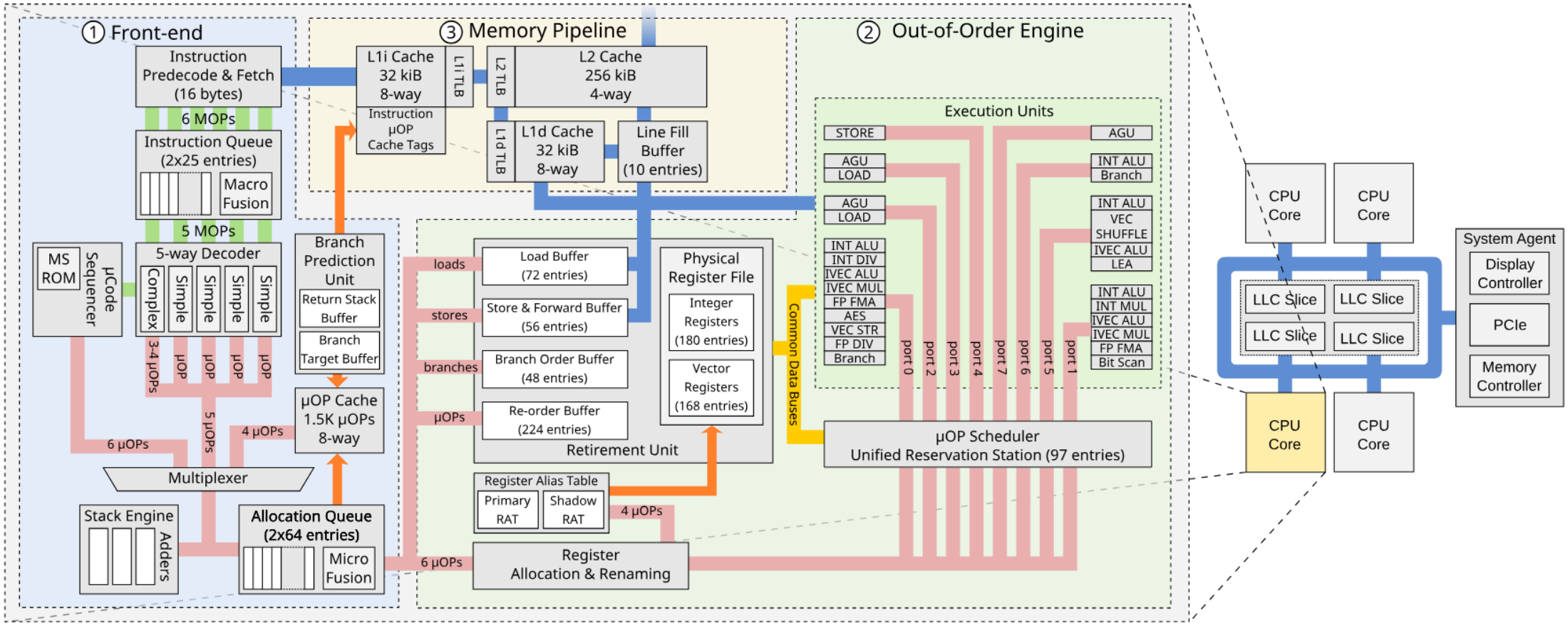
Voltage Glitching Intel Microcode

Federico Cerutti / ceres-c
Università degli Studi di Brescia
Previously: VU Amsterdam

TL;DR

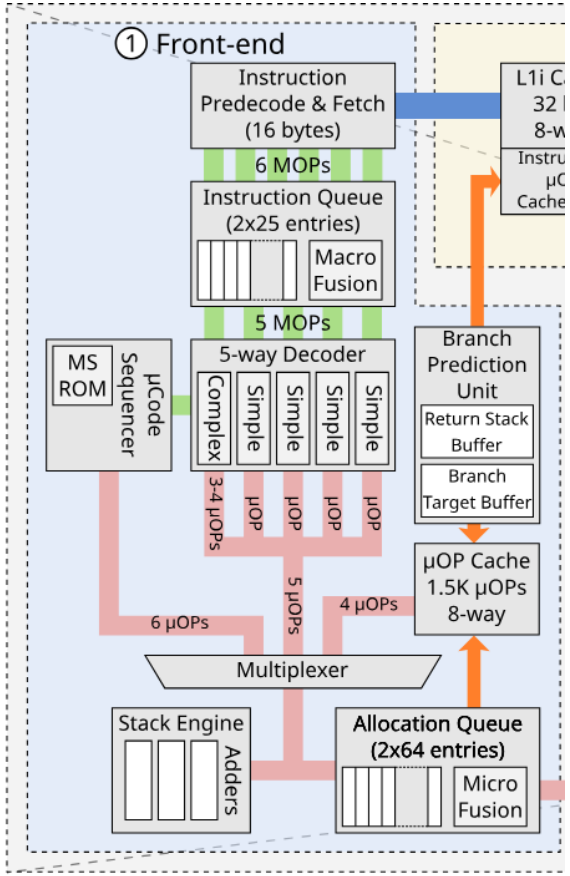
- **What:** Glitch Intel microcode
- **How:** Voltage glitching
- **Why:** Attack microcode update
- Did it **work**? *Kinda no?*

CPU 101



Custom Processing Unit: Tracing and Patching Intel Atom Microcode – Pietro Borrello, Martin Schwarzl

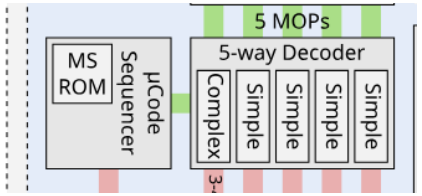
Arch → μArch instructions



1. **Fetch**
2. Decode queue
   (Instruction fusion)
3. **Decode**:
   a) Simple decoder — 1 x86 = 1 μ-op
   b) Complex decoder — 1 x86 = 3/4 μ-op
   c) **Microcode sequencer** — 1 x86 = *n* μ-op
4. Allocation queue
5. …

We pretend speculative execution does not exist

Microcode Sequencer



- Implements **complex** instructions
- Does **not execute** code directly, however has:
  - ROM
  - RAM
  - Registers
    - Instruction pointer
  - Peripherals
- **Updatable**
  - **Signed** updates

Previously on: Red Unlock

- Intel products have extensive testing & **debugging** features

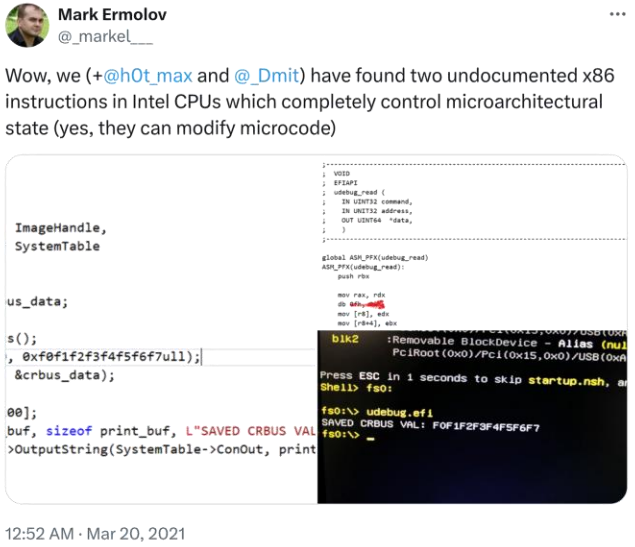
- Can be activated via:
  - Hardware straps (preproduction only)
  - Efuses (disabled on non-dev)
  - JTAG password
  - Software

Previously on: Red Unlock

Ermolov, Goryachy and Sklyarov

- Intel products have extensive testing & **debugging** features

- Can be activated via:
  - Hardware straps (preproduction only)
  - Efuses (disabled on non-dev)
  - JTAG password
  - **Software**



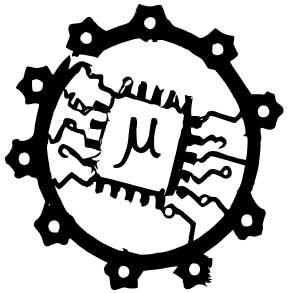
Introducing: Intel ME exploits on **Goldmont** SoCs

Previously on: Red Unlock - Goldmont

Ermolov, Goryachy and Sklyarov

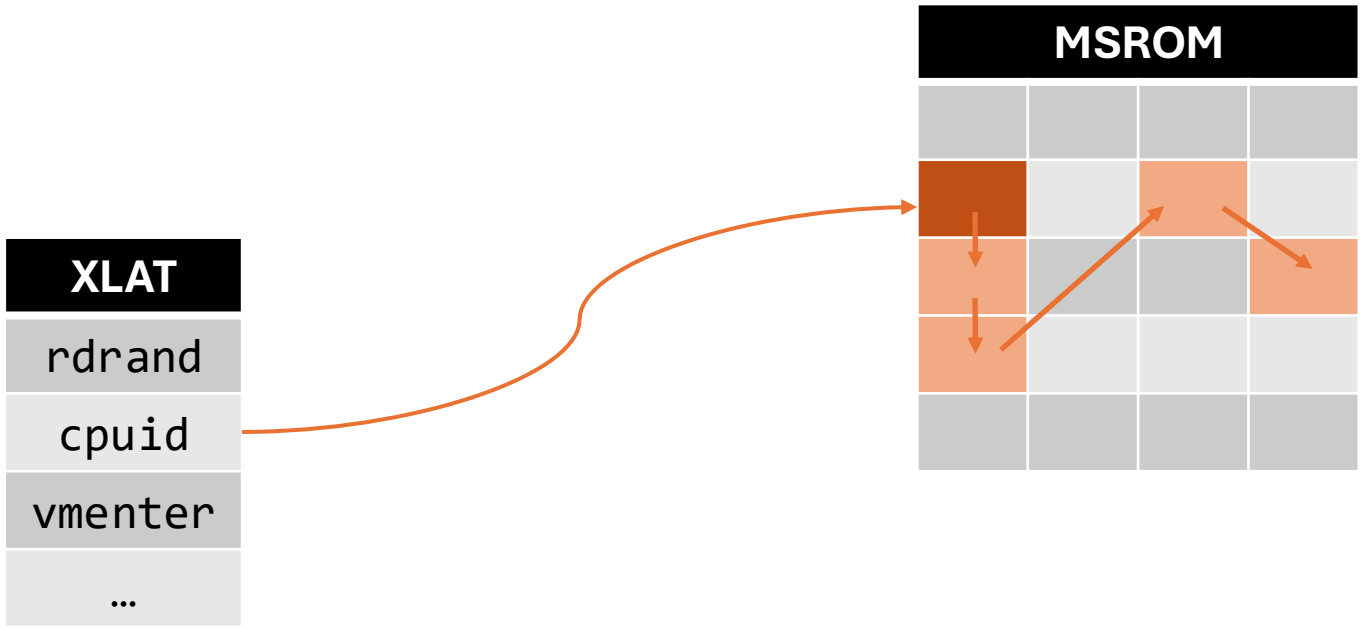
1. Exploit ME 11
2. Extract microcode with JTAG
3. Reverse engineer **architecture**
4. Find **architectural** instructions to **read/write** microarchitectural state
5. ???
6. Write **your own microcode**!

CustomProcessingUnit[1] lib-micro[2]

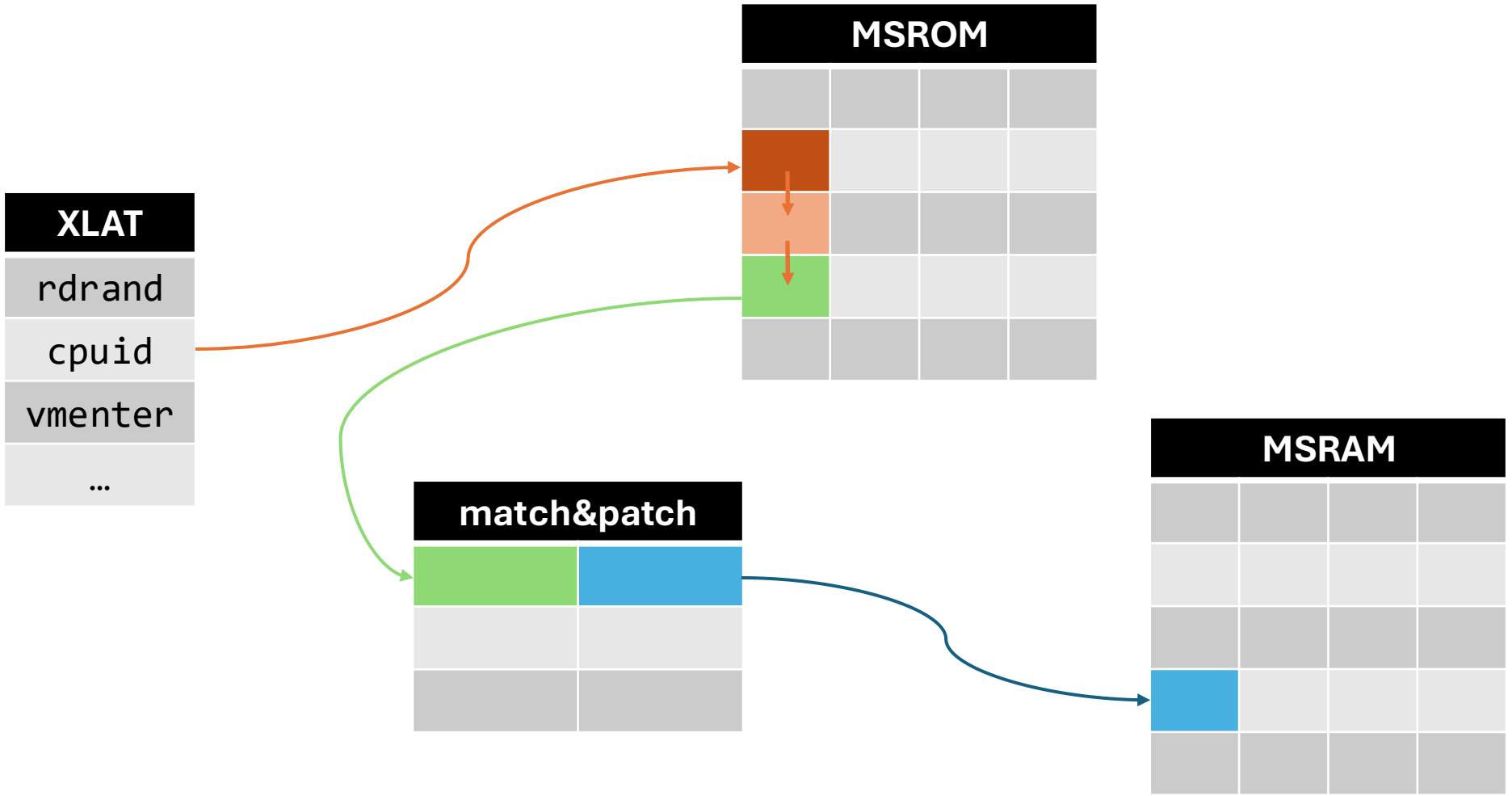


1: https://github.com/pietroborrello/CustomProcessingUnit/
2: https://github.com/zanderdk/lib-micro/

Custom Microcode



Custom Microcode

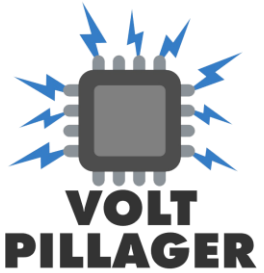


Previously on: x86 Voltage Glitching

- Intel and AMD CPUs proven vulnerable to **voltage fault injection**

- **No changes** to the power supply, leverage the PMIC[1]
- Two ways:
  - **Software** based: Plundervolt, VoltJockey
  - **Hardware** based: VoltPillager, PMFault

- Glitch **x86** or **secure enclaves**

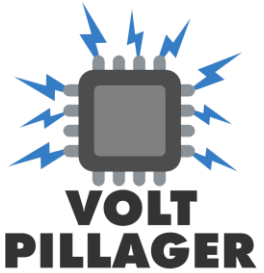


1: Power Management Integrated Chip

Previously on: x86 Voltage Glitching

- Intel and AMD CPUs proven vulnerable to **voltage fault injection**

- **No changes** to the power supply, leverage the PMIC[1]
- Two ways:
  - **Software** based: Plundervolt, VoltJockey
  - **Hardware** based: VoltPillager, PMFault



- Glitch **x86** or **secure enclaves**

How about **microcode**?

1: Power Management Integrated Chip

Roadmap

1. Find **hardware**
   - Must be **red unlocked**
   - Possibly simple I/O
2. Build **software** setup
   - x86 convoluted bootstrap
   - Make it **fast**
   - How2microcode?
3. Port **Plundervolt** x86 results
4. Attack custom **microprograms**
5. Attack microcode **update**

Pick (red unlocked) hardware

Gigabyte BRIX BPCE-3350C [1]

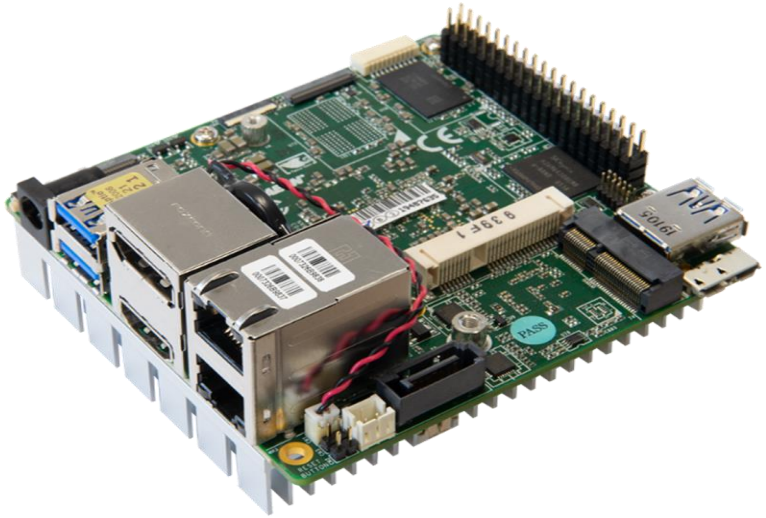
- Intel Celeron N3350
- AMI BIOS

- Ermolov et al.

Up Squared [2]

- Intel Pentium N4200
- **coreboot**
- **GPIO**

- lib-micro



1: https://github.com/ptresearch/IntelTXE-PoC
2: https://github.com/zanderdk/lib-micro/

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**
coreboot (32-bit) vs. microcode libs (64-bit)	

Software

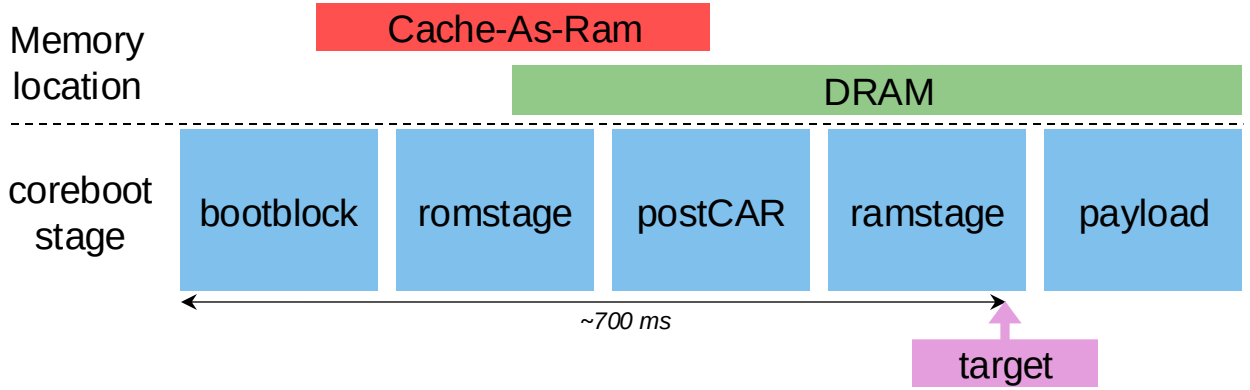
Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**
coreboot (32-bit) vs. microcode libs (64-bit)	**Port** lib-micro

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**
coreboot (32-bit) vs. microcode libs (64-bit)	**Port** lib-micro
Custom microcode in early boot **halts CPU**	

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**
coreboot (32-bit) vs. microcode libs (64-bit)	**Port** lib-micro
Custom microcode in early boot **halts CPU**	Find **stable** execution point

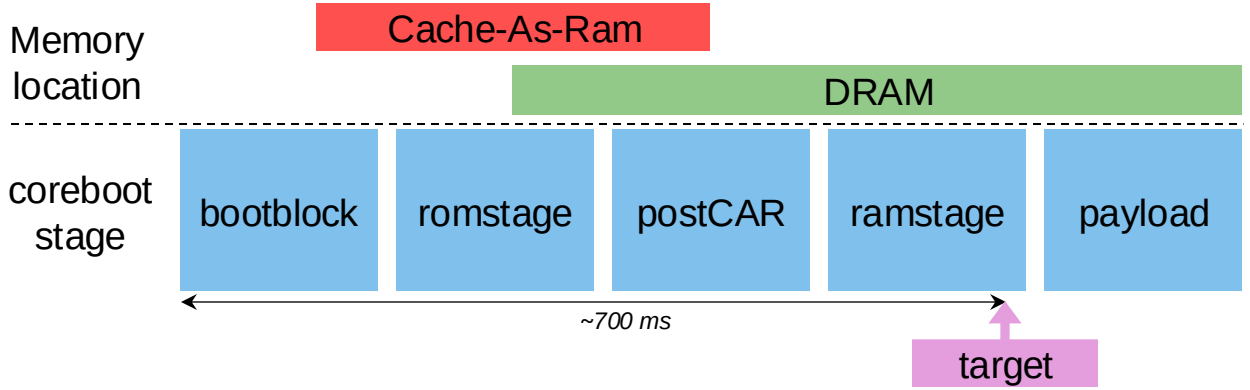


Up Squared + **coreboot** is the way to go

Software

Issue	Solution
BIOS boot time w/ red unlock > **2min**	Target code in **coreboot**
coreboot (32-bit) vs. microcode libs (64-bit)	**Port** lib-micro
Custom microcode in early boot **halts CPU**	Find **stable** execution point

(manually 🤡)



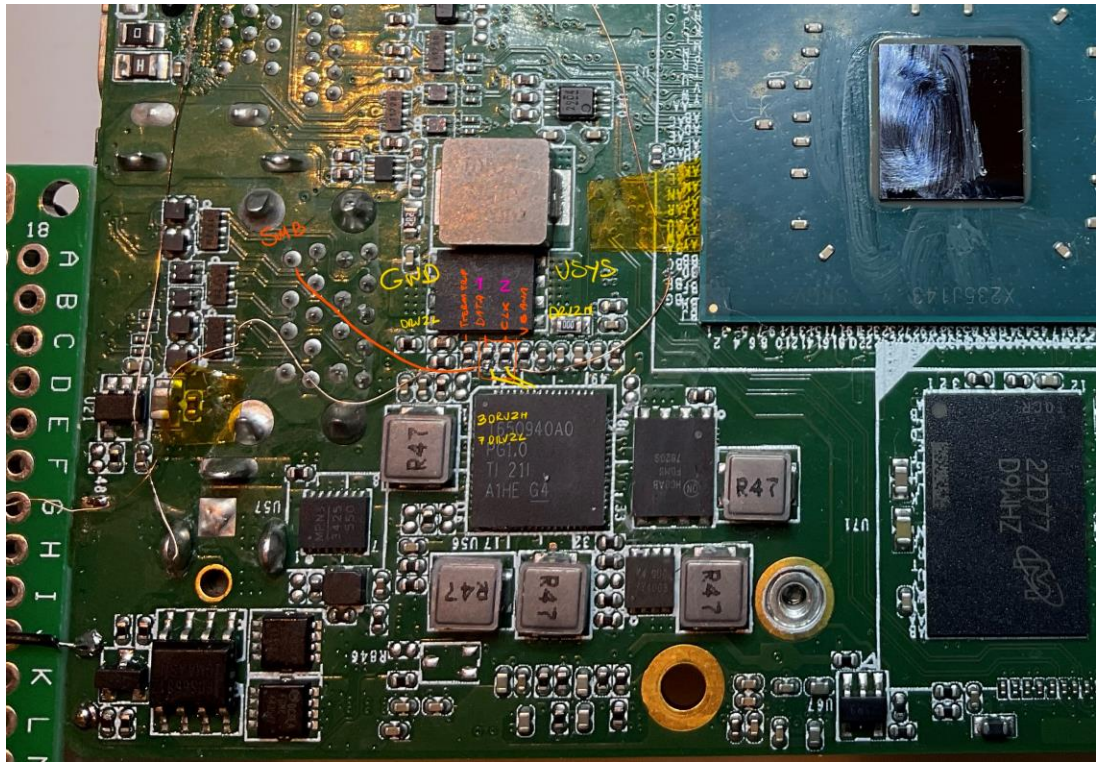
Up Squared + **coreboot** is the way to go

Inject the glitch

Control the **PMIC:**

- Software
- Hardware

Bonus: find **VCore**



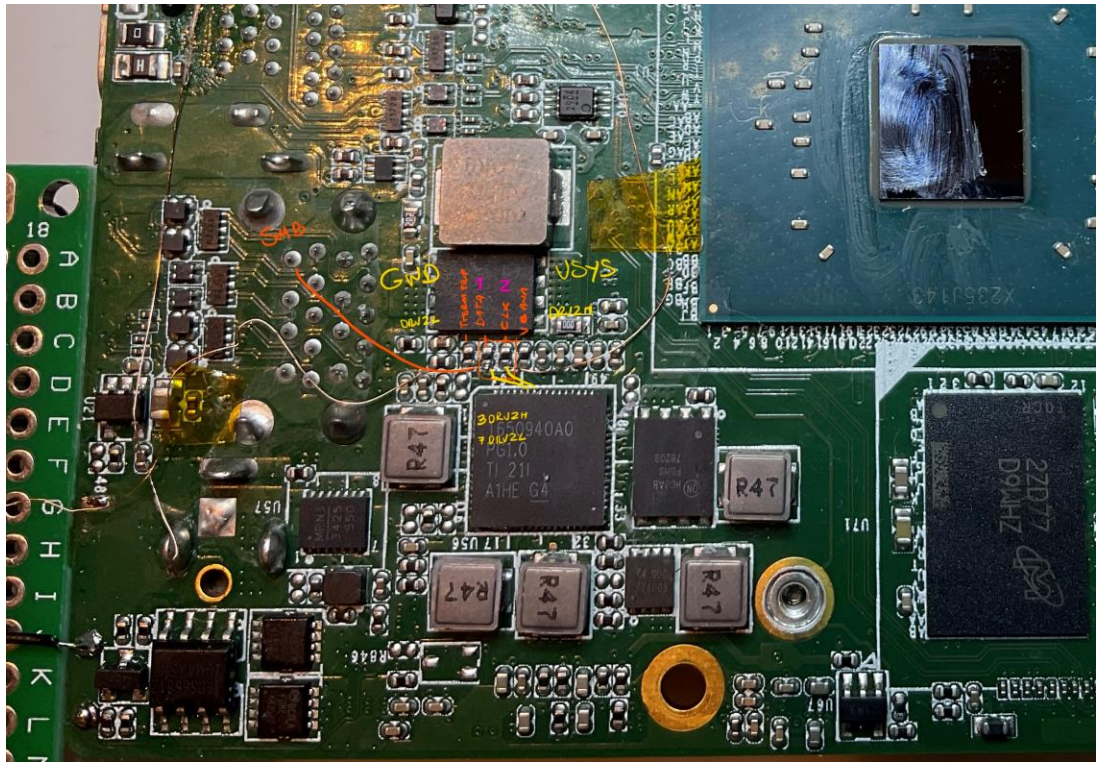
Thanks to Carlo Maragno

Inject the glitch

Control the **PMIC:**

- ~~Software~~
- **Hardware**

Bonus: find **VCore**

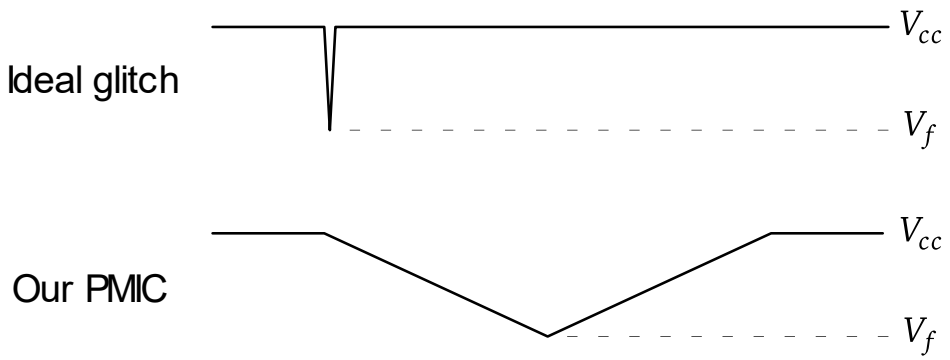


Thanks to Carlo Maragno

Interjection: Glitch characteristics

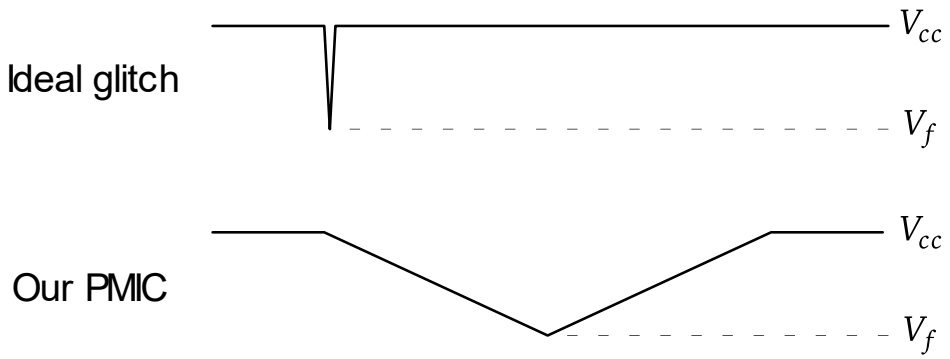
1: Inspired by VoltPillager

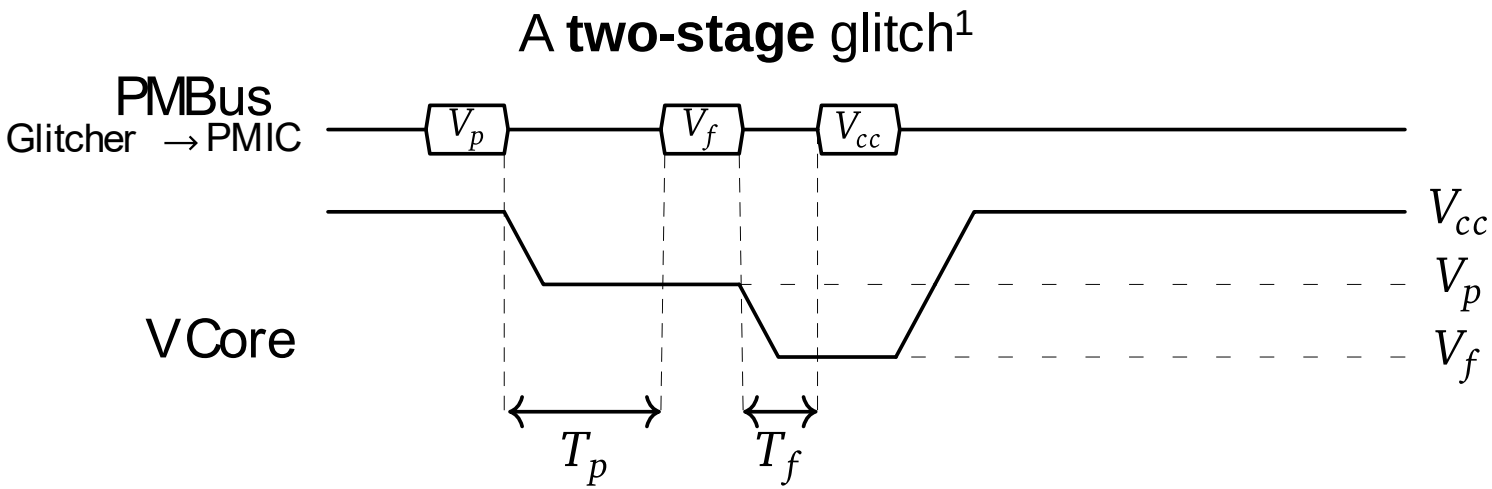
Interjection: Glitch characteristics



1: Inspired by VoltPillager

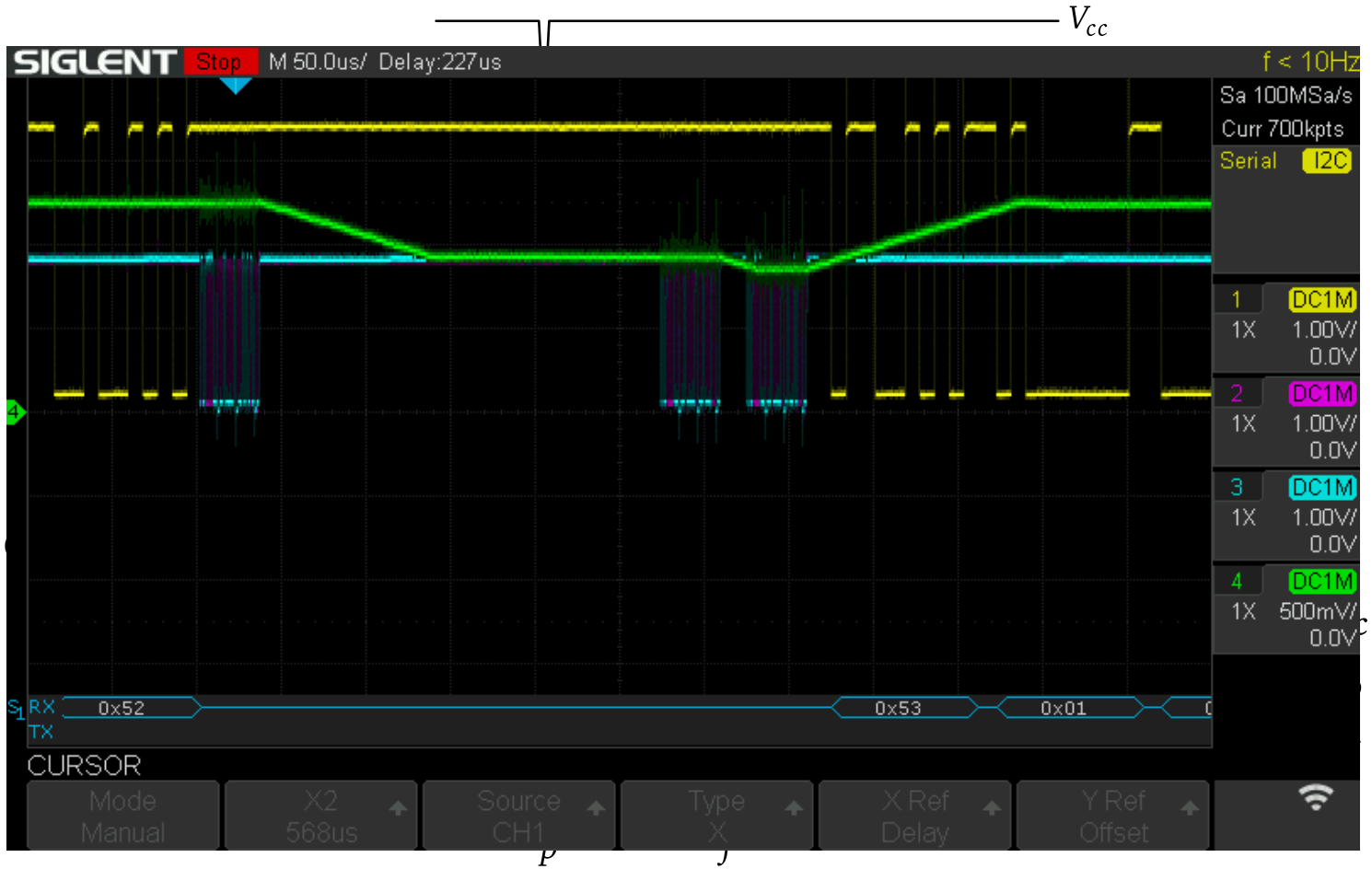
Interjection: Glitch characteristics





1: Inspired by VoltPillager

Slow PMIC - Workaround



1: Inspired by VoltPillager

Injecting the glitch

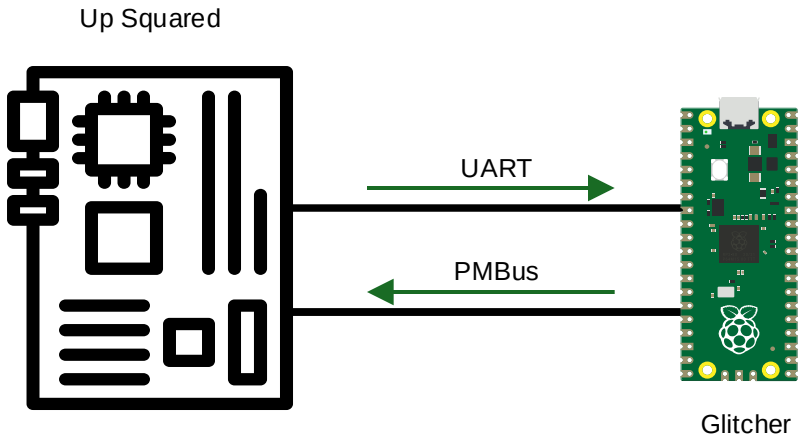
- Raspberry **Pi Pico** FTW
- GPIO @ **3v3** :(
- **Modify** Pi Pico
  - IOVDD @ 1v8
  - USB_VDD @ 3v3
  - Swap SPI flash

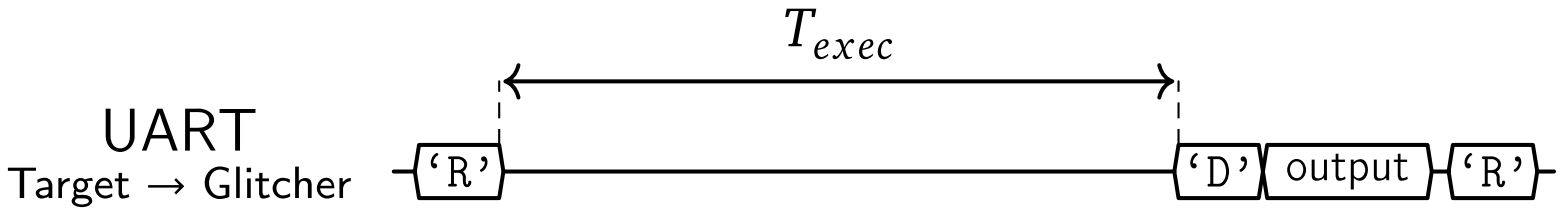
Getting them to talk

GPIO? Not really

CPLD proxy

UART will do!

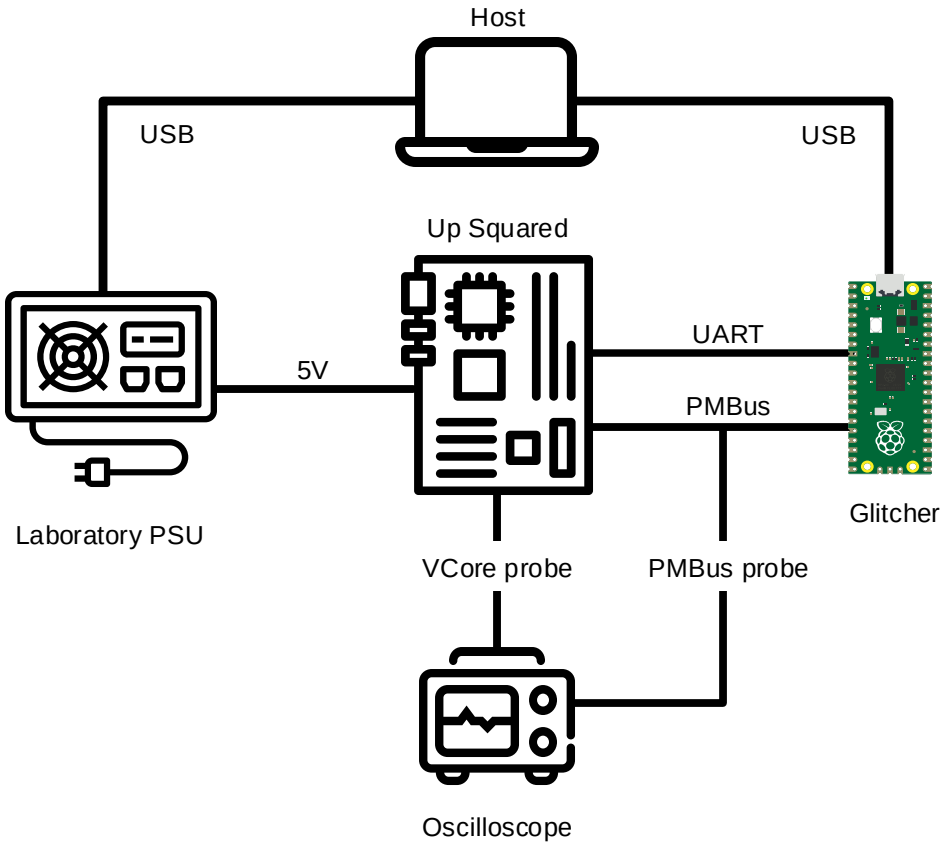




Setup overview

5 components:

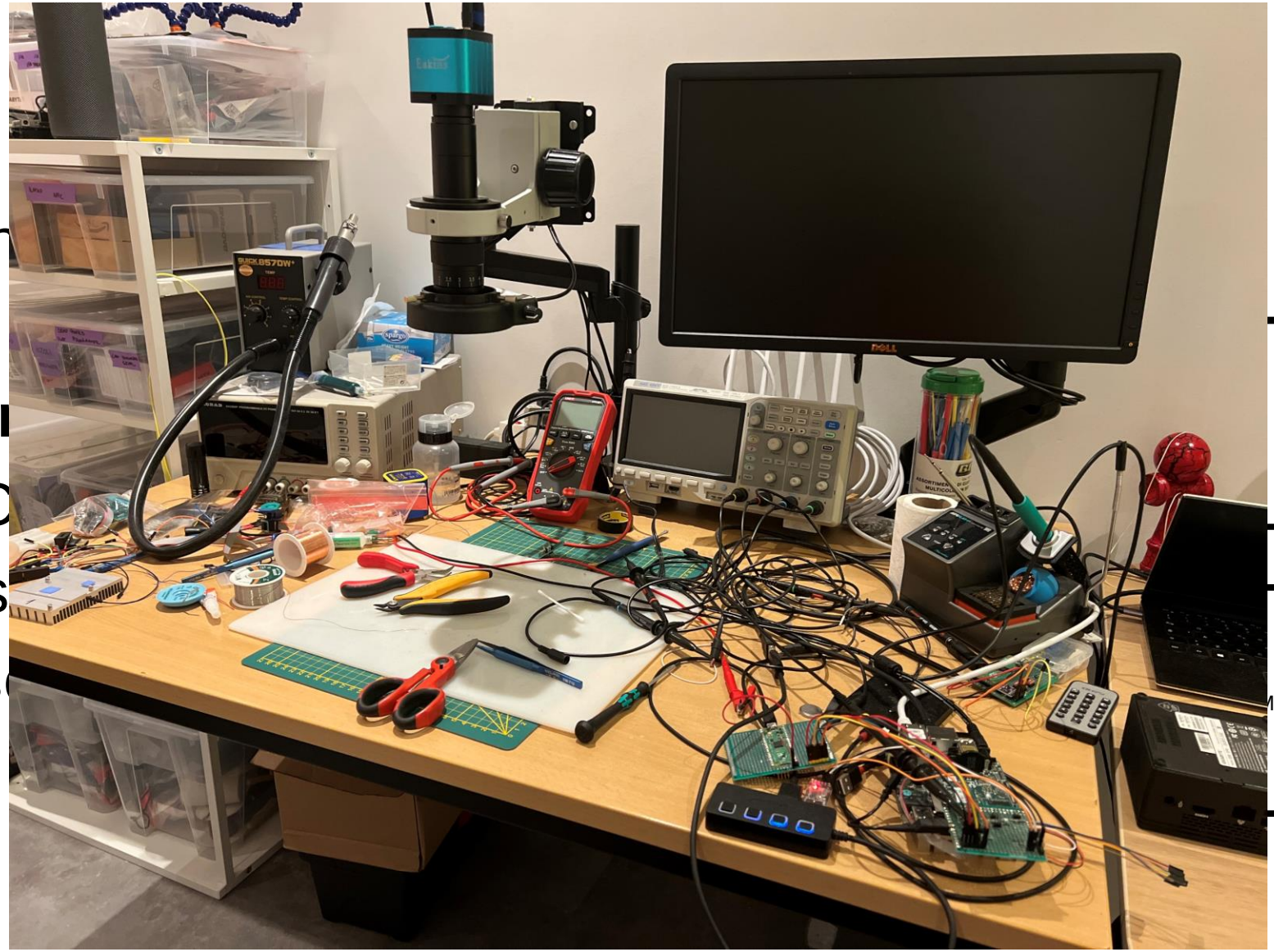
- **Target**: *Up Squared* board
- **Glitcher**: Raspberry Pi Pico
- Host PC
- Power supply
- Oscilloscope

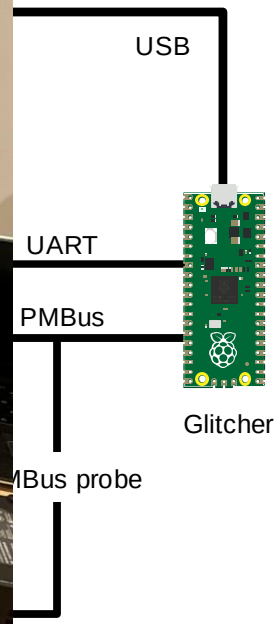


Setup

5 compon

- **Target**:
- **Glitche**
- Host PC
- Power s
- Oscillos





Previous research: Plundervolt

- Simple arithmetic operations can’t be glitched

  `ADD/SUB`, `OR/XOR/AND`: **nope**

- **`IMUL`** can be faulted with:
  - specific **operands**
  - specific operand **order**, e.g.
    - `0x80000 * 0x4`: **Faulty**
    - `0x4 * 0x80000`: **Not faulty**
    - `0x80001 * 0x4`: **Not faulty**

x86 assembly: `IMUL`

```
movl $0x80000, %eax;
movl $0x4, %ebx;
movl %eax, %edx;
imull %ebx, %edx;     # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;     # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;
```

1. Prepare operands
2. Multiply (twice)
3. Compare
4. Set `dl`*`[7:0]`* if different
5. Add `dl`*`[7:0]`* to `cl`*`[7:0]`*

Normal: `ecx = 0`

Fault: `ecx > 0`

x86 assembly: IMUL

IMUL **can** be glitched!

```
movl $0x80000, %eax;
movl $0x4, %ebx;
movl %eax, %edx;
imull %ebx, %edx;    # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;    # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;
```

x86 assembly: IMUL

```
movl $0x80000, %eax;
movl $0x4, %ebx;
movl %eax, %edx;
imull %ebx, %edx;    # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;    # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;
```

IMUL **can** be glitched:

- Regardless of operand **order**
  - `0x80000 * 0x4`: **Faulty**
  - `0x4 * 0x80000`: **Faulty**

x86 assembly: IMUL

```
movl $0x80000, %eax;
movl $0x4, %ebx;
movl %eax, %edx;
imull %ebx, %edx;     # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;     # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;
```

IMUL **can** be glitched:

- Regardless of operand **order**
  - `0x80000 * 0x4`: **Faulty**
  - `0x4 * 0x80000`: **Faulty**

Is it **CMP**?

x86 assembly: IMUL

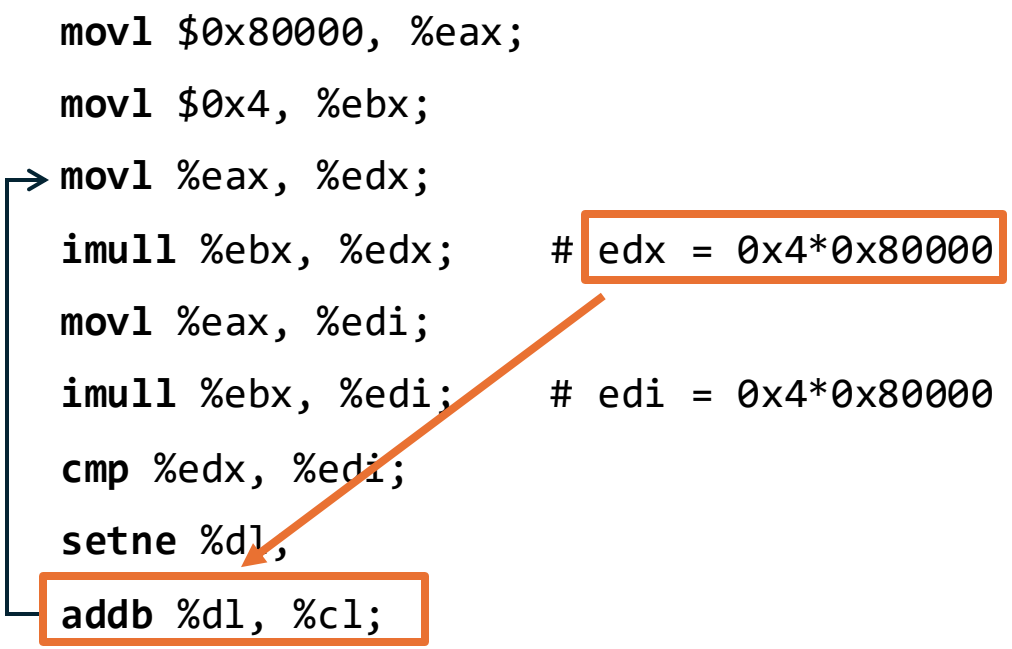
```
movl $0x80000, %eax;
movl $0x4, %ebx;
movl %eax, %edx;
imull %ebx, %edx;    # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;    # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;
```

IMUL **can** be glitched:

- Regardless of operand **order**
  - `0x80000 * 0x4`: **Faulty**
  - `0x4 * 0x80000`: **Faulty**

    Is it **CMP**?
- Sometimes
  - `ecx = 0x200000`

x86 assembly: IMUL



IMUL **can** be glitched:

- Regardless of operand **order**
  - `0x80000 * 0x4`: **Faulty**
  - `0x4 * 0x80000`: **Faulty**
  - Is it **CMP**?
- Sometimes
  - `ecx = 0x200000 (0x4*0x80000)`
  - Is **ADDB** adding 32 bits?
  - **Register file** issues?

x86 assembly: CMP

```
movl $0xAAAAAAAA, %eax;
movl $0xAAAAAAAA, %ebx;
cmp %eax, %ebx;
setne %dl;
addb %dl, %cl;
```

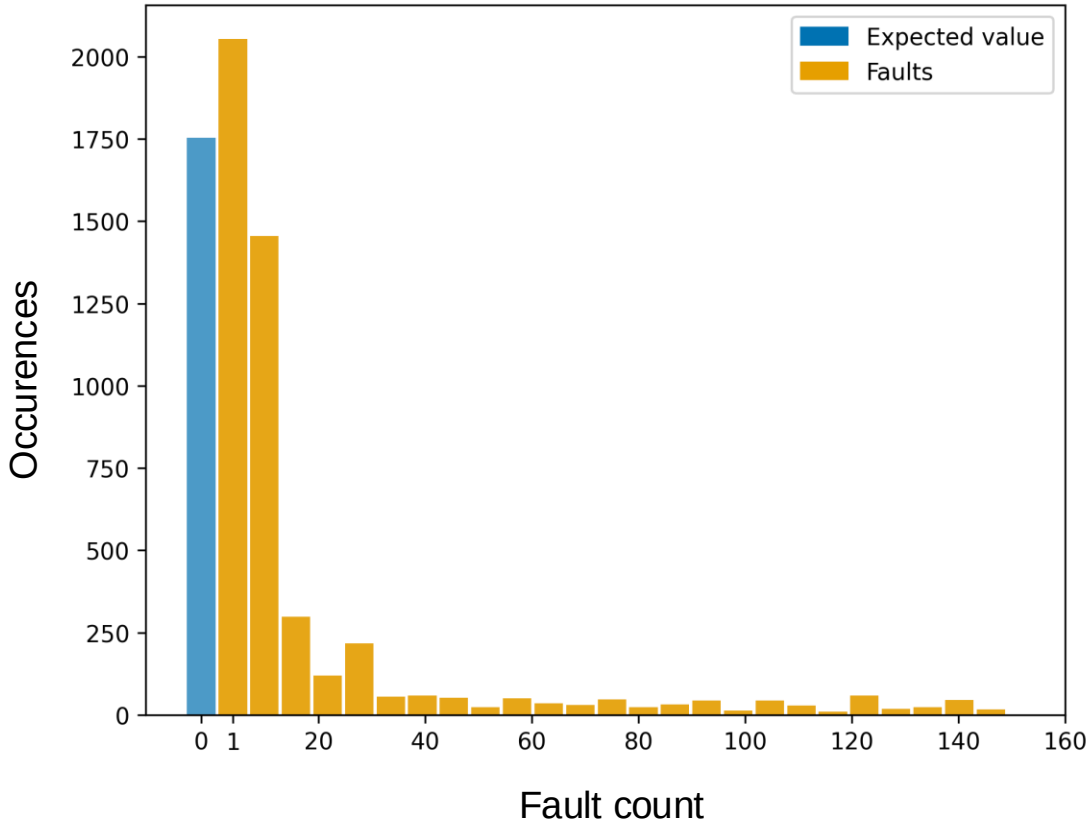
1. Prepare operands
2. Compare
3. Set `dl`*`[7:0]`* if different
4. Add `dl`*`[7:0]`* to `cl`*`[7:0]`*

Normal: `ecx = 0`

Fault: `ecx > 0`

x86 assembly: CMP

```
movl $0xAAAAAAAA, %eax;
movl $0xAAAAAAAA, %ebx;
cmp %eax, %ebx;
setne %dl;
addb %dl, %cl;
```



Yup, CMP can be glitched **75%** of the times

x86 assembly: Register file

```
mov $0x0101, %eax;
movb %al, %bl;
add %ebx, %ecx;
```

1. Prepare operand
2. Move al*[7:0]* to bl*[7:0]*
3. Add ebx*[31:0]* to ecx*[31:0]*

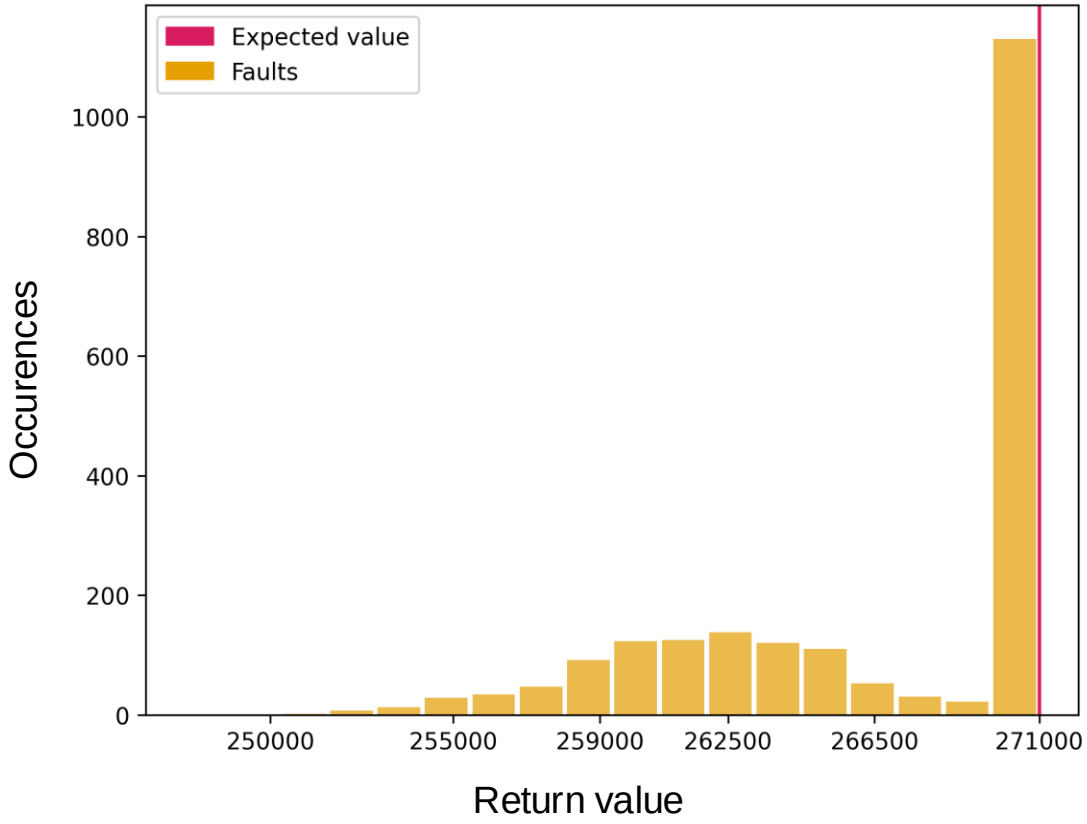
Normal: `ecx = i`

(*i* iterations)

Fault: `ecx > i`

x86 assembly: Register file

```
mov $0x0101, %eax;
movb %al, %bl;
add %ebx, %ecx;
```



Probably not register file, but **instructions skip**

Custom microcode: ADDs

Replace `RDRAND`

```
{
  ADD_DSZ64_DRI(RCX, RCX, 1),
  ADD_DSZ64_DRI(RCX, RCX, 1),
  ADD_DSZ64_DRI(RCX, RCX, 1),
  NOP_SEQWORD
}, /* ... */ {
  ADD_DSZ64_DRI(RCX, RCX, 1),
  NOP,
  NOP,
  END_SEQWORD
},
```

Add 1 to `ecx`, 10 times in a row

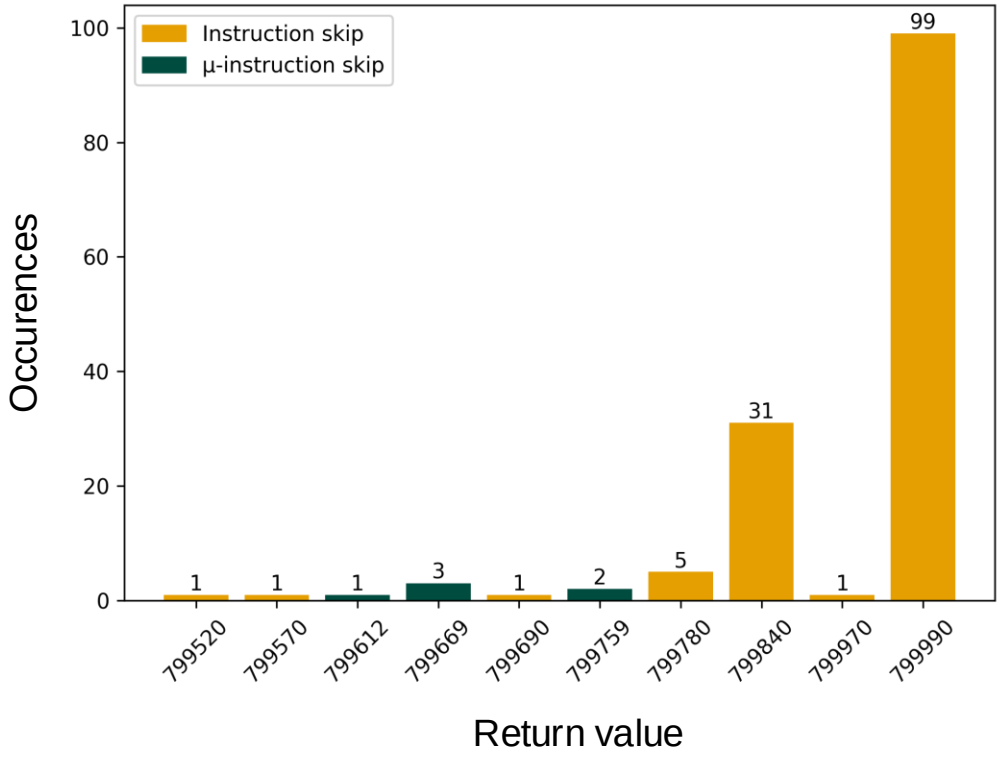
Normal: `ecx = i*10`
(*i* iterations)

Arch fault: `ecx % 10 = 0`
µarch fault: `ecx % 10 ≠ 0`

Custom microcode: ADDs

Replace `RDRAND`

```
{
  ADD_DSZ64_DRI(RCX, RCX, 1),
  ADD_DSZ64_DRI(RCX, RCX, 1),
  ADD_DSZ64_DRI(RCX, RCX, 1),
  NOP_SEQWORD
}, /* ... */ {
  ADD_DSZ64_DRI(RCX, RCX, 1),
  NOP,
  NOP,
  END_SEQWORD
},
```

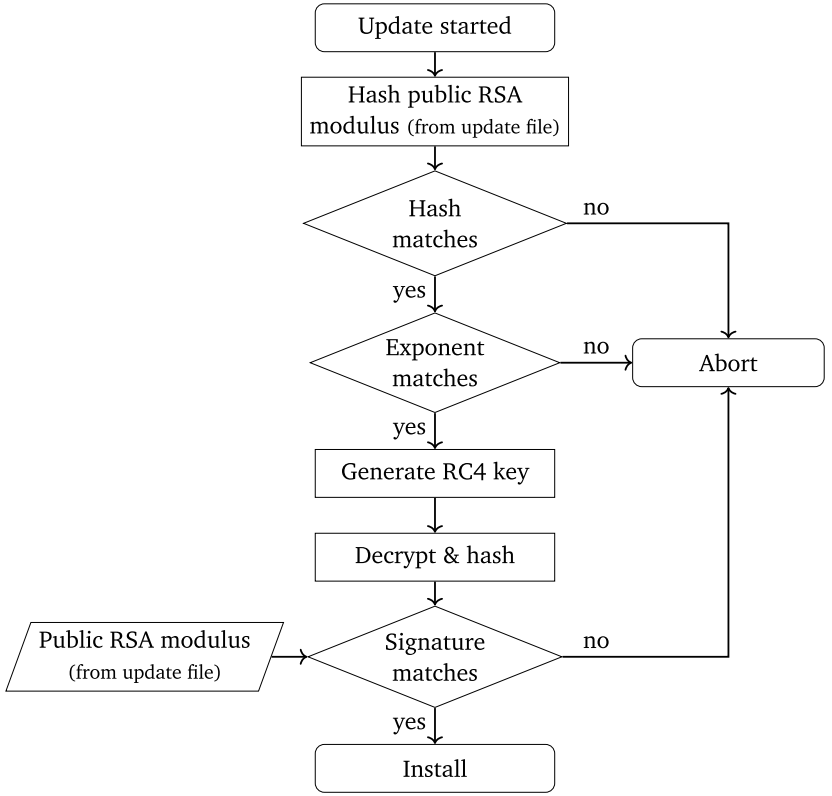


Both x86 and microcode **instruction skip**

Previously on: Microcode updates

- Documented in previous research
  - Treat it as a **gray box**
  - I **hope** it works like this

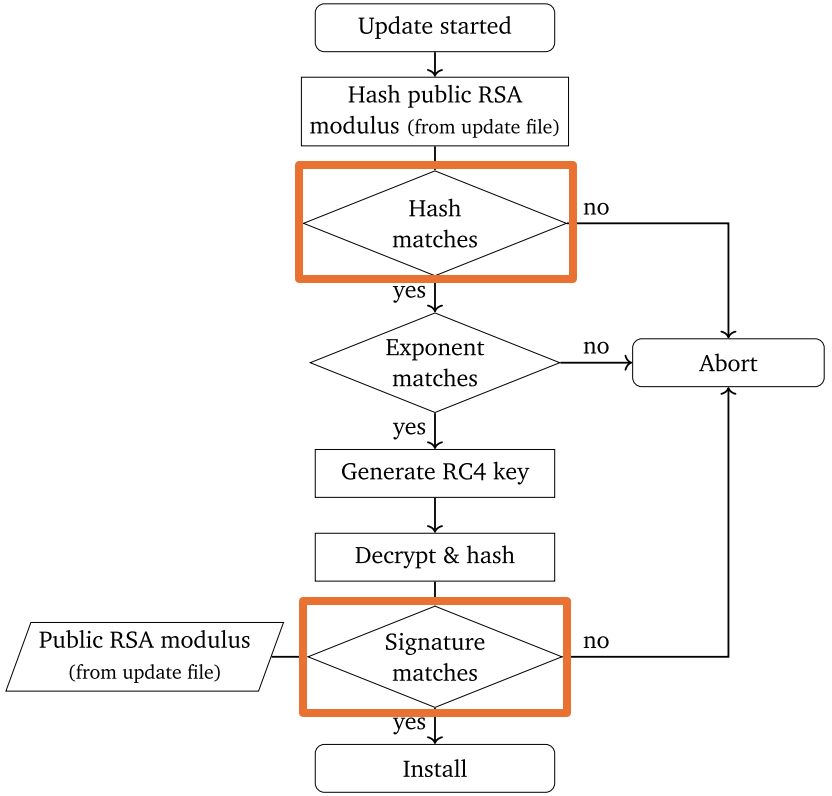
- **Secure** (enough) cryptographic primitives



Previously on: Microcode updates

- Documented in previous research
  - Treat it as a **gray box**
  - I **hope** it works like this

- **Secure** (enough) cryptographic primitives



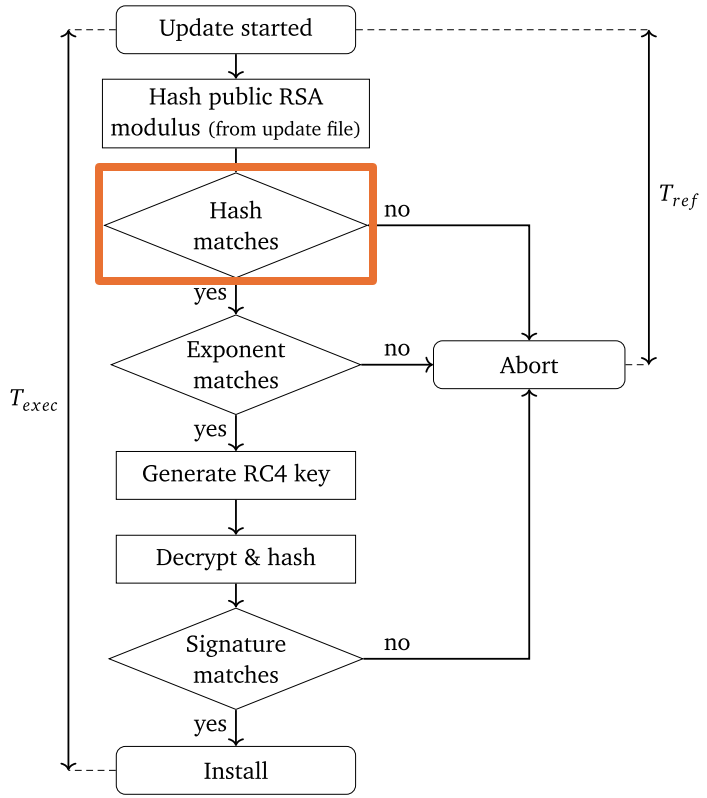
Can we attack these?

Microcode update: RSA modulus

Update is **not constant-time**

T_{exec} vs T_{ref}

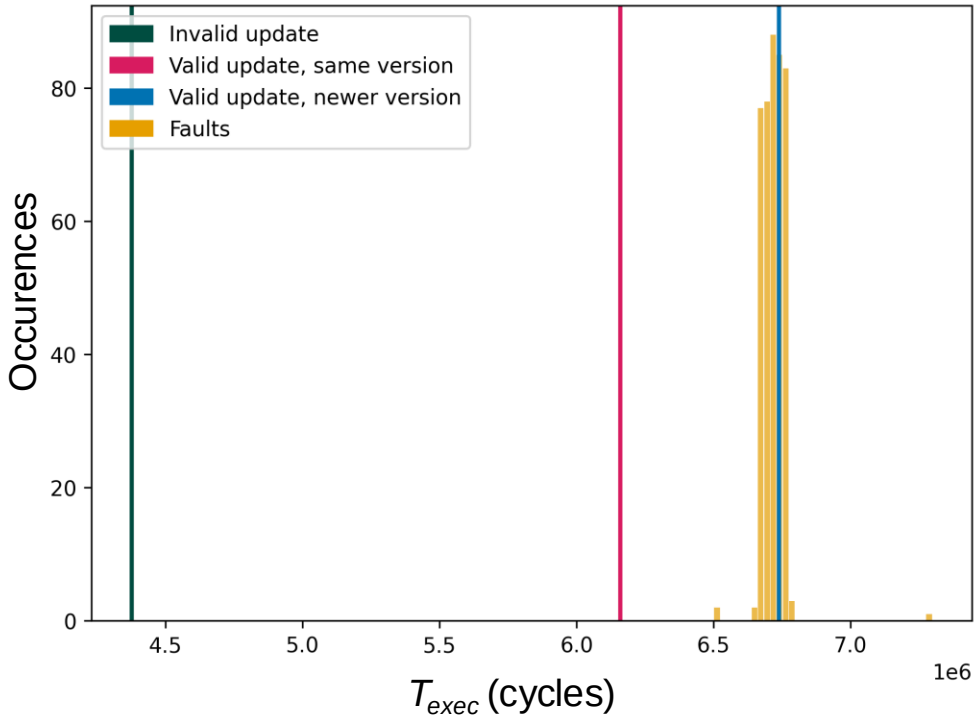
1. Change signature
2. Apply update (fail)
3. Measure T_{ref} (fail)
4. Glitch



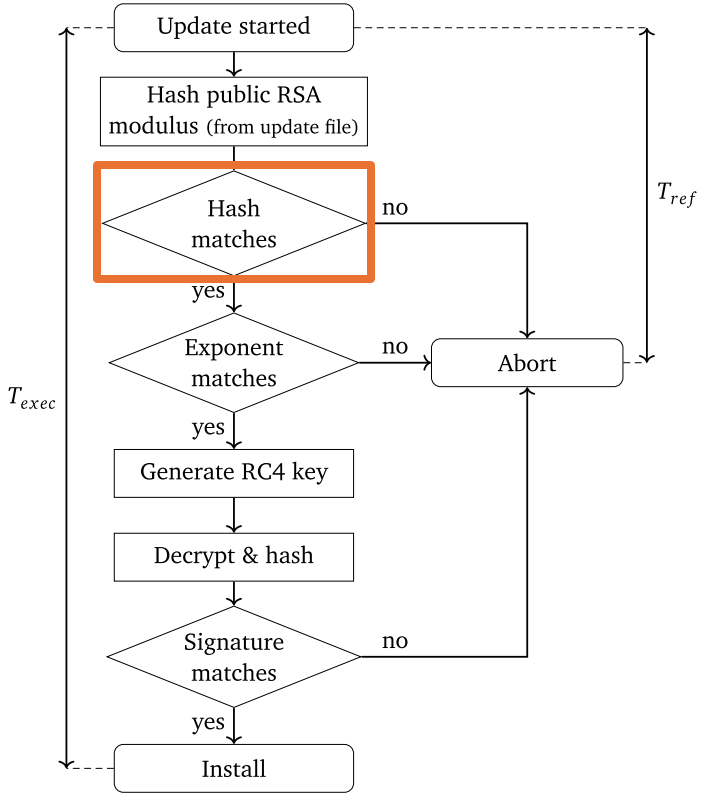
Normal: $T_{exec} = T_{ref}$

Fault: $T_{exec} > T_{ref}$

Microcode update: RSA modulus



Interesting timings :)



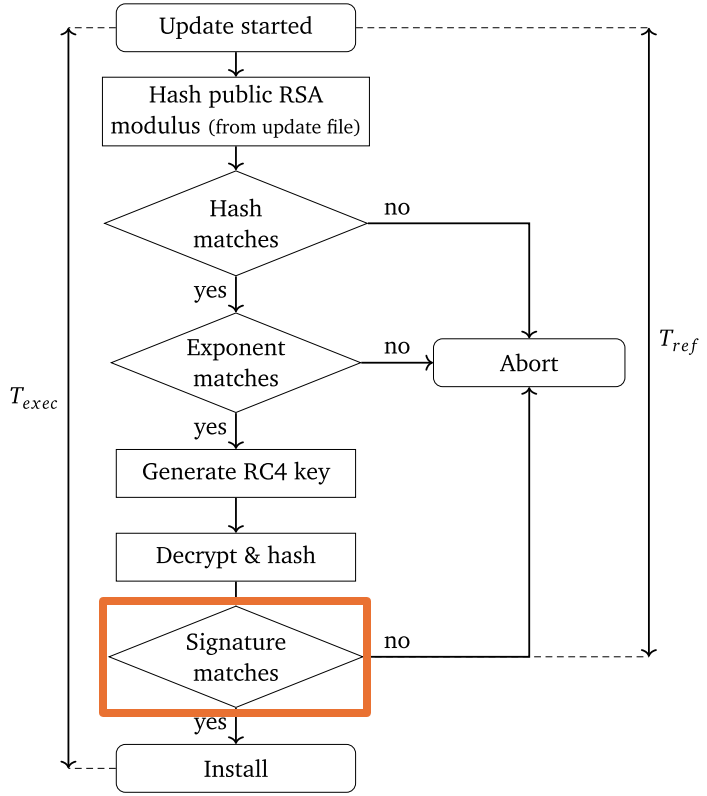
Fail update :(

Microcode update: Signature

Update is **not constant-time**

T_{exec} vs T_{ref}

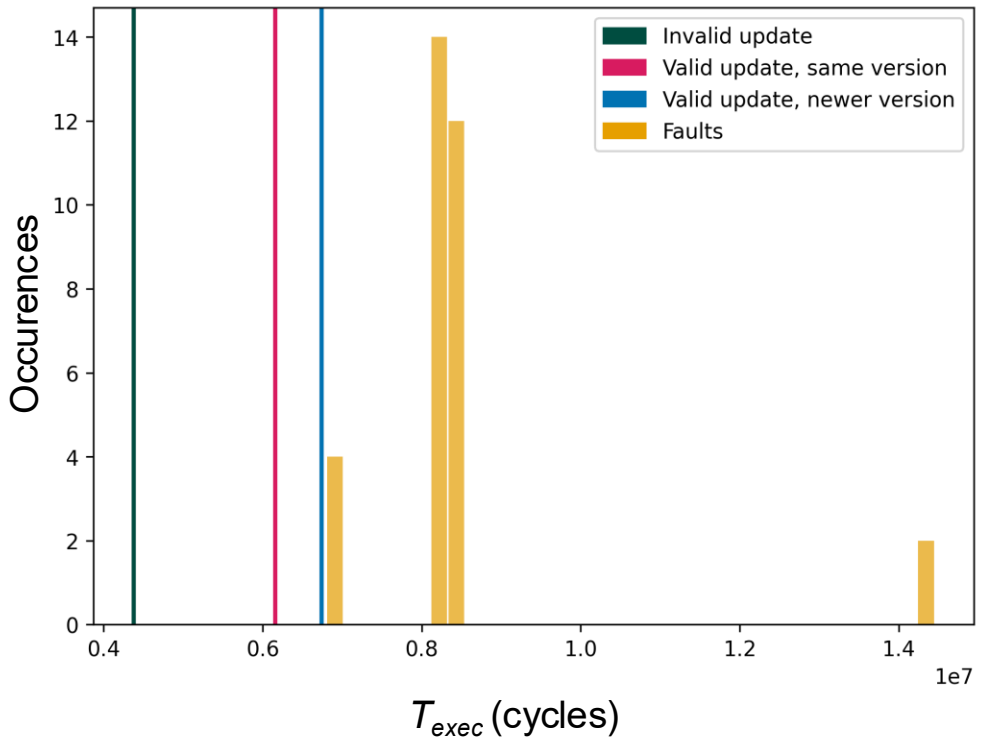
1. Change update content
2. Apply update (fail)
3. Measure T_{ref} (fail)
4. Glitch



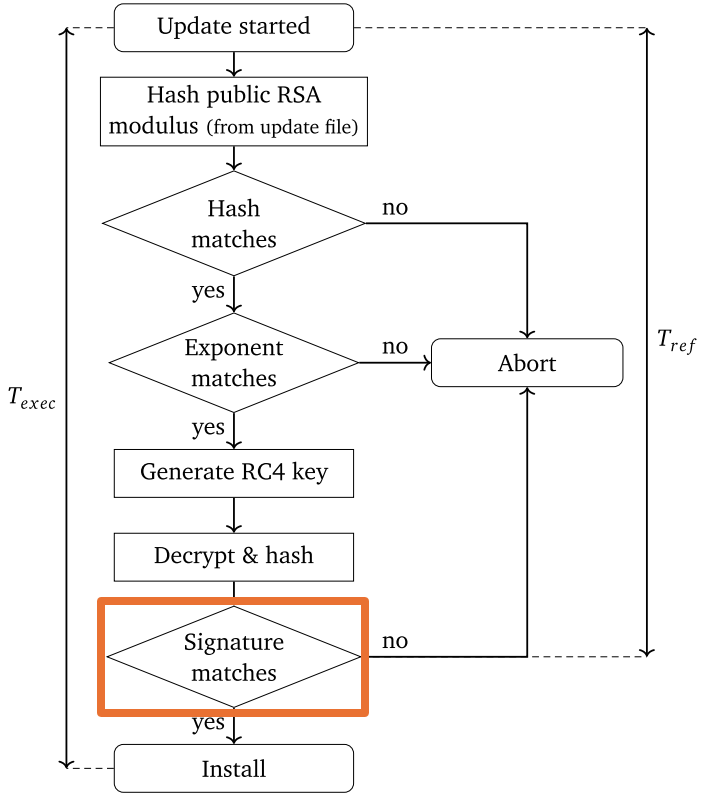
Normal: $T_{exec} = T_{ref}$

Fault: $T_{exec} > T_{ref}$

Microcode update: Signature



Meh timings :(



Fail update :(

Recap

Unsigned microcode update **rejected**

Evidence of:

- x86 **arithmetic faults**
- x86 instruction **skip**
- Register file **weirdness**
- µ-instruction **skip**
- Changed update program **behavior**

Future work:

- Reverse engineer microcode update
  - Why glitch not enough?
- Replace PMIC
- TOCTOU RSA modulus?

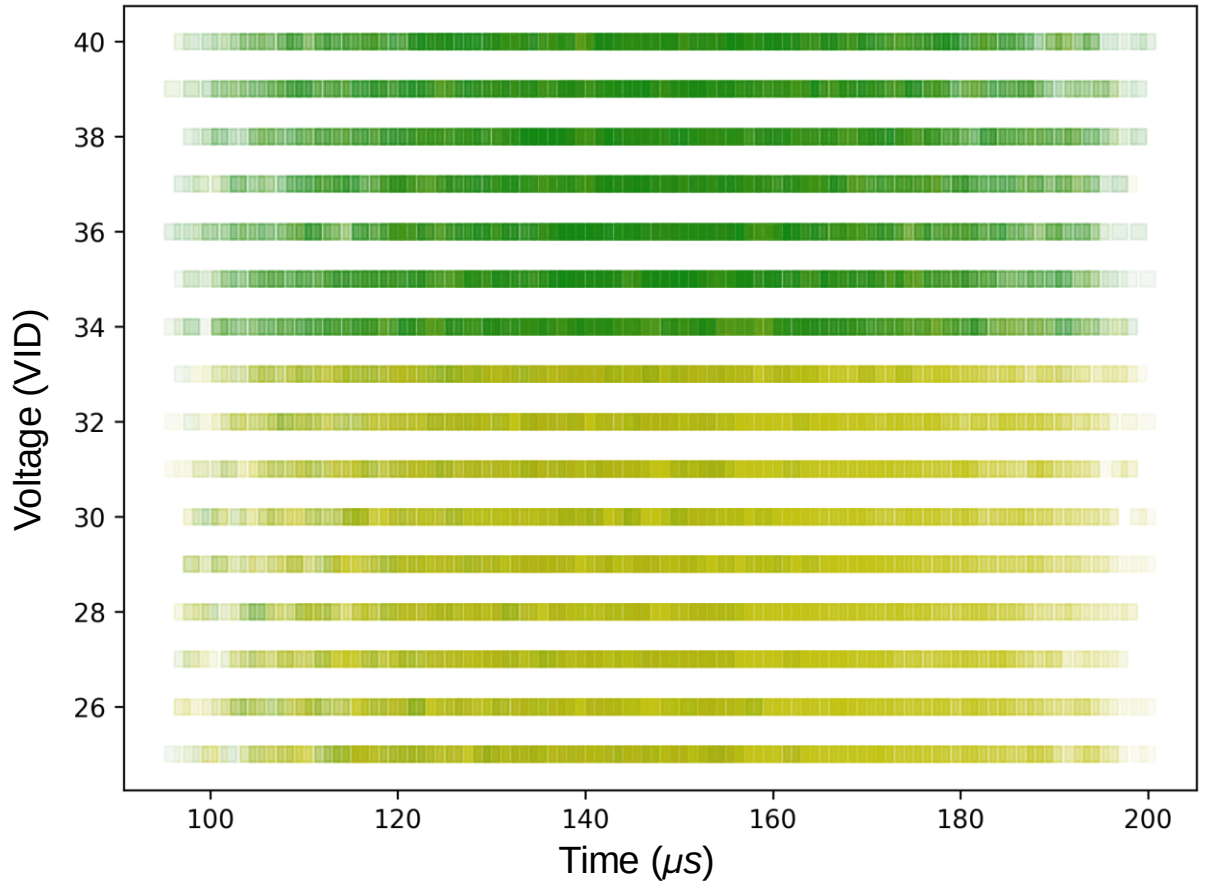
Questions, remarks, vituperation?

https://github.com/ceres-c/VU-Thesis

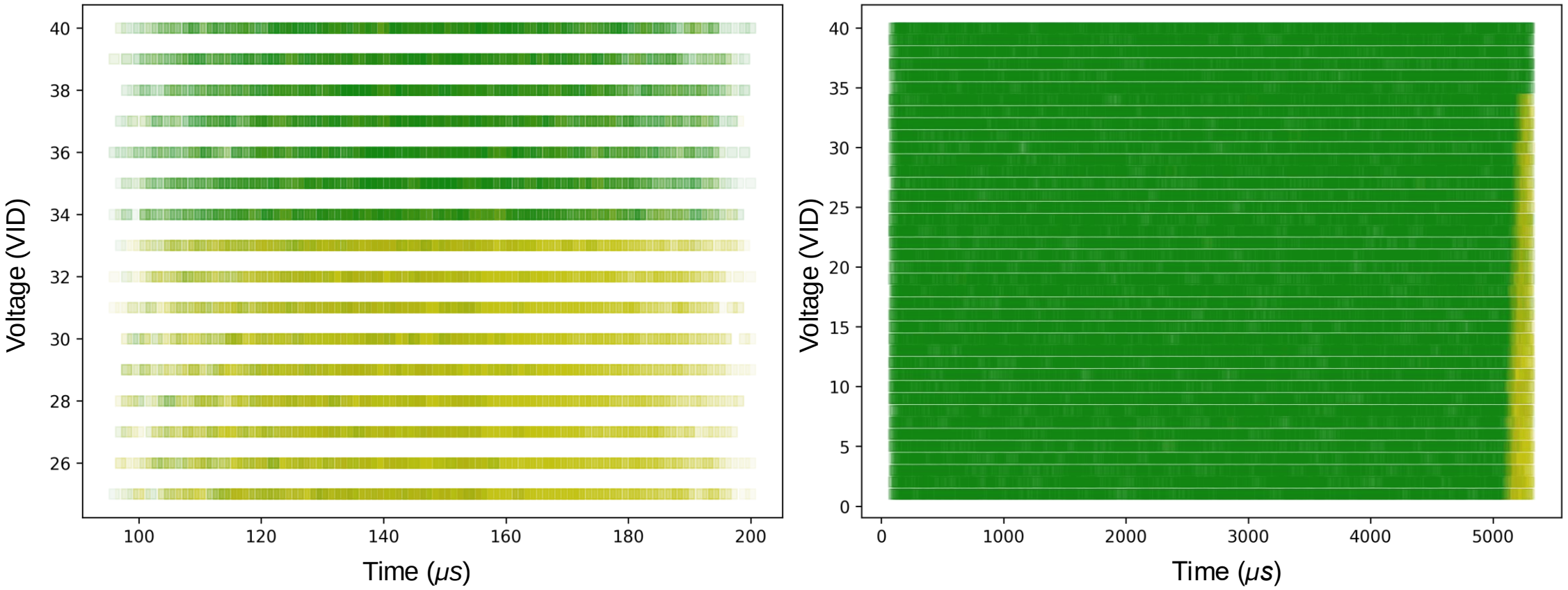
✉ federico@ceres-c.it https://ceres-c.it/

Backup slides

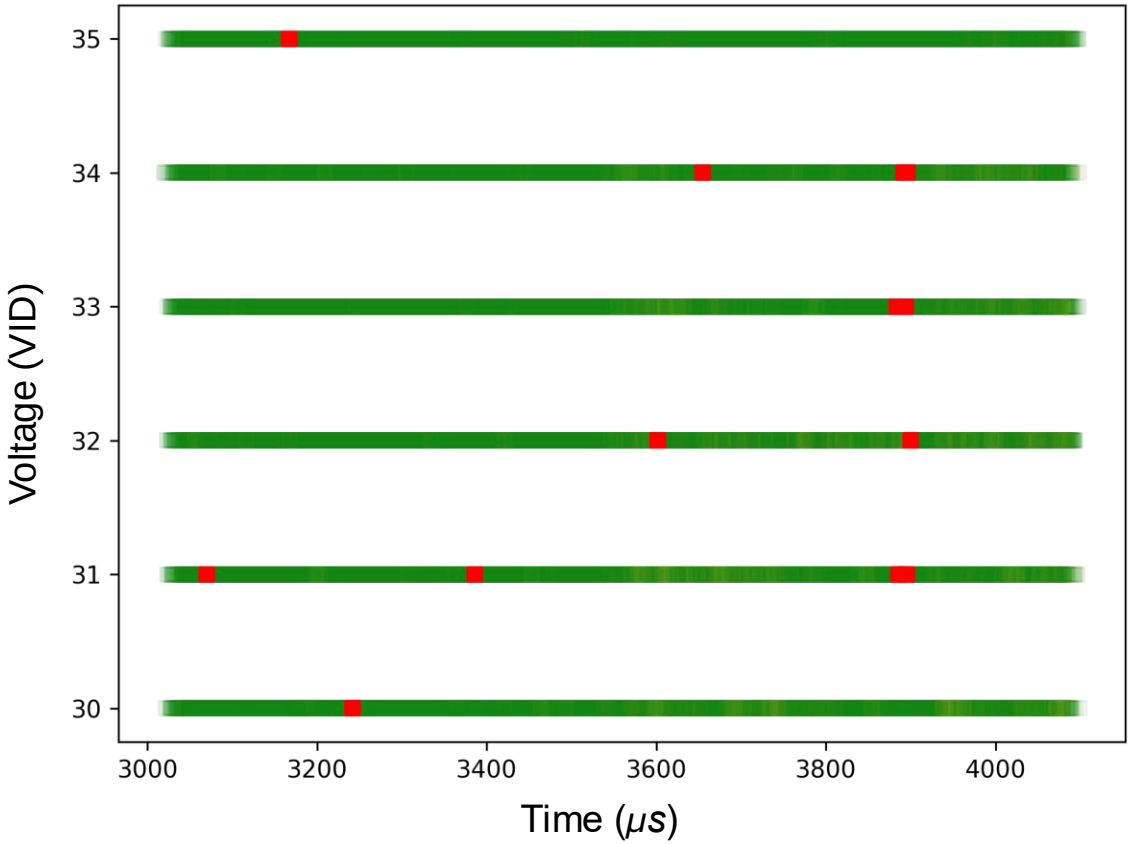
How to estimate voltages



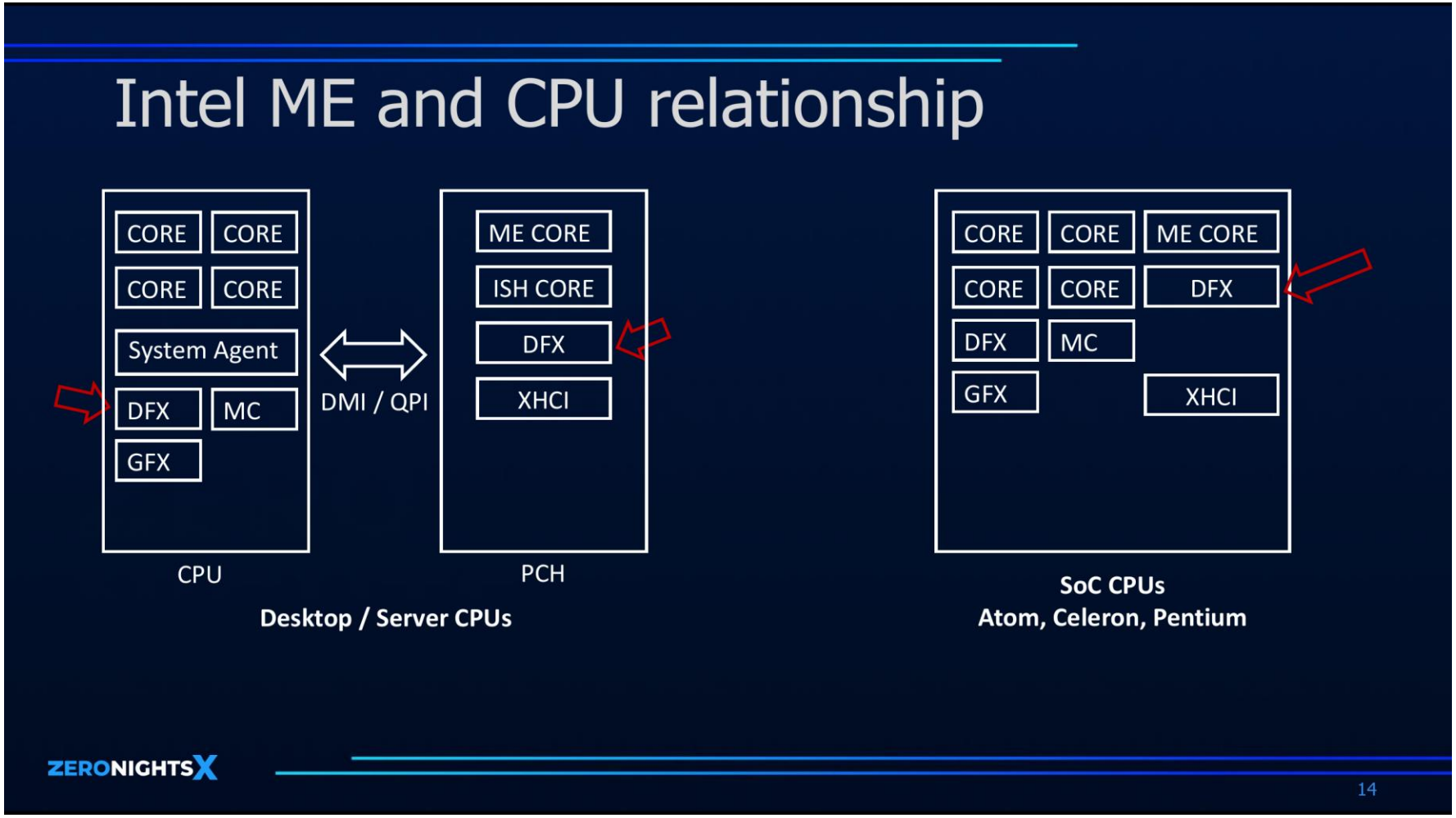
Signature check glitch fail



Signature check glitch fail

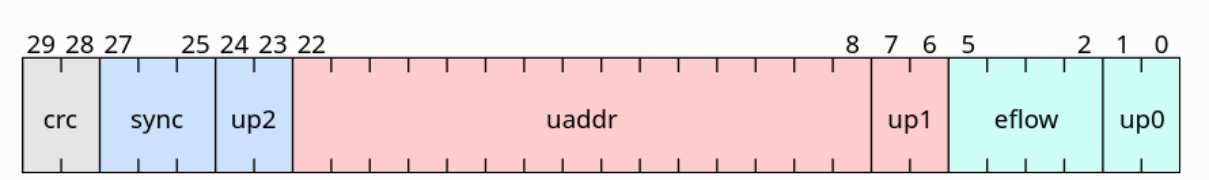


Why Goldmont only



Why microcode update is a lost cause

- We don’t have the **RC4 key**
- RC4 vulnerable to **bit flips**
- But each uinstr/seqword has a **CRC**



From lib-micro docs