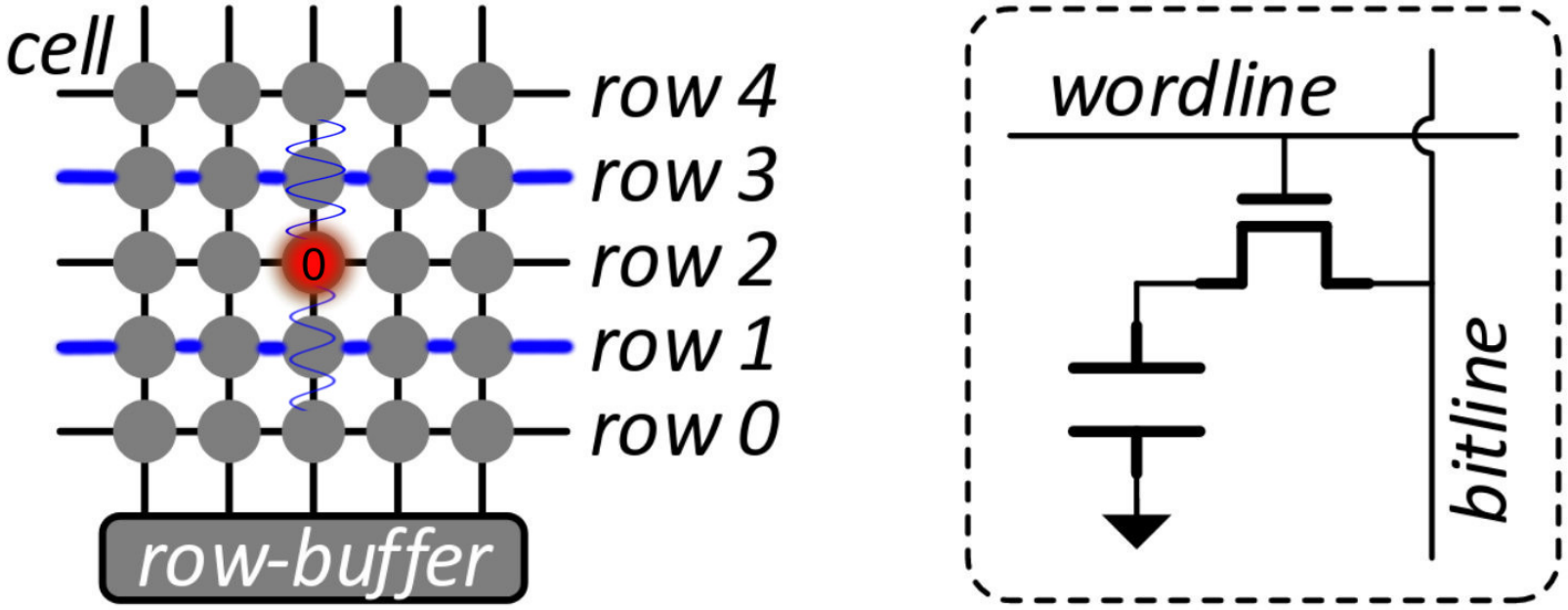
a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

Only bit erase from 1 to 0.

Rowhammer

- Discharging adjacent row capacitors.



a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

Only bit erase from 1 to 0.

Rowhammer

- Nowadays, DRAMs have a high density of capacitors.
- Seaborn *et al.*: Bit flip in PTE to change frame value.
- Our work takes place in a rowhammer attack context.

mov instruction syntax

FE 2C 08 76 79 F0 78 **8B 43 B0** C4 52 5D C6 7A 42 4F A4 7F 8C 5D 7C 8F 55 2B

INSTRUCTION SET

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m*

- **0x8B**: `mov` from memory to register instruction.
- **0x43**: (01 000 011)
  - `01`: Need **01** more byte: **0xB0**. Offset from a base memory address.
  - `000`: Register number (`%rbp`). Contains the base memory address.
  - `011`: Destination register number (`%rax`).

```
mov 0xB0(%rbp),%rax;
```

mov instruction syntax

```
mov 0xB0(%rbp),%rax;
```

- We can make a fault on registers, values, opcode, offset, ...
- Not everywhere: A fault in argument induces a disalignment.
- Instructions are put together within functions following a **calling convention:**
  - 32 bits: Use stack.
  - 64 bits: registers.<**Use this calling conventions on our faults**>
    - rdi: first parameter
    - rsi: second parameter
    - rdx: third parameter
    - rcx: fourth parameter
    - r8: fifth parameter
    - ...

Bellcore attack

- RSA signature with Chinese Remainder Theorem (CRT)

Input : Message M, private key $(p, q, d, i_q = q^{-1} \bmod p)$
Output: Signature $S = M^d \mod n$

1 $S_p = M^d \mod p$. /* Signature mod p */
2 $S_q = M^d \mod q$. /* Signature mod q */
3 $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ /* Garner's formula */

Bellcore attack

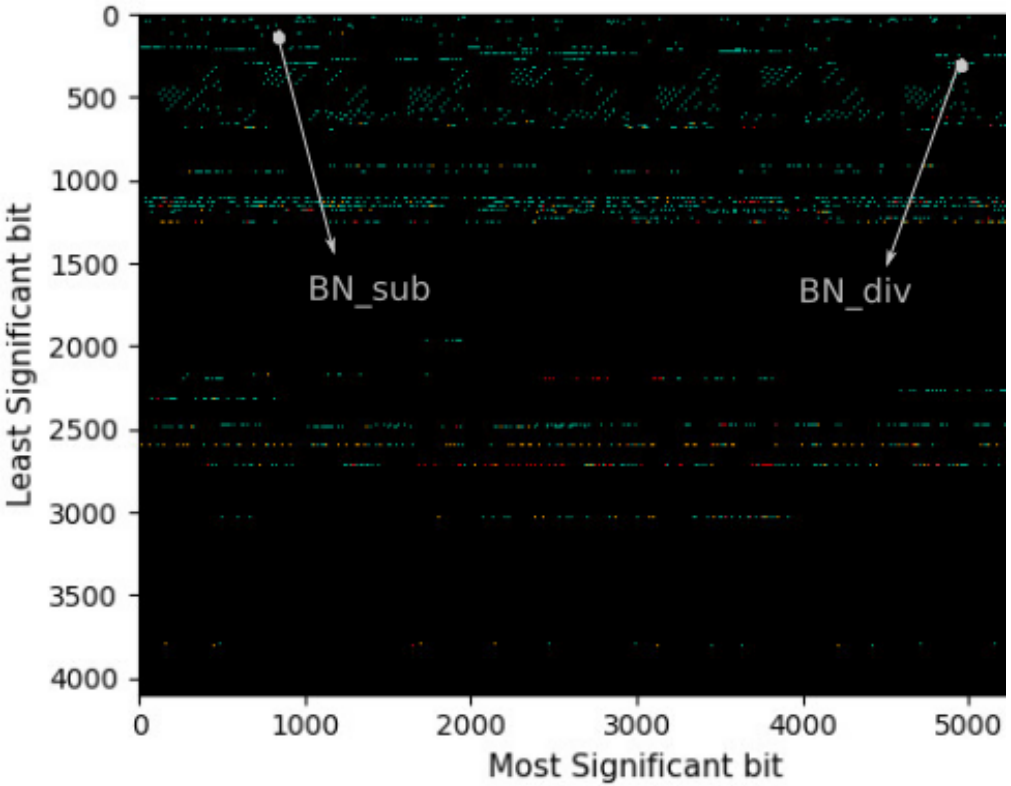
- Needs a correct signature S and a faulted one S'.
- Faults can target data (S_p) as well as instructions (- to + in Garner's formula)
- Faults can be on computation of S_p **or** S_q
- Faults can be on Garner's formula.
- If a fault occurs on S_p, $S - S' = 0 \bmod q$. $q = gcd(S' - S, n)$

Motivations

- Is OpenSSL protected against rowhammer attack? Where can we make an exploitable fault.
- Simulation of rowhammer attack.
  - Flip only one bit from 1 to 0.
  - Bellcore attack class: *gcd* used to test if a fault is exploitable.
  - OpenSSL is not correctly protected. Protection helps an attacker.

Results

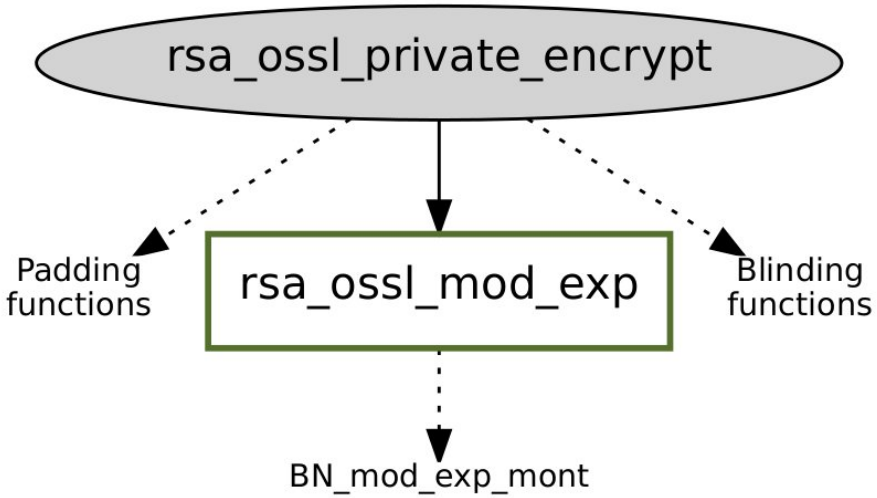
- Error handling
  - Signal handlers to handle segmentation faults, illegal instructions...
  - Watchdog to prevent infinite loop.
- Map of faults of the OpenSSL library (1 pixel = 1 bit)



●	Unexpected exit values.
●	SIGILL
●	SIGSEGV
●	Watchdog timeout

- **Black**: No fault or faults without effets.
- **White**: Exploitable faults.

OpenSSL signature implementation



- Use deterministic PKCS1-v1.5 padding.
- `rsa_ossl_mod_exp()` function computes $S = M^d \bmod n$
- `rsa_ossl_mod_exp()` uses Chinese Remainder Theorem(CRT) to compute S.
  - `BN_sub:` used in Garner's formula $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$
  - `BN_div:` used to reduce the message before modular exponentiations.
- `rsa_ossl_mod_exp()` includes protection against Bellcore faults attacks.
  - Check if signature is correct : $S^e - M = 0$.
  - If $S^e - M \neq 0$ recompute the signature without CRT.

OpenSSL signature implementation: `rsa_ossl_mod_exp()` - OpenSSL protection

- Seems to need at least two faults. One for CRT-RSA and another to break the protection.
- How to break the protection?
- Can we do that with only one bit flip.

BN_sub

- Called in two places
  - In Garner's recombination: `BN_sub(r0,r0,m1)` to compute $S_p - S_q$
  - In Protection: `BN_sub(vrfy,vrfy,I)` to check if $S^e - M = 0 \bmod n$

In `rsa_ossl_mod_exp`

```
mov    -0x48(%rbp),%rdx ;I
mov    -0x68(%rbp),%rcx ;vrfy
mov    -0x68(%rbp),%rax ;vrfy
mov     %rcx,%rsi
mov     %rax,%rdi
callq   0x7ffff775f620 <BN_sub>
```


In `BN_sub` — `BN_sub(vrfy,vrfy, I)`

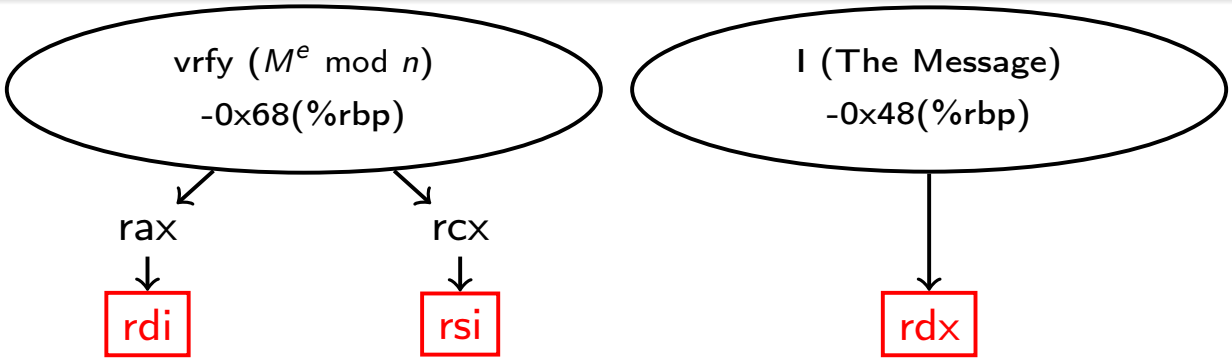
```
mov    %rdi,-0x28(%rbp)
mov    %rsi,-0x30(%rbp)
mov    %rdx,-0x38(%rbp)
```

BN_sub

- Called in two places
  - In Garner's recombination: `BN_sub(r0,r0,m1)` to compute $S_p - S_q$
  - In Protection: `BN_sub(vrfy,vrfy,I)` to check if $S^e - M = 0 \bmod n$

In `rsa_ossl_mod_exp`

```
mov    -0x48(%rbp),%rdx ;I
mov    -0x68(%rbp),%rcx ;vrfy
mov    -0x68(%rbp),%rax ;vrfy
mov     %rcx,%rsi
mov     %rax,%rdi
callq   0x7ffff775f620 <BN_sub>
```



In `BN_sub` — `BN_sub(vrfy,vrfy, I)`

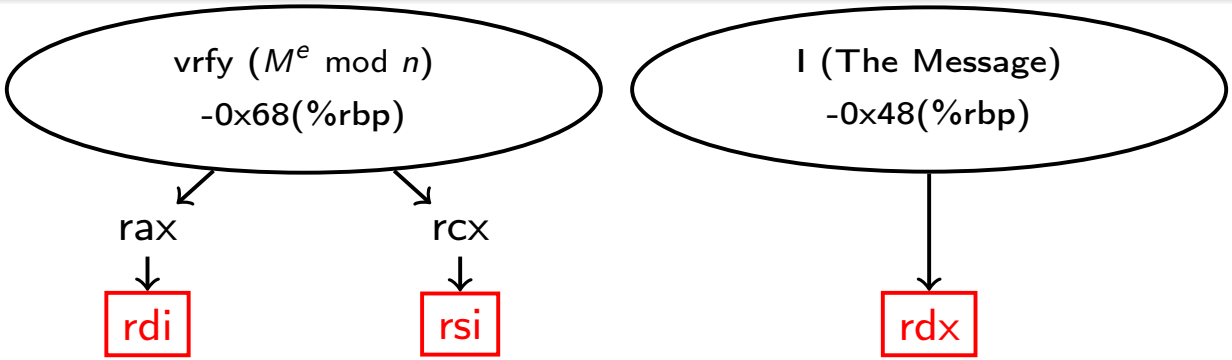
```
mov    %rdi,-0x28(%rbp)
mov    %rsi,-0x30(%rbp)
mov    %rdx,-0x38(%rbp)
```

BN_sub

- Called in two places
  - In Garner's recombination: `BN_sub(r0,r0,m1)` to compute $S_p - S_q$
  - In Protection: `BN_sub(vrfy,vrfy,I)` to check if $S^e - M = 0 \bmod n$

In `rsa_ossl_mod_exp`

```
mov    -0x48(%rbp),%rdx ;I
mov    -0x68(%rbp),%rcx ;vrfy
mov    -0x68(%rbp),%rax ;vrfy
mov     %rcx,%rsi
mov     %rax,%rdi
callq   0x7ffff775f620 <BN_sub>
```



In `BN_sub` — `BN_sub(vrfy,vrfy, I)`

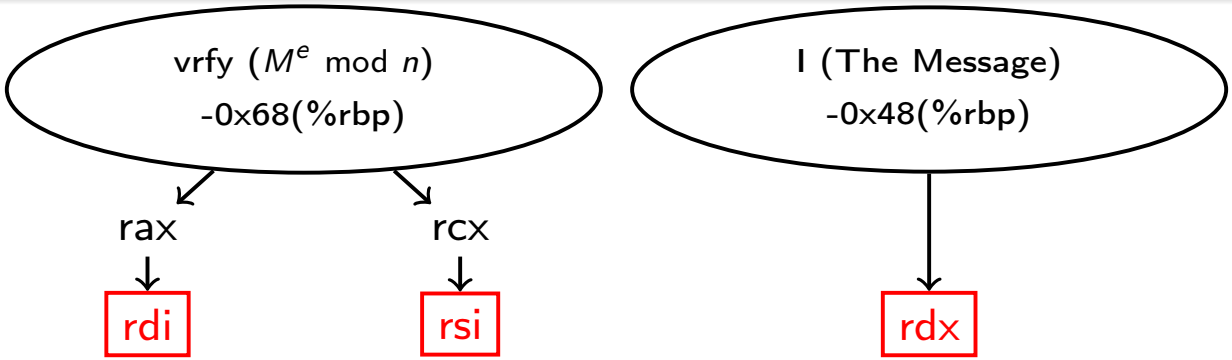
```
mov    %rdi,-0x28(%rbp)
mov    %rsi,-0x30(%rbp)
mov    %rdx,-0x38(%rbp)
```

BN_sub

- Called in two places
  - In Garner's recombination: `BN_sub(r0,r0,m1)` to compute $S_p - S_q$
  - In Protection: `BN_sub(vrfy,vrfy,I)` to check if $S^e - M = 0 \bmod n$

In `rsa_ossl_mod_exp`

```
mov    -0x48(%rbp),%rdx ;I
mov    -0x68(%rbp),%rcx ;vrfy
mov    -0x68(%rbp),%rax ;vrfy
mov     %rcx,%rsi
mov     %rax,%rdi
callq   0x7ffff775f620 <BN_sub>
```



In `BN_sub` — `BN_sub(vrfy,vrfy,I vrfy)` **NULL function**

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

BN_sub

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

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

BN_sub

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

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

BN_sub

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

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

BN_mod

Macro

```
#define BN_mod(rem,m,d,ctx) BN_div(NULL,(rem),(m),(d),(ctx))
```

```
                    BN_mod(r1,c,rsa->q,ctx)
                      ↓  ↓     ↓
BN_div(NULL,r1,c,rsa->q,ctx)
```

Call of BN_div — BN_mod(r1,c,rsa->q,ctx)

```
mov     0x40(%rax),%rcx
mov    -0x80(%rbp),%rsi
mov    -0x28(%rbp),%rdx
mov    -0x50(%rbp),%rax «««««« Second argument
mov    %rsi,%r8
mov    %rax,%rsi
mov    $0x0,%edi
callq   0x7ffff775e790 <bn_div>
```

BN_mod

Macro

```
#define BN_mod(rem,m,d,ctx) BN_div(NULL,(rem),(m),(d),(ctx))
```

```
                BN_mod(r1,c,rsa->q,ctx)
                    ↓  ↓      ↓
BN_div(NULL,r1,c,rsa->q,ctx)
```

Call of BN_div

BN_mod(~~r1~~c,c,rsa->q,ctx) $M = M \bmod q$

```
mov     0x40(%rax),%rcx
mov    -0x80(%rbp),%rsi
mov    -0x28(%rbp),%rdx
mov    -0x70-0x50(%rbp),%rax<<<<<< Second argument
mov    %rsi,%r8
mov    %rax,%rsi
mov    $0x0,%edi
callq   0x7ffff775e790 <bn_div>
```

BN_div

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

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

BN_div

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

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

BN_div

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

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

ARM AArch64

- AArch64 calling convention: use registers.
  - x0: first parameter
  - x1: second parameter
  - x2: third parameter
  - x3: fourth parameter
  - x4: fifth parameter

AArch64 - registers

64-bit registers

X0	X8	X16	X24
X1	X9	X17	X25
X2	X10	X18	X26
X3	X11	X19	X27
X4	X12	X20	X28
X5	X13	X21	X29
X6	X14	X22	X30*
X7	X15	X23	

* procedure_LR

{32-bit SP, 64-bit DP} scalar
FP / 128-bit vectors

V0	V8	V16	V24
V1	V9	V17	V25
V2	V10	V18	V26
V3	V11	V19	V27
V4	V12	V20	V28
V5	V13	V21	V29
V6	V14	V22	V30
V7	V15	V23	V31

	EL0	EL1	EL2	EL3	
Stack Ptr	SP_EL0	SP_EL1	SP_EL2	SP_EL3	(PC)
Exception Link Register		ELR_EL1	ELR_EL2	ELR_EL3	
Saved/Current Process Status Register		SPSR_EL1	SPSR_EL2	SPSR_EL3	(CPSR)

The Architecture for the Digital World® ARM

ARM AArch64

x86_64

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

AArch64

```
mov     x20, x1
ldr     w1, [x1,#16]
stp     x21, x22, [sp,#32]
mov     x19, x2
mov     x21, x0
ldr     w0, [ x2 ,#16]
cbz     w1, 9af38 <BN_sub+0xd0>
mov     w22, #0x1
cbnz     w0, 9aecc <BN_sub+0x64>
mov     x2,x19
mov     x1,x20
mov     x0,x21
```

ARM AArch64

x86_64

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

AArch64

```
mov     x20, x1
ldr     w1, [x1,#16]
stp     x21, x22, [sp,#32]
mov     x19, x2
mov     x21, x0
ldr     w0, [x2,#16]
cbz     w1, 9af38 <BN_sub+0xd0>
mov     w22, #0x1
cbnz     w0, 9aecc <BN_sub+0x64>
mov     x2,x19
mov     x1,x20
mov     x0,x21
```

Mitigations

Mitigations

- Use non-deterministic PSS padding instead of deterministic PKCS 1.5 padding.
- Code scrubbing (More space complexity)
- Good practice: Only detect fault but do not try to correct a faulted message. Instead, simply clear memory.
- Rowhammer attack mitigation: increase DRAM refresh frequency (More power consumption).

Conclusion and perspectives

Conclusion

- OpenSSL implemention of RSA signature is not secure enough against faults.
- Two faults with only one bit flip.
  - First one inside BN_sub function: restore Bellcore attack and break protection with the same fault.
  - Second one on BN_div function parameters passing: Do not restore Bellcore attack, Do no break the protection, exploit the protection to break RSA.

Perspective

- Multibit fault model.
- Other exploitable criteria.
- Going into practice instead of simulation.

- Thank you

Thank you.
Any question?

Bibliographical references I

[CWP+] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole.
Buffer overflows : Attacks and defenses for the vulnerability of the decade *.

[JZS+17] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng.
Towards efficient heap overflow discovery.
In *26th USENIX Security Symposium (USENIX Security 17)*, pages 989–1006, Vancouver, BC, 2017. USENIX Association.

[MHER14] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson.
Formal verification of a software countermeasure against instruction skip attacks.
CoRR, abs/1402.6461, 2014.

[os06] C0ntexb. open security.
How to hijack the global offset table with pointers for root shells.
2006.