Instruction Duplication: Leaky and Not Too Fault-Tolerant!

Lucian Cojocar (VU)
<l.cojocar@vu.nl>
Kostas Papagiannopoulos (RU)
<k.papagiannopoulos@cs.ru.nl>
Niek Timmers (Riscure)
<timmers@riscure.com>

CARDIS 2017

VU VRIJE UNIVERSITEIT AMSTERDAM Radboud Universiteit riscure Challenge your security

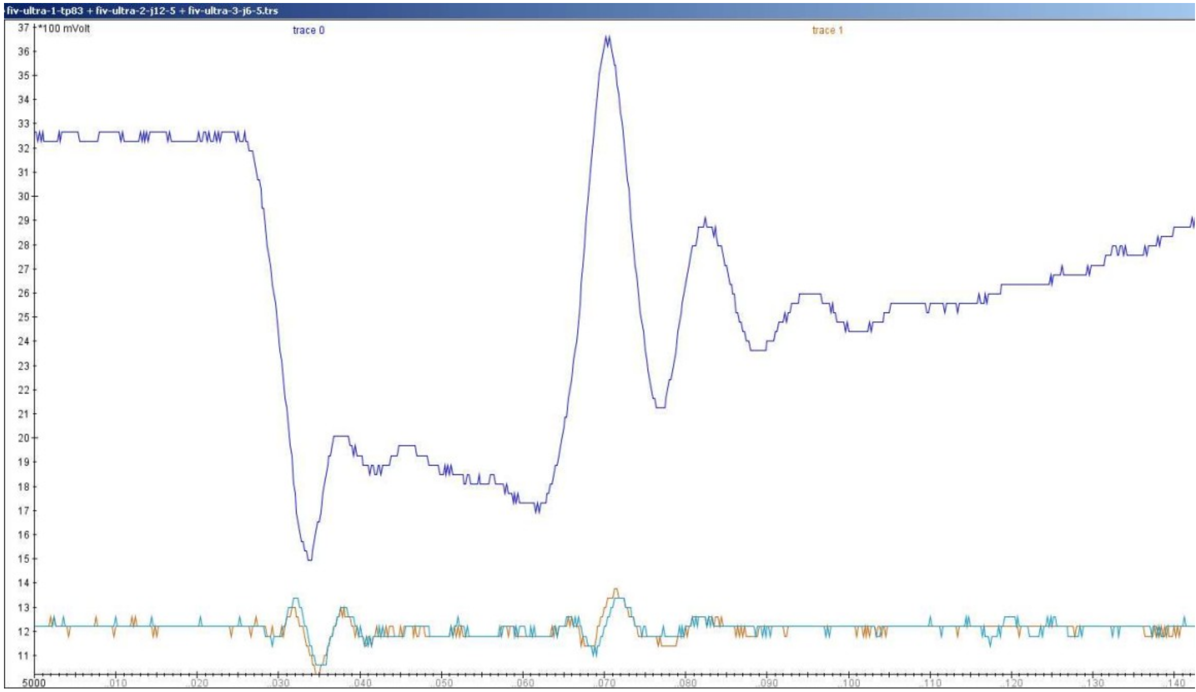
Software defenses …
to protect hardware

It might just work …

Fault injection (FI) definition

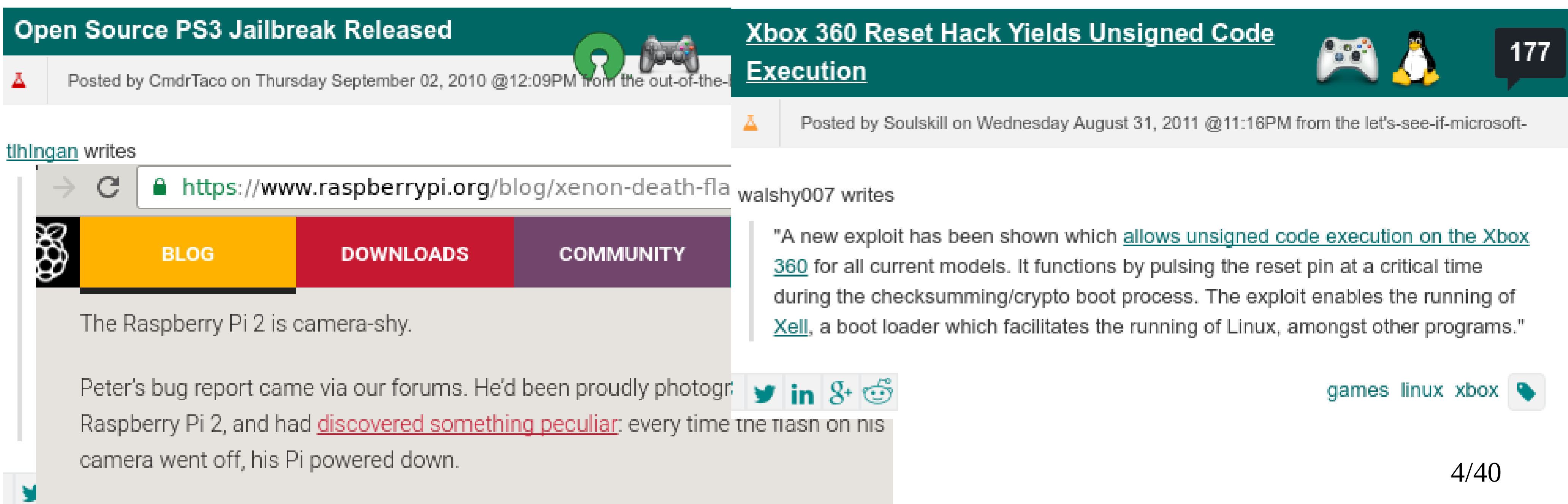
"Perturb the execution stream by manipulating the environment in which a device operates"

- Clock glitching
- Voltage glitching
- EMI pulses
- Temperature etc.



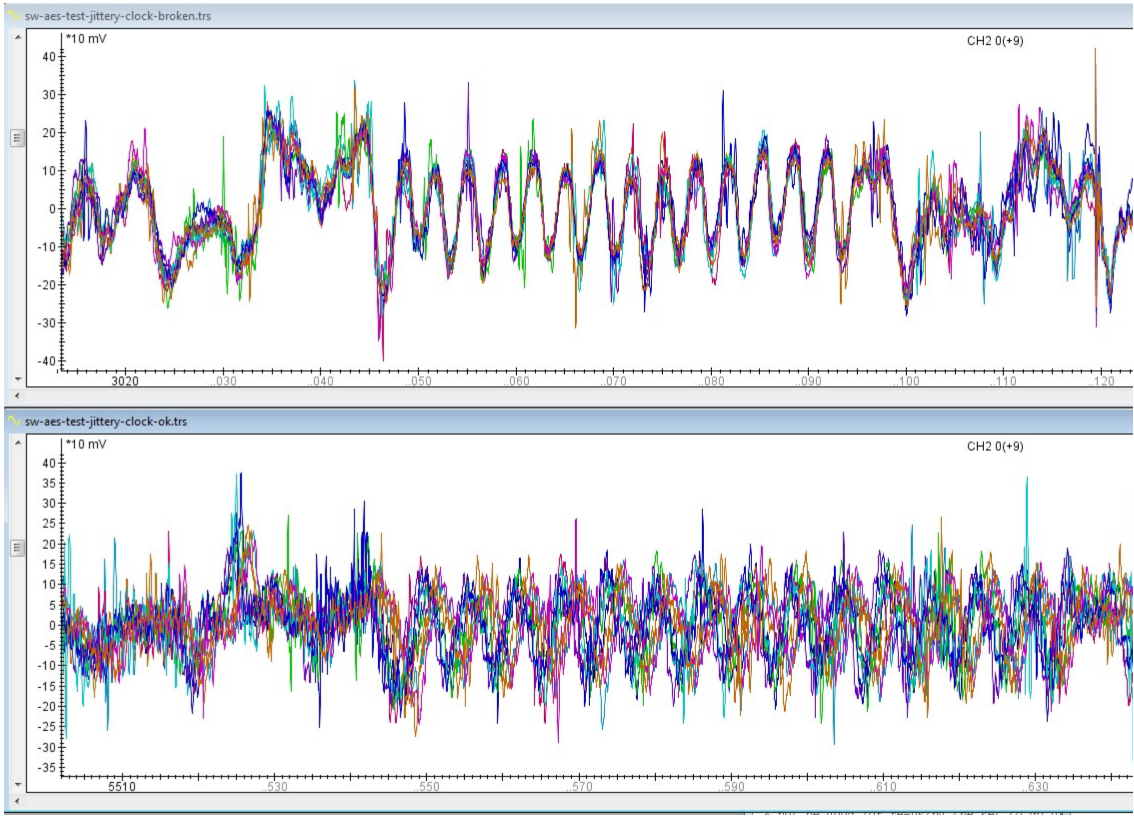
In practice

- *Boneh et al*.: “On the importance of checking cryptographic protocols for faults” ('97)
- *Bao et al*.: “Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults” (‘97)



Hardware defenses

- *Random delay*
- *Jittery clock*
- Shielding
- Brown out sensors



Cons:

- limited availability
- expensive

FI software defenses

- Instruction **redundancy**
  - **Duplication**, n-plication
- Load/Store verification
  - Used in practice
- Loop iteration duplication
- Control flow checks
  - Counter based

FI software defenses

- Instruction **redundancy**
  - **Duplication**, n-plication
- Load/Store verification
  - Used in practice
- Loop iteration duplication
- Control flow checks
  - Counter based

A FI model is needed

The instruction skip model

The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit

_exit: ...
```

The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

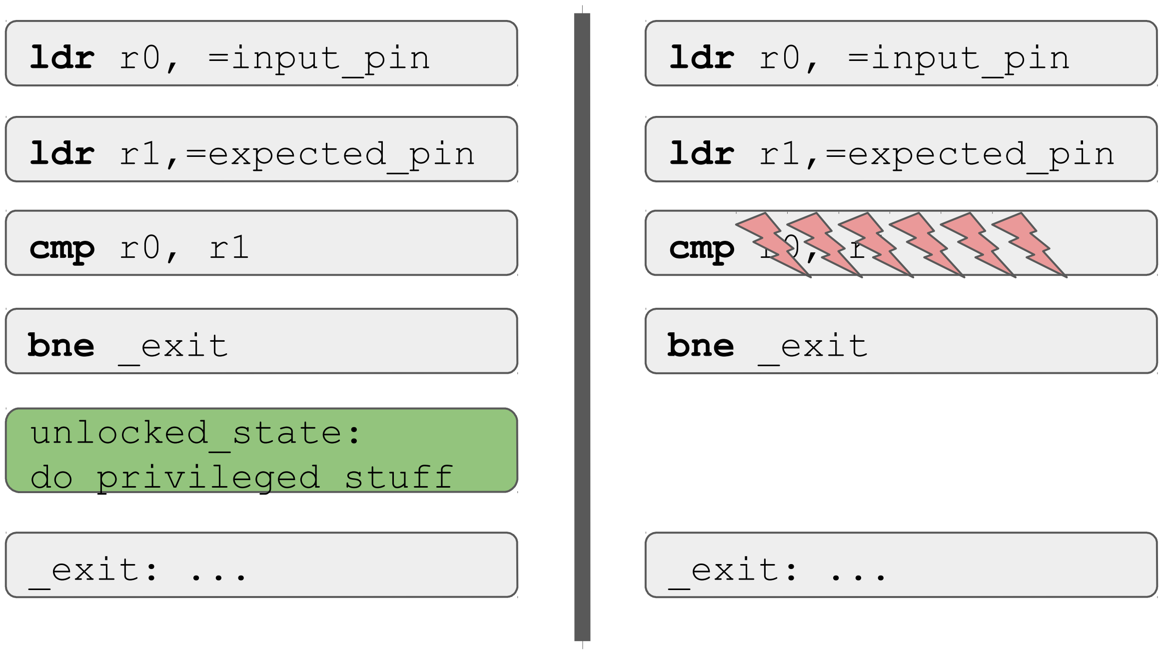
The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do_privileged_stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit

_exit: ...
```

The instruction skip model

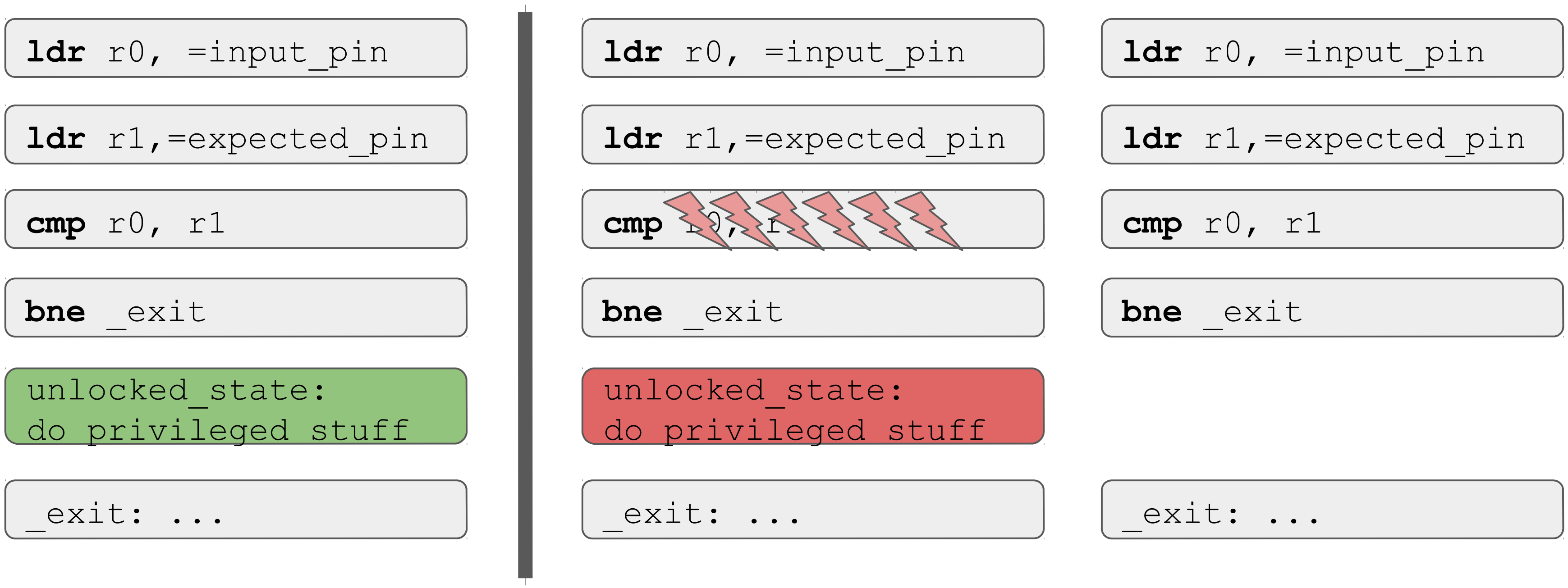


The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

The instruction skip model



The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit

_exit: ...
```

The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

The instruction skip model

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

```
ldr r0, =input_pin
ldr r1,=expected_pin
cmp r0, r1
bne _exit
unlocked_state:
do privileged stuff
_exit: ...
```

The side effect of *faulted* instructions is not observed

Instruction duplication (1/3)

- Fault detection

```
ldr r0, =input_pin
```

→

```
ldr r0, =input_pin
ldr r9, =input_pin
cmp r0, r9
bne _fi_detected
```

- Fault tolerance

```
ldr r0, =input_pin
```

→

```
ldr r0, =input_pin
ldr r0, =input_pin
```

Instruction duplication (2/3)

- Generic enough to work for any code →

Automatic deployment

```
000000a0 <p256_modmul>:
  a0:»   b5f0        » push»   {r4, r5, r6, r7, lr}
  a2:»   af03        » add»r7, sp, #12
```

```
000001b0 <p256_modmul>:
     1b0:»   f2ad 0a18 » subw»   sl, sp, #24
     1b4:»   f2ad 0a18 » subw»   sl, sp, #24
     1b8:»   46d5      » mov»sp, sl
     1ba:»   46d5      » mov»sp, sl
     1bc:»   f20d 0a18 » addw»   sl, sp, #24
     1c0:»   f20d 0a18 » addw»   sl, sp, #24
     1c4:»   e90a 41f0 » stmdb»  sl, {r4, r5, r6, r7, r8, lr}
     1c8:»   e90a 41f0 » stmdb»  sl, {r4, r5, r6, r7, r8, lr}
     1cc:»   af03      » add»r7, sp, #12
     1ce:»   af03      » add»r7, sp, #12
```

Instruction duplication (3/3)

- Fault injection model
  - Single instruction skip on “some” MCUs
    - *Moro et al.*: **EMI**, ARM Cortex-M3
- Practically applied
  - *Heydemann/Moro et al.* formal proof for **ARM**
    - > 2x run-time & space overhead
  - *Barry et al.* optimizes compiler
    - < 2x run-time overhead

Keywords (so far...)

- Software defenses
- Instruction duplication
- Instruction skip model
- Fault detection vs. <u>tolerance</u>
- Compiler assisted deployment

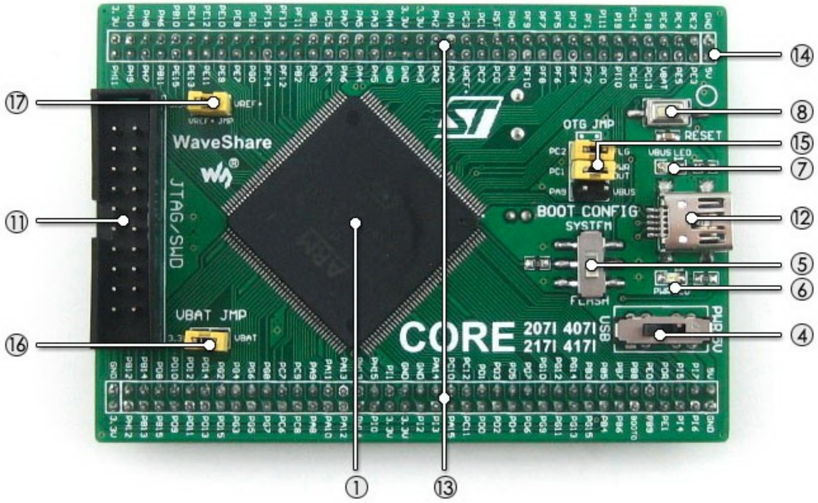
What we did?

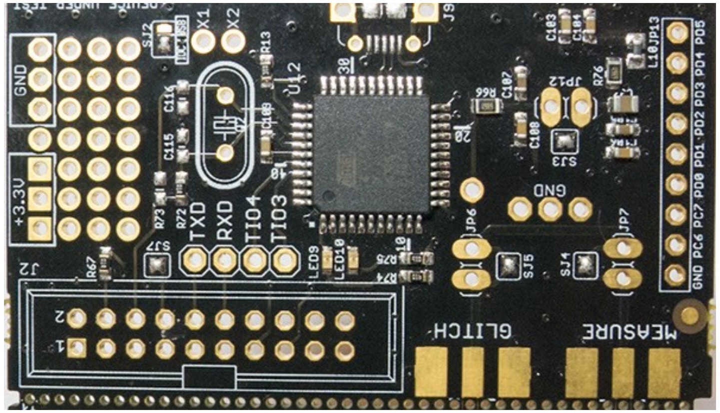
- *Holistic* evaluation of instruction dup.
  - Fault injection
  - Side channel analysis
- Instruction duplication compiler (LLVM)
  - Tests
  - Open source

Instruction duplication: SCA vs. FI

Experimental setup

- FI, power
  - Cortex-M3 @168MHz
  - Riscure's VC Glitcher
- SCA, power
  - XMEGA128, @7.9 MHz
  - ChipWhisperer

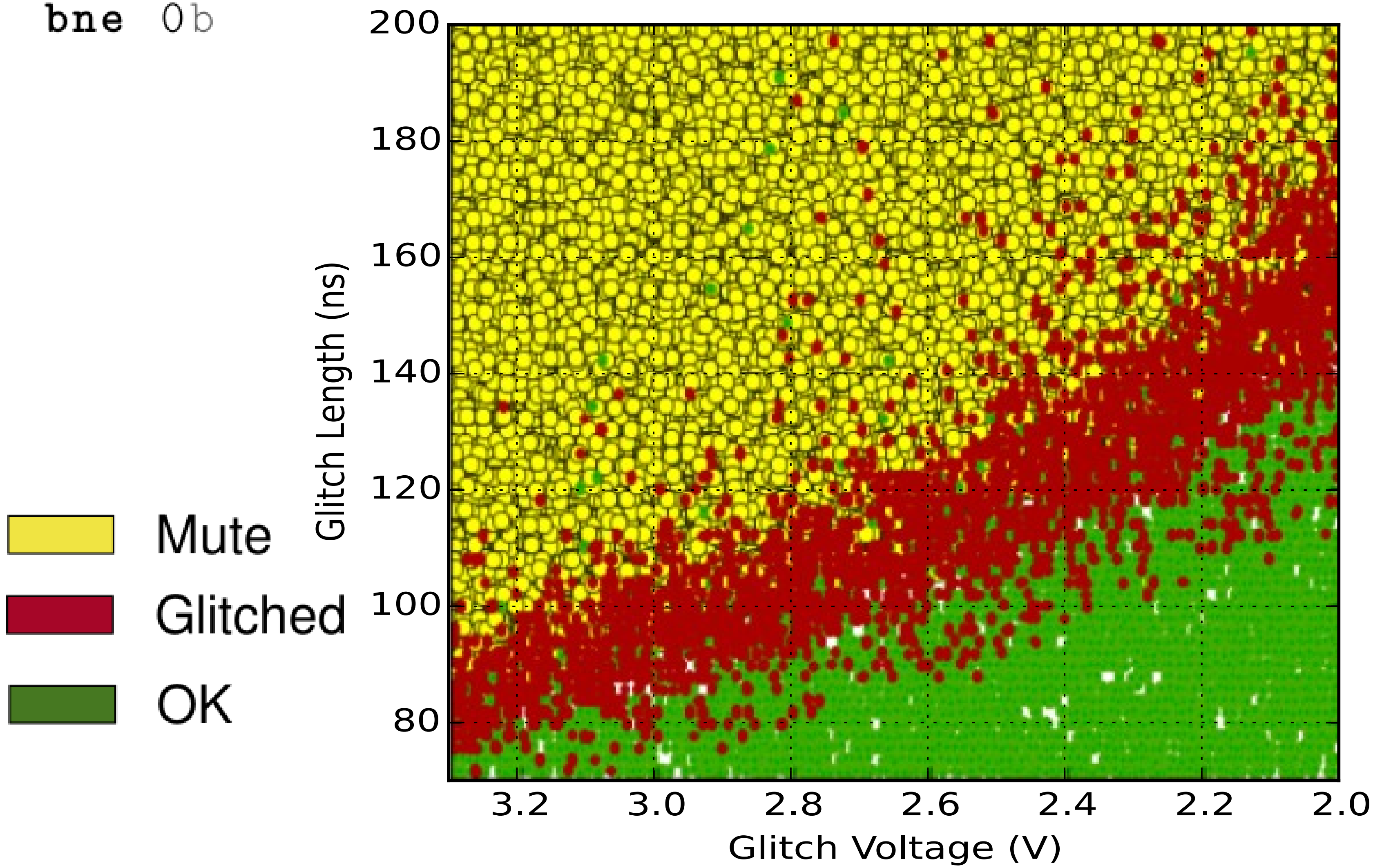




Real hardware

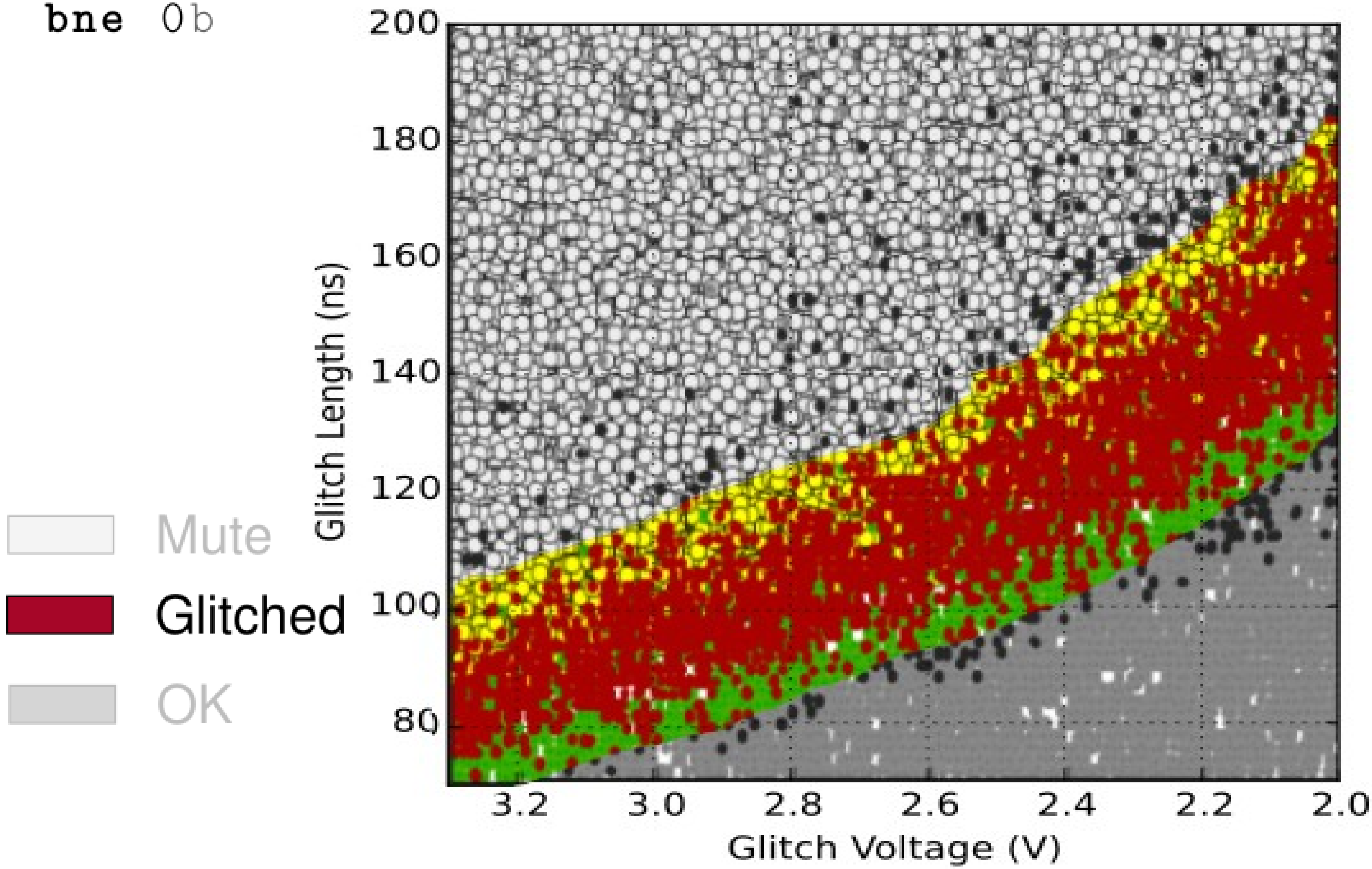
FI – the “Real” model

```
0:  ldm r0, {r4-r10}
    stm r0, {r4-r10}
    subs r1, #1
    bne 0b
```



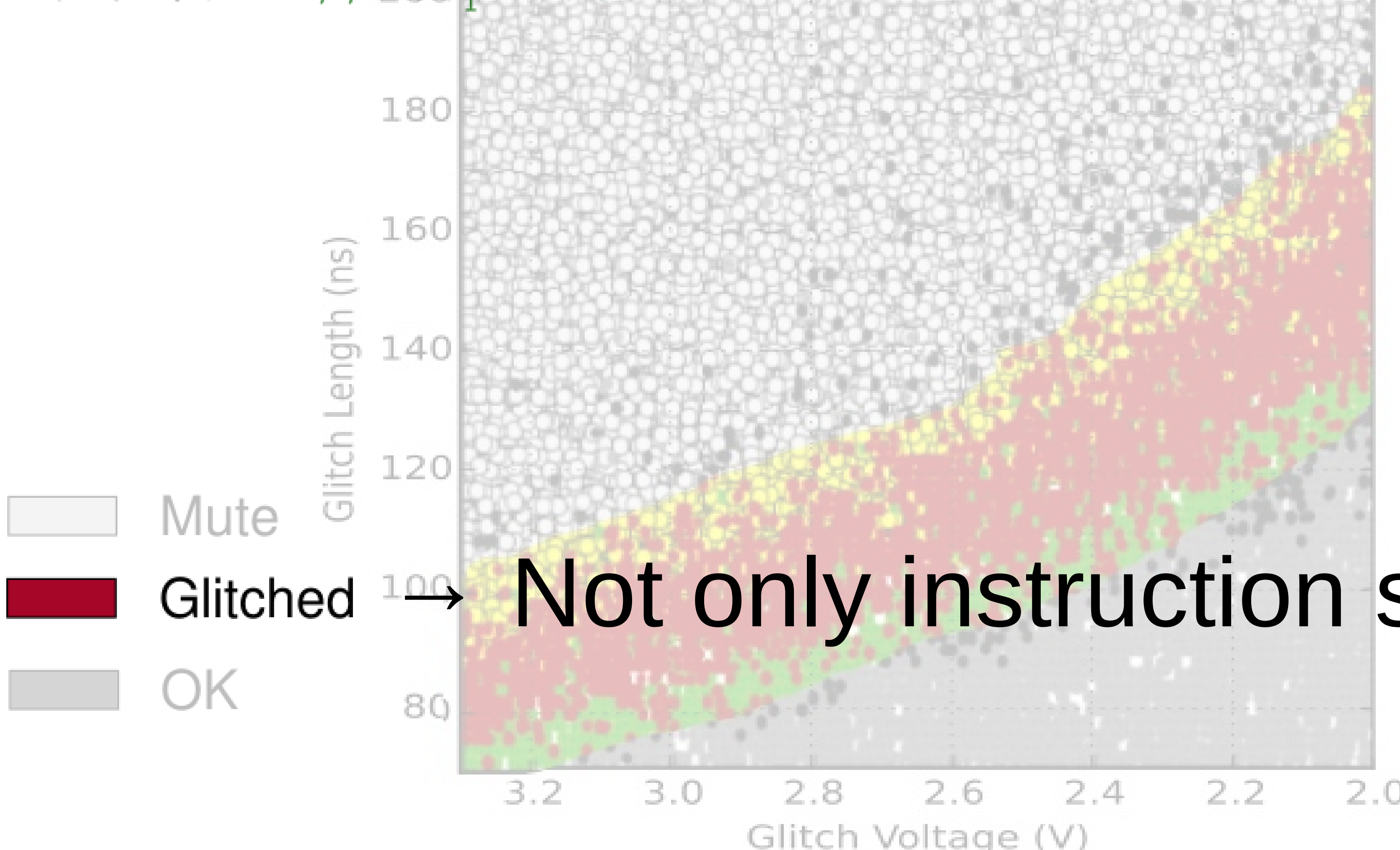
FI – the “Real” model

```
0:  ldm r0, {r4-r10}
    stm r0, {r4-r10}
    subs r1, #1
    bne 0b
```



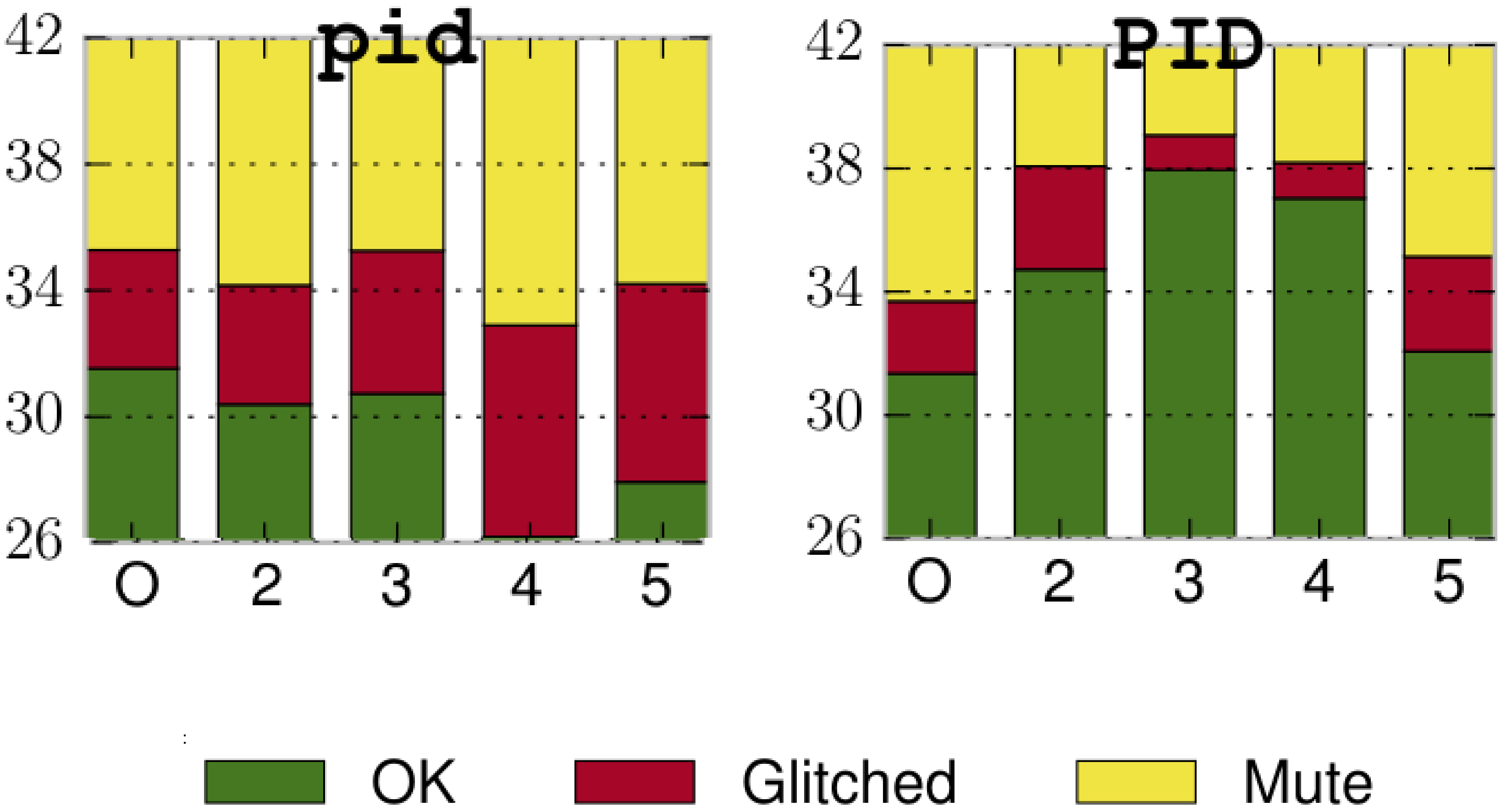
FI – the “Real” model

```
0:  ldm r0, {r4-r10}
    stm r0, {r4-r10}
    subs r1, #1
    bne 0b    //loop back
```

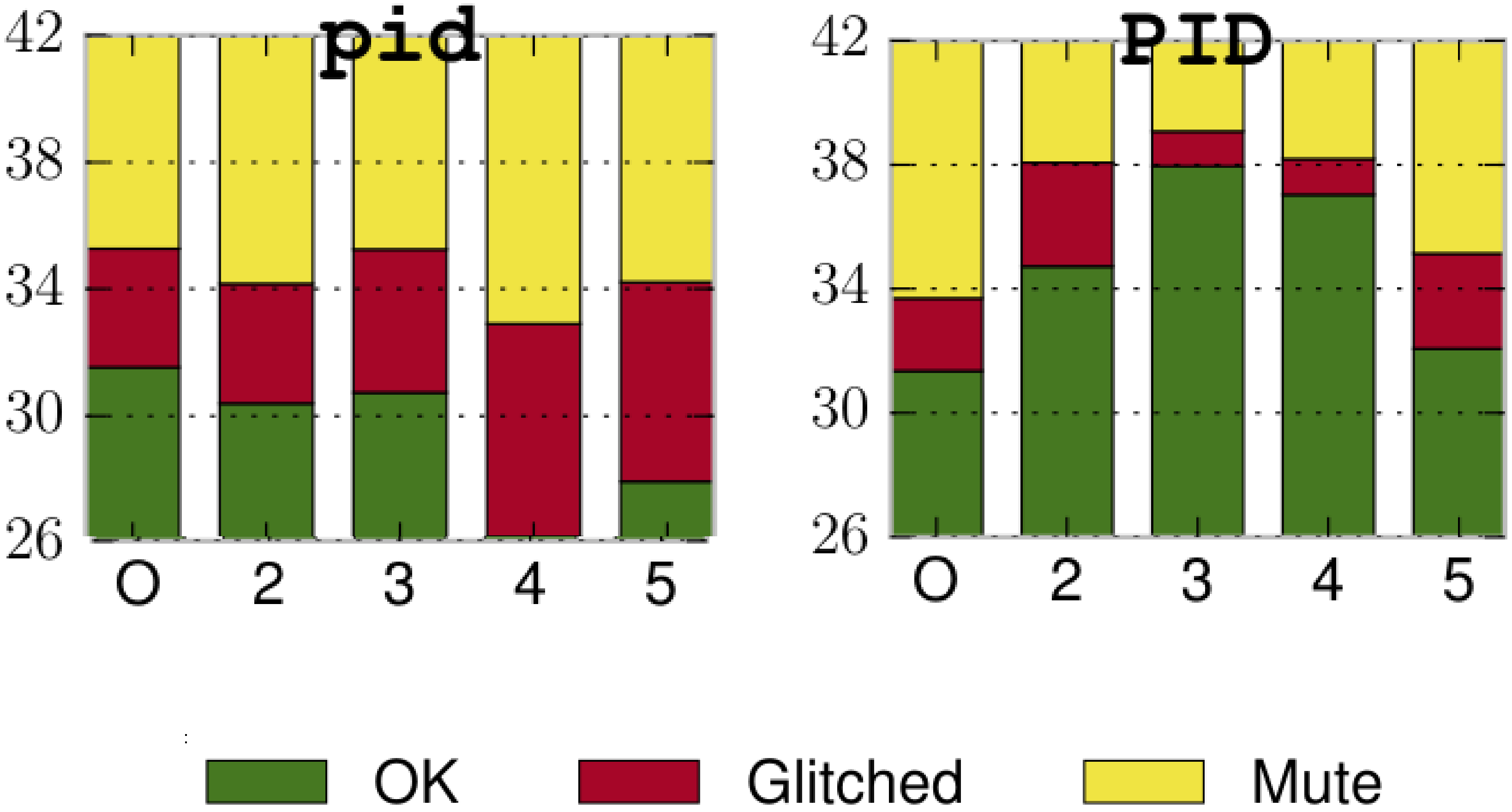


→ Not only instruction skip

FI – static model



FI – static model



Behavior influenced by the runtime config.

FI – conclusion

- The compiler is helpless
  - Unaware of the runtime configuration
  - Can reorder instructions
  - Register pressure is increased (more in the paper)

Instruction n-plication was not too effective

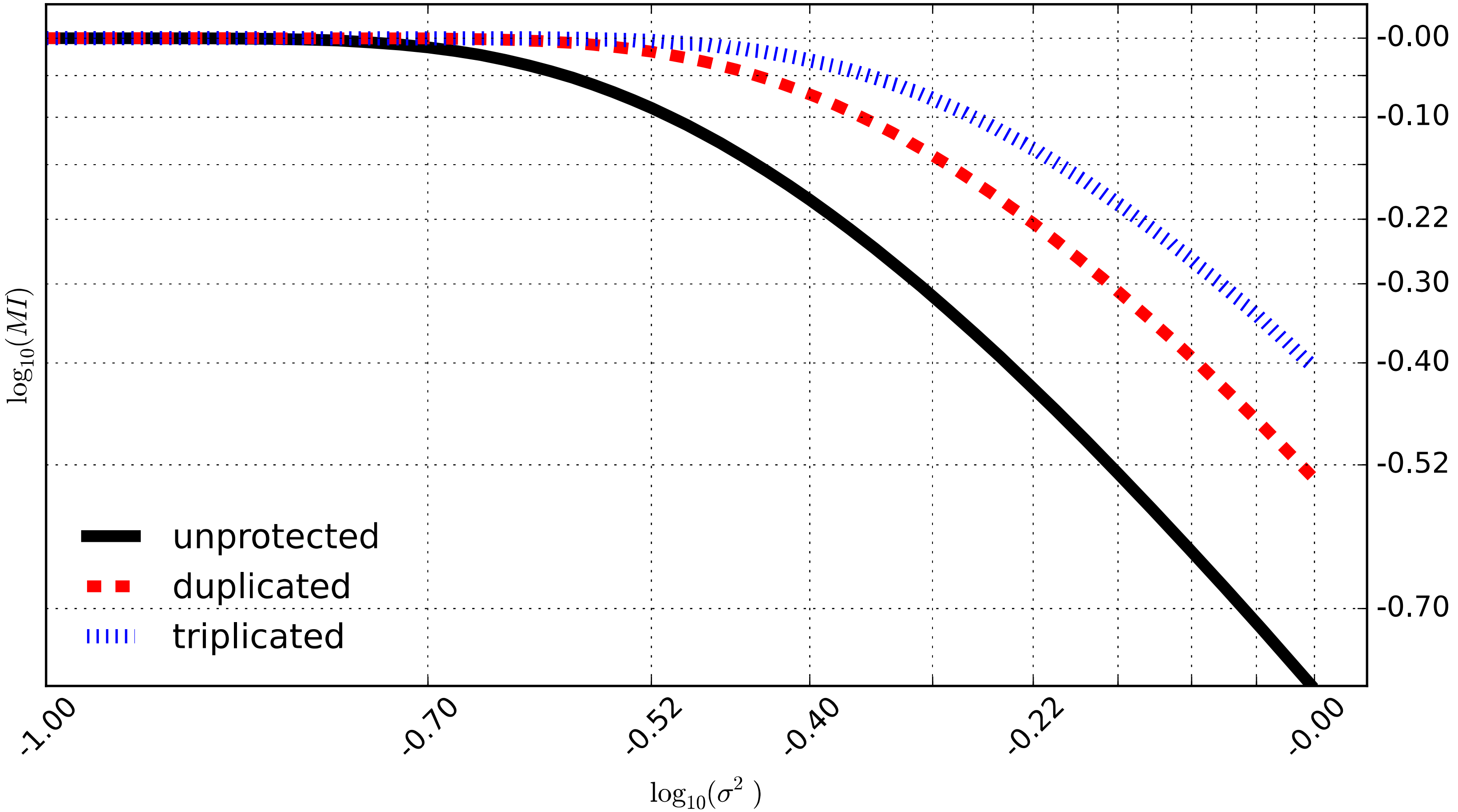
(SR of FI was 2%-12%)

SCA

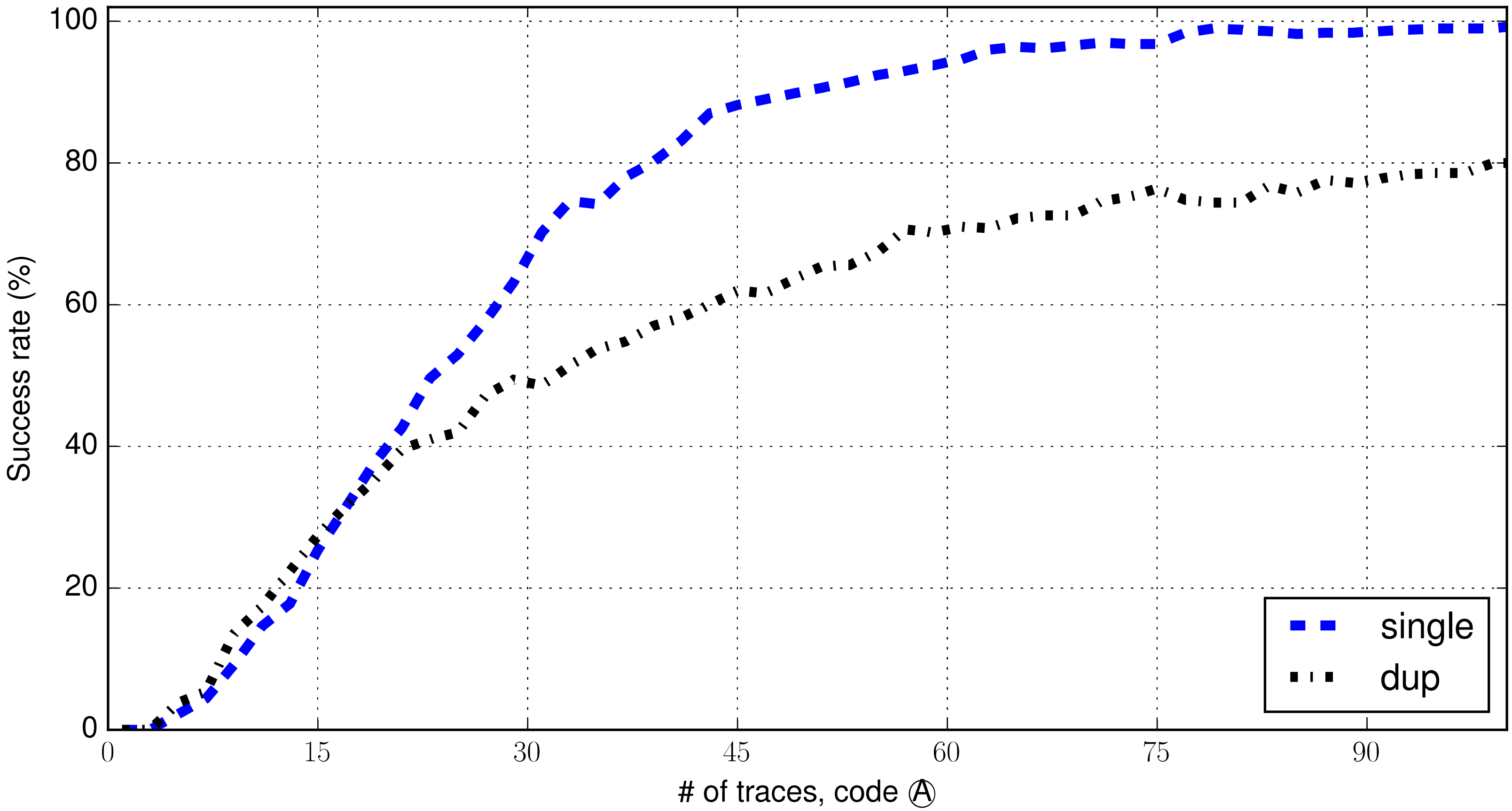
Twice as many instructions → easier to exploit?

SCA

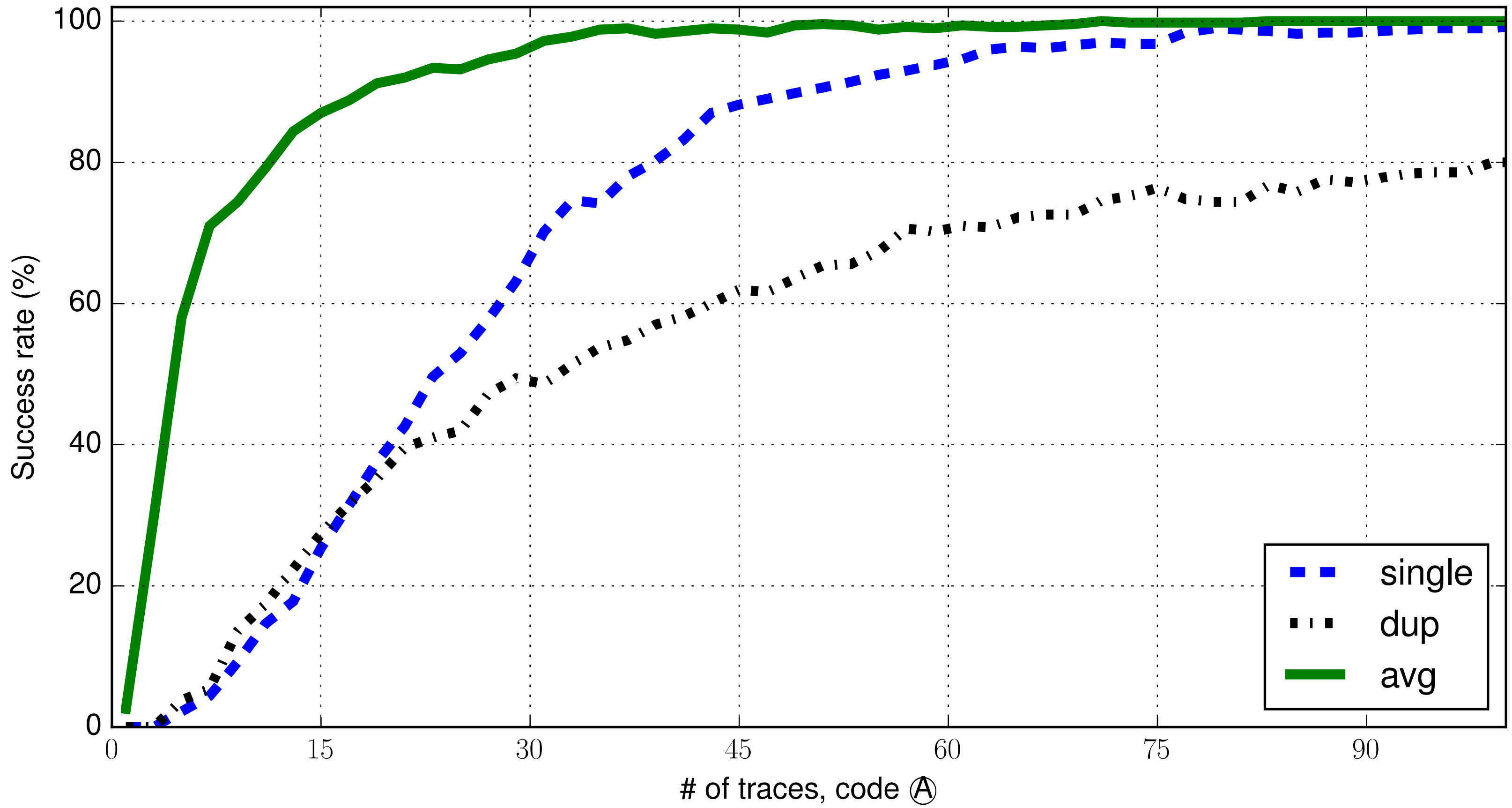
Twice as many instructions → easier to exploit?



Naive CPA – `xor` instruction



“Averaged” CPA – `xor` instruction



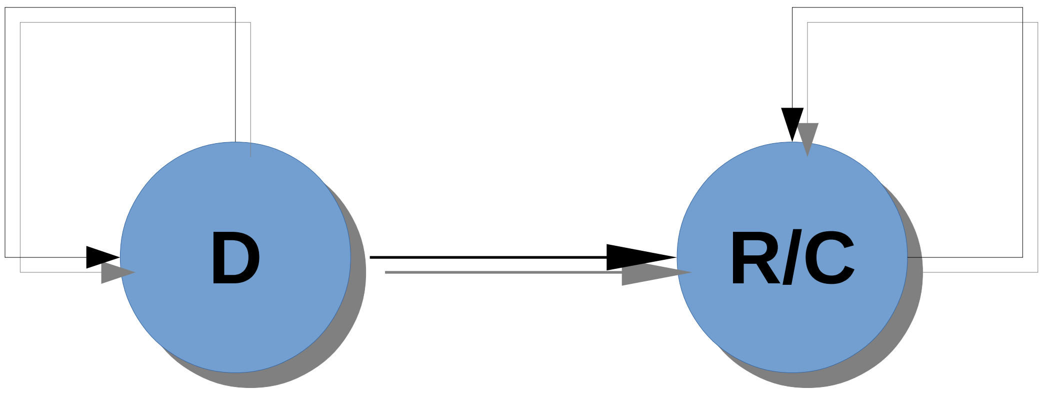
Horizontal Exploitation

- FI countermeasures rely on redundancy
- We need to exploit the trace horizontally
- Naive techniques may lure us into a false sense of security

1. We employ averaging techniques with CPA attacks
2. We employ multivariate templates using PCA dimensionality reduction (more in the paper)

SCA on Infective Countermeasures

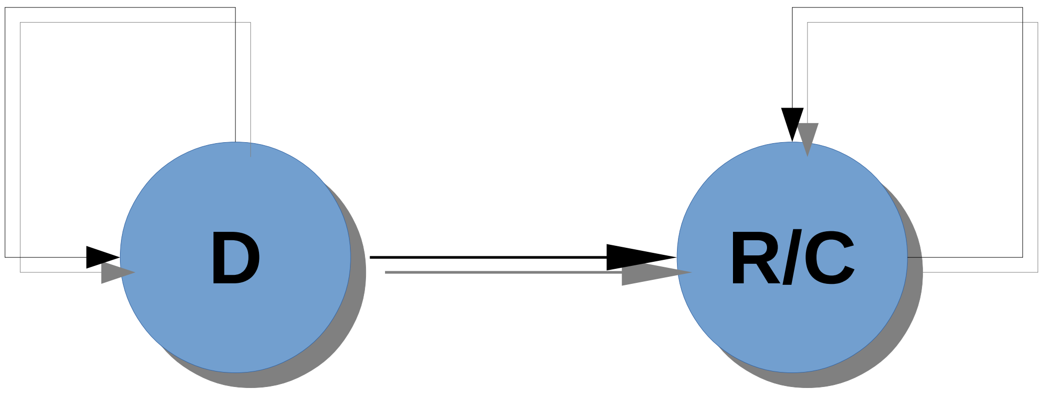
- Software redundancy can be implemented in less straightforward ways
- Infective countermeasure uses a random sequence of **D**ummy, **R**edundant and **C**ipher rounds



- Using the Hidden Markov Model helps us locate and exploit the **R**edundant and **C**ipher rounds

SCA on Infective Countermeasures

- Software redundancy can be implemented in less straightforward ways
- Infective countermeasure uses a random sequence of **D**ummy, **R**edundant and **C**ipher rounds



- Using the Hidden Markov Model helps us locate and exploit the **R**edundant and **C**ipher rounds

We reduce infective to n-plication

Summary

- Interaction between n-plication and its SCA
  - Horizontal exploitation technique
- Open source instruction duplication compiler
  - https://github.com/cojocar/llvm-iskip
- Other redundancy-based defenses have similar limitations
  - Infection

Future work

- Design a model that can take the runtime configuration into account
- Use data flow in the compiler
- Add fault detection to the constructions
- Take binary code layout into account

Conclusion

- Instruction skip model invalid in practice
  - "Instruction corruption"
- Side channel of duplicated code is **amplified**
- The fault model differs from target to target
  - Fine tunning according to runtime configuration
- Fine balance between performance and security

Instruction Duplication:
Leaky and Not Too Fault-Tolerant!