KERNELFAULT:

R00ting the Unexploitable using Hardware Fault Injection

Niek Timmers
Senior Security Analyst
@tieknimmers / niek@riscure.com

Cristofaro Mune
Product Security Consultant
@pulsoid / c.mune@pulse-sec.com

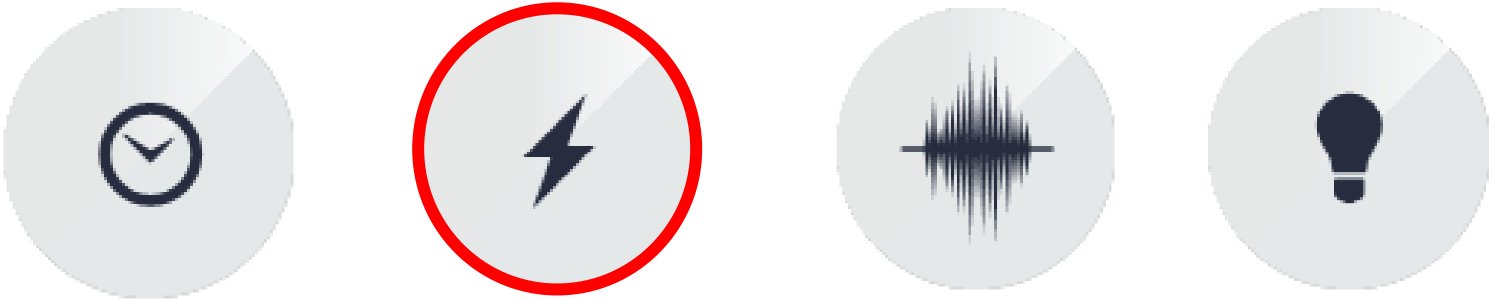
Fault Injection: a definition

"Introducing faults in a target to alter its intended behavior."

```
...
if( key_is_correct ) <-- Glitch here!
{
  open_door();
}
else
{
  keep_door_closed();
}
...
```

*How can we introduce these **faults**?*

Hardware fault injection techniques

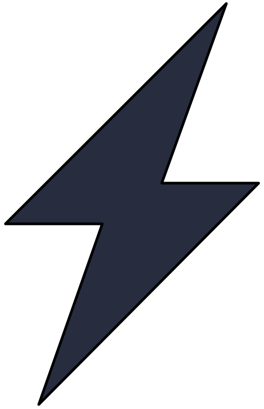


Clock *Voltage* *EM* *Laser*

- A controlled environmental change leads to altered behavior in a target
- They leverage a vulnerability in a hardware subsystem

Glitch

"A controlled environmental change."



*These **glitches** can result in **fault injection vulnerabilities**!*

Vulnerability

*"Susceptibility of a given **hardware subsystem** to a specific **fault injection technique,** which has an impact on security."*

- Located in hardware
- Cannot be identified by (code) review only
- Can only be identified by performing a successful attack
- Can only be entirely addressed in hardware

*These **vulnerabilities** lead to **faults**!*

Fault

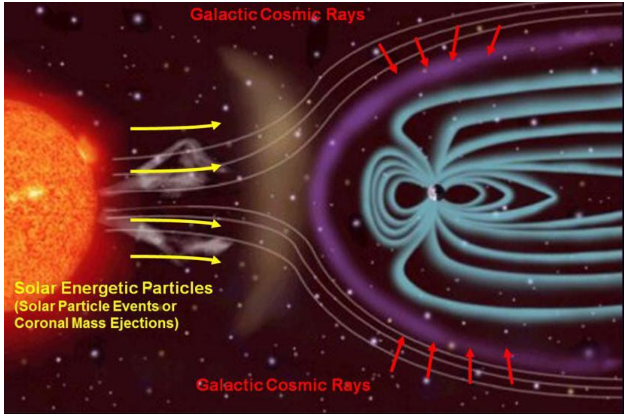
*"An unintended alteration of a target as a consequence of a **vulnerability**."*

- Happens at a specific moment in time
- May be (semi-)persistent
- May be mitigated in software

*These **faults** potentially lead to **compromised systems**!*

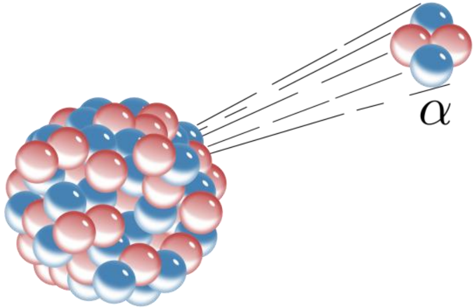
What do we need to glitch?

Natural phenomena



Cosmic rays

** Ziegler, Lanford – "Effects of cosmic rays on computer memories" (1979)*



Alpha decay

** May, Woods – "Alpha-particle-induced soft errors in dynamic memories" (1979)*

Cost: ???

High-end Tooling

- Great for security labs
- Different techniques:
  - VCC, Clock, EM, Laser,...
- Flexibility, speed, precision
- ***High control → Repeatability***

Cost ($): > 10,000

Other options...

Chipwhisperer Lite

~$250

Microcontroller

< $30

FPGA

~$99

Cost ($): < 300

Do we always need specialized tooling?

Software activated fault injection

- Possible when software can activate hardware vulnerabilities
- The vulnerabilities and faults are still in hardware!

Some recent examples...

- **Rowhammer** (Kim et al., 2014; many more afterwards)
  - Constantly reading a DDR address leads to bit flips in neighboring bits
- **CLKSCREW** (Tang et al., 2017)
  - Manipulating Digital Voltage Frequency Scaling (DVFS) registers
  - Operate the chip out of its specifications

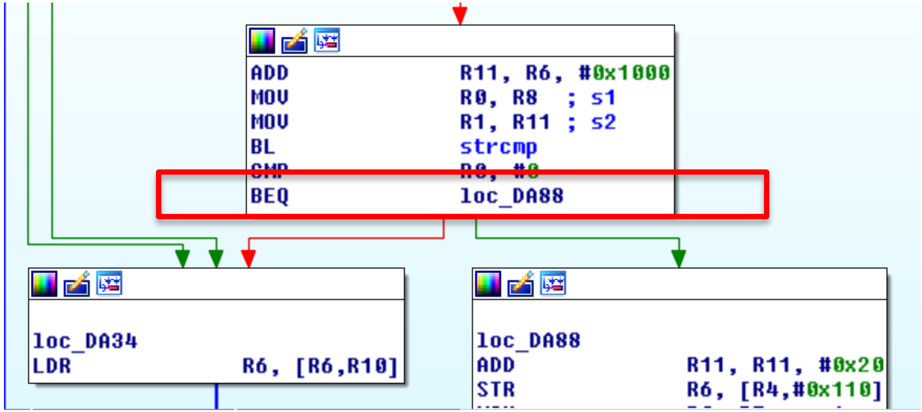
You can do this remotely without specialized tooling!

Hardware Fault Injection

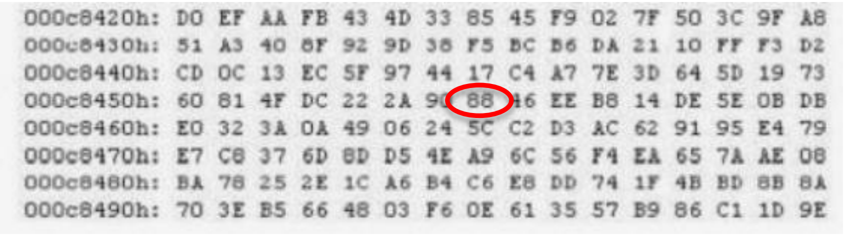
Some real world examples…

Traditional targets and models…

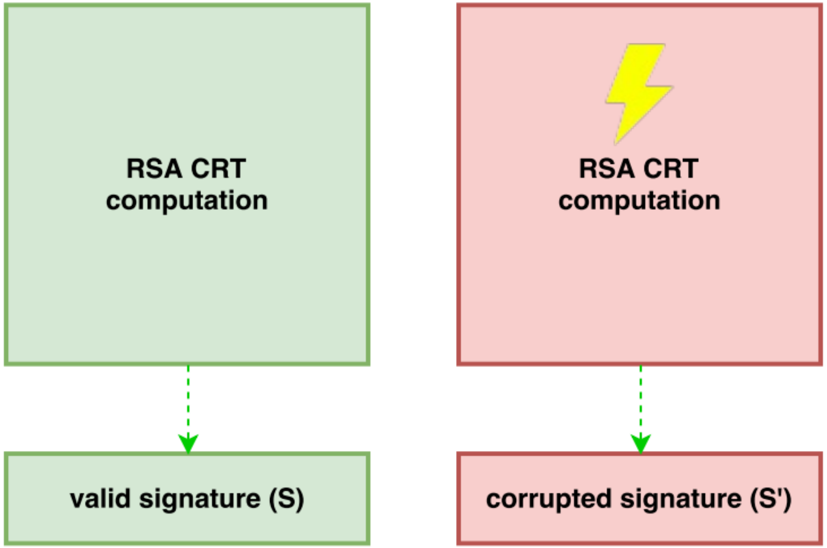
Control flow corruption
by skipping instructions



Data corruption
by flipping bits



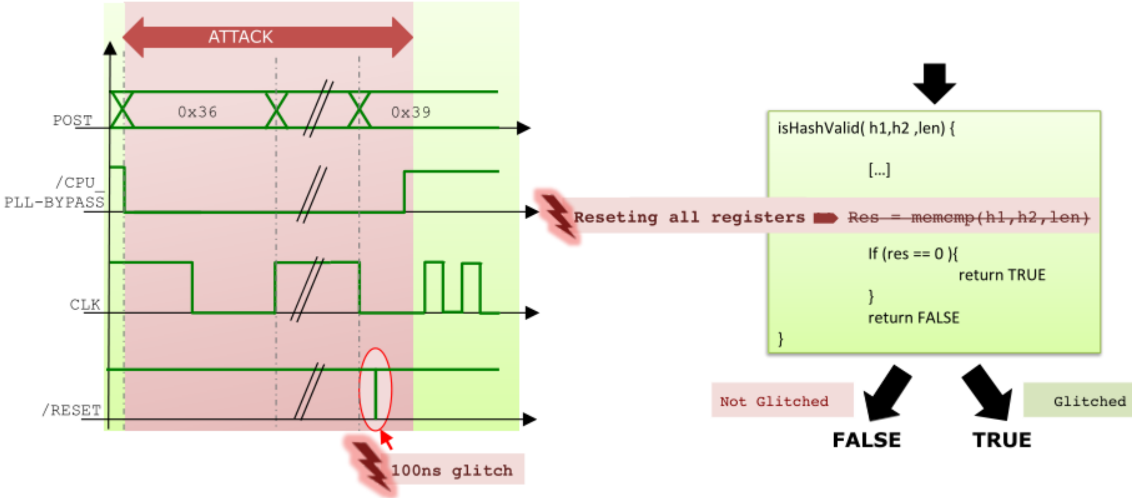
Differential fault analysis (DFA) – Recovering keys



The private key can be recovered by computing the GCD of (S - S') and the modulus (N) !

Similar attacks for most crypto algorithms!

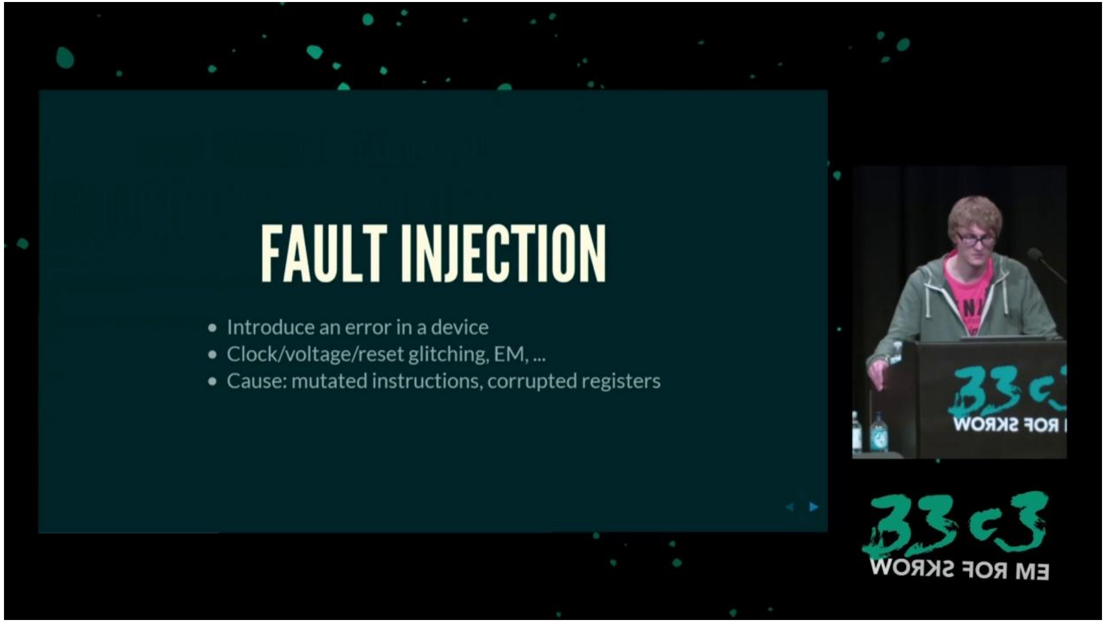
Xbox – Bypassing secure boot



Reference: *Video-game consoles architecture under microscope - R. Benadjila and M. Renard*

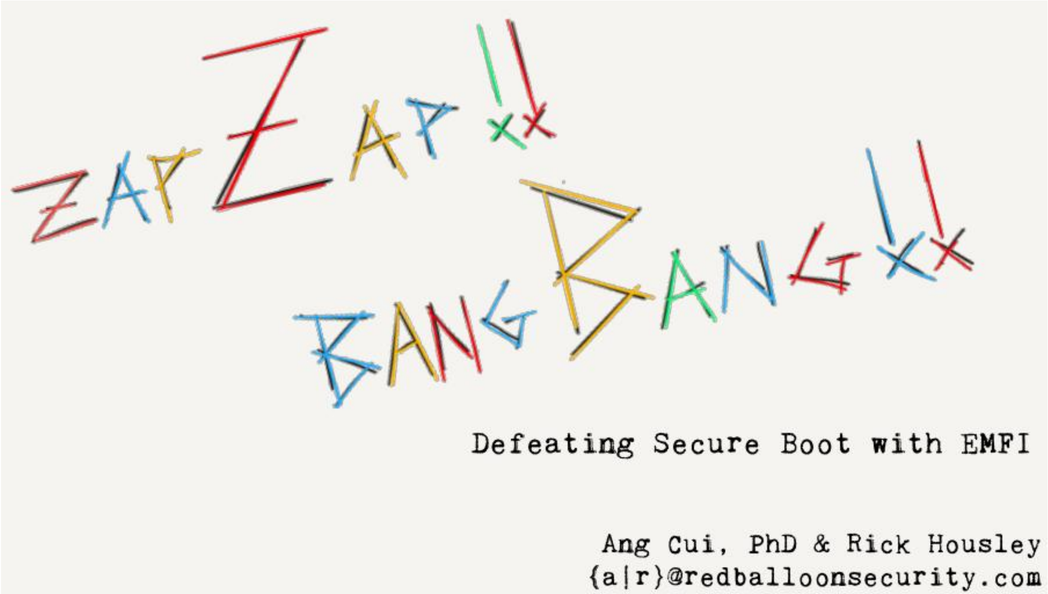
- ***Reset line glitch*** to reset registers' content
- Bypass hash comparison used by integrity check

Nintendo – Bypassing secure boot



- Use a **glitch** to bypass length check performed by software
- Code execution leads to dumping decryption key from memory

BADFET – Bypassing secure boot



- Using an electromagnetic glitch to bypass secure boot of a Cisco phone
- Not that invasive... (i.e. phone's housing can remain closed)

Trends

- Specialized equipment is becoming **cheaper** and **available to the masses**
- Equipment might **not** be **needed** at all (e.g. software activated fault injection)

How can these attacks be mitigated?

Traditional fault injection countermeasures

- **Hardware-based**
  - Specifically designed hardware logic for ***redundancy*** and ***detection***
  - Detection by hardware **close to** the glitch injection moment
  - May prevent injection (e.g. shielding)
  - Not implemented on standard embedded technology

- **Software-based**
  - Based on ***computational checks***, ***redundancy*** and ***random delays***
  - Detection by software **after** the glitch injection moment
  - Do not prevent injection

*Both can be effective at **lowering the probability** for a successful attack!*

Notes on software countermeasures

- They **do not** prevent fault injection but increase attack complexity
- They require software to be executed **after** the glitch is injected
- They (often) protect specific parts of the code
  - Critical decision points
  - Crypto operations
  - Data integrity

Are software fault injection countermeasures sufficient?

Most real world examples target secure boot…

Why not use fault injection at runtime?

Fault Injection meets *Linux*!

How is Linux usually compromised?

A summary of Linux CVEs

Year	DoS	Exec	Overflow	Corruption	Leak	PrivEsc
2015	55	6	15	4	10	17
2016	153	5	38	18	35	52
2017	92	166	35	16	78	29

Kernel software exploit: between $30k and $100k
(**Source:** Zerodium)

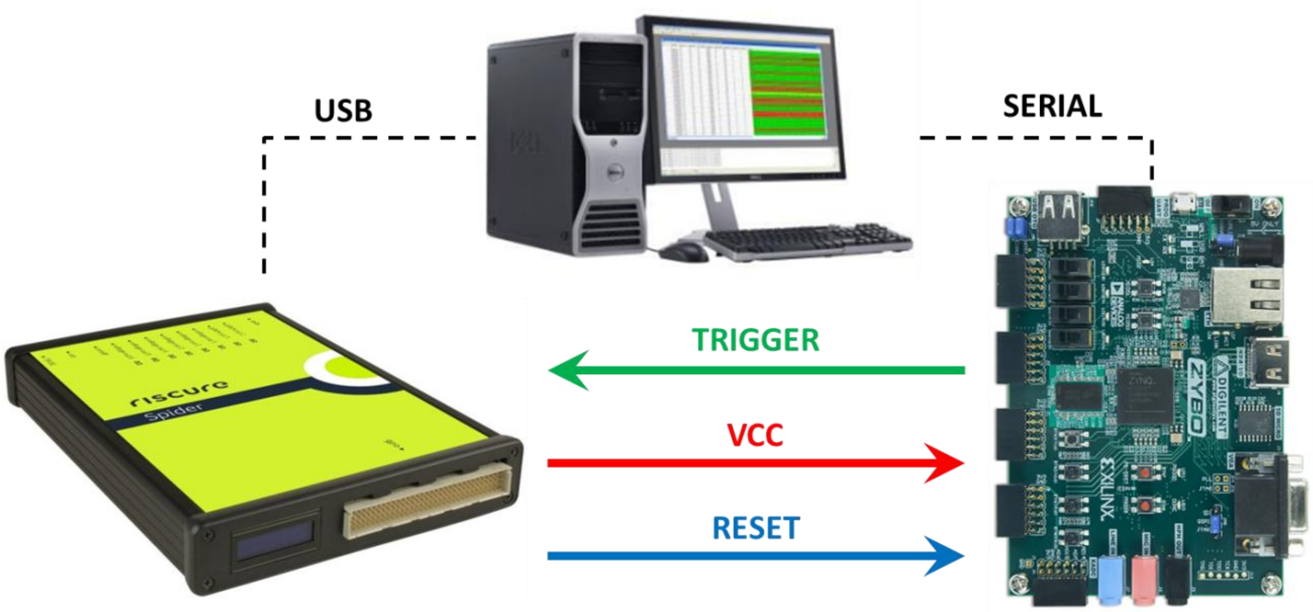
*What if they are **not known** or **not present?***

Others came to the same conclusion…

Reference: *https://derrekr.github.io/3ds/33c3/#/18*

Fault Injection!

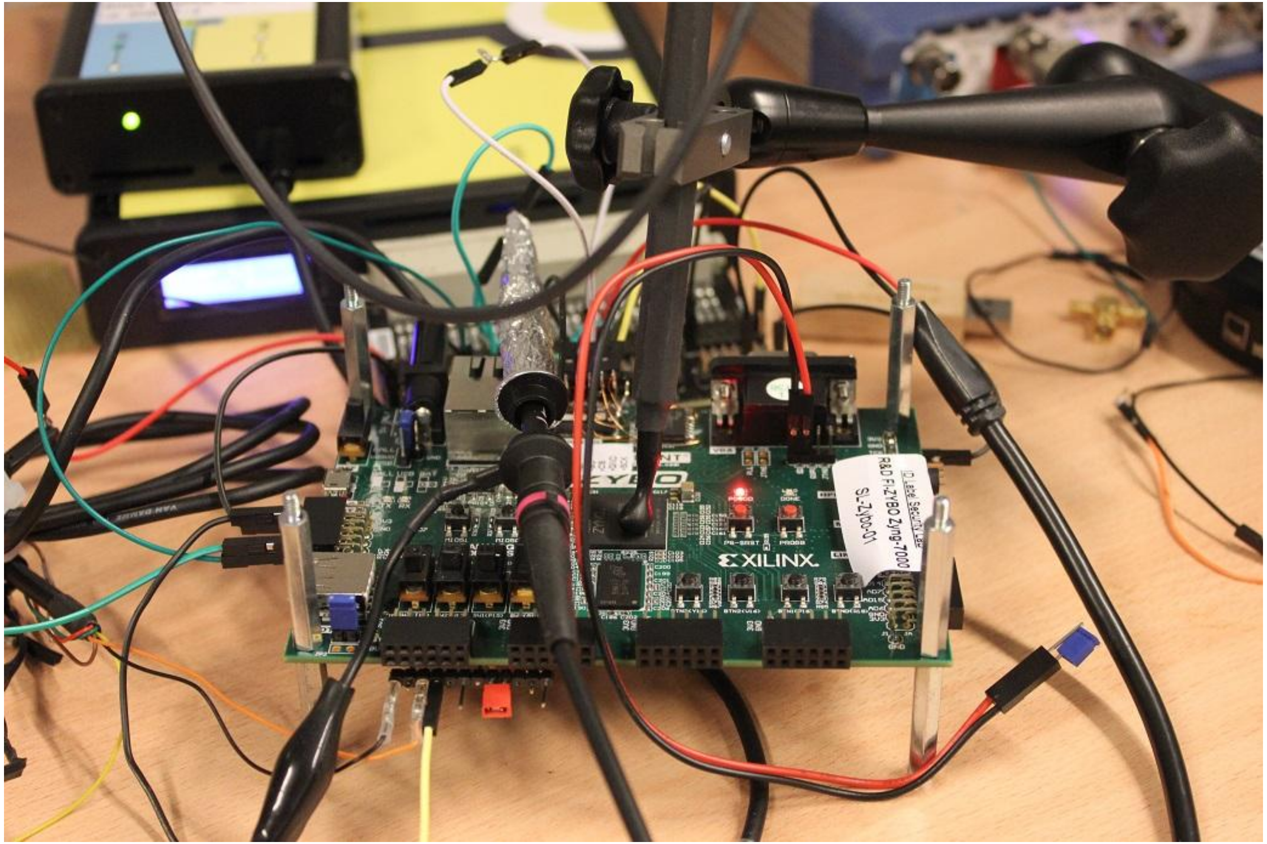
Voltage fault injection setup



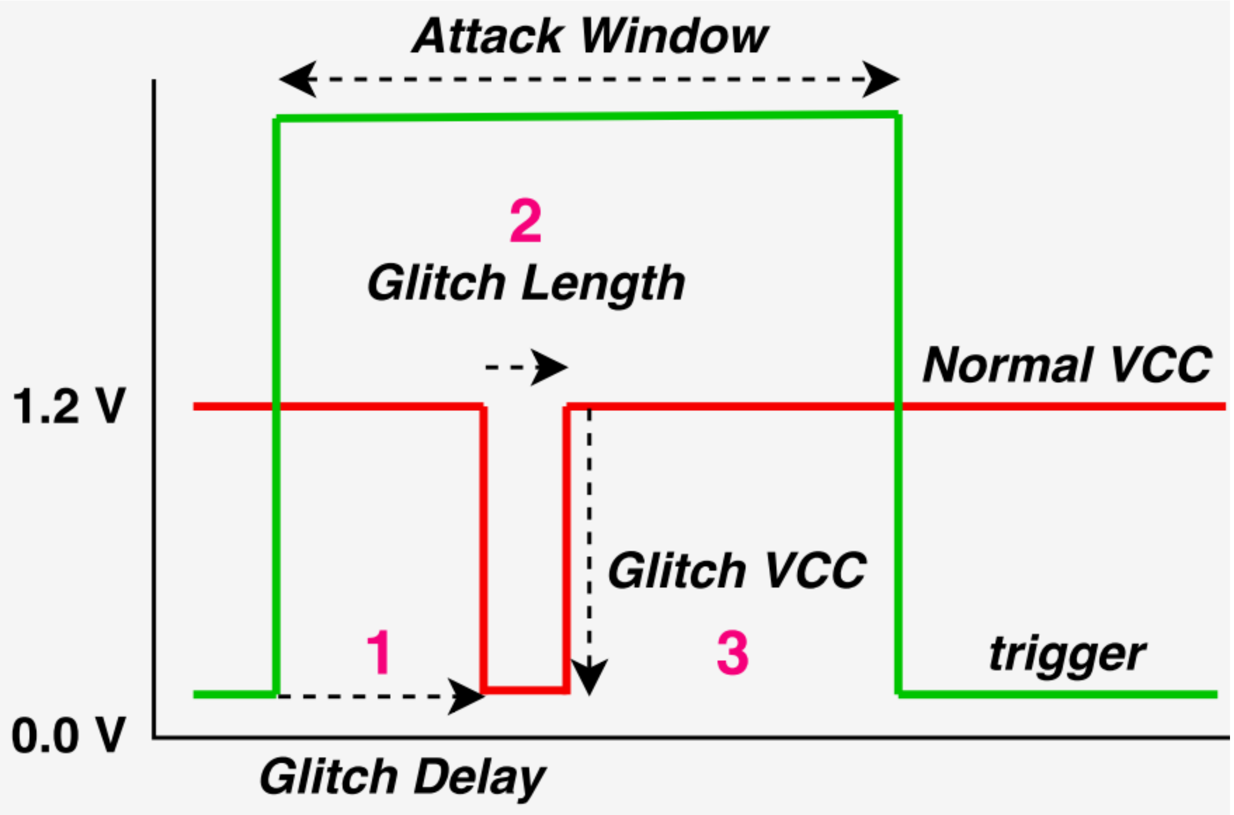
Target

- Fast and feature rich System-on-Chip (SoC)
- ARM Cortex-A9 (ARM32 / AArch32)
- Ubuntu 14.04 LTS (fully patched)

Typical setup



Voltage fault injection parameters



Characterization – Determining if target is vulnerable

```
. . .
set_trigger(1);

for(i = 0; i < 10000; i++) {   // glitch here
    j++;                        // glitch here
}                               // glitch here

set_trigger(0);
. . .
```

Characterization – Responses

Expected (too soft)

counter = 00010000

Mute (too hard)

counter =

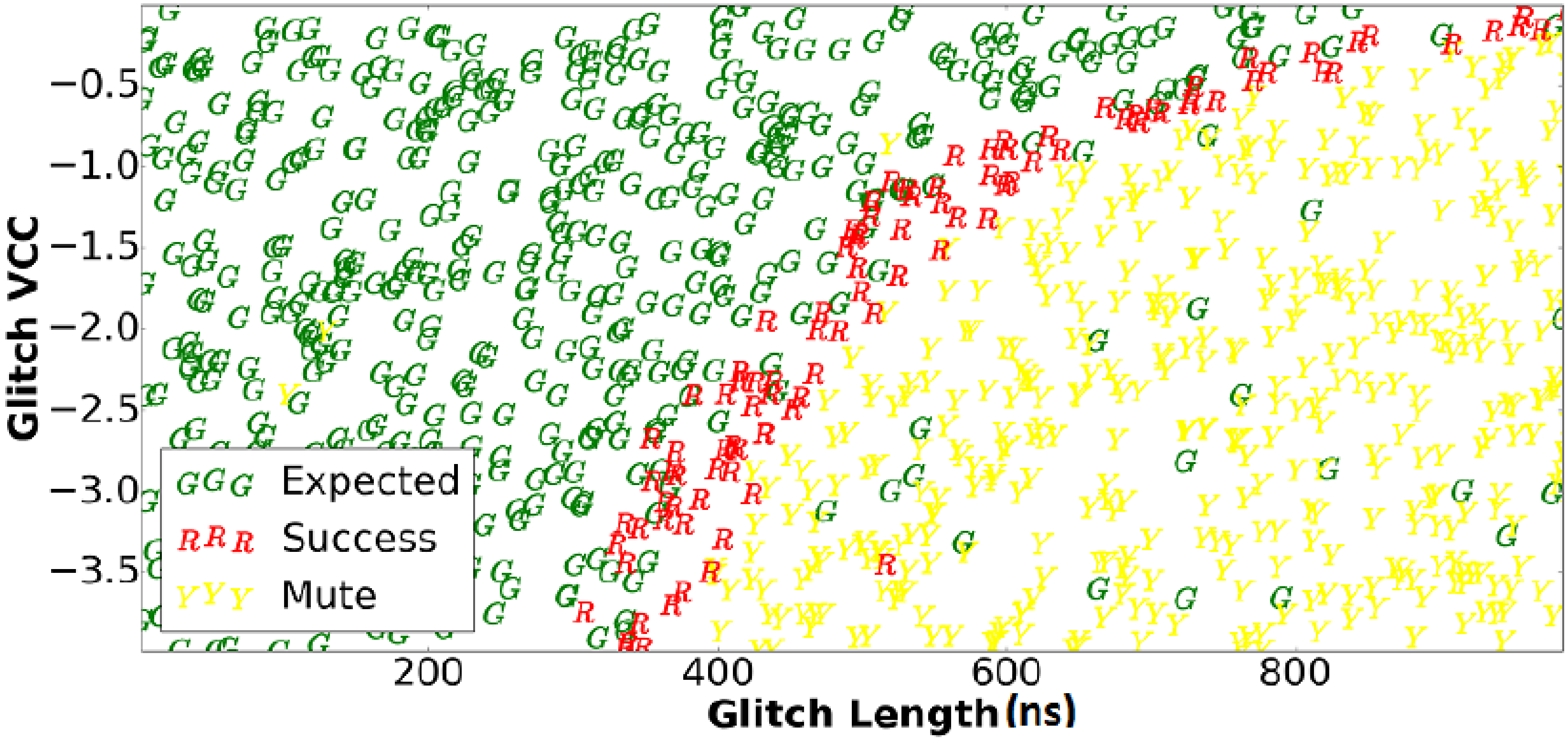
Success

counter = 00009999

counter = 00010015

counter = 00008687

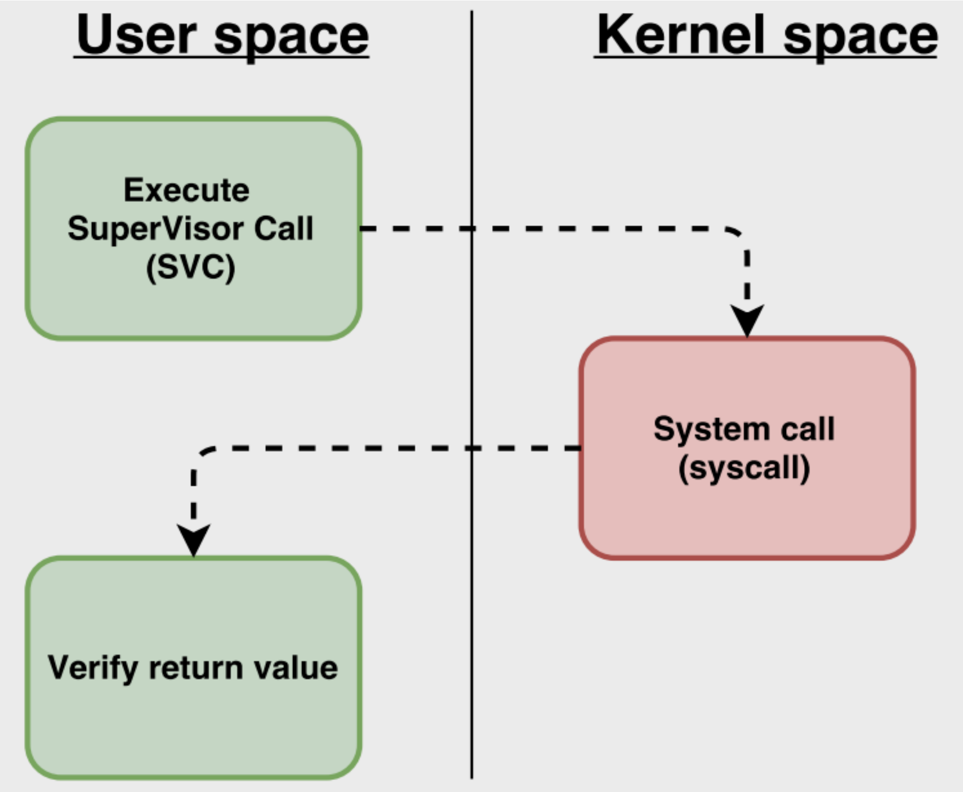
Characterization – Plot



Attacking Linux

More info: *https://www.riscure.com/publication/escalating-privileges-linux-using-fault-injection/*

Attacking Linux



Arbitrary memory mapping - Description

1. Open */dev/mem* using ***open*** syscall from userspace process
2. *Bypass checks performed by Linux kernel using a glitch*
3. Map arbitrary physical address in userspace

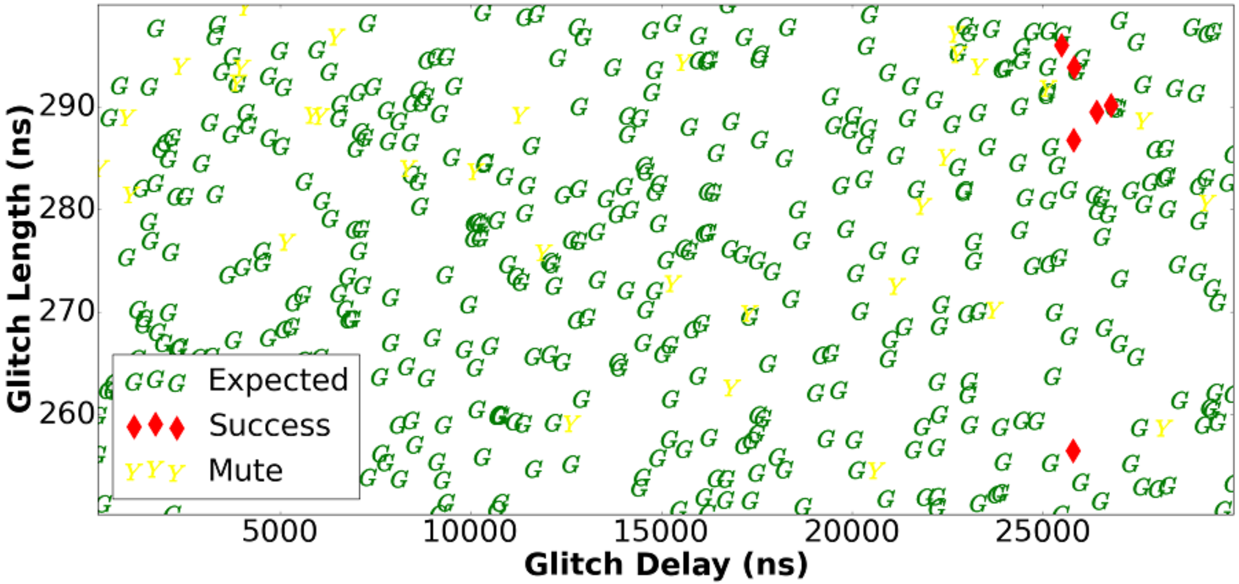
Full kernel memory access

Arbitrary memory mapping - Code

```
*(volatile unsigned int *)(trigger) = HIGH;

int mem = open("/dev/mem", O_RDWR | O_SYNC);

*(volatile unsigned int *)(trigger) = LOW;

if( mem == 4 ) {
  void * addr = mmap ( 0, ..., ..., mem, 0);
  printf("%08x\n", *(unsigned int *)(addr));
}
. . .
```

- Code running in userspace
- Linux syscall: sys_open (0x5)

Arbitrary memory mapping - Results



Remarks

- Performed 22118 experiments in 17 hours
- Success rate between 25.5 µs and 26.8 µs: 0.53%
- ***Kernel "pwned" every 10 minutes***

Escalating to a root shell - Description

1. Set all registers to 0 to increase success probability (*)
2. Perform ***setresuid*** syscall to set process IDs to root
3. Bypass checks performed by Linux kernel using a glitch
4. Execute shell using ***system*** function

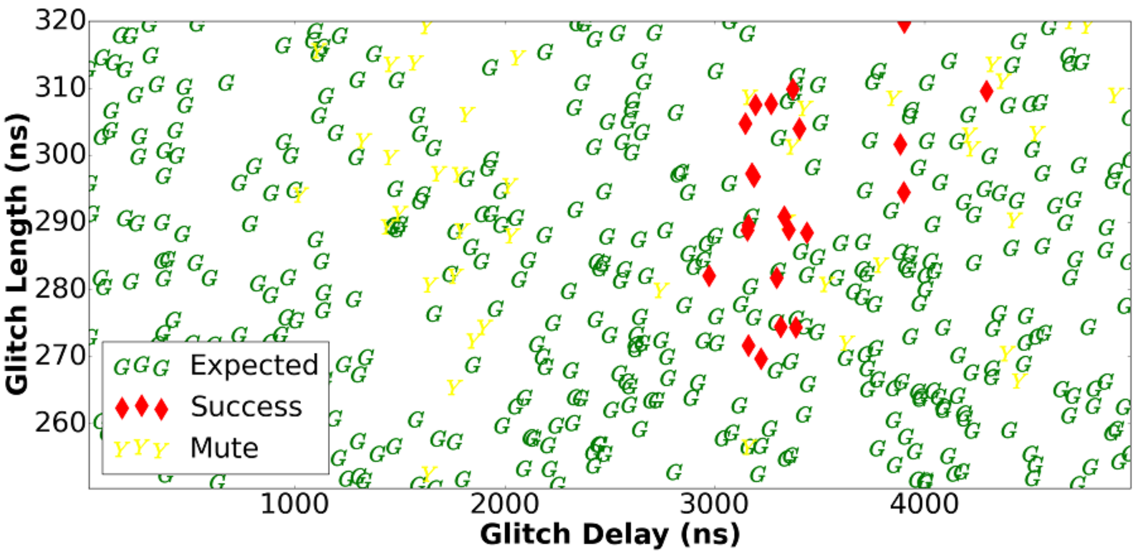
Shell with full root privileges

Escalating to a root shell - Code

```
*(volatile unsigned int *)(trigger) = HIGH;

asm volatile (
  "movw r12, #0x0;" // Repeat for other
  "movt r12, #0x0;" // unused registers
  . . .
  "mov r7, #0xd0;"  // setresuid syscall
  "swi #0;"         // Linux kernel takes over

  "mov %[ret], r0;" // Store return value in r0
  : [ret] "=r" (ret) : : "r0", . . ., "r12" )

*(volatile unsigned int *)(trigger) = LOW;

if(ret == 0) { system("/bin/sh"); }
```

- Code running in userspace
- Linux syscall: setresuid (0xd0)

Escalating to a root shell - Results



Remarks

- Performed 18968 experiments in 21 hours
- Success rate between 3.14 µs and 3.44 µs: 1.3%
- ***Kernel "pwned" every 5 minutes***

Summary

- Security boundary bypass
  - Full access to kernel memory
  - Root shell execution
- Not dependent on software vulnerabilities
- For these attack specific checks are targeted
  - No need not know which check exactly

*Traditional SW countermeasures **do** apply!*

Let’s go a little deeper…

Fault injection fault model

*"A **theoretical model** for describing **the effects of fault injection**."*

- Some examples: instruction skipping and bit flipping
- Are used for envisioning new attacks
  - Instruction skipping leads to bypassing conditional checks
  - Bit flips lead to cryptographic attacks
- Are used for identifying vulnerable targets
- Are used to invent new countermeasures

If it is not modeled…it may have not been researched. Yet.

Our fault model

*A generic one: “**instruction corruption**”*

Single-bit (MIPS)

```
addi $t1, $t1, 8     00100001001010010000000000001000
addi $t1, $t1, 0     00100001001010010000000000000000
```

Multi-bit (ARM)

```
ldr w1, [sp, #0x8]   10111001010000000000101111100001
str w7, [sp, #0x20]  10111001000000000100011111100111
```

Remarks

- Limited control over which bit(s) will be corrupted
- Also includes other fault models as sub-cases (e.g. instruction skipping)

Direct PC control

- ARM32 has an interesting ISA
- Program Counter (PC) is directly accessible

Valid ARM instructions

```
MOV r7,r1            00000001 01110000 10100000 11100001
EOR r0,r1            00000001 00000000 00100000 11100000
LDR r0,[r1]          00000000 00000000 10010001 11100101
LDMIA r0,{r1}        00000010 00000000 10010000 11101000
```

Corrupted ARM instructions may directly set PC!

```
MOV pc,r1            00000001 11110000 10100000 11100001
EOR pc,r1            00000001 11110000 00101111 11100000
LDR pc,[r1]          00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}    00000010 10000000 10010000 11101000
```

Attack variations (SP-control) also affect other architectures!

Direct PC control – Description

1. Set all registers to a specific value (e.g. 0x41414141)
2. Execute random Linux system calls
3. Load the arbitrary value into the PC register using a glitch

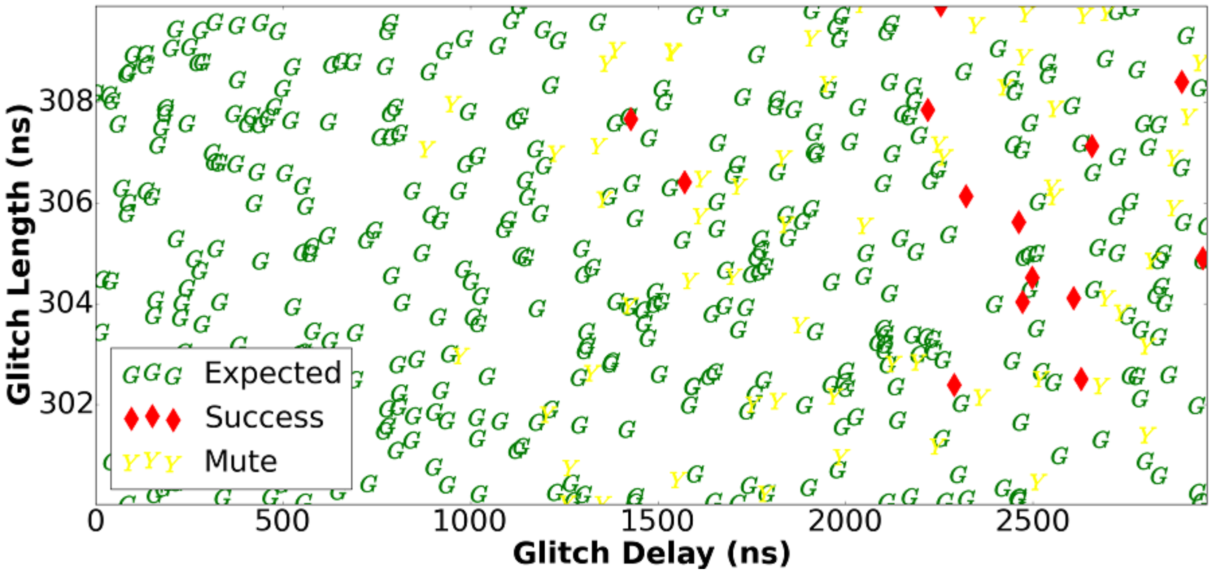
Control flow hijacked

Direct PC control – Code

```
. . .
int rand = random();
*(volatile unsigned int *)(trigger) = HIGH;

volatile (
  "movw r12, #0x4141;" // Repeat for other
  "movt r12, #0x4141;" // unused registers
  . . .
  "mov r7, %[rand];" // Random syscall nr
  "swi #0;"          // Linux kernel takes over
  . . .

*(volatile unsigned int *)(trigger) = LOW;
. . .
```

- Code running in userspace
- Linux syscall: initially random
- Found to be more effective: **getgroups** and **prctl**

Direct PC control – Results



Remarks:

- Performed 12705 experiments in 14 hours
- Success rate between 2.2 µs and 2.65 µs: 0.63%
- ***Control of PC in Kernel mode gained every 10 minutes***

Video demonstration

Direct PC control – Summary

- Security boundary bypass
  - Kernel level code execution
- Not dependent on SW vulnerabilities
- Any instruction is a potential target

Why is this attack so special?

- **New** — *Yields software control with one successful fault*
- **Global** — *Any software instruction can be a target*
- **Direct** — *Software control is achieved immediately*
- **Precise** — *Load arbitrary values into arbitrary registers*
- **Powerful** — *Bypass security boundaries*
- **Unpredictable** — *Creates exec primitives out of thin air (e.g. a data only operation can be turned into an execution primitive)*

Impact

- Hardware FI countermeasures are fully applicable
  - They can target the injected glitch
- Software FI countermeasures are likely not executed
  - A successful attack hijacks control flow immediately
- Localized software FI countermeasures are insufficient
  - Any instruction is a potential target

Traditional software FI countermeasures are ineffective!

Exploit mitigations

- **Effective:** Limiting usage of an hijacked control flow
  - DEP/NX
  - ASLR
  - CFI
  - …
- **Not effective:** Preventing control flow hijacking:
  - Stack cookies
  - SEHOP
  - …

Wrapping up

Fault injection attack trends

- Reaching a wider audience
- Equipment is becoming accessible
  - May not even be needed!
- Research is increasing
- New powerful techniques subverting software boundaries
- Current fault injection countermeasures are mostly insufficient
- Fault injection attacks can be cheaper than a software exploit

Improving products

- Include fault injection attacks in your threat model
- Design and implement fault injection resistant hardware
  - Start from early design.
  - Test during implementation cycles
  - Test, test…and test again!
- Implement software with strong exploit mitigations
- Make critical assets inaccessible to software
  - E.g. Using *“real”* hardware

Conclusions

1. ***Fault injection attacks are coming to the masses.***
   (and will not go away)

2. ***They can easily subvert typical software security models.***
   (Adjust your threat models)

3. ***Any unprotected device is vulnerable.***
   (Factor in countermeasures from the start)

Questions?

Niek Timmers
Senior Security Analyst
@tieknimmers / niek@riscure.com

Cristofaro Mune
Product Security Consultant
@pulsoid / c.mune@pulse-sec.com

https://www.riscure.com/careers