NOP-Oriented Programming: Should we Care?

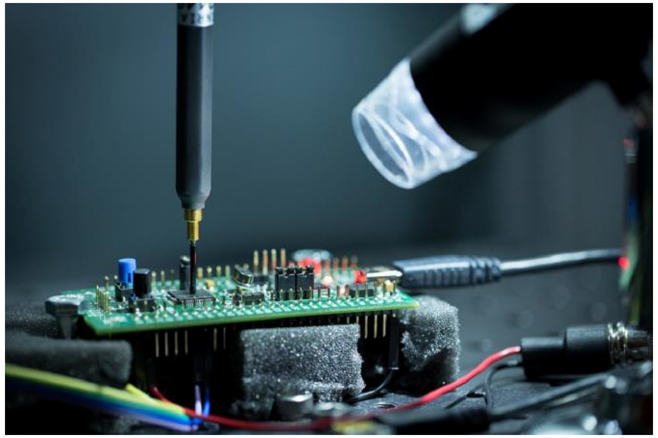
Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, Erven Rohou

Univ Rennes, Inria, CNRS, IRISA

SILM workshop - Friday, September 11th 2020

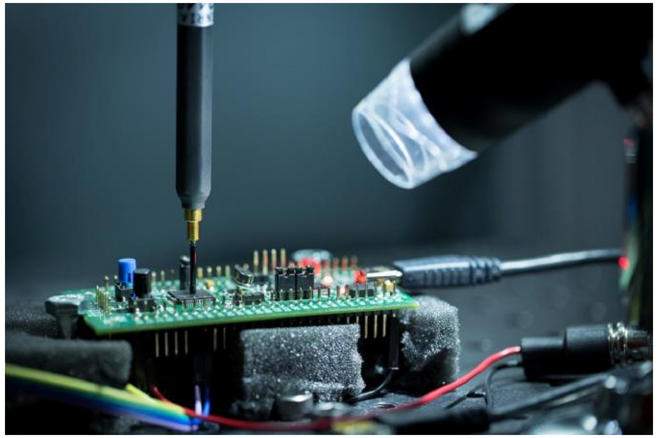
Introduction

- Fault injection nowadays
  - ElectroMagnetic Pulse (EMP)
  - Laser injection
  - Clock glitch
  - …



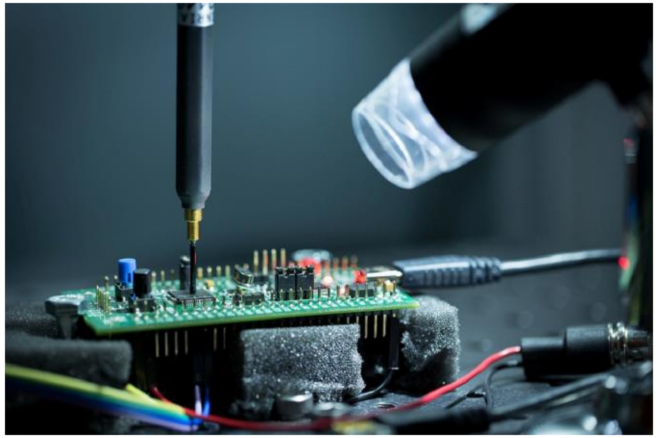
Introduction

- Fault injection nowadays
  - ElectroMagnetic Pulse (EMP)
  - Laser injection
  - Clock glitch
  - …
- Efficient but limited to 1 or few injections



Introduction

- Fault injection nowadays
  - ElectroMagnetic Pulse (EMP)
  - Laser injection
  - Clock glitch
  - …
- Efficient but limited to 1 or few injections
- Lack of precision
  - EMP/laser: unavoidable delay between 2 injections



Approach and questions

- What if an attacker can overcome these limitations?
  - No delay between injection
  - High precision (instruction-level)
  - Unlimited number of faults

- Questions:
  1. What are the possibilities for an attacker?
  2. Can we simulate this?

Fault model : NOP-Oriented programming

- Base fault model: instruction skip*
  - An attacker is able to entirely skips a specific instruction
  - Skipping an instruction replaces this instruction by a NOP

*Chong Hee and Quisquater. "Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures."*International Workshop on Information Security Theory and Practices*, 2007.

Fault model : NOP-Oriented programming

- Base fault model: instruction skip*
  - An attacker is able to entirely skips a specific instruction
  - Skipping an instruction replaces this instruction by a NOP
- Our model: instruction-skip by a factor of hundreds/thousands
  - Program mainly driven by NOP
  - Select which instruction you want to execute

*Chong Hee and Quisquater. "Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures."*International Workshop on Information Security Theory and Practices*, 2007.

Fault model : NOP-Oriented programming

- Base fault model: instruction skip*
  - An attacker is able to entirely skips a specific instruction
  - Skipping an instruction replaces this instruction by a NOP
- Our model: instruction-skip by a factor of hundreds/thousands
  - Program mainly driven by NOP
  - Select which instruction you want to execute
- That’s what we call NOP-Oriented Programming

*Chong Hee and Quisquater. "Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures."*International Workshop on Information Security Theory and Practices*, 2007.

Theoretical analysis

Possibilities with a NOP-Oriented programming model

Assumptions

- The binary contains a minimal set of instructions:
  - load/store
  - move
  - add
  - sub
- The binary is bug-free
- No backdoor is necessary
- ARM instruction set
  - Could be applied to other ISA

Control Flow Hijacking (attack 1)

- Any instruction can be skipped to reach any address
- From an address *A*, any address *A'* where *A'* > *A* can be reached

Control Flow Hijacking (attack 1)

- Any instruction can be skipped to reach any address
- From an address *A*, any address *A'* where *A'* > *A* can be reached

```
@A     : inst. 1
@A+2   : inst. 2
          …
          …
@A'-1  : inst. n-1
@A'    : inst. n
```

Control Flow Hijacking (attack 1)

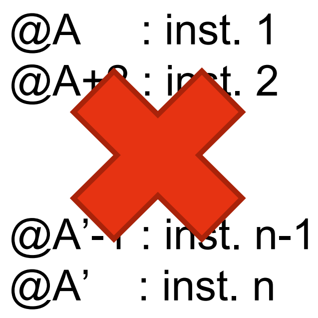
- Any instruction can be skipped to reach any address
- From an address *A*, any address *A'* where *A'* > *A* can be reached

@A : inst. 1
@A+2 : inst. 2

@A'-1 : inst. n-1
@A' : inst. n

Control Flow Hijacking (attack 1)

- Any instruction can be skipped to reach any address
- From an address *A*, any address *A’* where *A’* > *A* can be reached



- With branches, almost any address could be reached
  - Starting from @A, how to reach @B?

Control Flow Hijacking (attack 1)

- Any instruction can be skipped to reach any address
- From an address *A*, any address *A'* where *A'* > *A* can be reached

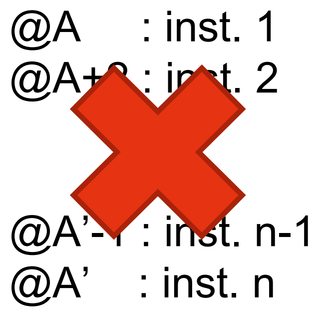
@A : inst. 1
@A+[illegible] : inst. 2
@A'-1 : inst. n-1
@A' : inst. n

- With branches, almost any address could be reached
  - Starting from @A, how to reach @B?

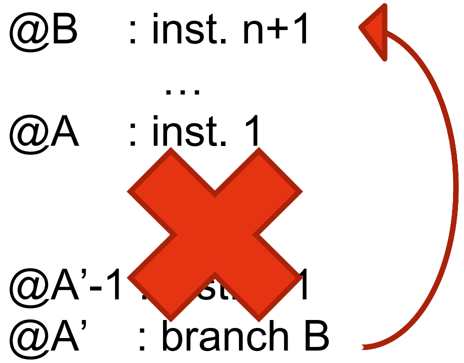
@B : inst. n+1
…
@A : inst. 1
…
…
@A'-1 : inst. n-1
@A' : branch B

Control Flow Hijacking (attack 1)

- Any instruction can be skipped to reach any address
- From an address *A*, any address *A'* where *A'* > *A* can be reached



- With branches, almost any address could be reached
  - Starting from @A, how to reach @B?



Control loop iteration (attack 2)

- Do fewer iterations by:

- Do more iterations by:

```
        …
  mov r4, #10
label:
        …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
        …
```

Control loop iteration (attack 2)

- Do fewer iterations by:
  - Replace entire body by NOP

```
        …
   mov r4, #10
label:
        …
   su        #1
   cmp
   bne abe

endloop:
        …
```

- Do more iterations by:

Control loop iteration (attack 2)

- Do fewer iterations by:
  - Replace entire body by NOP
  - NOP the conditional branch at the end and exit
- Do more iterations by:

```
        …
  mov r4, #10
label:
        …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
        …
```

Control loop iteration (attack 2)

- Do fewer iterations by:
  - Replace entire body by NOP
  - NOP the conditional branch at the end and exit
- Do more iterations by:
  - NOP the instruction which controls the loop condition

    Typically a subtraction on a counter

…
mov r4, #10
label:
…
~~sub r4, r4, #1~~
cmp r4, 0
bne label

endloop:
…

Control loop iteration (attack 2)

- Do fewer iterations by:
  - Replace entire body by NOP
  - NOP the conditional branch at the end and exit

- Do more iterations by:
  - NOP the instruction which controls the loop condition
    Typically a subtraction on a counter
  - NOP the compare instruction
    This relies on the current state of the control flags

…
mov r4, #10
label:
…
sub r4, r4, #1
~~cmp r4, 0~~
bne label

endloop:
…

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

```
    …
    mov r0, #0
    mov r4, #10
label:
    …
    add r0, r0, #1
    …
    mov r3, r0
    …
    sub r4, r4, #1
    cmp r4, 0
    bne label

endloop:
    …
```

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

1. Use a controlled loop (attack 2)
   - We control the number of iterations

```
    …
    mov r0, #0
    mov r4, #10
label:
    …
    add r0, r0, #1
    …
    mov r3, r0
    …
    sub r4, r4, #1
    cmp r4, 0
    bne label

endloop:
    …
```

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

1. Use a controlled loop (attack 2)
   - We control the number of iterations
2. Use a register *Rs* whose content is controlled

```
      …
   mov r0, #0
   mov r4, #10
label:
      …
   add r0, r0, #1
      …
   mov r3, r0
      …
   sub r4, r4, #1
   cmp r4, 0
   bne label

endloop:
      …
```

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

1. Use a controlled loop (attack 2)
   - We control the number of iterations
2. Use a register *Rs* whose content is controlled
3. Use a move instruction from *Rs* into *Rd*

```
    …
  mov r0, #0
  mov r4, #10
label:
    …
  add r0, r0, #1
    …
  mov r3, r0
    …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
    …
```

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

1. Use a controlled loop (attack 2)
   - We control the number of iterations
2. Use a register *Rs* whose content is controlled
3. Use a move instruction from *Rs* into *Rd*
4. Exit the loop (attack 1)

```
        …
    mov r0, #0
    mov r4, #10
label:
        …
    add r0, r0, #1
        …
    mov r3, r0
        …
    sub r4, r4, #1
    cmp r4, 0
    bne label

endloop:
        …
```

Write any possible value in a register (attack 3)

- This relies on the presence of
  - Instruction(s) to increment a register
  - move or load

1. Use a controlled loop (attack 2)
   - We control the number of iterations
2. Use a register *Rs* whose content is controlled
3. Use a move instruction from *Rs* into *Rd*
4. Exit the loop (attack 1)

```
        …
  mov r0, #0
  mov r4, #10
label:
        …
  add r0, r0, #1
        …
  mov r3, r0
        …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
        …
```

- This can be extended to a set of registers (see paper)

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store

```
        …
    mov r0, #0
    mov r1, #0
    mov r4, #10
label:
        …
    add r0, r0, #1
    add r1, r1, #1
        …
    sub r4, r4, #1
    cmp r4, 0
    bne label

endloop:
        …
        …
    ldr r1, [r0]
        …
        …
    str r1, [r0]
```

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store
- Both registers content are controlled (attack 3)

```
    …
  mov r0, #0
  mov r1, #0
  mov r4, #10
label:
    …
  add r0, r0, #1
  add r1, r1, #1
    …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
    …
    …
  ldr r1, [r0]
    …
    …
  str r1, [r0]
```

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store
- Both registers content are controlled (attack 3)

- To read in memory, reach a load that uses *Rm*
  - No need of *Rs* for reading

…
mov r0, #0
mov r1, #0
mov r4, #10
label:
…
add r0, r0, #1
add r1, r1, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
ldr rX, [r0]
…
…
str r1, [r0]

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store
- Both registers content are controlled (attack 3)

- To read in memory, reach a load that uses *Rm*
  - No need of *Rs* for reading
- To write in memory, reach a store that uses *Rm* and *Rs*

…
mov r0, #0
mov r1, #0
mov r4, #10
label:
…
add r0, r0, #1
add r1, r1, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
ldr
…
…
str r1, [r0]

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store
- Both registers content are controlled (attack 3)

- To read in memory, reach a load that uses *Rm*
    - No need of *Rs* for reading
- To write in memory, reach a store that uses *Rm* and *Rs*
    - If no such instruction, use other registers with move (attack 3)

…
mov r0, #0
mov r1, #0
mov r4, #10
label:
…
add r0, r0, #1
add r1, r1, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~
endloop:
…
ldr
…
str r1, [r0]

Load & store from a register (attack 4)

- *Rm* represents a memory address
- *Rs* represents a value to store
- Both registers content are controlled (attack 3)

- To read in memory, reach a load that uses *Rm*
  - No need of *Rs* for reading
- To write in memory, reach a store that uses *Rm* and *Rs*
  - If no such instruction, use other registers with move (attack 3)

- This can be extended to a set of registers (see paper)

```
   …
  mov r0, #0
  mov r1, #0
  mov r4, #10
label:
   …
  add r0, r0, #1
  add r1, r1, #1
   …
  sub r4, r4, #1
  cmp r4, 0
  bne label

endloop:
   …
  ldr r
   …
   …
  str r1, [r0]
```

Jump to any address (attack 5)

- *Rd* represents the destination of a branch

```
        …
    mov r0, #0
    mov r4, #10
label:
        …
    add r0, r0, #1
        …
    sub r4, r4, #1
    cmp r4, 0
    bne label

endloop:
        …
        …
        …
    blx r0
```

Jump to any address (attack 5)

- *Rd* represents the destination of a branch
- *Rd* is a controlled register

…
mov r0, #0
mov r4, #10
label:
…
add r0, r0, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
…
…
blx r0

Jump to any address (attack 5)

- *Rd* represents the destination of a branch
- *Rd* is a controlled register

- Find an unconditional branch to *Rd*:
  blx *Rd*

…
mov r0, #0
mov r4, #10
label:
…
add r0, r0, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
…
…
blx r0

Jump to any address (attack 5)

- *Rd* represents the destination of a branch
- *Rd* is a controlled register

- Find an unconditional branch to *Rd*:
  blx *Rd*
- Execute

…
mov r0, #0
mov r4, #10
label:
…
add r0, r0, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
blx r0

Jump to any address (attack 5)

- *Rd* represents the destination of a branch
- *Rd* is a controlled register

- Find an unconditional branch to *Rd*:
  blx *Rd*
- Execute

- Use the stack:
  push *Rd*
  pop *pc*

…
mov r0, #0
mov r4, #10
label:
…
add r0, r0, #1
…
~~sub r4, r4, #1~~
cmp r4, 0
~~bne label~~

endloop:
…
blx r0

Summary of possibilities

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

Summary of possibilities

Direct dependency

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

Summary of possibilities

Direct dependency

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

Summary of possibilities

Direct dependency

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

Summary of possibilities

Direct dependency

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

Summary of possibilities

Direct dependency

1) CFG Hijacking
2) Control loop iteration
3) Control register(s) content

4) Load/Store from register(s)
5) Jump to any address

This is Turing-Complete (proof in the paper)

Application to (almost) real life

NOP-Oriented programming in a nutshell

Disclaimer

- We present two attacks:
  1) How to retrieve the encryption key used in AES
  2) How to write user-defined data in memory
- However, this is **not specific to AES**

*Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH computer architecture news 39.2 (2011).

Disclaimer

- We present two attacks:
  1) How to retrieve the encryption key used in AES
  2) How to write user-defined data in memory
- However, this is **not specific to AES**

- We only need a minimum set of instructions
- Our target: **ARM embedded systems**
  - No memory protection

*Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH computer architecture news 39.2 (2011).

Disclaimer

- We present two attacks:
  1) How to retrieve the encryption key used in AES
  2) How to write user-defined data in memory
- However, this is **not specific to AES**

- We only need a minimum set of instructions
- Our target: **ARM embedded systems**
  - No memory protection

- Realised in the **gem5* simulator**
  - Replay fault model has been implemented
  - Few attacks tested on real board

*Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH computer architecture news 39.2 (2011).

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**
  - N is the size of the instruction buffer
  - N = 1 in our experiments

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**
  - N is the size of the instruction buffer
  - N = 1 in our experiments
- Limits the attacker
  - cannot repeat a PC-relative load for example

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**
  - N is the size of the instruction buffer
  - N = 1 in our experiments
- Limits the attacker
  - cannot repeat a PC-relative load for example

```
ldr  r0, [pc, #-32]
add  r0, r0, r1
```

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**
  - N is the size of the instruction buffer
  - N = 1 in our experiments

- Limits the attacker
  - cannot repeat a PC-relative load for example

```
ldr   r0, [pc, #-32]        ldr   r0, [pc, #-32]
add   r0, r0, r1       →    ldr   r0, [pc, #-32]
```

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Adaptation of the fault model

- Theory: skipping an instruction **has no side-effect**
  - NOP-Oriented Programming
- Experienced fault model*: skipping an instruction **repeats the N previous ones**
  - N is the size of the instruction buffer
  - N = 1 in our experiments

- Limits the attacker
  - cannot repeat a PC-relative load for example

ldr *r0*, [*pc*, #-32]
add *r0*, *r0*, *r1*

→

ldr *r0*, [*pc*, #-32]
ldr *r0*, [*pc*, #-32]

→

Possible side-effect on *r0*

*Riviere, Lionel, et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

Base program

```
memset(cipher, 0, BUF_SIZE);
sprintf(plain, "%s", "thisisaplaintext");
sprintf(key, "%s", "0123456789ABCDEF");
```

Init phase

Base program

```
memset(cipher, 0, BUF_SIZE);
sprintf(plain, "%s", "thisisaplaintext");
sprintf(key, "%s", "0123456789ABCDEF");

AESEncrypt(cipher, plain, key);
printf("%s\n", cipher);
```

Init phase

Compute phase

Base program

```
memset(cipher, 0, BUF_SIZE);
sprintf(plain, "%s", "thisisaplaintext");
sprintf(key, "%s", "0123456789ABCDEF");

AESEncrypt(cipher, plain, key);
printf("%s\n", cipher);
```

Init phase

Compute phase

- Goal: retrieve the key

Attack #1 : get the encryption key (1/2)

- AESEncrypt(cipher, plain, key);
  → key is in r2 (function call convention)

Attack #1 : get the encryption key (1/2)

- AESEncrypt(cipher, plain, key);
  → key is in r2 (function call convention)

- printf("%s\n", cipher);
  → String to print is in r1 (function call convention)

Attack #1 : get the encryption key (1/2)

- AESEncrypt(cipher, plain, key);
  → key is in r2 (function call convention)
- printf("%s\n", cipher);
  → String to print is in r1 (function call convention)
- Idea: move r2 into r1, then call printf()

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
```

1) Move r2 into r1
2) Call printf()

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

1) Move r2 into r1
2) Call printf()

key is in r2

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
```

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

1) Move r2 into r1
2) Call printf()

key is in r2

Store at $fp-248

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
```

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

1) Move r2 into r1
2) Call printf()

key is in r2

Store at $fp-248

Load into r1

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
```

1: inst.
to reach

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

1) Move r2 into r1
2) Call printf()

key is in r2

Store at $fp-248

Load into r1

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

1: inst.
to reach

Attack #1 : get the encryption key (2/2)

1) Move r2 into r1
2) Call printf()

```
<abort>:
        ...
<main>:
        ...
```

key is in r2

Store at $fp-248

Load into r1

```
<Encrypt>:
105d0: push  {fp, lr}
105d4: add   fp, sp, #4
105d8: sub   sp, sp, #248
105dc: str   r0, [fp, #-240]
105e0: str   r1, [fp, #-244]
105e4: str   r2, [fp, #-248]
105e8: sub   r3, fp, #24
105ec: ldr   r1, [fp, #-248]
105f0: mov   r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

1: inst. to reach

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
```

1) Move r2 into r1
2) Call printf()

key is in r2

Store at $fp-248

Load into r1

```
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

1: inst. to reach

2: nop burst

Attack #1 : get the encryption key (2/2)

1) Move r2 into r1
2) Call printf()

```
<abort>:
        ...
<main>:
        ...

<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

key is in r2

Store at $fp-248

Load into r1

1: inst. to reach

2: nop burst

3: jump

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
10594: ldr r2, [fp, #-16]
10598: ldr r1, [fp, #-8]
1059c: ldr r0, [fp, #-12]
105a0: bl  105d0 <Encrypt>
105a4: ldr r1, [fp, #-12]
105a8: ldr r0, [pc, #28]
105ac: bl 17cc4 <_IO_printf>
        ...
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

key is in r2

Store at $fp-248

Load into r1

4: nop burst

3: jump

1: inst. to reach

2: nop burst

1) Move r2 into r1

2) Call printf()

Attack #1 : get the encryption key (2/2)

<abort>:
...
<main>:
...
10594: ~~ldr r2, [fp, #-16]~~
10598: ~~ldr r1, [fp, #-8]~~
1059c: ~~ldr r0, [fp, #-12]~~
105a0: ~~bl 105d0 <Encrypt>~~
105a4: ~~ldr r1, [fp, #-12]~~
105a8: ldr r0, [pc, #28]
105ac: bl 17cc4 <_IO_printf>
...
<Encrypt>:
105d0: push {fp, lr}
105d4: add fp, sp, #4
105d8: sub sp, sp, #248
105dc: str r0, [fp, #-240]
105e0: str r1, [fp, #-244]
105e4: str r2, [fp, #-248]
105e8: sub r3, fp, #24
105ec: ldr r1, [fp, #-248]
105f0: ~~mov r0, r3~~
...
<__assert_fail_base>:
...
11ef4: bl 10170 <abort>

key is in r2

Store at $fp-248

Load into r1

4: nop burst

3: jump

1: inst. to reach

2: nop burst

1) Move r2 into r1
2) Call printf()

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
10594: ldr r2, [fp, #-16]
10598: ldr r1, [fp, # 8]
1059c: ldr r0, [fp, #-12]
105a0: bl  105d0 <Encrypt>
105a4: ldr r1, [fp, #-12]
105a8: ldr r0, [pc, #28]
105ac: bl 17cc4 <_IO_printf>
        ...
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

key is in r2

Store at $fp-248

Load into r1

4: nop burst

3: jump

1: inst. to reach

2: nop burst

1) Move r2 into r1
2) Call printf()

- Two bursts of nops are necessary

Attack #1 : get the encryption key (2/2)

```
<abort>:
        ...
<main>:
        ...
10594: ldr r2, [fp, #-16]
10598: ldr r1, [fp, #-8]
1059c: ldr r0, [fp, #-12]
105a0: bl  105d0 <Encrypt>
105a4: ldr r1, [fp, #-12]
105a8: ldr r0, [pc, #28]
105ac: bl 17cc4 <_IO_printf>
        ...
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]
105e8: sub  r3, fp, #24
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...
<__assert_fail_base>:
        ...
11ef4: bl 10170 <abort>
```

key is in r2

Store at $fp-248

Load into r1

1: inst. to reach

2: nop burst

3: jump

4: nop burst

1) Move r2 into r1
2) Call printf()

- Two bursts of nops are necessary
- More generic attacks to retrieve the key are presented in the paper

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

```
memset(cipher, 0, BUF_SIZE);
AESEncrypt(cipher, plain, key);
printf("%s\n", cipher);
```

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init →

```
memset(cipher, 0, BUF_SIZE);
AESEncrypt(cipher, plain, key);
printf("%s\n", cipher);
```

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init →
Attack →

```
memset(cipher, 0, BUF_SIZE);
AESEncrypt(cipher, plain, key);
printf("%s\n", cipher);
```

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init →	memset(cipher, 0, BUF_SIZE);
Attack →	AESEncrypt(cipher, plain, key);
Display →	printf("%s\n", cipher);

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init →	`memset(cipher, 0, BUF_SIZE);`
Attack →	`AESEncrypt(cipher, plain, key);`
Display →	`printf("%s\n", cipher);`

- How? Take control of a loop to:

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init → memset(cipher, 0, BUF_SIZE);
Attack → AESEncrypt(cipher, plain, key);
Display → printf("%s\n", cipher);

- How? Take control of a loop to:
  1) Re-create the address of cipher in register *Rm*

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init → memset(cipher, 0, BUF_SIZE);
Attack → AESEncrypt(cipher, plain, key);
Display → printf("%s\n", cipher);

- How? Take control of a loop to:
  1) Re-create the address of cipher in register *Rm*
  2) Set the decimal value for a character in register *Rs*

Attack #2: Write custom data in memory (in theory)

- Idea: hijack cipher buffer to write custom data
  → ASCII characters in this example

Init → memset(cipher, 0, BUF_SIZE);
Attack → AESEncrypt(cipher, plain, key);
Display → printf("%s\n", cipher);

- How? Take control of a loop to:
  1) Re-create the address of cipher in register *Rm*
  2) Set the decimal value for a character in register *Rs*
  3) Find a store instruction that use *Rm* and *Rs*

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()

<AESEncrypt>:

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()

```
<AESEncrypt>:
    …
    mov r0, #0        // the memory address
    mov r1, #0        // the ASCII value
    …
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()

```
<AESEncrypt>:
        …
  mov r0, #0          // the memory address
  mov r1, #0          // the ASCII value
        …
loop:






  cmp
  bne loop
endloop:
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary

```
<AESEncrypt>:
        …
    mov r0, #0          // the memory address
    mov r1, #0          // the ASCII value
        …
loop:




    cmp
    bne loop
endloop:
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary

```
<AESEncrypt>:
        …
   mov r0, #0          // the memory address
   mov r1, #0          // the ASCII value
        …
loop:
        …
   add r0, r0, #1      // Executed N times, with N = @cipher


   cmp
   bne loop
endloop:
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary

```
<AESEncrypt>:
        …
  mov r0, #0          // the memory address
  mov r1, #0          // the ASCII value
        …
loop:
        …
  add r0, r0, #1      // Executed N times, with N = @cipher
  add r1, r1, #1      // Executed 72 times (‘H’ character)
        …              // then always skipped
  cmp
  bne loop
endloop:
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary
3. Exit the loop

```
<AESEncrypt>:
        …
  mov r0, #0          // the memory address
  mov r1, #0          // the ASCII value
        …
loop:
        …
  add r0, r0, #1      // Executed N times, with N = @cipher
  add r1, r1, #1      // Executed 72 times (‘H’ character)
        …             // then always skipped
  cmp
  bne loop
endloop:
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary
3. Exit the loop
4. Store the value

```
<AESEncrypt>:
        …
  mov r0, #0          // the memory address
  mov r1, #0          // the ASCII value
        …
loop:
        …
  add r0, r0, #1      // Executed N times, with N = @cipher
  add r1, r1, #1      // Executed 72 times (‘H’ character)
        …              // then always skipped
  cmp
  bne loop
endloop:
        …
  str r1, [r0]
```

Attack #2: Write custom data in memory (in practice)

- Illustration with “Hello World!”

1. Call AESEncrypt()
2. Do as many iteration as necessary
3. Exit the loop
4. Store the value
5. Restart

```
<AESEncrypt>:
        …
  mov r0, #0          // the memory address
  mov r1, #0          // the ASCII value
        …
loop:
        …
  add r0, r0, #1      // Executed N times, with N = @cipher
  add r1, r1, #1      // Executed 72 times (‘H’ character)
        …              // then always skipped
  cmp
  bne loop
endloop:
        …
  str r1, [r0]
```

Conclusion

Let’s try to summarize and answer our questions

Summary

- Q1. What are the possibilities for an attacker?
  - Hijack CFG
  - Control registers
  - Write in memory

  → The attacker executes what he wants (full control)

Summary

- Q1. What are the possibilities for an attacker?
  - Hijack CFG
  - Control registers
  - Write in memory

  → The attacker executes what he wants (full control)

- Q2. Can we simulate this fault model?
  - gem5 simulator
    - Setup available at https://gitlab.inria.fr/gem5-nop/gem5
    - Simulator modifications + all source code and binaries
  - Successfully retrieve a key with AES
  - Successfully altered memory

Future Works

- More realistic use-cases
  - Proof of concept
  - Extends to real applications (embedded OS?)
- Fault model in gem5 has to be enhanced (on-going internship)
  - More realistic fault model (replay more than one inst.)
- Propose countermeasures
  - Hardware (could depend on how the injection is made)
  - Software (HW independent)

Thank You!

NOP-Oriented Programming: Should we Care?

Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, Erven Rohou