MicroSpark: Testing Voltage Glitches on Intel Microcode

Federico Cerutti
Università degli Studi di Brescia *

Alvise de Faveri Tron
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Abstract

The ability to cause precise faults during the execution of *microprograms*, e.g., the microcode update routine of a CPU, can constitute a powerful and stealthy primitive to compromise a CPU at the microarchitectural level. However, the complexity of modern CPUs inherently limits the possibility of inducing such controlled faults in microarchitectural components. It is therefore unclear to what extent an attacker might be able to use hardware fault injection attacks against microcode.

In this work, we present a robust and reproducible experimental platform that pairs a low-cost glitcher with a coreboot-based environment on a "Red Unlocked" Intel Goldmont SoC. Leveraging the low-noise execution environment provided by this setup, as well as the possibility of crafting custom microcode routines due to Red Unlock, we successfully identified previously undocumented fault types, including instruction skips at both the architectural and microarchitectural levels on Apollo Lake CPUs. This provides a foundation for future investigations into Intel microcode security, and the potential for broader applications across various CPU models.

1 Introduction

The x86 instruction set relies on microcode to implement complex instructions, a design principle established with the 8086 microprocessor [23]. This means that certain instructions, deemed too complex to implement in hardware, are executed as elaborate microprograms in the CPU backend.

The "Red Unlock" of Intel Goldmont SoCs by Ermolov and Goryachy [11] unlocked powerful debugging features on production CPU units. It made possible to dump the microcode ROM, reverse engineer the architecture, release a disassembler [16], and even identify some undocumented x86 opcodes used to debug microcode [12]. Ultimately, this allows to write and execute custom microprograms on Intel Goldmont CPUs, which can be used to study the processor's microarchitecture.

Concurrently, separate research efforts demonstrated the feasibility of voltage fault injection attacks on Intel CPUs, leveraging hardware on the motherboard to inject glitches in x86 cores. Researchers demonstrated that they could fault some operations (`IMUL`) as well as extract cryptographic keys from the Intel Software Guard Extensions (SGX) secure enclave [9, 8, 28, 24, 30].

We aim to investigate whether the findings from these two distinct areas of research can be combined to manipulate the behavior of the CPU at the microcode level.

To this end, we built a robust setup to experiment voltage fault injection on the Goldmont platform. This has two main advantages:

1. The extensive public research on Goldmont microcode structure gives the ability to modify x86 instructions behavior with custom microprograms [12], and it allows us to test individual aspects of the hardware.

2. Open firmware projects support Goldmont CPUs, making it possible to easily create a stable test environment.

We show how our setup is able to achieve instruction skips at both architectural and microarchitectural level on Goldmont CPUs, and present a characterization of fault types for microcoded instructions.

Our contributions can be summarized as follows:

1. We present a coreboot-based open-source tool[1] to efficiently conduct (microcode) glitching experiments on Goldmont CPUs.

2. We provide a characterization of new behaviors in Intel CPUs under fault conditions, such as (micro)instruction skips and faults in basic arithmetic operations, which were previously considered immune to faults [28].

3. We present the first documented instances of microcode-level glitches occurring in customized microprograms.

*This project was developed while at the *Vrije Universiteit Amsterdam*

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

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 [18]. As such, all IP blocks now incorporate extensive DFX features. These pervasive debugging features are not limited to preproduction chips, but they remain accessible on final production devices to analyze faults that might appear on the final user's hardware. Since they enable fine-grained debugging of 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 [21]: 1. Red - Intel internal 2. Orange - BIOS vendors 3. Green - Customers

Green unlock provides access to the architectural state only 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 [12]: 1. Hardware straps; 2. Efuses; 3. JTAG password; 4. Software. To unlock DFX mode via software, another privileged component of the CPU or Platform Controller Hub (PCH), e.g., the Power Control Unit (PCU) or the Management Engine (ME), must write the value corresponding to a specific DFX level to the `PERSONALITY` register of the DFX AGG.

On desktop and server CPUs there are two different DFX AGGs, one for the PCH, and one for the ME. As such, 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 [15]. Through a code execution exploit on Intel ME version 11, Goryachy et al. [15] were able to achieve Red Unlock on Intel Goldmont/Apollo Lake SoCs, unlocking debug of all IP cores.

2.2 From Assembly to Microcode

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

Each instruction coming from the Instruction Cache 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. Such decoders can be divided in:

1. **Simple Decoders**, which directly map an x86 instruction to a µ-instructions.
2. **Multi-Instructions Decoders**, which translate instructions to a sequence of 2-4 µ-instructions.

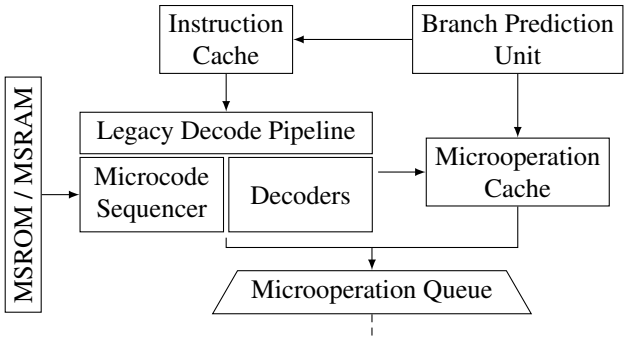

Figure 1: Intel Skylake Pipeline Frontend, simplified [22, Figure 2-4][31]

3. a **Microcode Sequencer** that maps complex instructions like `CPUID` or `RDRAND` to long µ-instructions chains, called microprograms [22, §B.5.7.4][5].

The Microcode Sequencer uses components that resemble those of a full CPU, such as an instruction pointer, a set of registers and memory [16].

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

2.3 Microcode Internals

By gaining complete debug access on Goldmont through Red Unlock [15], researchers were able to read the Microcode Sequencer ROM (MSROM), reverse engineer its structure and functionality, and figure out how microcode can be patched [12]. Further research resulted in a UEFI-based framework [5] and a Linux library [25], both capable of assembling and installing custom microcode patches.

2.3.1 Microcode Structure

Microcode execution in Goldmont CPUs is performed in *triads*, i.e., triplets of µ-instructions that are executed in parallel. The individual result of each µ-instruction is combined to the other components of the triad based on the directives contained in a separate sequence word (seqword) [15]. Such triplets, and their corresponding seqwords, are stored in the MSROM.

In particular, the MSROM is composed of two different arrays: one contains quartets of µ-instructions (three are populated, the fourth is zero-filled), and the other stores the corresponding seqwords. There is a direct address correspondence between the two arrays: the seqword at a given address con-

trols the μ-instruction triad located at the same address in the μ-instruction array [15].

Finally, microprograms can also be *patched*. Microcode patches are stored in the Microcode Sequencer RAM (MSRAM) in two arrays that mirror the MSROM structure.

2.3.2 Match/Patch Registers

Match and Patch register pairs are used to implement the microcode patching functionality. In particular, each pair forces the Microcode Sequencer to transparently steer execution from a matched address, to a new patch destination [14, 5]. They can be used to decide whether the Microcode Sequencer should execute the base version of a microcode program, which is stored at production time in the MSROM, or a patched version stored in the MSRAM. After writing microcode patches in MSRAM, the execution unit must be informed of which MSROM address should be replaced with the new code. On Goldmont CPUs there are 64 such 31-bits registers.

2.4 Voltage Fault Injection

Voltage-based fault injection, also known as voltage *glitching*, is a well-known, cheap technique that can be used to inject faults on security-sensitive software, either through a software interface or through direct physical access. In scenarios in which an attacker has physical access to the target, it generally involves modifying the power rail of the Device Under Test (DUT) to inject a glitch, either directly via the power supply or through crowbar glitching [29]. While in the general case such glitches could completely disrupt the hardware's ability to execute software, when accurately injected they allow for attacks such as skipping instructions that perform checks or inserting faults in a cryptographic algorithm to recover the key through Differential Fault Analysis (DFA) [4, 3].

However, applying these methods to modern x86 CPUs is non-trivial, as doing so requires managing both their high current consumption and complex bring-up sequences.

2.4.1 Glitching Through Voltage Scaling

As research on these techniques has advanced, a new voltage fault injection vector has emerged: generating glitches through the Power Management Integrated Chip (PMIC), the power supply used for CPU frequency/voltage scaling. These attacks can be divided in two groups, based on whether they require physical access to the target system:

1. **Remote access**: CPU voltage is controlled indirectly through the software interface exposed by the CPU. This was exploited in Plundervolt [28] and V0LTpwn [24], and Intel quickly mitigated these attacks disabling the voltage control interface via a microcode update.
2. **Physical access**: VoltPillager [9], PMFault [8] and Buhren et al. [7] attacks require modifications the DUT, gaining access to the data lines that connect the CPU to the PMIC. In particular, the attacker needs to add an external hardware device that instructs the PMIC to change the supplied voltage, which grants higher reliability and finer control on the glitch shape. This can be done accurately with a relatively low-budget setup.

The authors of the first attack, Plundervolt [28], found the `IMUL` instruction to be susceptible to bit flips when using specific operands in a certain order, whereas simpler 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 applets through DFA. Building upon this work, the researchers behind VoltPillager [9] addressed Intel's mitigation, which removed software access to voltage controls, connecting directly to the motherboard's voltage regulator chip. This hardware-based approach not only replicated the results of Plundervolt on fully patched systems, but also demonstrated new fault types, and allowed for more precise voltage control, leading to higher success rates.

2.4.2 Intel Voltage Scaling

On Intel CPUs specifically, it is possible to find multiple power management systems that operate at different granularity levels [10]:

1. **Power Management Controller (PMC)**: A coprocessor in the PCH that manages power at the package level and handles power state transitions [6].
2. **Power Control Unit (PCU)**: A coprocessor in the CPU that runs custom PCODE and controls the CPU power state.
3. **Power Management Unit (PMU)**: Hardwired logic to handle power management in the PCH.
4. **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:

1. **SVID**: An SPI-like Intel proprietary communication protocol documented in CPU datasheets [20, Table 34-4], and used in the VoltPillager attack [9] to inject faults in the CPU.

2. **PMBus**: An open-standard 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

This research is a study into the feasibility of voltage fault injection at the microarchitectural level. We consider an attacker with physical access to the victim hardware, 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 remote attacker. Furthermore, the authors of PMFault [8] were able to use the Baseboard Management Controller (BMC) to remotely inject faults, which makes the physical access constraint optional when targeting specific devices.

While our work concentrates on Goldmont CPUs since they provide an experimental ground truth for microprograms, it should be noted that our hardware/software setup does not intrinsically require Red Unlock to attack architectural operations and pre-existing microprograms, and our infrastructure can be in principle used to test to other CPU families.

4 Overview

In this work, we aim to evaluate the ability of an attacker to inject precise faults on a modern CPU with a minimal setup, by injecting messages into the bus between the PMIC and the PCH. To test for the effects of such glitches on *microcode* specifically, we target Intel Goldmont CPUs, for which we can craft arbitrary sequences of microinstructions by leveraging the Red Unlock privileges.

Since the target market for Goldmont processors differs substantially from that of the Core-series processors used in previous fault injection studies (Section 2.4), there are notable differences in both microarchitectural design of the CPU and in the electrical properties of the PMICs. Hence, to establish a baseline for our specific hardware, we first tested known architectural faults on this target.

After verifying that voltage fault injection is effective 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 which μ-instructions the CPU is executing.

4.1 Challenges

Successfully injecting and characterizing microcode faults presents several key challenges. First, the target CPU supports exclusively PMBus, which limits the temporal resolution of the injected glitches due to its lower speed. A direct communication channel is also needed to synchronize the external glitching hardware with the target. We use the onboard UART for this purpose, allowing the target to signal when each experiment begins. Finally, the unpredictable nature of voltage glitches demands a carefully controlled, noise-free execution environment. Running the experiments on top of a full operating system would introduce significant variability due to task switching and background processes, making it impossible to reliably attribute observed faults. A minimal firmware-based setup is therefore essential to isolate execution to only the target code, enabling a clear characterization of the effects on specific instructions and microinstructions.

5 Experimental Setup

To address the challenges described in subsection 4.1 we built an experimental setup based on inexpensive and widely available components. Figure 2 provides a summary of the setup, while a complete picture can be found in Figure 9.

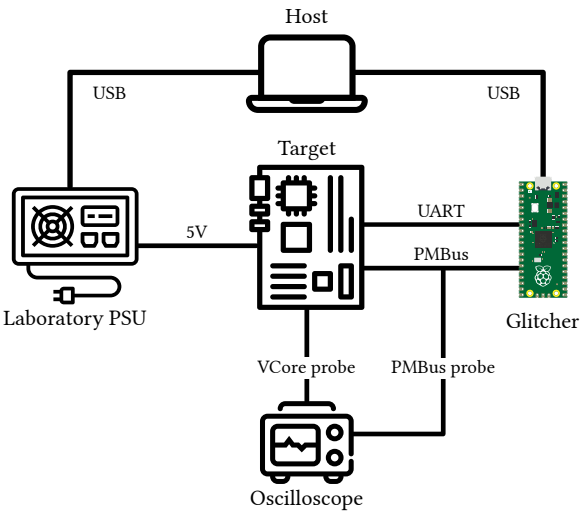

Figure 2: Experimental setup overview

The DUT (*Target*) runs code in a loop, and is essentially independent from the rest of the system. A cheap microcontroller board (*Glitcher*) injects control packets to the target's PMIC and monitors the state of the target. A computer (*Host*) orchestrates the glitching campaign, configuring the glitcher and deciding when to reset the DUT based on the results received from the glitcher. A programmable power supply (*PSU*), connected to the main power input of the DUT, is used to power-cycle the target when it becomes unresponsive.

Finally, an *Oscilloscope* is used to get an independent measurement of the CPU Core power supply line (VCore) and check the state of the PMBus bus for debugging purposes.

5.1 Target

We chose the *UP Squared* Single Board Computer (SBC) with a Pentium N4200 CPU as our target, utilizing multiple boards with both 4GB and 8GB of soldered RAM to validate the repeatability of our results across different samples. The Red Unlock exploit by Ermolov and Goryachy [11] has been recently ported to this device by Krog and Skovsende [26]. Notably, the authors of the port have also released a complete BIOS image that embeds both the exploit and the vulnerable Intel ME binary, making the attack highly reproducible with publicly available software (the file is also available on our repository[2]). Since the N4200 CPU does not allow for software-based voltage control, we are only able to inject commands through the power management bus, which is PMBus for this board. This is done by soldering the Glitcher wires to the PCB traces that connect the CPU to the PMIC.

5.1.1 Hardware

The *Squared* uses a Texas Instruments TPS65094 PMIC, specifically tailored for Apollo Lake CPUs, which automatically manages voltage sequencing and chip bring up. We modify the target *Squared* board to access:

1. PMBus signals (SDA, SCL): to send the PMIC our control packets.

2. VCore: The tension supplied to core circuitry (execution units) [20, Table 41-5].

The TPS65094 datasheet [19] provides a reference implementation that helps in locating the PMBus traces on the circuit board.

To communicate with the glitcher, we use the UART serial bus on connector CN16 [2]. While it would be preferable to use UART flow control pins like RTS or CTS as a trigger due to their atomicity, we are unable to identify them on the board and conclude that they are not exposed.

5.1.2 BIOS

In our tests, booting any Red Unlockable board with the stock BIOS and the Red Unlock exploit enabled takes multiple minutes. This would severely limit the feasibility of this study, as we anticipate a high number of resets for the target, especially in the exploratory phase of our attack with broad glitching settings ranges. The *UP Squared*, however, is supported by the coreboot project[3]. This provides us with a minimal, open-source BIOS codebase that we can modify to fit our needs. Furthermore, since coreboot is a single-threaded application, there is no scheduling or resource contention, which yields 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 0*x*88). Normally, this operation would be instantaneous, but when the Red Unlock exploit [13] is included in the ME image, this step takes minutes to execute, before eventually booting successfully. We speculate this delay is due to the exploit itself, 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 out, and return control to coreboot, which proceeds to launch its payload. Notably, the FSP notify phase can also be skipped, and the target will quickly boot into Linux.

In order to execute experiments as quickly as possible after power-up, all of our experiments run directly within coreboot, before the FSP notify phase. Since we also want to leverage the possibility of installing custom microcode in the CPU, we modified lib-micro[4] [25] to work in 32-bit protected execution mode, which is the standard for coreboot. Empirically we observed that a stable experimental environment for microcode can be setup only after the firmware configures the package power limits, otherwise the CPU would hang.

The setup we obtained is able to execute code within 700 ms of power-up, and can successfully load custom microcode patches.

5.1.3 Software

Each experiment running on the target is composed of the following operations (cf. Listing 1):

1. The target sends a synchronization character (`R`) to the glitcher via UART to signal the start of the experiment.

2. Testcase runs on the target.

3. The target sends a second synchronization character to notify completion (`D`).

4. The result of the experiment is sent to the glitcher (cf. Figure 3).

The communication is unidirectional and the target runs independently of any other component of the setup. This procedure 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.

[2] https://github.com/ceres-c/coreboot/blob/thesis-24.02.01/blobs_libmicro/coreboot.rom

[3] https://coreboot.org/

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

```
uint32_t output;
while (true) {
  uart_tx('R');
  /* Unrolled asm 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. Moreover, since loops ending in a conditional jump would activate the branch predictor, and are themselves a potential injection target, we manually unroll any loops in the testcase to make sure that any effects observed in the output come from the testcase instructions.

5.2 Glitcher

5.2.1 Hardware

The glitcher hardware does not have stringent requirements: any microcontroller supporting I2C and UART communication would be suitable. For this work, we selected the cheap and widely available Raspberry Pi Pico board, which mounts an RP2040 microcontroller. To ensure compatibility with the DUT, which utilizes a PMBus bus voltage of 1.8 V, we modified the Pico board to power the RP2040 with an external 1.8 V power supply, as stated in the datasheet [27, §2.9.7.3].

Since the glitcher also communicates with the target on the UART bus, which operates at 3.3 V, we added a Texas Instruments TXS0102 Bidirectional Voltage-Level Translator to step the UART voltage down to a safe level for the Pico.

We note that our modifications to the Pico board, which required SMD soldering, can be bypassed by using a board that natively operates at 1.8 V, such as the ST NUCLEO-U575ZI-Q.

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 the latter if the target becomes unreachable.

Figure 3 describes the sequence of operations for the glitcher. Once the glitcher is armed by the host computer, it will wait for the next synchronization character `R` from the target, and inject a PMBus packet to trigger a voltage drop. If the target is still online after the glitch, the `output` of the current testcase execution is received from target and forwarded to the host.

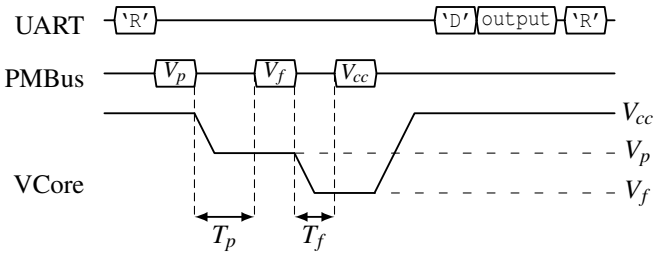

Figure 3: Glitch signal sequence

5.2.3 Two-Stage Glitching

While the *Squared* board is one the few targets that has both a working Red Unlock exploit readily available and support for Coreboot, we found that its PMIC's slew rate is limited to $\sim$3 mV/µs [19]. For comparison, the PMIC used in the VoltPillager attack [9] had a 20 mV/µs slew rate, and specialized tools like the ChipWhisperer can achieve rates over 1500 mV/µs [29], 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.

When designing the shape of our glitch, we must accommodate for such limitation. Our initial attempts to inject a glitch using a direct, single-slope voltage drop proved ineffective due to the PMIC's low slew rate. To overcome this, we adopted a two-stage glitch approach described by Chen et al. [9]. This technique involves lowering the voltage to a *preparation voltage* V_p, where the target remains operational, before initiating the final voltage drop for the glitch itself. While this does not change the effective slew rate, it allows the target to stabilize at a lower voltage, and results in voltage drops with higher temporal precision. Our glitches (cfr Figure 3) are therefore characterized by the following parameters:

1. V_p: Preparation voltage, the lower bound of the CPU stability voltage range.
2. V_f: Fault voltage, the target voltage during the glitch.
3. T_p: Preparation time, the external offset of the glitch, or for how long V_p must be held (µs).
4. 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 for its clean signal and absence of over/undershoot

relative to the set point voltage when enabling the output. This model can be controlled over USB, simplifying the process of rebooting the DUT. The *Squared* is powered at 5.3 V, as measured from the original power supply, and consumes on average 1.1 A when running coreboot.

5.4 Host

The host PC 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 using the laboratory power supply. We have developed an extensible Python library that provides convenient APIs for interacting with the glitcher.

After every testcase, the result of the current operation on the DUT is analyzed by the host to identify faulty executions. Each individual result (normal output, fault or crash) is collected as a data point in an SQLite database [1] and visualized with Matplotlib in a Jupyter notebook.

6 Glitching Experiments

The goal of our experiments is to first assess if our specific target device is vulnerable to known architectural glitches, and then expand to microarchitectural components of the CPU by crafting ad-hoc microprograms. For our microcode experiments, we first target microinstructions that map to architectural faults, to study such faults in detail. Secondly, we target instructions that are highly likely to cause security-sensitive side-effects if glitched, e.g. skipping comparison instructions.

We prepare experiments that execute small snippets of x86 assembly in an unrolled loop, as mentioned in Listing 1, to characterize specific aspects of DUT when attacked with voltage glitching. Finally, we repeat similar experiments on microprograms, by crafting custom microcode patches and executing the patched instructions multiple times in a sequence. We test the same snippet with different values for V_f and T_f ("depth" and "width" of the glitch, respectively) and report the result of each individual run in a plot:

- Green dots represent normal executions.
- Yellow dots represent unexpected states (hangs, garbled data, no response, etc.).
- Red dots represent successful glitches.

In the plots, color intensity indicates datapoint density. Since each combination of settings is tested multiple times, a darker shade represents a higher concentration of identical results at that position.

Throughout all our experiments, voltages are expressed as PMIC Voltage IDs (VIDs) as per the PMIC datasheet [19, Table 6-3], and time measurements are reported in µs. Assembly code is in AT&T syntax, as used in coreboot, and x86 microcode uses lib-micro [25] C macros syntax.

6.1 Identifying Glitch Parameters

Each experiment requires an exploratory 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 preliminary stage with two considerations:

1. We aim for V_p to be near the lowest voltage at which the CPU executes the specific payload without faults. The lower V_p is, the faster we can drop to V_f (cf. Figure 3).

   Setting $V_f = V_{cc}$ 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 interacting negatively with the target, we maintain VCore at V_p for a long period T_p, and verify the DUT is not crashing due to the voltage drop.

   As an example, in Figure 4 data points for $V_p > 35$ are mostly green (i.e. normal execution): an interval around $V_p = 35$ is a good initial range.

2. T_p does not need to be precise when running characterization experiments. Each iteration of the (unrolled) loop execute the same code thousands of times, 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 would be critical when attacking the microcode update, as in that case it would be necessary glitch a specific operation within the whole update procedure.

6.2 Assembly Experiments

6.2.1 `IMUL`

We first replicate the attack against `IMUL` presented in Plundervolt: Murdock et al. [28] 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.

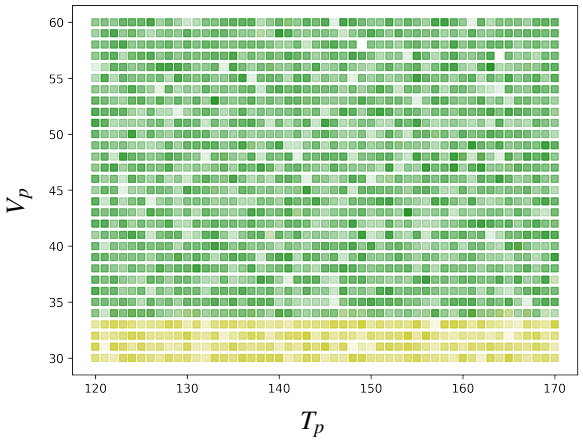

Figure 4: Estimating V_p for `IMUL` glitching

```
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 rate of $\sim$4%[5]. 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). Figure 5 shows the results of our campaign.

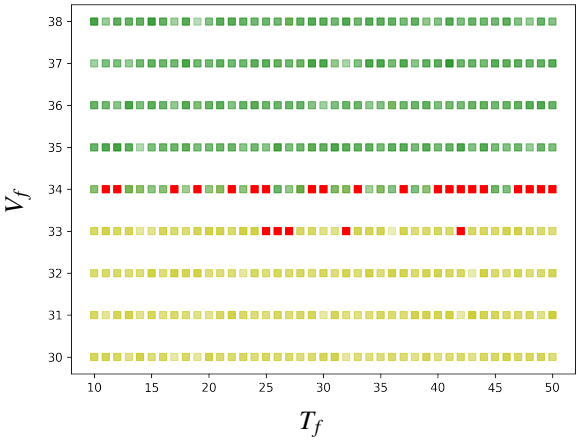

Figure 5: `IMUL` glitching

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` [6].
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[7]. 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`. A possible explanation is that, due to the injected glitch, the full 32-bit register is being added to `ecx` instead of only the low 8-bits.

These two facts suggest that the component affected by the voltage drop might not be the multiplier, but rather 1. the `CMP` instruction of the snippet in Listing 2 or 2. the Register File.

6.2.2 CMP

As our next target, we choose to test `CMP` instructions. These instructions are known to be an interesting glitching target, as software can use comparisons to check protections, access rights, or signature validity.

To investigate this behavior, we further reduce the code in Listing 2. In the new target code, `CMP` compares two fixed

[5] Database [1] table _02a46ea_mul_2
[6] Database [1] tables _02a46ea_mul_2 and _02a46ea_mul_swap
[7] Database [1] table _02a46ea_mul_2

operands, and a counter value is incremented whenever they are found to be different (cf. Listing 3). It's worth noting that this code also relies on the Arithmetic Logic Unit (ALU), because `CMP` on x86 is implemented as a `SUB`, the result of which is then discarded.

```
movl $0xAAAAAAAA, %eax;
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 6) reporting the two values to be non-equal at least once[8]. Since the optimal glitch parameters (V_p and V_f) were such that the DUT was not resetting often, we achieve a 72 glitch/s rate. 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.

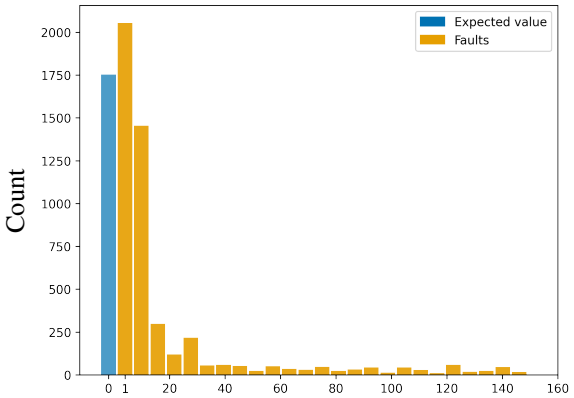

Figure 6: Distribution of faults in `CMP` test

6.2.3 Register File

With this experiment we want to test whether the induced glitches are able to affect the values stored in the Register File. To this end, 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 an experiment runtime that allows VCore to drop to sufficiently low values, the (unrolled) loop is executed 271 000 times, with the expected value in `ecx` also being 271 000.

[8] Database [1] table _5e872de_cmp_2

```
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 rather than just the single byte `0x01`, hence the final result will be larger. However, as shown in Figure 7, we observe values that are either slightly smaller than the expected 271 000, or normally distributed around 262 500[9], hinting at instructions skip. We found 2385 non-expected results on a total of 11 854 attempts, a 19% success rate, with 0.375 glitch/s. Specifically:

- 944 (40%) result $<$ 269500.
- 1289 (54%) 269500 $<$ result $<$ 271000.
- 152 (6%) result $>$ 271000, but these values were orders of magnitude greater than the expected value, so we classify them as false positives.

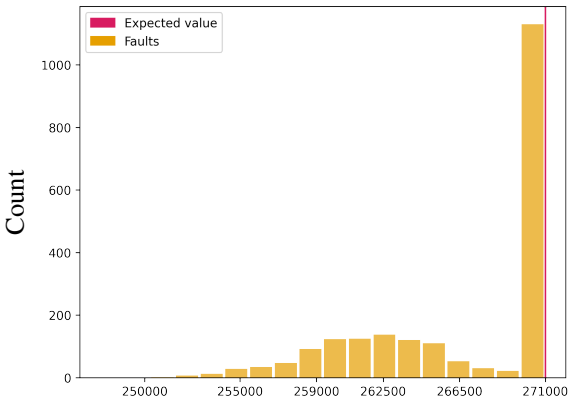

Figure 7: Distribution of result values in register file test (successful glitches only)

This analysis yields two key findings. First, it demonstrates that Intel's low-power SoCs are susceptible to fault injection attacks previously observed only in their high-power Core CPU counterparts. Second, we found that even simple instructions can be glitched or skipped.

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

[9] Database [1] table _03168d2_reg

our focus on injecting glitches in microarchitectural components. We choose μ-instructions that are closely related to architectural instructions we found to be vulnerable, and that could help in glitching real microprograms.

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 32-bit compatible version of lib-micro[10] [25], we move our test code "inside" the `RDRAND` operation. Thus, the body of the assembly loop from Listing 1 now consists of calls to `RDRAND` only.

6.3.1 CMP Branchless

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[11], but the number of faulted instructions are not coherent with our expected values, and we deem them to be false positives with corrupted data from the target misinterpreted as a successful glitch. This result suggests that the x86 instructions in Listing 3, which was successfully glitched, have a more complex implementation than the basic microcode we are now testing.

6.3.2 CMP Branching

To further investigate the the architectural faults observed in 6.2.2, we implemented a more complex version of the CMP microprogram that utilizes conditional jumps. Due to implementation details of jumps in the microcode sequencer [16], we created two distinct microprograms to thoroughly test this target: one with a speculatively taken branch on equality (Listing 6) and another with a speculatively taken branch on inequality (Listing 7).

[10] https://github.com/ceres-c/lib-micro
[11] Database [1] table _5e872de_rdrand_cmp_ne_3

Extensive testing of both variants, for durations up to 63 h[12], failed to produce any meaningful faults. We therefore rule out the comparison logic circuit, microcode jump operations, and branch misprediction as possible root causes of the previously observed faults. It is then possible to conclude that the faults in the `CMP` x86 operation were caused by architectural components of the CPU.

6.3.3 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 8 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 `ADD`s. Conversely, if the value is not a multiple of 10, then a μ-instruction was skipped.

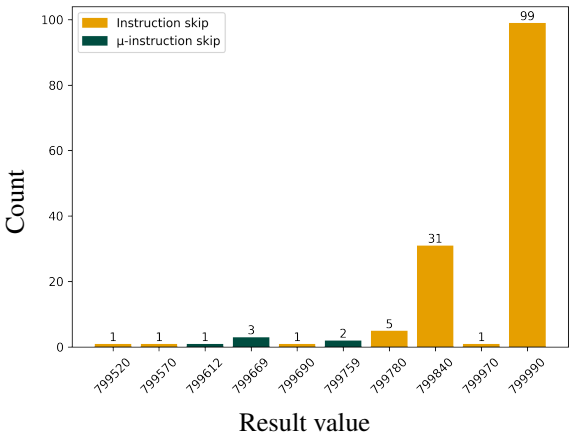

Figure 8: Distribution of result values in μ-instruction skip test (successful glitches only)

After tuning our glitch settings, we found 160 faulty executions after 106 000 attempts, a 0.1% success rate[13]. 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 our settings (cf. Figure 8). The high prevalence of 799 990 also tells that in most experiments only a single x86 instruction is skipped.

[12] Database [1] table _8Gb_ab3a109_ucode_cmp_jmp_ne
[13] Database [1] table _4a2b3cf_customucode_add10

6.3.4 Bit-Flipping in µ-instructions

We modify the target from 6.3.3 to consolidate all the additions in a single architectural RDRAND call. A microcode-only loop implementation (Listing 9) further isolates the execution to the microcode engine.

This experiment did not produce any instruction skips. Instead, all faults were data corruptions, producing values like `0x39fffd` and `0xe9fffd`, consistently one or two bitflips away from the expected value of `0x29fffd`. The location of these bitflips, predominantly within the 3rd byte of the result, corresponds to the architectural pattern reported in Plundervolt [28], suggesting that the original fault mode extends to the microarchitectural level. There were 61 such data corruptions over 102 222 attempts, a 0.05% success rate at $\sim$1 glitch/min[14].

6.3.5 URAM

To test if energy-intensive components of the microarchitecture were more vulnerable, we targeted the Microcode RAM (URAM). Our microprogram executes a loop that writes a value to URAM, reads it back, and compares the two, incrementing an architectural register if a mismatch is detected (Listing 10). Despite extensive testing (64 h), this experiment failed to produce any successful glitches, and we conclude the URAM is highly robust against the supply voltage fluctuations we can induce.

These findings indicate that, predictably, the microarchitecture is a more challenging target for fault injection than the architectural level. We anticipated this difficulty because voltage glitches would likely crash other sensitive, power-hungry CPU components before affecting our intended targets. Despite this, our most crucial discovery is that these components are not immune. We successfully induced faults analogous to those at the architectural level, such as bit-flips and instruction skips, proving that the microarchitecture can be manipulated via voltage glitching.

6.4 Microcode Updates

Having identified faults using minimal, custom-designed microprograms, we discuss how our methodology could be applied to a more complex, real-world target, i.e., the native microcode update procedure. The internal logic for the update routine has been extensively reverse-engineered by previous research, which allows to treat it as a gray-box problem rather than a black box. Furthermore, the update process presents a fundamentally distinct target for fault injection, as the update routine operates in a minimal environment where hyper-threading is disabled, data is not accessed through caches [5][15], and likely various other microarchitectural units are powered down. Crucially, this implies that the power profile in this context differs significantly from the conditions of our prior experiments, potentially changing how voltage glitches affect operation.

6.4.1 Testing Setup Modifications

Target code in Listing 1 can be changed to perform a microcode update. In addition to checking the installed microcode version after the update, the execution time of the microcode update can be monitored through `RDTSC`. This provides two different ways to evaluate the outcome of a glitch:

1. Comparing the final installed microcode **version number** with the version automatically loaded at boot. This can be useful when modifying the update file, as the update process should always detect the that the update is not correctly signed and abort before completion.

2. Comparing the **duration** of the update T_u to the median duration $\text{Med}(T_u)$ observed when the same update file is used without injecting faults. Since the microcode update process is not constant-time [17], this metric tells us if the update microprogram continued further as a result of glitching. This could happen if some check is skipped or a loop counter is corrupted.

We measured that a successful update to a newer microcode revision takes $T_u \approx 6\,740\,000$ cycles ($\sim$6.2 ms), but once the update has been applied, each subsequent execution of the update procedure will be slightly shorter $\text{Med}(T_u) = 6\,160\,509$ cycles ($\sim$5.6 ms), as the CPU will not apply the same version twice. However, when attacking the update routine, it is expected that each attempt to install the malicious update will always fail unless the CPU is successfully glitched.

6.4.2 Algorithm Analysis

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, which prevents key reuse attacks. RSA public modulus and exponent are provided in the update file, but before being used their hash is checked against an hardcoded value in the CPU to prevent an attacker from re-signing the microcode.

We identify the following possible injection points:

1. **RSA public modulus check**. To ensure the RSA modulus value 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.

2. **Signature Check**. The hash of the decrypted update content is checked against the signature in the file. If an

[14] Database [1] table _8Gb_405526d_ucode_loop_add_4

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

attacker were to skip the signature check through fault injection, they could load a modified microcode.

7 Conclusions

Our work demonstrates that x86 instructions and microcode can be faulted using voltage glitching on Intel Goldmont CPUs. We believe that the software infrastructure we developed using coreboot provides an environment for highly reliabile experiments in a controlled, low-noise setting. It can serve as a solid foundation for further fault injection research, particularly with more advanced glitch delivery mechanisms.

Experiments were performed across multiple *UP Squared* boards with the same CPU model, but different RAM configurations to ensure the reproducibility of our findings. Consistent results were observed on all samples, suggesting the identified fault modes are not device-specific anomalies but indicative of a repeatable phenomenon. We show evidence of novel types of architectural faults on x86, that we identify as data corruptions in the register file, and instructions skip. We also find μ-instructions skip and bit flips in custom microprograms.

8 Limitations and Future Work

Improving Glitch Rate. A current limitation of our setup is the inherent low precision of PMIC-based fault injection: while we were able to inject faults, the low slew rate of the voltage regulator makes the glitches less localized and harder to control. Exploring other fault injection methods, such as EM or laser, could allow more targeted attacks on microcode.
Applicability Beyond Red-Unlocked CPUs. Our current results are limited to Red Unlocked Goldmont CPUs. While this platform gives unparalleled access to microarchitectural features, it is not necessarily representative of other CPUs. To generalize our findings, future work can test our experimental setup and software framework on other Intel CPUs, especially those without Red Unlock support. Of course, in such cases, testing custom microprograms will no longer be possible, but it is possible to test architectural programs and compare with our result.
Microcode Updates. In future work, we plan to expand our testing to the microcode update routine and investigate the feasibility of applying faults during the update phase.

9 Related Work

Voltage fault injection in x86 CPUs via the onboard PMIC was pioneered by Plundervolt [28] and VoltJockey [30], 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. [12], the microcode update reverse engineering effort by Borrello et al. [5], and the lib-micro microcode patching library by Krog and Skovsende [25]. To build a usable microcode glitch setup, we embed lib-micro [25] in coreboot, modifying it to support 32-bit registers. We believe this design 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.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by NWO through project "INTERSECT" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] Experiments result database, 2025. URL https://github.com/ceres-c/MicroSpark/blob/master/notebooks/glitch2.db_PLACEHOLDER.

[2] AAEON. Up squared user's manual, 2021. URL https://newdata.aaeon.com.tw/DOWNLOAD/MANUAL/UP%20Squared%20(UPS-APL)%20Manual%206th%20Ed.pdf.

[3] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, 1997. URL https://api.semanticscholar.org/CorpusID:12376527.

[4] Dan Boneh, Richard Demillo, and Richard Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14, 07 1999. doi: 10.1007/s001450010016.

[5] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. Customprocessingunit: Reverse engineering and customization of intel microcode. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 285–297, 2023. doi: 10.1109/SPW59333.2023.00031.

[6] Peter Bosch. Intel management engine deep dive. Slideshow presented at 36C3, 2019.

[7] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. CCS '21. Association for Computing Machinery, 2021. ISBN 9781450384544. doi: 10.1145/3460120.3484779. URL https://doi.org/10.1145/3460120.3484779.

[8] Zitai Chen and David Oswald. 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. doi: 10.46586/tches.v2023.i2.1-23. URL https://tches.iacr.org/index.php/TCHES/article/view/10275.

[9] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 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, August 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[10] Mark Ermolov. Tweet on abbreviations relating to power management of intel chips, Sep 2019. URL https://twitter.com/_markel___/status/1175851055676043267.

[11] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. Slideshow presented at Black Hat Europe 2017, 2017.

[12] Mark Ermolov, Dmitry Sklyarov, and Maxim Goryachy. 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. doi: 10.1007/s11416-022-00438-x.

[13] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. Intel txe poc, 2018. URL https://github.com/ptresearch/IntelTXE-PoC.

[14] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. glm-ucode, 2020. URL https://github.com/chip-red-pill/glm-ucode.

[15] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. Chip red pill. Slideshow presented at OffensiveCon22, 2022.

[16] Maxim Goryachy, Dmitry Sklyarov, and Mark Ermolov. ucodedisasm, 2022. URL https://github.com/chip-red-pill/uCodeDisasm.

[17] Ben Hawkes. Notes on intel microcode updates, Mar 2013. URL https://web.archive.org/web/20230217224318/http://inertiawar.com/microcode/.

[18] Yeoh Eng Hong, Lim Seong Leong, Wong Yik Choong, and Mahmud Adnan. An overview of advanced failure analysis techniques for pentium and pentium pro microprocessors. volume 2, 1998. URL https://www.intel.com/content/dam/www/public/us/en/documents/research/1998-vol02-iss-2-intel-technology-journal.pdf#page=2.

[19] Texas Instruments. Tps65094 pmic for intel apollo lake platform, 2019. URL https://www.ti.com/lit/ds/symlink/tps65094.pdf.

[20] Intel. Intel atom processor c3000 product family, 2018. URL https://www.mouser.com/datasheet/2/612/c3000-family-datasheet-1623704.pdf.

[21] Intel. Intel debug technology, 2021. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/intel-debug-technology.html.

[22] Intel. Intel 64 and ia-32 architectures optimization reference manual, 2023. URL https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[23] Andrew Jenner. 8086 microcode disassembled, 2020. URL https://www.reenigne.org/blog/8086-microcode-disassembled/.

[24] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020. ISBN 978-1-939133-17-5.

[25] Alexander Krog and Alexander Skovsende. lib-micro documentation, 2023. URL https://libmicro.dev/.

[26] Alexander Krog and Alexander Skovsende. Backdoor in the core, 2023. URL https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Alexander%20Dalsgaard%20Krog%20Alexander%20Skovsende%20-%20Backdoor%20in%20the%20Core%20-%20Altering%20the%20Intel%20x86%20Instruction%20Set%20at%20Runtime.pdf.

[27] Raspberry Pi Ltd. Rp2040 datasheet, 2021. URL https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf.

[28] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482, 2020. doi: 10.1109/SP40000.2020.00057.

[29] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptol. ePrint Arch.*, 2016:810, 2016. URL https://api.semanticscholar.org/CorpusID:8502986.

[30] Pengfei Qui, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Abusing the processor voltage to break arm trustzone. *GetMobile: Mobile Comp. and Comm.*, 24(2), 2020. doi: 10.1145/3427384.3427394.

[31] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 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)*, pages 624–637, 2018. doi: 10.1109/ISCA.2018.00058.

A Experiments Code

Assembly code is in AT&T syntax, as used in coreboot, and x86 microcode uses lib-micro [25] C macros syntax.

```
{
    SUB_DSZ64_DRR(TMP0, RAX, RBX),
    UJMPCC_DIRECT_NOTTAKEN_CONDNZ_RI(TMP0, (
        PATCH_ADDR + 0x04)),
    ADD_DSZ64_DRI(R64SRC, R64SRC, 0),
    ( SEQ_UEND0(2) | SEQ_NEXT | SEQ_SYNCFULL(1) )
}, {
    ADD_DSZ64_DRI(R64SRC, R64SRC, 1), /* UJMP
        destination */
    NOP,
    NOP,
    ( SEQ_UEND0(0) | SEQ_NEXT | SEQ_NOSYNC )
},
```

Listing 6: µ-instructions `CMP`, add 1 on non-equal, speculatively equal

```
{
    SUB_DSZ64_DRR(TMP0, RAX, RBX),
    UJMPCC_DIRECT_NOTTAKEN_CONDZ_RI(TMP0, (
        PATCH_ADDR + 0x04)),
    ADD_DSZ64_DRI(R64SRC, R64SRC, 1),
    ( SEQ_UEND0(2) | SEQ_NEXT | SEQ_SYNCFULL(1) )
}, {
    ADD_DSZ64_DRI(R64SRC, R64SRC, 0), /* UJMP
        destination */
    NOP,
    NOP,
    ( SEQ_UEND0(0) | SEQ_NEXT | SEQ_NOSYNC )
},
```

Listing 7: µ-instructions `CMP`, add 1 on non-equal, speculatively not equal

```
{
    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 8: µ-instructions skip target

```
{
    ZEROEXT_DSZ64_DI(TMP0, 0x000D),
    CONCAT_DSZ16_DRI(TMP0, TMP0, 0xFFFF),
    NOP,
    NOP_SEQWORD,
}, {
    SUB_DSZ64_DIR(TMP0, 1, TMP0),
    ADD_DSZ64_DRI(R64SRC, R64SRC, 1),
    ADD_DSZ64_DRI(R64SRC, R64SRC, 1),
    NOP_SEQWORD,
}, {
    ADD_DSZ64_DRI(R64SRC, R64SRC, 1),
    UJMPCC_DIRECT_NOTTAKEN_CONDNZ_RI(TMP0, (
        PATCH_ADDR + 0x04)),
    NOP,
    ( SEQ_UEND0(2) | SEQ_NEXT | SEQ_SYNCFULL(1) ),
}
```

Listing 9: `ADD` loop target

```
{
    ZEROEXT_DSZ64_DI(TMP0, 0x0007),
    CONCAT_DSZ16_DRI(TMP0, TMP0, 0xFFFF),
    ZEROEXT_DSZ64_DI(TMP1, 0x5555),
    NOP_SEQWORD,
}, {
    ZEROEXT_DSZ64_DI(R64SRC, 0x0000),
    ZEROEXT_DSZ64_DI(TMP2, 0x0000),
    WRITEURAM_RI(TMP1, 0x48),
    NOP_SEQWORD,
}, {
    READURAM_DI(TMP2, 0x48),
    SUB_DSZ16_DRR(TMP3, TMP2, TMP1),
    UJMPCC_DIRECT_NOTTAKEN_CONDNZ_RI(TMP3, (
        PATCH_ADDR + 0x10)),
    ( SEQ_NOP | SEQ_NEXT | SEQ_SYNCFULL(2) )
}, {
    SUB_DSZ64_DIR(TMP0, 1, TMP0),
    UJMPCC_DIRECT_NOTTAKEN_CONDNZ_RI(TMP0, (
        PATCH_ADDR + 0x08)),
    NOP,
    ( SEQ_UEND0(2) | SEQ_NEXT | SEQ_SYNCFULL(1) )
}, {
    ZEROEXT_DSZ64_DI(R64SRC, 0x0001),
    NOP,
    NOP,
    END_SEQWORD
}
```

Listing 10: URAM target

B Setup picture

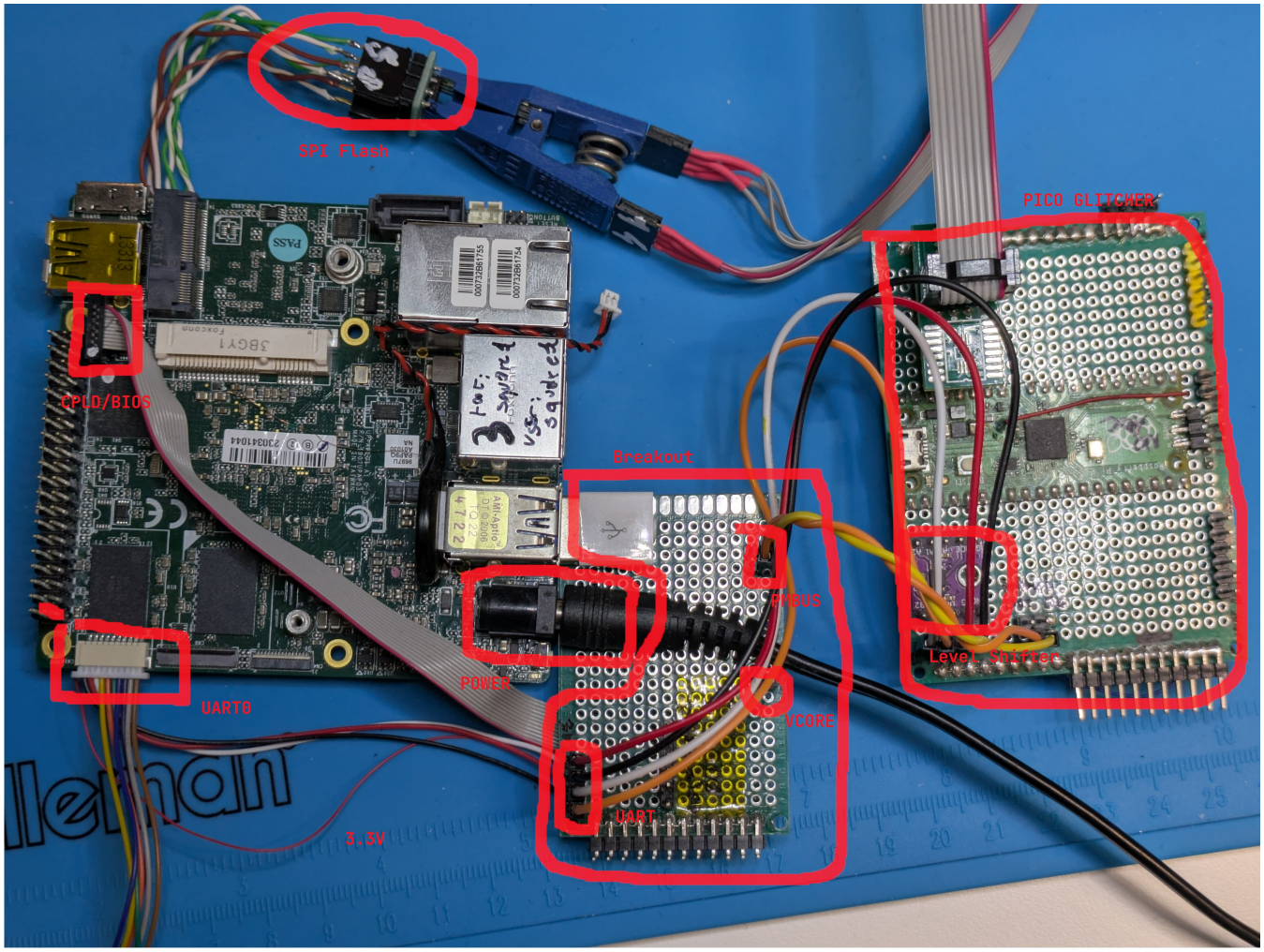

Figure 9: The hardware setup with all the components highlighted