Master's Thesis
(Course Code: XM_0123)

MicroSpark: Voltage Glitching on Intel Microcode

by

Federico Cerutti
(Student Number: 2763015)

Submitted in partial fulfillment of the requirements
for the degree of
'Computer Security' Master of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

September 8, 2024

Certified by .....
Cristiano Giuffrida
Associate Professor
First Supervisor

Certified by .....
Alvise de Faveri Tron
PhD Student
Daily Supervisor

Certified by .....
Erik van der Kouwe
Assistant Professor
Second Reader

MicroSpark: Voltage Glitching on Intel Microcode

Federico Cerutti
Vrije Universiteit Amsterdam
Amsterdam, NL
f.cerutti@student.vu.nl

ABSTRACT

Recent developments in security research made possible to Red Unlock Intel Goldmont CPUs, paving the way for in-depth exploration of the instruction set underlying x86: microcode. If an attacker was able to bypass the security systems employed by the microcode update process, they would obtain a powerful and stealthy attack primitive. Previous work has shown the possibility of glitching architectural code on CPUs, but it is unclear if such techniques could be used at the microcode level.

This thesis explores the feasibility of compromising Intel CPU microcode via voltage fault injection, targeting both custom microprograms and the microcode update process. We build a robust glitching setup, and identify novel fault types in architectural and microarchitectural components of the processor. Our research builds on prior work in Intel Red Unlock and x86 CPU voltage glitching, finding new interesting results in this area. With our experiments, we demonstrate how the microcode behavior can be altered with voltage glitching. We provide a foundation for future investigations into the security of microcode updates, and the potential for broader applications across various CPU models.

1 INTRODUCTION

Ever since its introduction with the 8086 microprocessor, part of the x86 instruction set is implemented in microcode [28]. This means that certain instructions, deemed too complex to implement in hardware, are executed as elaborate microprograms in the CPU backend. From the P6 microarchitecture, first appeared in 1995, the microcode of Intel CPUs can also be updated in the field [21], but the mechanism behind such updates is considered a trade secret, and has never been officially documented in details beyond a patent [40]. These updates are primarily utilized to fix CPU bugs and address security vulnerabilities: they can change how an instruction behaves [12] or how a CPU subsystem is configured [11]. Due to their capabilities, microcode updates are potentially a powerful, invisible and persistent attack vector, as they can undermine secure code in a completely invisible way.

In recent years, Ermolov and Goryachy [13] achieved "Red Unlock" on Intel Goldmont SoCs, and unlocked debugging features on final production units. This allowed them to dump the microcode ROM, reverse engineer the unknown architecture, release a disassembler [20], and even identify some undocumented x86 opcodes used to debug microcode [14]. They were finally able to analyze the microcode update process in detail, and understand the encryption and signature schemas [4, 19]. However, it is still impossible to install custom microcode updates due to the secure signature check employed by the update procedure.

Parallel to these advancements, another line of research has explored voltage fault injection attacks on Intel CPUs, leveraging hardware on the motherboard to glitch x86 cores. Researchers demonstrated that they could change the result of some operations (`IMUL`) as well as extract cryptographic keys from Intel Software Guard Extensions (SGX) secure enclave [8, 9, 29, 36, 38].

We aim to investigate whether the findings from these two distinct areas of research could be combined to achieve the final goal of installing unsigned microcode updates. While this is not necessary on Goldmont, due to the aforementioned Red Unlock [13], we choose this platform as it provides us with a well-understood baseline regarding the microcode structure and update mechanisms. Furthermore, the ability to modify x86 instructions behavior with custom microprograms [14] allows us to test individual aspects of the hardware.

We present a characterization of fault types for microcoded instructions on Goldmont CPUs, as well as the first empirical evidence of microcode update glitching. We show how our setup is able to achieve instruction skips on both architectural and microcoded instructions.

Our main contributions through this research are:

(1) A coreboot-based open-source tool[1] to efficiently conduct (microcode) glitching experiments on Goldmont CPUs
(2) Identification of new behaviors in Intel CPUs under fault conditions: Our research identified (micro)instruction skips and faults in basic arithmetic operations, which were previously considered immune to faults [36]
(3) Microcode glitches: We present the first documented instances of microcode-level glitches occurring both in customized microprograms and during the microcode update routine
(4) Characterization of microcode update faults: We identify key moments during the microcode update procedure where injecting faults result in abnormal behavior

These contributions pave the way for further research into microcode manipulation and security, potentially to a broader range of devices.

2 BACKGROUND

2.1 Red Unlock

Since the Pentium Pro era, Intel considers Design For Debug/Validation/Testability (DFX) a crucial technological advantage, essential for product success [23]. As such, all IP blocks now incorporate extensive DFX features. These pervasive debugging features are not only available on preproduction chips, but remain accessible on production devices to analyze faults that might appear on the final user's hardware. Since they enable fine-grained debug of

[1] https://github.com/ceres-c/coreboot/

all the components of a microprocessor, unconditional access to these tools would raise obvious security concerns. Intel then implemented an authentication system, enforced by the DFX Aggregator (henceforth DFX AGG), that supports 3 different DFX unlock levels [26]: (1) Red - Intel internal (2) Orange - BIOS vendors (3) Green - Customers

Green unlock permits only architectural state debugging while, on the other end of the spectrum, Red unlock guarantees complete control of DFX features and access to microarchitectural aspects. Red unlock can be achieved in 4 different modes [14]: (1) Hardware straps (2) Efuses (3) JTAG password (4) Software. To unlock DFX mode via software, another privileged component (e.g. Power Control Unit (PCU), Management Engine (ME)…) of the CPU or Platform Controller Hub (PCH) must write the value corresponding to a specific DFX level to the `PERSONALITY` register of the DFX AGG. Since on desktop and server CPUs there are two different DFX AGGs, setting the personality of either will grant access only to the corresponding realm.

On Intel SoCs, however, there is only one DFX AGG, and setting the personality there guarantees unrestricted access to the whole platform [18]. Through a code execution exploit on Intel ME version 11, Goryachy et al. [18] were able to achieve Red Unlock on Intel Goldmont/Apollo Lake SoCs.

2.2 From assembly to microcode

This section will briefly describe the microcode decoding process within the core pipeline frontend, depicted in Figure 1.

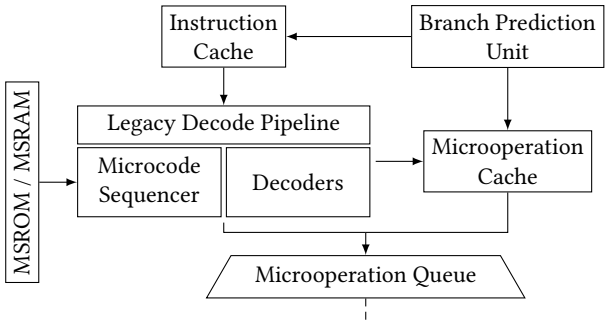

Figure 1: Intel Skylake Pipeline Frontend, simplified [27, Figure 2-4][41]

When an instruction is in the instruction cache, it is decoded by the Legacy Decode Pipeline, which performs the conversion from x86 instructions to microinstructions (henceforth μ-instructions). First, it decodes the length of the instruction, then the bytes are fed into one of the decoders in the pipeline. There are several simple decoders for x86 instructions that map directly to single μ-instructions, at least one decoder that translates to 2-4 μ-instructions, and a complex Microcode Sequencer for long μ-instructions chains [27, §B.5.7.4][4]. The Microcode Sequencer has some features of a CPU, like an instruction pointer, a set of registers and memory [20]. It maps complex x86 instructions like `CPUID` or `RDRAND` to specific microprograms [4, lables.txt (sic)].

The decoded instructions are fed to both the Microoperation Cache and into the Microoperation Queue from where, after various additional optimizations, they will finally be executed. When a loop is detected, the Legacy Decode Pipeline is disabled and the Microoperation Cache becomes the μ-instructions source for the Queue, until a cache miss happens.

2.3 Microcode internals

By gaining complete debug access on Goldmont through Red Unlock [18], researchers were able to read the Microcode Sequencer ROM (MSROM), reverse engineer its structure and functionality, and figure out how microcode can be patched [14]. Further research resulted in an Unified Extensible Firmware Interface (UEFI)-based framework [4] and a Linux library [33] capable of assembling and installing custom microcode patches.

2.3.1 Microcode structure. Within the execution core there are three microcode execution units that operate in parallel, each executing one μ-instruction. The individual results are combined based on the directives in sequence words (seqwords) [18].

The MSROM is composed of two different arrays: one contains quartets of μ-instructions (three are active, the fourth is zero-filled), and the other stores the corresponding seqwords. There is a direct address correspondence between the μ-instruction array and the seqwords array: the μ-instructions triad at address e.g. $0x348$ is controlled by the seqword at $0x348$ in the seqwords array [18].

Microcode patches are stored in the Microcode Sequencer RAM (MSRAM) in two arrays that mirror MSROM structure.

2.3.2 Microinstructions. Goldmont microcode uses 48-bit fixed-length μ-instructions, with 12-bits microopcodes (μ-ops), and a complex source/destination/immediate encoding [20, 33]. Each microinstruction also includes a 2-bit CRC (even/odd bits parity) and, if that is incorrect, the microcode execution unit will hang without raising any architectural exception; exceptions are explicitly raised with the μ-op `SIGEVENT`. When an execution unit encounters an error, the whole CPU is halted in an unrecoverable way.

2.3.3 Sequence words. Seqword are 30-bits fields which regulate the flow of operations, both within one triad and between triads; they also contain a 2-bit parity CRC [20, 33].

2.3.4 Match/Patch. Match registers are used to decide whether the Microcode Sequencer should execute the base version of a microcode program from MSROM or the patched version: when execution hits an address in a match register, it switches to the code in MSRAM [4, 17]. After writing microcode patches in MSRAM, is thus necessary to instruct the execution unit on which MSROM address should be replaced with the new code. On Goldmont CPUs there are 64 such 31-bits registers.

2.3.5 Microcode update. Previous research identified structures in the update file [35] and patterns in update process [22] that hint to well-known cryptographic primitives. Despite this, no further study of the actual content of the updates was possible because the decryption routine and key were stored in the unreadable MSROM, fused in the CPU during production. Once Goldmont CPUs were Red Unlocked, it was possible to reverse engineer the microprogram handling the update process [4, 19], and identify all the cryptographic routines and keys used to decrypt and verify an updated microcode revision.

The microcode update is encrypted with RC4, and signed with RSA. The RC4 key is generated concatenating a value in the update file with a secret stored in MSROM, this approach prevents key reuse attacks. RSA public modulus and exponent are included in the update file, but before being used they are checked against a known hash to prevent an attacker from re-signing the microcode. The decrypted content of the update is parsed linearly, beginning at byte zero: based on this initial byte, the update routine determines which function to call. Each function consumes a certain amount of bytes and advances the index; some functions write data to peripherals, others execute code provided in the update file. This continues in a loop until a termination marker is found. The routine that applies a microcode patch parses the special encoding used in microcode updates for μ-instructions triads and seqwords, writes them to MSRAM [4], and then it writes a match register with the RAM address.

2.4 Voltage fault injection

The implementation of any algorithm, when executed on hardware as opposed to its theoretical version, is vulnerable to what is broadly known as *hardware attacks*. Such attacks aim at breaking either the confidentiality or the integrity of the executed program by exploiting the physical properties of the computing device. In particular, these attacks are generally divided into two main families:

- *Side channels*: The attacker is able to infer the internal state of the target observing information leaked over different channels (e.g. timing [31], power consumption [7, 32], electromagnetic emission [15]).
- *Fault injection*: The attacker actively interacts with the target, modifying its behavior to insert a fault (an unexpected state). Multiple vectors can be used to inject the fault (e.g. supply voltage, electromagnetic fields [30], light [39]) with the final goal of modifying the internal state in some way favorable to the attacker. This could mean skipping instructions that perform checks, inserting faults in a cryptographic algorithm to recover the key through Differential Fault Analysis (DFA) [2, 3], etc.

Voltage-based fault injection generally involves modifying the power rail of the Device Under Test (DUT) to inject glitches directly via the power supply or through crowbar glitching [37]. Both, however, can not be trivially applied to modern x86 CPUs due to the high current consumption. The custom power supply circuit should also support the complex bring up sequence of these devices.

2.4.1 Glitching through voltage scaling. Recently, a new voltage fault injection vector emerged: generating glitches through the Power Management Integrated Chip (PMIC), the power supply used for CPU frequency/voltage scaling. There are two main families of such attacks, based on whether they require physical access to the target system:

- Remote access: CPU voltage is controlled indirectly through the software interface exposed by the CPU. This was exploited in Plundervolt [36] and V0LTpwn [29], and Intel quickly mitigated these attacks disabling the voltage control interface via a microcode update.
- Physical access: VoltPillager [9], PMFault [8] and Buhren et al. [6] attacks require modifications the DUT, gaining access to the data lines that connect the CPU to the PMIC: the attacker adds an external hardware device that instructs the PMIC to change the supplied voltage. This technique grants higher reliability and finer control on the glitch shape.

Plundervolt [36] authors found that the `IMUL` multiplication was susceptible to bit flips when using specific operands in a certain order, whereas other arithmetic operations such as `OR`, `XOR` and `AND` were never affected. Furthermore, by injecting faults during cryptographic operations, they successfully recovered AES and RSA keys from SGX through DFA.

2.4.2 Intel voltage scaling. Intel CPUs have multiple power management systems at different granularity levels [10]:

- Power Management Controller (PMC): A Synopsys ARC coprocessor in the PCH that manages power at the package level and handles power state transitions [5]
- Power Control Unit (PCU): A coprocessor in the CPU that runs custom PCODE and controls the CPU power state
- Power Management Unit (PMU): Hardwired logic to handle power management in the PCH
- Power Management Engine (PME): Hardwired logic to handle power gating (component de/activation) in each IP core

The PMC communicates with the external PMIC, which in turn drives the Voltage Regulator Modules (VRMs) to obtain the required voltage. Depending on the aggregate power consumption of the package, the PMC will ask the PMIC to adjust the voltage of the appropriate power rail. The PMC and PMIC can communicate with two protocols

- SVID: An SPI-like Intel proprietary communication protocol, used in the VoltPillager attack [9] to inject faults in the CPU, and documented in Intel CPU datasheets [25, Table 34-4].
- PMBus: An open standard, a variant of SMBus which is itself based on the I2C protocol. It was used for the PMFault attack [8].

SVID is faster, reaching speeds up to 25 MHz, while PMBus is limited to 5 MHz, the maximum speed of I2C. In reality, most PMBus PMICs support only up to 1 MHz. For voltage glitching, a higher communication speed is advantageous, as it reduces the communication overhead when generating a glitch.

3 THREAT MODEL

In this work, we consider an attacker with unrestricted access who wants to invisibly modify the behavior of *any* software executed on the target. This includes secure enclaves or System Management Mode (SMM) code, which operate in isolation and are typically beyond reach of a privileged attacker. Furthermore, the authors of PMFault [8] were able to use the Baseboard Management Controller (BMC) to inject faults without physical access, so the physical access constraint can be lifted when targeting specific devices.

While we concentrate on Goldmont CPUs since they provide an experimental ground truth for our evaluation, it should be possible to generalize our results to other CPU families. However, the hardware differences between Goldmont and other microarchitectures

might drastically change the likeliness of success. We are planning to test whether the applicable attacks are feasible on different hardware.

4 OVERVIEW

To achieve our final goal of installing unsigned microcode updates, we need to assess if voltage glitching is a viable attack vector against microcode. This requires combining previous research on Red Unlock with voltage-based fault injection attacks.

Our first objective is to replicate with Red Unlockable hardware the results discussed in 2.4 on architectural operations. This might prove challenging, as previous research on fault injection focused on Core-series CPUs, while we are working on Goldmont SoCs. Hardware differences in the CPU or different PMIC specifications might prevent the attack from working.

Once we confirm voltage fault injection works on the DUT, we leverage the microcode patching capabilities on Red Unlocked hardware to replace the implementation of x86 instructions with minimal microprograms, and attempt to glitch them. This provides insight on the effects of our glitch attempts, as we are aware of the μ-instructions being executed in the CPU.

Finally, we attack the update procedure treating it as a gray box: with knowledge from previous research [4, 19] we observe its measurable behavior and how that changes with glitching.

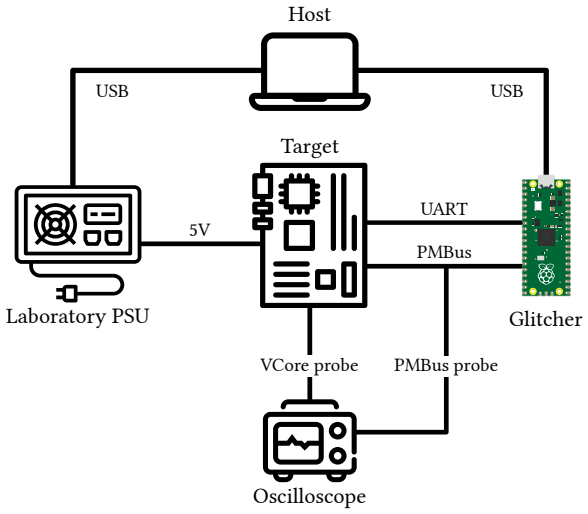

Figure 2: Experimental setup overview

Figure 2 shows all the components in our setup and their interconnections. The host computer controls the campaign, configures the glitcher and decides when to reset the DUT based on the results it gets from the glitcher. The glitcher injects PMBus control packets for the PMIC, while the target is operating independently of the rest of the system.

5 EXPERIMENTAL SETUP

This section will analyze in depth each component of our setup.

5.1 Target

We experiment with two different Single Board Computer (SBC)s known to be vulnerable to the exploit by Ermolov and Goryachy [13]:

(1) *Gigabyte BRIX GB-BPCE-3350* (Celeron N3350): the board where the attack was first released [16]
(2) *UP Squared* (Pentium N4200): a device where the exploit has been recently ported [33]

Since neither of these CPUs allow for software-based voltage control, we construct a setup similar to VoltPillager [9]. However, because the onboard PMIC does not support SVID, we use PMBus instead. We finally choose the *Squared* board for reasons that will be explained in 5.1.2.

5.1.1 Hardware. Both of the SBCs in question use the Texas Instruments TPS65094: a PMIC specifically tailored for Apollo Lake CPUs which automatically manages voltage sequencing and chip bring up. As a result, there is no basis for favoring one over the other in this regard.

We modify the target *Squared* board to access

- VCore: The tension supplied to core circuitry (execution units) [25, Table 41-5]
- PMBus signals (SDA, SCL): to send the PMIC our control packets

The PMBus traces can be identified on the circuit board using as a guide the reference implementation in the TPS65094 datasheet [24].

To communicate with the glitcher we use UART, available on connector CN16 [1]. While it would be preferable to use UART flow control pins like RTS or CTS as a trigger, we are unable to locate them on the board and conclude that they are not exposed.

5.1.2 Software. Neither of the two SBCs initially appears suitable for this project because booting a BIOS with the Red Unlock exploit [16] takes multiple minutes. This would have severely limited the feasibility of the attack, as we anticipate a high number of resets, especially in the explorative phase with broad settings ranges. The *Squared*, however, is particularly interesting because Krog and Skovsende [33] had ported the Red Unlock exploit to this board on a coreboot[2] BIOS image. This gives us clear understanding of the BIOS code's operations, and the flexibility to modify the firmware source code. Furthermore, since coreboot is a single-threaded application, there is no scheduling or resource contention, which implies less noise on the system and higher experiment reliability.

Booting coreboot with debug output, we are able to determine the boot is being delayed when Intel Firmware Support Package (FSP) is notified of the imminent transfer of control from coreboot to its payload (POST code $0x88$). Normally, this operation would be instantaneous, but when the Red Unlock exploit [16] is included in the ME image, this step takes minutes to execute, before eventually booting successfully. We speculate this delay is due to ROP chain used in the exploit, which introduces an infinite loop in Intel ME code after unlocking the CPU. When the FSP is notified of the imminent execution of the OS, it might need to communicate with ME, which is not able to respond. Eventually, the FSP will time

[2] https://coreboot.org/

out, and return control to coreboot, which proceeds to launch its payload.

We decide to integrate our code into coreboot to ensure it executes as quickly as possible after power-up. Since we also want to leverage the possibility of installing custom microcode in the CPU, we modify lib-micro[3] [33] to work in 32-bit protected execution mode, which is the standard for coreboot. This is possible thanks to the detailed report on the undocumented x86 opcodes [14, §2], which hints at support for 32-bit mode (combination of `rbx:rdx`).

After adapting lib-micro for 32-bit mode, we discover that the CPU hangs if custom microcode is loaded too early in the boot process. For the microcode patch to run successfully, package power limits need to be set; currently, we have no explanation for this behavior.

Finally, we obtain a setup that executes our code within 700 ms of power-up, and can successfully load custom microcode patches. The target runs a very simple loop (cf. Listing 1): it sends a synchronization character (`R`) to the glitcher via UART, executes relevant opcodes for the current experiment, notifies completion (`D`), and sends the result to the glitcher. The communication is unidirectional and the target runs independently of any other component of the setup. This system also enables detection of whether the target resets due to the glitch (`D` not received by the glitcher) or if it is transmitting garbled data because it ended in some unexpected state.

```
uint32_t output;
while (true) {
  uart_tx('R');
  // < Unrolled assembly loop modifying 'output' >
  uart_tx('D');
  uart_tx(output);
}
```

Listing 1: Pseudocode running on the target

We minimize the target code to reduce the risk of inadvertently glitching CPU components that we are not interested in. A `for` loop with a conditional jump would activate the branch predictor, and potentially load different code in the instructions cache. Furthermore, the conditional jump itself could be source of variability, as the Arithmetic Logic Unit (ALU) would be doing comparisons with the counter variable and write a relative offset to the Program Counter (PC) instead of consistently incrementing it.

5.2 Glitcher

5.2.1 Hardware. We build our glitcher around the inexpensive Raspberry Pi Pico board with the RP2040 microcontroller. On the DUT the PMBus bus voltage is 1.8 V, while the Pico GPIOs are normally operating at 3.3 V. We then modified the Pico board to power the RP2040 with an external 1.8 V power supply, as stated in the datasheet [34, §2.9.7.3]. However, since the target UART operates at 3.3 V, we added a Texas Instruments TXS0102 Bidirectional Voltage-Level Translator to step the voltage down to a safe level for the Pico.

To avoid modifications to the Pico board, it might be possible to connect the glitcher to PMBus through a TXS0102, and directly wire the UART to the glitcher. This method should work equally well.

5.2.2 Software. The glitcher acts as a middleman between target and host, injecting a glitch with the configuration provided by the host, and notifying it if the target becomes unreachable.

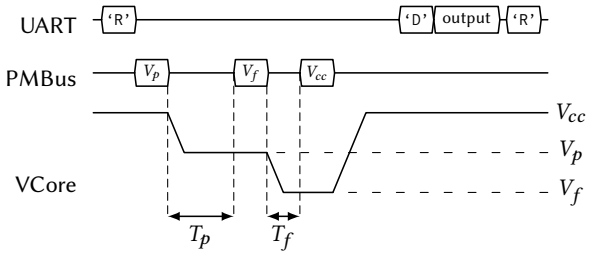

Figure 3: Glitch signal sequence

As shown in Figure 3, once the glitcher is armed, it will wait for the next synchronization character `R` from the target, and inject the glitch via PMBus. The `output` of the current execution of Listing 1 on the target is then sent to the host.

In designing the shape of our glitch, we must accommodate the limitations of the PMIC when it is misused to inject glitches, specifically its low ~3 mV/µs slew rate [24]. For comparison, the PMIC used in VoltPillager [9] had a 20 mV/µs slew rate, and specialized tools like the ChipWhisperer can achieve rates over 1500 mV/µs [37], delivering rapid and precise voltage drops. The slew rate is a significant aspect of a glitch: if the voltage drops too slowly the glitch might have no effect on the target device, it might affect the behavior of many components and become uncontrollable, or the target could detect a power loss and reset autonomously.

Due to the low slew rate, we use a two-staged glitch (cfr Figure 3) as Chen et al. [9] did. This allows us to more quickly reach the critical voltage V_f and achieve a narrower glitch. Our glitch is characterized by the following parameters

- V_p: Preparation voltage, the lower bound of the CPU stability voltage range
- V_f: Fault voltage, the target voltage during the glitch
- T_p: Preparation time, the external offset of the glitch, or for how long V_p must be held (µs)
- T_f: Fault time, the width of the glitch, or for how long V_f must be held (µs)

Timings are also influenced by the transmission time of each PMBus command sent by the glitcher to the PMIC.

5.3 Power Supply

As a power supply for the DUT we utilize a KORAD KA3305P, known for its clean signal and absence of over/undershoot relative to the set point voltage when enabling the output. This model can be controlled with Virtual Instrument Software Architecture (VISA) commands over USB, simplifying the process of rebooting the DUT.

[3] https://github.com/ceres-c/lib-micro

The *Squared* is powered at 5.3 V, as measured from the original power supply, and consumes a maximum of 1.1 A in coreboot.

5.4 Host

The host orchestrates the entire setup, communicating with both the glitcher and the power supply via USB. If the glitcher indicates that the target is unresponsive, the host will reset it with a power flush using the laboratory power supply. We have developed an extensible Python library that provides convenient APIs for interacting with the glitcher. The result of the current operation on the target is analyzed on the host to distinguish between executions where the glitch had no effect and those where the output deviates from the expected result.

We collect data points in a SQLite database and visualize them with Matplotlib in a Jupyter notebook.

6 EVALUATION

We design experiments to characterize specific aspects of the DUT. The unrolled loop mentioned in Listing 1 consists of a small x86 assembly snippet, repeated up to 270,000 times to achieve an interval between two consecutive UART bytes of at least 400 µs, enough to let VCore drop to the required V_f and subsequently return to V_{cc}. In all the scatter plots in the current section and in section 9:

- Voltages are expressed as PMIC Voltage IDs (VIDs) as per [24, Table 6-3]
- Times are in µs
- Green dots represent normal executions
- Yellow dots represent unexpected states (hangs, garbled data, no response…)
- Red dots represent successful glitches

Assembly code is in AT&T syntax, as used in coreboot, and x86 microcode uses lib-micro [33] C macros syntax.

6.1 Identifying glitch parameters

Each experiment requires an explorative phase to characterize the behavior of the DUT with the specific workload. Since each component of the CPU has a different instability voltage, we initially have to evaluate every target with broad settings ranges. Furthermore, due to the aforementioned limitations of the PMIC (cf. 5.2.2), the glitch we introduce has four configuration parameters, expanding the search space exponentially. We can however optimize this stage with two considerations:

- We aim for V_p to be close to the lowest possible voltage at which the CPU can execute our specific payload without faults. The lower V_p is, the faster we can drop to V_f (cf. Figure 3).
  Setting V_f to a safe voltage like V_{cc}, we can establish an appropriate value for V_p independently of other settings, essentially reverting to a one-stage glitch. To ensure this V_p value is not posing issues to the target, we maintain VCore at V_p for a long period T_p, and verify the DUT is not influenced by this voltage drop.
  As an example, in Figure 4 data points for $V_p > 35$ are mostly green (i.e. normal execution): an interval with 35 as a lower bound is a good initial search range.
- T_p does not need to be precise when running characterization experiments. All iterations of the (unrolled) loop execute the same code, which means that VCore can be dropped at any point in time to V_f, and the results should be the same. We are not aiming at a specific fragment of a long procedure, but rather evaluating if the current workload is affected by voltage drops. In this scenario T_p needs only to be enough for VCore to stabilize at V_p. T_p will become critical when attacking the microcode update, as we will be trying to glitch a specific operation in the update procedure.

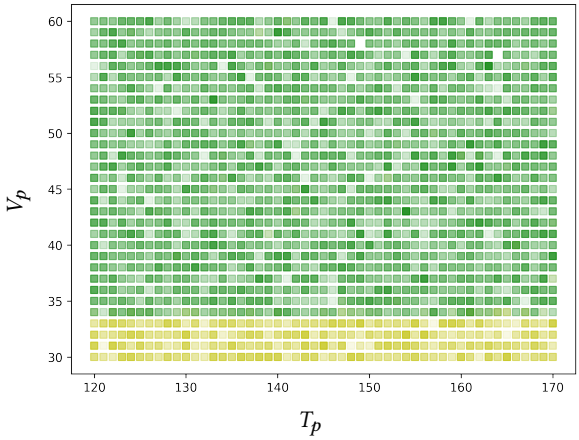

Figure 4: Estimating V_p for `IMUL` glitching

6.2 x86 assembly instructions

6.2.1 `IMUL`. We first replicate the attack against `IMUL` presented in Plundervolt: Murdock et al. [36] observed that the result of the multiplication had flipped bits when certain operands were arranged in a specific order.

The target executes code in Listing 2, which is comparing the results of two multiplications and adding one to `ecx` if the results are different.

```
movl $0x80000, %eax;      # operand1
movl $0x4, %ebx;          # operand2
movl %eax, %edx;          # loop body start
imull %ebx, %edx;
movl %eax, %edi;
imull %ebx, %edi;
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;            # loop body end
```

Listing 2: Target code for `IMUL`

We confirm the DUT is vulnerable to fault injection and, after optimizing the glitch parameters, achieve a success ratio of ~4% of

attempts (database[4] table _02a46ea_mul_2). Throughout the glitching campaign, the target code executed at 2.4 Hz due to the eventual resets, yielding a glitch ratio of 0.016 glitch/s (1 min/glitch).

Furthermore, we found notable differences with Plundervolt results:

(1) We can corrupt the result of the multiplication regardless of the order of the operands `0x80000` and `0x4` (database tables _02a46ea_mul_2 and _02a46ea_mul_swap).
(2) While generally the target only reports a handful of glitches, occasionally the glitch count becomes `0x200000`, which corresponds to the product of the two operands (database table _02a46ea_mul_2). Given the target code in Listing 2, we can see that the register `edx` is used both as a destination for the multiplication result, and as storage for `SETNE`. The lowest byte of `edx` is then added to the fault counter `ecx` with `ADDB`. Due to the glitch, the full 32-bit register is being added to `ecx` instead of the bottom 8-bits.

These two facts suggest that the peripheral affected by the voltage drop might not be the ALU multiplier, but rather the register file.

6.2.2 `CMP`. Since `IMUL` does not seem to be affected by order of operands, we speculate the results we saw might be generated by the `CMP` operation rather than the multiplication. To investigate this behavior, we further reduce the code in Listing 2.

In the new target code, `CMP` is used to compare two fixed operands, and increment a counter value whenever they are found to be different (cf. Listing 3). It's worth noting that this code also relies on the ALU, because `CMP` on x86 is implemented as a `SUB` the result of which is then discarded.

```
movl $0xAAAAAAAA, %eax;    # 0b10101010...
movl $0xAAAAAAAA, %ebx;
cmp %eax, %ebx;            # loop body start
setne %dl;
addb %dl, %cl;             # loop body end
```

Listing 3: Target code for `CMP`

In this test we find a significant increase in glitches, with approximately 74% of executions (cf. Figure 5) reporting the two values to be non-equal at least once (database table _5e872de_cmp_2). We reach a 72 glitch/s rate, as the optimal glitch parameters (V_p and V_f) were such that the DUT was not resetting. This higher success rate can be attributed to the reduced number of instructions in this target; without the two `IMUL` operations, `CMP` is executed more frequently per unit of time, increasing the probability of it being faulted.

6.2.3 Register file. To test the hypothesis that the glitch may occur in the register file, we employ the code shown in Listing 4. This code truncates a fixed value (`0x0101` to `0x01`) and accumulates result of truncation in the `ecx` register. To achieve the standard ~400 µs runtime, the (unrolled) loop is executed 271,000 times, with the expected value in `ecx` also being 271,000.

[4] https://github.com/ceres-c/VU-Thesis/blob/master/notebooks/glitch2.db

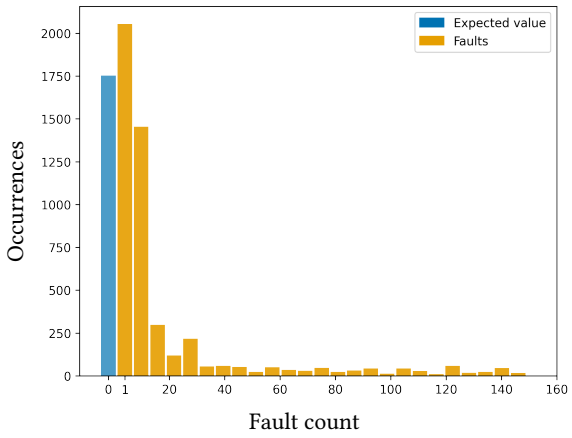

Figure 5: Distribution of faults in `CMP` test (all results)

```
mov $0x0101, %eax;
movb %al, %bl;             # loop body start
add %ebx, %ecx;            # loop body end
```

Listing 4: Target code for register file

If our hypothesis holds true, we expect to see values greater than 271,000: the full value `0x0101` will be added to the counter instead of the byte `0x01`, hence the final result will be larger. However, as shown in Figure 6, we observe values that are either slightly smaller than the expected 271,000, or normally distributed around 262,500 (database table _03168d2_reg), hinting at instructions skip. Some results are actually greater than the expected value, but by orders of magnitude, and amount to merely the 6% of successes, so we classify them as false positives.

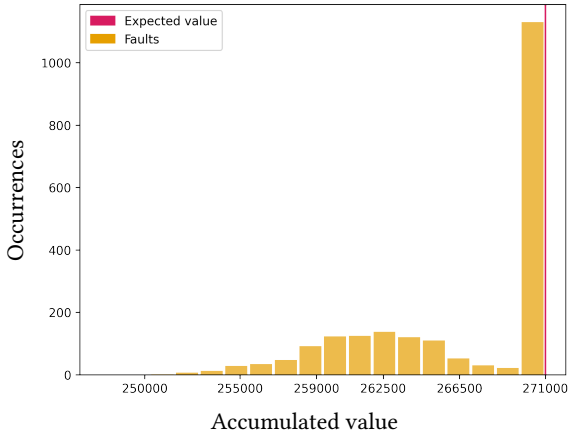

Figure 6: Distribution of accumulated values in register file test (successful glitches only)

This test had a 19% success ratio, with 0.375 glitch/s.

6.3 µ-instructions

Given that the target device has been confirmed to be vulnerable to glitching attacks at the architectural level, we now shift our focus on injecting glitches in microarchitectural components.

On Red Unlocked hardware, the implementation of microcoded x86 instructions can be replaced with custom microprograms. We apply to µ-instructions the same approach we used before: create minimal microcode snippets that target specific aspects of the DUT. Using our modified version of lib-micro[5] [33], we move our test code "inside" the RDRAND operation. Thus, the body of the assembly loop from Listing 1 now consists of call to RDRAND only.

Notably, some microcode patches built with lib-micro work only on a fully booted device in Linux; when the same microcode is executed in coreboot instead, the target hangs. We observed this affects specifically conditional jumps, and that the patches cannot be longer than 4 µ-instructions triads. While the limited patch length can be explained by our code overwriting some other microcode, we don't have a justification for the issue with conditional jumps. Our only hypothesis is that this is related to the CPU operating in protected mode rather than long mode. Various microcode details are still obscure, and this can lead to unexpected behaviors: some µ-instructions like MOVE_DSZ32_DR don't work in coreboot altogether, while other like ADD_DSZ64_DRR that operate on 64-bit registers, work correctly despite 64-bit registers not being available architecturally.

6.3.1 CMP. We patch RDRAND to implement Listing 3 in a single x86 opcode: compare the values in two registers, and add 1 to the accumulator if they are different. In Listing 5, SUB_DSZ32_DRR will set TMP0's per-register flags, that are then used by SETCC_CONDNZ_DR to set the temporary register TMP1 to 1 when the subtraction result is non-zero. Finally, the conditional value of TMP1 is added to the architectural register rcx, accessible as ecx in protected mode.

```
{
    SUB_DSZ32_DRR(TMP0, RAX, RBX),
    SETCC_CONDNZ_DR(TMP1, TMP0),
    ADD_DSZ32_DRR(RCX, TMP1, RCX),
    END_SEQWORD
}
```

Listing 5: CMP target implemented in µ-instructions.

Despite using wide settings ranges and testing for up to 36 h, we were not able to inject any meaningful fault in this target. There are less than 10 tests reported as successes in the database (table _5e872de_rdrand_cmp_ne_3), but the number of faulted instructions are not coherent with our expected values, and we deem them to be false positives with garbled data from the target interpreted as a successful glitch. This result suggests that the x86 instructions in Listing 3, which we successfully glitched, have a more complex implementation than the basic microcode we are now testing.

6.3.2 µ-instruction skip. To investigate the instruction skip highlighted by the experiment in 6.2.3, we create a target to identify both architectural and microarchitectural faults of this kind. The code in Listing 7 performs 10 separate additions to the value in the accumulator register ecx, adding 1 each time.

RDRAND is invoked 80,000 times in this target, thus the expected final value of the accumulator is 800,000. If an instruction is skipped, then the final value is <800,000, and it can either be a multiple of 10 or not. In the first case, a call to RDRAND was skipped, thus missing 10 ADDs. Conversely, if the value is not a multiple of 10, then a µ-instruction was skipped.

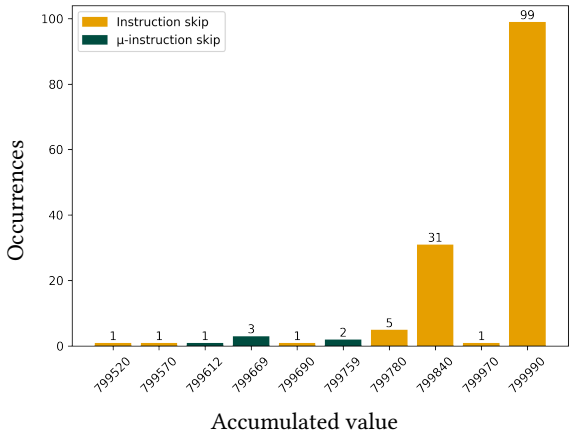

Figure 7: Distribution of accumulated values in µ-instruction skip test (successful glitches only)

After tuning our glitch settings, we found 160 faulty executions after 106,000 attempts, a 0.1% success ratio (database table _4a2b3cf_customucode_add10). Of these:

- 6 are not multiples of 10
- 139 are multiples of 10
- 15 are spurious results orders of magnitude away from the expected value

We conclude architectural instruction skip is more common than microarchitectural skip with these settings (cf. Figure 7). The high prevalence of 799,990 also suggests that in most experiments only a single x86 instruction is skipped.

6.4 Microcode updates

Although we have not been able to inject numerous faults in microarchitectural components so far, we still proceed to evaluate the glitch resistance of the microcode update procedure. In microcode experiments up to this point, we replaced the implementation of the architectural instruction RDRAND with our target code, and proceeded to attack that. To trigger our microcode, we thus invoked RDRAND, and execution went through the complete pipeline frontend shown in Figure 1. The microcode update process differs significantly, as it is a long routine completely implemented in microcode, taking up to 5.3 ms from start to finish. Once an update is initiated with WRMSR, it is executed by the Microcode Sequencer only, and the architectural components of the CPU are not involved.

[5] https://github.com/ceres-c/lib-micro

To attack the microcode update procedure, we first analyze the logic of the update process to identify interesting targets. The update procedure is treated a gray box, leveraging previous work by Borrello et al. [4][6] and focusing on two main attack vectors:

- Glitching the RSA public modulus check: the update file contains the RSA public modulus which is used to verify the integrity of the file itself. To ensure the modulus remains unaltered, its hash is checked against a value hardcoded in MSROM. If the attacker were to glitch this comparison, they could sign the update with any private key; however, the attacker would still need to know the RC4 key to decrypt the update and sign the decrypted content.
- Glitching the signature check: The hash of the decrypted update content is checked against the signature in the file. If an attacker were to skip the signature check through fault injection, they could load a modified microcode. Since RC4 is subject to bit flipping attacks, it might even be possible to flip bits in the ciphertext without knowledge of the key, and obtain valid, different, microcode plaintext after decryption.

We change the target code in Listing 1 to perform a microcode update and a `NOP` sled, necessary for VCore to return to V_{cc} before UART communication begins (cf. Listing 8). Once the update is done, the target transmits both the installed microcode version after the update, and the execution time of the microcode update (in `RDTSC` cycles). This gives us two different ways to evaluate the success of a glitch:

- Compare the final installed microcode version number with the version automatically loaded at boot. Since we modify the update file, the update process should always abort before completion and the installed version must never change if not glitched.
- Compare the duration of the update T_u to the median duration Med (T_u) observed when the same update file is used without injecting faults. Since the microcode update process is not constant-time [22], this metric tells us if some check has been skipped or a loop counter has been corrupted. It is worth noting that the first update after a power flush will take more time because the CPU is slower right after boot, while code is being cached. This is not an issue with our setup because, after a reset, the execution speed has already stabilized when the target is considered ready for another experiment.
  With this basic measurement, we can notice potentially interesting changes in the behavior of the update routine, even if the update is not applied and the installed microcode version remains unchanged.

A successful update to a newer microcode revision takes $T_u \approx$ 6,740,000 cycles (~6.2 ms), but once the update has been applied, each subsequent execution of the update procedure will be slightly shorter Med (T_u) = 6,160,509 cycles (~5.6 ms), as the CPU will not apply the same version twice. This is not an issue for our tests, as none of the updates we are trying to install will ever complete successfully without a glitch.

[6] Additional details at https://github.com/pietroborrello/CustomProcessingUnit/blob/master/Notes.md#ucode-update

When glitching the microcode update process, T_p becomes crucial. Previously, while attacking our target loops, T_p was only required to let VCore drop to V_p. However, given the extended duration and complexity of the microcode update process, we now have the resolution to introduce a glitch within a single update, instead of across multiple iterations of a high-speed target. Consequently, identifying optimal glitch parameters requires exploring wider T_p ranges, since a complete microcode update averages at 5.3 ms.

6.4.1 Low voltage tolerance. While estimating V_p as explained in 6.1, we notice an unexpected behavior of the DUT: VCore can be set to any non-zero value with no effect on the stability of the target, as long as VCore is restored to a safe voltage before the microcode update is done. For example, V_p can be dropped from $V_{cc} = 1.24$ V to VID 1, equivalent to 0.5 V [24, Table 6-3], and the CPU would not reset. This means that we can expand our search to voltages much lower than those we tested before, and potentially inject more faults.

This pattern can be seen in Figure 8: the target does not reset, regardless of the voltage V_p and the width of the drop T_p; we see a cluster of resets only at the extreme right of the graph, for $T_p >$ 5000 µs. These can be explained considering the low slew rate of the PMIC: when VCore is extremely low, it can not be raised to a safe value before the CPU starts executing x86 assembly, therefore it hangs.

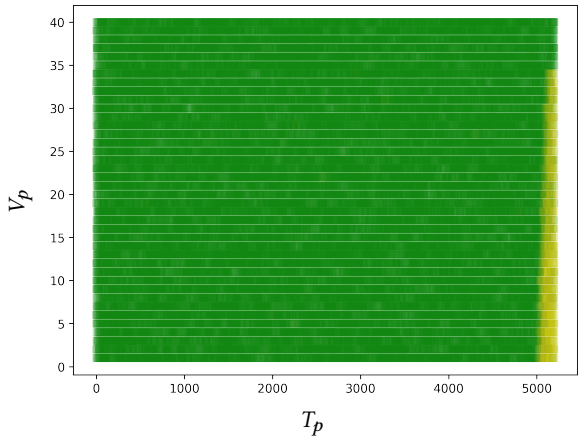

Figure 8: Estimating V_p for microcode update glitching. Notice the x-axis extends to 5.2 ms

6.4.2 RSA public modulus. To attack the RSA modulus check, we modify a microcode update file[7], changing the modulus and the signature accordingly. The signature is calculated on the decrypted content of the update, so this attack is only feasible on Goldmont devices, where the per-family secret RC4 key has been leaked from MSROM [4, 19].

To identify a sensible range for T_p, we compare the duration of an update with a valid signature Med (T_u) = 6,160,509 cycles (~5.6 ms) against that of an update with a modified modulus Med (T_u) =

[7] https://github.com/ceres-c/VU-Thesis/blob/master/ucode_update_mod/06-5c-0a_signed

4,375,591 cycles (~4 ms). Knowing the modulus is checked early in the update process, but there is a cleanup phase at the end of the procedure, and that the PMIC requires some time to bring VCore back to V_{cc}, we start the glitch campaign with $T_p \in [3; 3.8]$ ms.

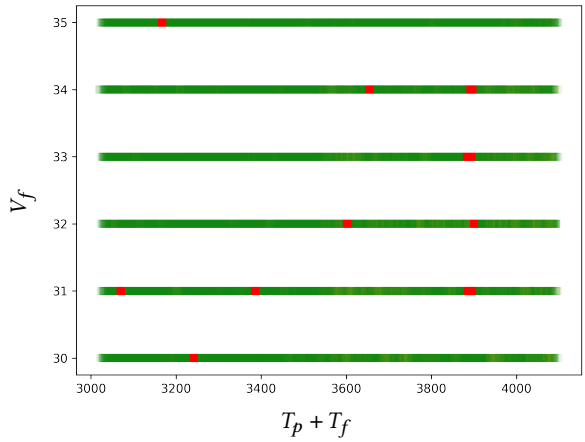

Figure 9: Faults in RSA modulus glitch campaign (success: $T_u > 5{,}000{,}000$)

In Figure 9 (database table _90ec1b5_ucode_update_rsamod_3) we plot the x-axis as $T_p + T_f$, because both these settings contribute to identifying the right instant of the update for a glitch. With this reference system, we can clearly see faults clustered around $T_p + T_f = 3900$.

Tuning the glitch parameters, we reliably inject faults in the microcode update procedure, bringing T_u within the expected range for successful updates, as can be seen in Figure 10 (database table _99074c0_ucode_update_signed). However, despite T_u being in the appropriate range, the microcode update process does not seem to complete successfully: the installed version after the update remains the original $0x20$ instead of $0x28$ contained in the update file. We speculate there might be a secondary check in the update procedure that is unaffected by the glitch. The success rate in this test is 0, 17%, with 0.015 glitch/s (68 s/glitch).

6.4.3 Signature check. Knowing the RC4 key for Goldmont CPUs, we are able to modify the μ-instructions in the update and re-encrypt it, obtaining a new valid update, albeit not signed. The modified update file will be rejected by the CPU after Med (T_u) = 6,050,103 cycles (~5.55 ms) because the post-decryption signature check stage of the update will fail. Following the approach used in 6.4.2, we start the campaign with $T_p \in [3.6; 5.2]$ ms. T_u values for successes is this test are not as neatly distributed as they were in the RSA modulus test (shown in Figure 11), and again the DUT never accepts the modified update.

The update file we use for this test is semantically valid: we change a `NOP` to `ADD_DSZ32(tmp0, 0x00000000)`, which essentially still is a no-op. We can modify precisely a single μ-instruction, and replace it with another valid operation because we know the RC4 encryption key [4, 19]. However, if this attack had an adequate success ratio, this approach would potentially be feasible without knowing the RC4 key.

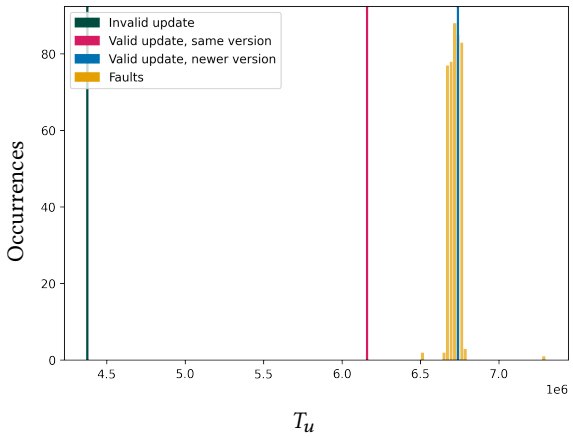

Figure 10: Distribution of successes in RSA modulus test $T_p \in [3575 : 3580]$, $T_f \in [150 : 250]$

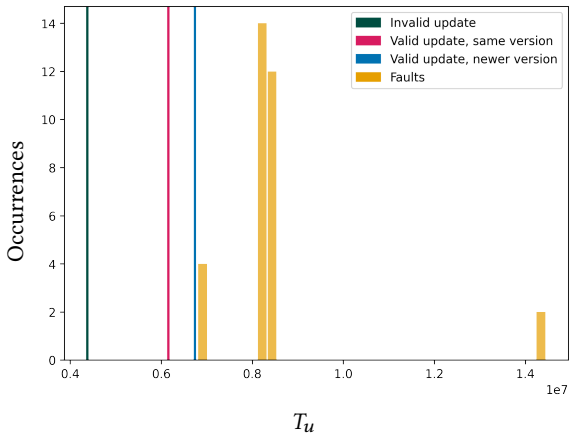

Figure 11: Distribution of successes in signature check test $T_p \in [3600 : 5200]$, $T_f \in [20 : 250]$

Since RC4 is a keystream, it is vulnerable to bit-flipping attacks: changing a bit in the ciphertext will also flip that bit in the plaintext after decryption, without corrupting the rest of the data. A microcode update contains different types of data (cf. 2.3.5), generally including raw data that is written to memory areas, and μ-instructions that the update will execute. With a bit-flipping attack, we would be able to change both which data is written where, and which μ-instructions are executed.

It is important to highlight that μ-instructions and seqwords include a CRC (cf. 2.3.2), which is a parity check on even/odd bits: if a bit is flipped in the instruction, the corresponding CRC bit must also be flipped to preserve integrity. Thus, the only data integrity structure in a decrypted microcode update can be circumvented by knowing (or brute forcing) the distance of a bit to the CRC, obtaining another valid μ-instruction after a bit-flip.

6.5 Results evaluation

We confirm that Intel Goldmont SoCs, similarly to Core CPUs, are vulnerable to voltage fault injection attacks targeting x86 architectural operations. Furthermore, we observe different types of faults, and a glitch ratio significantly higher than that of previous research [8, 9, 29, 36], detecting instability in the register file and instructions skip.

The Microcode Sequencer proves to be less susceptible to faults, likely due to a greater voltage tolerance compared to other components of the CPU, but we have evidence of μ-instruction skip. When μ-instructions are executed together with other architectural components of the CPU, it is not possible to reach a voltage low enough for faults to appear which also maintains the CPU stable. However, we define a time-based metric to evaluate the success of a glitch in the microcode update procedure, which shows clear patterns when attacked with specific parameters.

7 DISCUSSION

Ultimately, we were unable to install unsigned microcode updates, but we successfully managed to glitch x86 instructions and, to a lesser extent, microcode. Specifically, we demonstrated the susceptibility of microcode to voltage fault injection attacks, despite the suboptimal specifications of the PMIC on the DUT. Moreover, we exposed new categories of faults in Intel x86 CPUs that had not been identified in previous research [8, 9, 29, 36].

It is clear from our experiment that it is possible to change the behavior of small microcode targets (6.3.2), as well as complex microprograms such as the microcode update (6.4.2). We believe there is opportunity for further research in this field, starting from an in-depth analysis of the microcode update process on Goldmont CPUs. This study could explain why our current glitch attempts failed, and shall be conducted with a specific focus on glitching in order to identify potential targets and exclude hardened code.

Moving forward, we want to verify if conditional jumps (`JUMPCC`) are equally vulnerable as conditional set (`SETCC`) (6.2.2), as this could prove useful when attacking architectural code. The shape of the glitch we inject could also be modified, with multiple consecutive $V_p \rightarrow V_f$ voltage drops, which may lead to more pronounced effects on (micro)code. Finally, we aim to determine why long microprograms crash the DUT (6.3). After clarifying that, we will evaluate whether the resistance to low voltage observed during the microcode update (6.4.1) applies to our custom microprograms as well. This test will clarify whether the device survived low voltages during the update due to power gating in preparation for the update, or because the CPU was executing only code from the Microcode Sequencer. Our software setup can also be used with different fault injection systems, such as EM or lasers, which would probably be more accurate and precise than the current PMIC system.

As a final observation, the vulnerability of x86 CPUs should not come as a surprise, because: "tampering of internal hardware to compromise SGX is out of scope for SGX threat model"[8]. General purpose CPUs are not designed to be resistant to glitching, as it is not part of the threat model for these devices; secure hardware makes trade-offs incompatible with the performance expected from a modern CPU.

[8] https://zt-chen.github.io/voltpillager/#is-there-a-cve-and-how-did-intel-respond

8 RELATED WORK

Voltage fault injection in x86 CPUs via the onboard PMIC was pioneered by Plundervolt [36] and VoltJockey [38], which exploited a software voltage control interface that has since been disabled by Intel. In this study we use the same concept of Voltpillager [9] and PMFault [8]: inject commands in the bus that connects the PMIC to the CPU. We build a setup similar to Voltpillager, but use the same communication protocol employed in PMFault, as the PMIC on the DUT supports PMBus only. In our work, we focus on microarchitectural components of the CPU that were not analyzed in the aforementioned papers, but also show new architectural fault types with a higher success rate.

We leverage previous research on Goldmont microcode, especially: the seminal work by Ermolov et al. [14], the microcode update reverse engineering effort by Borrello et al. [4], and the convenient microcode patching library by Krog and Skovsende [33]. To build a usable microcode glitch setup, we embed lib-micro [33] in coreboot, modifying it to support 32-bit registers. We believe this choice is crucial, as it increases the execution speed of our tests by at least 3 orders of magnitude, and streamlines the setup, minimizing the number of active components during the glitch.

9 CONCLUSION

The first contribution of this research is an efficient glitching setup to attack architectural and microarchitectural code on Intel Goldmont SoCs. Using an inexpensive Raspberry Pi Pico board to control the onboard PMIC, we inject faults on the *Up Squared* board running coreboot. We also show evidence of novel types of architectural faults on x86, that we identify as corruptions in the register file and assembly instructions skip. We then show the existence of μ-instructions skip in the microcode, but are not successful in installing unsigned microcode updates. However, we prove that cryptographic checks during the microcode update process can be faulted.

Our research extends the boundaries of previous fault injection attacks on Intel CPUs, proving that the microcode is also affected. Building upon our results, future research should focus on investigating the effect of voltage glitches on microarchitectural components such as microcode or DFX security systems. Finally, other fault injection systems must also be evaluated to assess their practicality against complex target such as modern general purpose CPUs.

REFERENCES

[1] AAEON. 2021. UP Squared User's Manual. https://newdata.aaeon.com.tw/DOWNLOAD/MANUAL/UP%20Squared%20(UPS-APL)%20Manual%206th%20Ed.pdf
[2] Eli Biham and Adi Shamir. 1997. Differential Fault Analysis of Secret Key Cryptosystems. In *Annual International Cryptology Conference*. https://api.semanticscholar.org/CorpusID:12376527
[3] Dan Boneh, Richard Demillo, and Richard Lipton. 1999. On the Importance of Eliminating Errors in Cryptographic Computations. *Journal of Cryptology* 14 (07 1999). https://doi.org/10.1007/s001450010016
[4] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. 2023. CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode. In *2023 IEEE Security and Privacy Workshops (SPW)*. 285–297. https://doi.org/10.1109/SPW59333.2023.00031
[5] Peter Bosch. 2019. Intel Management Engine Deep Dive. (2019). Slideshow presented at 36C3.
[6] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's

Secure Encrypted Virtualization *(CCS '21)*. Association for Computing Machinery. https://doi.org/10.1145/3460120.3484779
[7] Suresh Chari, Josyula Rao, and Pankaj Rohatgi. 2002. Template Attacks, Vol. 2523. 13–28. https://doi.org/10.1007/3-540-36400-5_3
[8] Zitai Chen and David Oswald. 2023. PMFault: Faulting and Bricking Server CPUs through Management Interfaces: Or: A Modern Example of Halt and Catch Fire. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 2 (Mar. 2023). https://doi.org/10.46586/tches.v2023.i2.1-23
[9] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai
[10] Mark Ermolov. 2019. Tweet on abbreviations relating to Power Management of Intel Chips. https://twitter.com/_markel___/status/1175851055676043267
[11] Mark Ermolov. 2021. Reverse engineering an Intel microcode update that fixes a hardware bug. https://twitter.com/_markel___/status/1465801588468035587
[12] Mark Ermolov. 2024. Tweet about reverse engineering an Intel microcode update. https://twitter.com/_markel___/status/1774955302720254279
[13] Mark Ermolov and Maxim Goryachy. 2017. How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine. (2017). Slideshow presented at Black Hat Europe 2017.
[14] Mark Ermolov, Dmitry Sklyarov, and Maxim Goryachy. 2022. Undocumented x86 instructions to control the CPU at the microarchitecture level in modern Intel processors. *Journal of Computer Virology and Hacking Techniques* 19 (08 2022). https://doi.org/10.1007/s11416-022-00438-x
[15] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems — CHES 2001*, Çetin K. Koç, David Naccache, and Christof Paar (Eds.). Springer Berlin Heidelberg.
[16] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. 2018. Intel TXE PoC. https://github.com/ptresearch/IntelTXE-PoC
[17] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. 2020. glm-ucode. https://github.com/chip-red-pill/glm-ucode
[18] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. 2022. Chip Red Pill. (2022). Slideshow presented at OffensiveCon22.
[19] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. 2022. MicrocodeDecryptor. https://github.com/chip-red-pill/MicrocodeDecryptor
[20] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. 2022. uCodeDisasm. https://github.com/chip-red-pill/uCodeDisasm
[21] Linley Gwennap. 1997. P6 Microcode Can Be Patched. *Microprocessor Report* (Sep 1997). https://web.archive.org/web/20091221182054/https://www.ele.uva.es/~jesman/BigSeti/ftp/Cajon-Desastre/MPR/111204.pdf
[22] Ben Hawkes. 2013. Notes on Intel Microcode Updates. https://web.archive.org/web/20230217224318/http://inertiawar.com/microcode/
[23] Yeoh Eng Hong, Lim Seong Leong, Wong Yik Choong, and Mahmud Adnan. 1998. An Overview of Advanced Failure Analysis Techniques for Pentium and Pentium Pro Microprocessors. *Intel Technology Journal* 2. https://www.intel.com/content/dam/www/public/us/en/documents/research/1998-vol02-iss-2-intel-technology-journal.pdf#page=2
[24] Texas Instruments. 2019. TPS65094 PMIC for Intel Apollo Lake Platform. https://www.ti.com/lit/ds/symlink/tps65094.pdf
[25] Intel. 2018. Intel Atom Processor C3000 Product Family. https://www.mouser.com/datasheet/2/612/c3000-family-datasheet-1623704.pdf
[26] Intel. 2021. Intel Debug Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/intel-debug-technology.html
[27] Intel. 2023. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html
[28] Andrew Jenner. 2020. 8086 microcode disassembled. https://www.reenigne.org/blog/8086-microcode-disassembled/
[29] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.
[30] Chong Kim and Jean-Jacques Quisquater. 2007. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures, Vol. 4462. 215–228. https://doi.org/10.1007/978-3-540-72354-7_18
[31] Paul Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.). Springer Berlin Heidelberg.
[32] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer Berlin Heidelberg.
[33] Alexander Krog and Alexander Skovsende. 2023. redunlock-lib-micro. https://libmicro.dev/
[34] Raspberry Pi Ltd. 2021. RP2040 Datasheet. https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf
[35] Plato Mavropoulos. [n.d.]. Intel Microcode Extra Undocumented Header. https://github.com/platomav/MCExtractor/wiki/Intel-Microcode-Extra-Undocumented-Header
[36] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1466–1482. https://doi.org/10.1109/SP40000.2020.00057
[37] Colin O'Flynn. 2016. Fault Injection using Crowbars on Embedded Systems. *IACR Cryptol. ePrint Arch.* 2016 (2016), 810. https://api.semanticscholar.org/CorpusID:8502986
[38] Pengfei Qui, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2020. VoltJockey: Abusing the Processor Voltage to Break Arm TrustZone. *GetMobile: Mobile Comp. and Comm.* 24, 2 (2020). https://doi.org/10.1145/3427384.3427394
[39] Sergei Skorobogatov and Ross Anderson. 2002. Optical Fault Induction Attacks. *Optical Fault Induction Attacks* 2523, 2–12. https://doi.org/10.1007/3-540-36400-5_2
[40] James Sutton. 2002. Microcode patch authentication. US Patent 20030196096A1.
[41] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 624–637. https://doi.org/10.1109/ISCA.2018.00058

IMAGE ATTRIBUTIONS

EXPERIMENTS PLOT

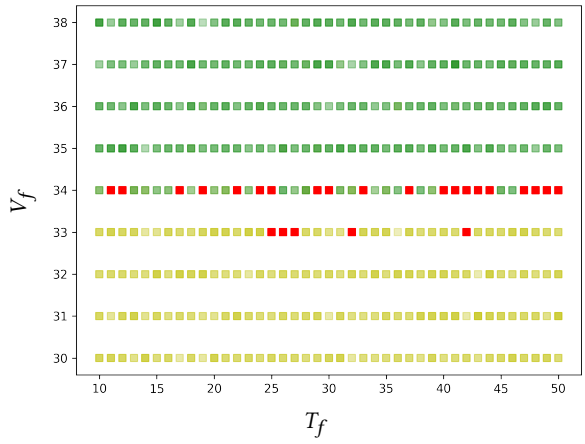

Figure 12: IMUL glitching (db table _02a46ea_mul)

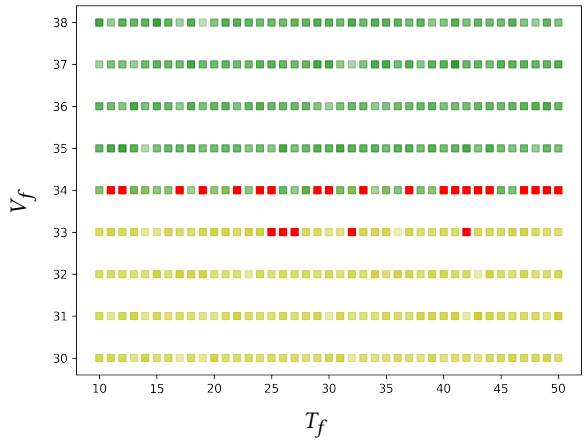

Figure 13: IMUL glitching (db table _02a46ea_mul)

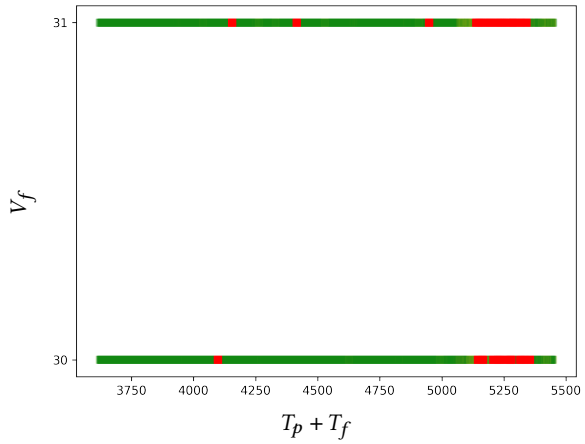

Figure 14: Faults in signature check glitch campaign (success: $T_u > 6{,}500{,}000$)

EXPERIMENTS CODE

Assembly code is in AT&T syntax, as used in coreboot, and x86 microcode uses lib-micro [33] C macros syntax.

```
movl $0x80000, %eax;  # operand1
movl $0x4, %ebx;      # operand2
movl %eax, %edx;      # loop head
imull %ebx, %edx;
movl %eax, %edi;
imull %ebx, %edi;
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;        # goto loop head
```

Listing 6: IMUL target code

```
{
    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),
    ADD_DSZ64_DRI(RCX, RCX, 1),
    ADD_DSZ64_DRI(RCX, RCX, 1),
    NOP_SEQWORD
},
{
    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
}
```

Listing 7: µ-instructions skip target

```
#define IA32_BIOS_UPDT_TRIG    0x79
while (true) {
  uart_tx('R');
  uint64_t ucode_tsc_start = timestamp_get();
  __asm__ __volatile__ (
    "wrmsr;\t\n"
    "nop;\t\n" // Repeat 500.000 times
    : : "c" (IA32_BIOS_UPDT_TRIG), "a"
    ↪ (ucode_patch_addr), "d" (0)
  );
  uint64_t ucode_tsc_end = timestamp_get();
  uart_tx('D');
  putu32(uart_base, read_microcode_rev());
  putu32(uart_base, ucode_tsc_end - ucode_tsc_start);
}
```

Listing 8: Microcode update target code