VoltPillager:

Chen, Zitai; Vasilakis, Georgios; Murdock, Kit; Dean, Edward; Oswald, David; Garcia, Flavio

VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface

Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia

School of Computer Science, University of Birmingham, UK

Abstract

Hardware-based fault injection attacks such as voltage and clock glitching have been thoroughly studied on embedded devices. Typical targets for such attacks include smartcards and low-power microcontrollers used in IoT devices. This paper presents the first hardware-based voltage glitching attack against a fully-fledged Intel CPU. The transition to complex CPUs is not trivial due to several factors, including: a complex operating system, large power consumption, multi-threading, and high clock speeds. To this end, we have built VoltPillager, a low-cost tool for injecting messages on the Serial Voltage Identification bus between the CPU and the voltage regulator on the motherboard. This allows us to precisely control the CPU core voltage. We leverage this powerful tool to mount fault-injection attacks that breach confidentiality and integrity of Intel SGX enclaves. We present proof-of-concept key-recovery attacks against cryptographic algorithms running inside SGX. We demonstrate that VoltPillager attacks are more powerful than recent software-only undervolting attacks against SGX (CVE-2019-11157) because they work on fully patched systems with all countermeasures against software undervolting enabled. Additionally, we are able to fault security-critical operations by delaying memory writes. Mitigation of VoltPillager is not straightforward and may require a rethink of the SGX adversarial model where a cloud provider is untrusted and has physical access to the hardware.

1 Introduction

Modern computing platforms allow the operating system to self-regulate the processor's core frequency and voltage in order to manage heat and power consumption. Several authors [37, 24, 40] have shown that an adversary can abuse this feature to inject bit flips into computations, including those inside an Intel Software Guard Extensions (SGX) enclave (cf. CVE-2019-11157). Using the software-exposed interface Model Specific Register (MSR) `0x150`, attacks against Intel SGX were mounted by undervolting from (untrusted) software running with root privileges. Intel have addressed this vulnerability by providing features to disable software undervolting through this MSR. Because SGX was compromised, Intel have initiated Trusted Computing Base (TCB) recovery and modified remote attestation to verify that software-based undervolting is disabled. This requires Microcode (μCode) and BIOS updates.

Hardware fault injection considers a different adversarial model where the adversary has physical access to the device under attack. When targeting an SGX enclave running on a fully patched system (with the latest μCode and BIOS updates), software-based fault attacks have been fully mitigated and that is where hardware-based attacks become relevant. Fault attacks induce a computation fault in the target processor, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability or any code execution privileges. Voltage fault injection (aka, glitching) in particular has the advantage of being very powerful whilst not requiring expensive lab equipment.

1.1 Our Contribution

In this paper, we analyse the dynamic voltage scaling features of x86 systems at the hardware level. We found that a three-wire bus, Serial Voltage Identification (SVID), is used to send the currently required voltage to an external Voltage Regulator (VR) chip on the motherboard. The VR then adjusts the voltage supplied to the CPU. We reverse-engineered the

communication protocol of SVID and developed a small microcontroller-based board that can be connected to the SVID bus. As there is no cryptographic authentication of the SVID packets, we were able to inject our own commands to control the CPU voltage. With this, we reproduced Plundervolt's [37] open-source Proof-of-Concept (PoC) attacks, including against code running inside an SGX enclave. Beyond that, we also found (and document) faults not previously observed. These faults affect elementary operations such as memory accesses. Because the software interface MSR `0x150` is not used, Intel's countermeasures do not prevent this attack. The main contributions of this paper are:

- We showcase the (to our knowledge) first hardware-based attack that directly breaches SGX's integrity guarantees. We demonstrate its practicality with end-to-end secret-key recovery attacks against mbed TLS and the unmodified `file-encryptor` sample enclave from Microsoft Open Enclave.
- We show that Intel's countermeasures for CVE-2019-11157 do not prevent fault-injection attacks from adversaries with physical access. This challenges the widely accepted belief that SGX can protect enclave integrity against a malicious cloud provider (cf. e.g., [2, 5, 27, 8]).
- We demonstrate novel fault effects discovered through hardware-based undervolting, in particular by briefly delaying memory writes.
- We present VoltPillager, an open-source hardware device to inject SVID packets. VoltPillager is based on a low-cost, widely available microcontroller board, the Teensy 4.0, and can be built for approximately $ 30. We also document the internal power management interfaces on modern motherboards, SVID and System Management Bus (SMBus).

1.2 Responsible Disclosure

We reported this issue to Intel on 13 March 2020. Intel evaluated our report and concluded on 5 May that "... opening the case and tampering of internal hardware to compromise SGX is out of scope for SGX threat model. Patches for CVE-2019-11157 (Plundervolt) were not designed to protect against hardware-based attacks as per the threat model", and, therefore, they will not further address the issue. Intel have not requested an embargo for the vulnerabilities described in this paper. We discuss the implications of Intel's response in relation to the widely adopted threat model of SGX in Section 1.4.

1.3 Related Work

Since their introduction by Boneh et al. [6], fault-injection attacks with physical access have been widely investigated in the context of embedded devices. Those attacks are based on the fact that the execution semantics of an IC can change when it is operated outside the specified operating conditions. Examples of fault injection include: over and undervolting ("voltage glitching"), overclocking, exposure to high or low temperature, or laser light [3, 53].

The fault injection threat model changed with the discovery of *software-based* attacks. In 2014, Kim et al. reported the Rowhammer effect: bits could be flipped in DRAM by accessing neighbouring rows but not the actual target location [26]. Several authors [43, 17, 28] have since discovered applications, variations, and improvements of the original attack, including the successful bypass of countermeasures in recent DDR4 DRAM chips [14]. While Rowhammer can be performed from unprivileged software, another class of software-based fault injection attacks require the adversary to have root privileges. These generally target a Trusted Execution Environment (TEE) such as ARM TrustZone or Intel SGX, which should defend the code running inside the TEE even against a privileged adversary.

CLKSCREW [50] was the first attack of this type: it exploited the software-controlled overclocking features on the ARM processor of a Nexus 6 smartphone. CLKSCREW was able to extract cryptographic secrets from TrustZone and to bypass signature checks, leading to code execution inside TrustZone. Qiu et al. later found a similar attack, VoltJockey, against TrustZone, this time controlling the CPU's core *voltage* from privileged software [41]. This line of work continued with voltage fault injection attacks on Intel SGX enclaves [37, 24, 40], which use the software-exposed MSR `0x150` to undervolt during enclave execution and thus trigger bit flips in certain operations, e.g., multiplications, vector instructions, and cryptographic operations.

Hardware-based attacks against TEEs have, so far, received less attention. Cui et al. showed that electro-magnetic fault injection can be used to bypass the TrustZone-based secure boot process of a Broadcom ARM CPU [9]. Similarly, Roth et al. presented fault injection attacks with physical access to ARMv8-M processors, among others breaking the TrustZone-M security on certain CPUs [46].

Lee et al. presented a side-channel attack [29] on SGX by physically connecting to, and eavesdropping on, the DRAM memory bus. They showed that by observing the pattern of the (encrypted) memory ac-

cesses, they can recover secret information from a range of example enclaves. Notably, their attack requires specialized and expensive test equipment (e.g., $ 170,000 for a JLA320A signal analyzer).

1.4 Attacker Model

We are using the widely adopted SGX adversary model with physical access to the target CPU and full control over all software running outside the enclave, including BIOS and operating system. Crucially, our attacks do not require expensive lab equipment (e.g., for invasive attacks on the CPU die), but can be mounted with an inexpensive microcontroller board and only require board-level access as opposed to e.g., chip decapsulation. It is sufficient that the adversary can connect two wires to the SVID bus on the motherboard.

In the research community, SGX is widely assumed to provide protection against such an adversary, e.g., in the form of an untrusted cloud provider with physical access to the server hardware. A substantial amount of research explicitly relies on SGX protecting integrity and confidentiality even for a malicious cloud operator, cf. e.g., [2, 5, 27, 8]. Accordingly, cloud providers with SGX support, such as Microsoft Azure, state that SGX "safeguard[s] data from malicious and insider threats while it's in use" [36, 34]. Fortanix, the developers of widely used runtime software for SGX, similarly claim that "Intel SGX allows you to run applications on untrusted infrastructure (for example public cloud) without having to trust the infrastructure provider with access to your applications." [13]. Similarly, the Enarx project considers the enclave host as untrusted in their threat model [11]. Finally, SGX was also originally designed for client-side applications such as Digital Rights Management (DRM) (e.g., in early versions of the Netflix 4K client), user authentication, and fingerprint matching [47].

1.5 Experimental Setup

For the experiments in this paper, we mainly used three different systems, with 7th and 9th generation Intel CPUs and SGX support. We upgraded the BIOS and μCode to the most recent available version. The systems are detailed in Table 1 and all use SVID as the main interface for controlling the supply voltage. We initially used a fourth motherboard, a Gigabyte Z170X Gaming 3 with an i3-7100, but this was damaged due to inadvertently short-circuiting the SVID lines, and is therefore not listed in Table 1. However, as we used this motherboard during SVID reverse engineering, we occasionally refer to it as i3-7100-GZ170 in Section 4.

We used 64-bit Ubuntu 18.04.3 LTS as our operating system with stock Linux 5.0.0-23-generic kernel, Intel SGX driver V2.6 and Intel SGX-SDK V2.8. We publicly release all source code at `https://github.com/zt-chen/voltpillager`.

1.6 Outline

The remainder of this paper is structured as follows: first, in Section 2, we discuss Intel's mitigation for CVE-2019-11157. In Section 3, we then describe the two main interfaces for controlling CPU voltage on modern systems. We introduce our open-source tool VoltPillager for injection of SVID packets in Section 4. The use of VoltPillager for hardware-based undervolting attacks on SGX is detailed in Section 5 and Section 6. We discuss possible countermeasures and the implications of our findings in Section 7, before concluding in Section 8.

2 Intel's Mitigation for Software-based Undervolting Attacks on SGX Enclaves

The mitigation deployed by Intel to address CVE-2019-11157 effectively disables software access to the voltage control features of the system. It consists of two main parts: (*i*) a BIOS update supplied by the BIOS vendor to disable the undervolting functionality at boot, and (*ii*) a μCode update to interact with the updated BIOS and include the software undervolting status (enabled or disabled) in SGX's remote attestation functionality.

The exact implementation of the BIOS update differs by vendor. On our test systems i3-7100-AZ170 and i3-9100-MZ370, even the most recent BIOS still allowed undervolting. In contrast, the latest update of our Intel-manufactured i3-7100U-NUC added a new BIOS option ("Real-Time Performance Tuning"), which, when disabled, removes the ability to undervolt via MSR `0x150`. We ran all experiments for the i3-7100U-NUC with software undervolting disabled via this BIOS option, and confirmed, practically, that writes to MSR `0x150` no longer cause voltage changes. We also verified that the OC mailbox interface (which is used for undervolting) is disabled on i3-7100U-NUC using `intel-oc-mbox` [12].

Other manufacturers, such as Dell [15], removed software-controlled undervolting with recent updates in response to CVE-2019-11157, without any configuration options available at the BIOS level. The μCode update deployed by Intel includes the

Device	Motherboard	BIOS version	CPU	µCode	VR IC	VR on SMBus
i3-7100-AZ170	ASRock Z170 E4	P7.50	i3-7100	0xca	ISL95856	`0x40`
i3-9100-MZ370	MSI Z370-A Pro	E7B48IMS.2B0	i3-9100	0xca	UP9508	`0x45`
i3-7100U-NUC	NUC7i3BNH	0082.2020.0505	i3-7100U	0xca	ISL96853	✗

Table 1: CPUs and motherboards used for experiments in this paper, including BIOS and µCode version, and the used VR IC. We also indicate if the VR is connected to the SMBus (and at which address) in addition to SVID. A 4$^{\text{th}}$ system, i3-7100-GZ170, left out here as it was only used in initial reverse engineering.

status of the software undervolting interface in the remote attestation process, similar to other functionality such as hyperthreading and the internal graphics card. Specifically, the SGX attestation service returns a `CONFIGURATION_NEEDED` response if software undervolting (or any of the other problematic features such as hyperthreading) is enabled [20].

3 Power Management Interfaces

In modern computers, there are usually one or more VRs connected to the CPU on the motherboard. They are used for managing the performance and power consumption of the system by changing the core voltage (and other voltages) supplied to the CPU. When the CPU runs at lower frequencies or is in idle mode, it sends commands to the VR to reduce the voltage. Vice versa, when the CPU operates under heavy load and/or at high frequency, it requests the VR to increase the voltage. We found two main interfaces to the VR that can be used for changing the CPU voltage and hence to conduct undervolting attacks: the SVID interface and the SMBus interface (more specifically Power Management Bus (PMBus) in this context). The overall architecture of the voltage supply on an x86 system is shown in Figure 1. We now introduce those VR interfaces in detail and discuss their use for undervolting attacks.

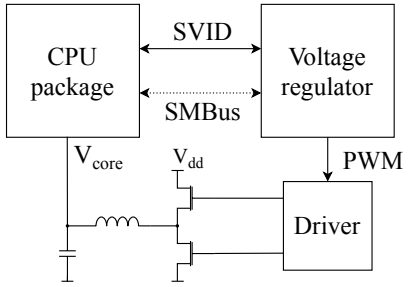

Figure 1: Architecture of voltage supply on x86 systems. The CPU has an SVID and (optional, dotted) SMBus connection to the VR. The VR drives a Pulse Width Modulation (PWM) output to generate the core voltage from the main supply voltage.

3.1 Serial Voltage Identification

SVID is the interface used by the CPU to send the currently required voltage (and other related data) to an external VR IC. To the best of our knowledge, Intel does not provide any detailed documentation for this interface. From the CPU documentation [21], we found that SVID uses the three pins `VCLK`, `VDIO`, and `ALERT#`. The first two pins are used for a bi-directional serial interface similar to common serial protocols like Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI). The `ALERT#` line is asserted by the VR when a voltage change has been completed [18]. The SVID bus uses voltage levels of 0 V (low) and 1 V (high) and is clocked at 25 MHz. Both clock (`VCLK`) and data (`VDIO`) lines are realised as open-drain outputs: by default, the lines are held high by pullup resistors to 1 V, and actively driven low by the CPU or VR when they exchange data. Note that this allows multiple devices to be connected to SVID, we will later use this to connect our own device for command injection.

Locating VR ICs The identification of the VRs on the motherboard is the first step required for further analysis of a particular system. Some vendors provide schematic diagrams of their boards, which greatly simplifies the process. Unfortunately, most vendors do not publish such detailed documentation of their hardware. However, in our experience, VRs are commonly placed in close proximity to the CPU and to large switching transistors and inductors, making them easy to identify by visual inspection and oscilloscope probing. Additionally, the SVID signals are commonly connected to small resistors and/or available on test pads, simplifying connecting devices for analysis and packet injection. Figure 2 shows the ISL96853 VR on i3-7100U-NUC. The large transistors generating the actual core voltage are visible on the right, while the relevant SVID

pins are located around the top-left corner to the VR IC.

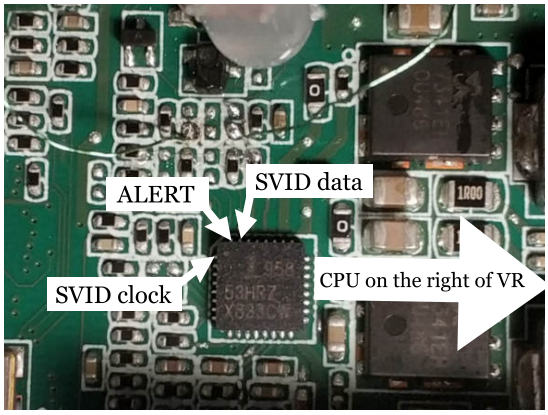

Figure 2: ISL96853 VR on the motherboard of i3-7100U-NUC with relevant SVID pins annotated.

Protocol Reverse-Engineering We identified the clock and data lines on the motherboard used for our experiments using a Rigol DS1074Z oscilloscope [45], and connected a DSlogic logic analyser [10] to the SVID bus.

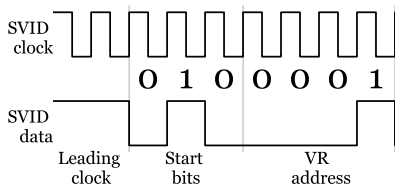

Figure 3: SVID data and clock lines during the first cycles of a command sent to the VR.

By observing the bus and setting known values for the voltage and with the help of a screenshot of a SVID protocol analyzer [54], we reverse-engineered the relevant commands used for configuring the voltage output by the voltage regulator. Note that this is only a one-time process required to understand the working of the SVID protocol and is not necessary for subsequent packet injection attacks. Figure 3 shows an example of the clock and data signals when transmitting a bit sequence over SVID. Based on this, we analysed the command that configures the voltage and its response, and document their format in Figure 4 and Figure 5. Further details on SVID can be found in Appendix C.

Bus Activity and Command Injection We found that the SVID bus on our test motherboards

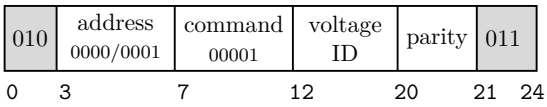

Figure 4: Format of 24-bit SVID command from CPU to VR to set the current core voltage. Gray background indicates fixed bits.

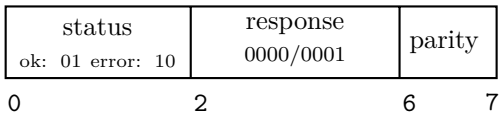

Figure 5: 7-bit response from VR to CPU to the set voltage command from Figure 4.

was active even when a fixed core voltage was configured in the BIOS. Regular commands from the CPU to the VR are still sent, although at a reduced rate compared to normal, dynamic voltage control.

Understanding bus activity is crucial for SVID. Our experiments (as described in Section 4) show that the CPU freezes if there are multiple subsequent collisions between an injected and a "real" SVID packet. From our experiments, we concluded that the CPU stops after less than a second of failed transmission attempts. Hence, it is crucial to limit SVID command injection to a short burst.

Conversely, for reverse-engineering and analysis of VR behaviour, the ability to temporarily "remove" the CPU from the bus was useful: once the CPU has frozen, we could send our own commands to the VR without interference from the CPU. This allowed us to verify our understanding of the SVID commands during the initial analysis of the protocol, and to test our tool, VoltPillager.

3.2 System Management Bus and Power Management Bus

Apart from SVID, some VRs support another voltage control interface named SMBus (also referred to as PMBus in this case) [49, 48]. Such VRs supporting SMBus are mainly found on server and gaming motherboards. Similar to SVID, SMBus is a serial two-wire interface consisting of a clock line (maximum frequency 1 MHz) and a data line. Multiple devices are connected to the SMBus, with the motherboard's Southbridge acting as bus master. Each device on the bus, including the VR if connected, is assigned a unique address. Typically, these addresses are assigned by the motherboard vendor and not publicly documented. Hence, determining the address of the VR on the SMBus requires probing

the bus and/or analysis of manufacturer tuning software.

Among our test systems, the i3-7100-AZ170 and i3-9100-MZ370 had the VR chip connected to the SMBus. We found that on both systems, when setting a fixed core voltage through the BIOS, this configuration takes place via the SMBus. Subsequently, we determined the VR address (cf. Table 1) and the respective commands to adjust the CPU voltage. We also noted that any voltage change made through SMBus overrides any subsequent settings made through SVID.

While it is technically possible to inject packets into the SMBus, we opted for using SVID as our main attack interface for several reasons: First, with the higher clock frequency of SVID, voltage changes can be made more quickly, allowing for more accurate fault injection. Second, we found that SVID commands were the same across our test systems, while SMBus commands varied between VRs. Finally, SVID is used by all modern motherboards and CPUs, while SMBus is only present on certain motherboards and VRs.

4 VoltPillager for SVID Command Injection

In this section, we present VoltPillager, our custom device for SVID command injection based on a Teensy 4.0 microcontroller development board [38]. In contrast to other widely used interfaces like SPI, we could not find a Microcontroller (µC) with a dedicated hardware peripheral for SVID. We initially attempted to implement the protocol in software using the General Purpose I/O (GPIO) pins, but we found that the relatively high clock frequency of 25 MHz of SVID makes this difficult in practise even on relatively high-end µCs like the ARM Cortex-M7 on the Teensy 4.0, which runs at 600 MHz.

However, as SVID is similar to SPI in so-called mode two (clock idles at high voltage and data line stable and sampled on the falling edge of the clock), we were able to use the SPI hardware peripheral of the Teensy 4.0 for packet injection. Note that an additional required feature is the support for large SPI frames, which allows for the transmission of a complete SVID transaction. With our implementation, a complete transmission of one SVID packet takes 96 clock cycles (3.84 µs), including 40 clock cycles before sending the data, the 24-bit data frame, and 32 clock cycles for receiving the response. To adapt the output voltage levels of 3.3 V of the Teensy 4.0 to the 1 V levels of SVID, we used two open-drain drivers [51] for the data and clock lines.

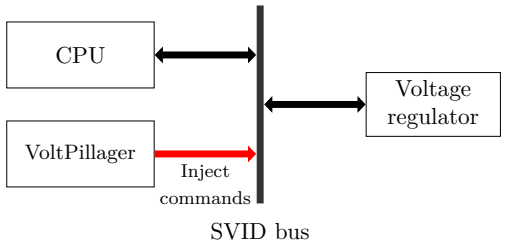

Figure 6: Command injection into SVID with the VoltPillager connected in parallel to the bus.

As shown in Figure 7, we also made sure to keep the wires between the actual bus lines and the driver ICs short to minimise additional inductive and capacitive load on the bus. Note that VoltPillager can be connected in parallel to the bus without—when inactive—affecting the normal SVID traffic from CPU to VR.

We also carried out initial experiments with a Field Programmable Gate Array (FPGA) board to obtain a man-in-the-middle position by splitting the SVID bus: in this situation, the attacker simulates normal behaviour of the VR to the CPU, while injecting commands that are only visible to the VR. However, due to the strict timing constraints of SVID and the necessary level conversions, we did not further investigate this approach for the present paper.

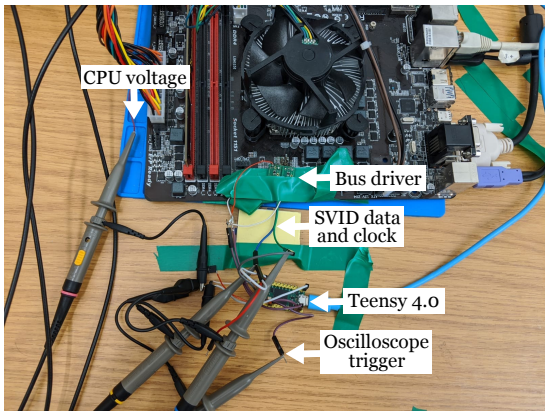

Figure 7: Hardware setup (i3-7100-GZ170) for analysis and packet injection into the SVID bus. Oscilloscope probes are attached to CPU core voltage and SVID lines, while Teensy injects SVID packets.

4.1 Implementation of VoltPillager

VoltPillager has two main components, the firmware of the Teensy 4.0 and PC-side software which controls the device. We developed the Teensy 4.0

firmware using the open source Arduino IDE [1] and the official Teensyduino [39] library, which provides the basic SPI functionality. We configured this to match the structure of an SVID message. The controlling PC communicates with the VoltPillager through USB using the protocol shown in Figure 8. Firstly, the software on the controlling computer specifies the undervolting parameters, including the required voltage, number of SVID packets being sent, and additional parameters as detailed in Table 2. It then arms the glitch.

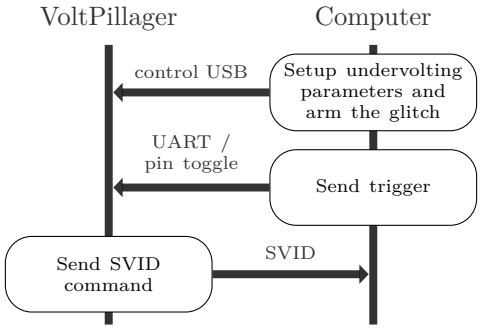

Figure 8: Protocol of VoltPillager for configuring and initiating undervolting from a controlling PC.

Trigger After that, VoltPillager is armed and waits for an active-low *trigger* input on a specific pin. When the trigger signal is asserted (*i.e.*, the trigger pin pulled low), VoltPillager sends the prepared SVID packets according to the configuration to the target VR, *i.e.*, initiates a hardware-based undervolting attack. The trigger is generated by the controlling computer from untrusted code directly before the target code executes. In our experiments, the control PC typically is the machine that also runs the target SGX enclave, although this is not a strict requirement.

To provide a precisely-timed trigger signal, we utilise the fact that some motherboards, such as the ones used in i3-7100-AZ170 and i3-9100-MZ370, offer a legacy onboard RS232 Universal Asynchronous Receiver Transmitter (UART) port, including the so-called `DTR` signal, which can be controlled from software through an `ioctl()` system call. Because this interface is controlled by the motherboard's Southbridge through the "super I/O" chip, it is more reliable in timing than sending a trigger command over the control USB interface.

However, for systems without hardware UART, e.g., i3-7100U-NUC, we additionally implemented a less timing-stable trigger over USB. After the undervolting has been triggered, the control program evaluates whether it has succeeded (*i.e.*, that an unexpected result occurred in the target code), and outputs the fault results or repeats the process until a fault has been found.

Adjustment of Undervolting Parameters The parameters from Table 2, defining the undervolting glitch shape, are illustrated in Figure 9. Note that these parameters affect the stability of the system-under-attack and hence, can be adjusted to minimise system crashes. Note that the VR changes

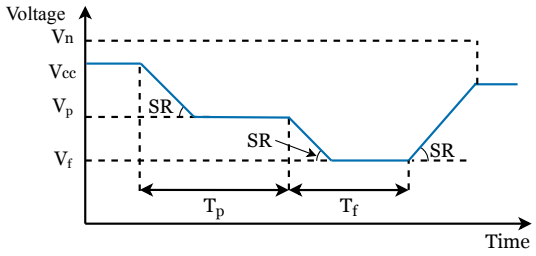

Figure 9: Undervolting waveform with the parameters described in Table 2. V_n can be $\leq V_{cc}$.

the core voltage CPU with a finite slew rate SR, typically 20 mV/µs, however on i3-7100U-NUC we observed a higher slew rate (approximately 40 mV/µs). The limited slew rate can reduce the timing precision of undervolting, as it adds delay between the reception of the injected SVID command from the VR and the physical change of the core voltage to the fault voltage V_f. The preparation voltage V_p reduces the time to reach the target voltage V_f before the actual attack: the system is still stable at V_p, and the adjustment to V_f at finite SR is quicker compared to the higher default voltage V_{cc}. The reset voltage V_n is set after the fault. It can take any values and is used to stabilize the system. Figure 10 shows an oscilloscope capture of an actual undervolting injected by VoltPillager on i3-9100-MZ370. In this case, we set $V_p = V_n = V_{cc} = 1.050\,V$, $V_f = 0.810\,V$, $T_p = 10\,\mu s$ and $T_f = 32\,\mu s$.

System Stability One of the major factors affecting the system stability is the amount of undervolting. When the voltage is too low, the CPU will "freeze" or crash, while a too high voltage will not yield successful faults. Figure 11 shows the undervolting for the fault to occur and the value at which the system crashes for different CPU frequencies. We observed crashes before a fault happens below

Parameter	Symbol	Description
Delay after trigger	T_d	The time between assertion of trigger and start of first undervolting.
Number of glitches	N	Number of times to repeat the undervolting.
Preparation voltage	V_p	Voltage before undervolting. Typically $\leq V_n$, but system stable at V_p.
Preparation width	T_p	Time of voltage slew to V_p plus time for which V_p is held.
Fault voltage	V_f	Voltage used for actual fault injection.
Fault width	T_f	Time of voltage slew to V_p plus time for which V_f is held.
Normal voltage	V_n	Stable operating voltage after V_f.
Slew rate	SR	Rate of voltage change (in mV/µs). Most VRs support a single SR.

Table 2: Parameters defining a specific undervolting experiment with VoltPillager

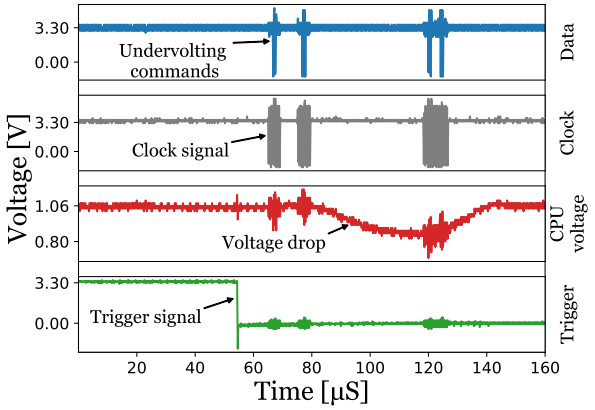

Figure 10: Oscillocope capture of undervolting. Data and clock captured before the voltage level shifters.

1.4 GHz on i3-7100-AZ170 and 2.9 GHz on i3-9100-MZ370. For both systems, there is a gap of ≈ 20 mV (4 VID steps) which can be used for fault injection without affecting system stability.

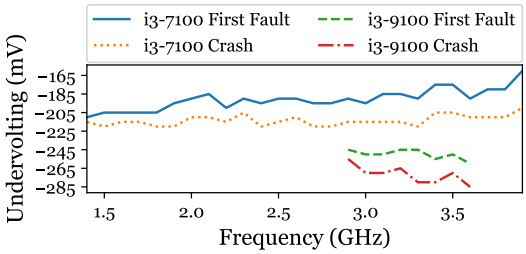

Figure 11: Undervolting for first fault (solid) and crash (dotted) for different frequencies on i3-7100-AZ170 (max. 3.9 GHz) and i3-9100-MZ370 (max. 3.6 GHz), using the PoC from Listing 1 and VoltPillager. 5 repetitions per frequency. $V_n = V_p = V_{cc}$.

5 Fault Injection into SGX Enclaves with Hardware-based Undervolting

In this section, we demonstrate how VoltPillager can be used to inject faults into the CPU even when the software-controlled interface has been disabled. To this end, we first show that we can reproduce faults observed with software-based fault attacks through hardware-based undervolting. We show how this can be used in an end-to-end attack scenario to extract in-enclave secrets. We then describe a novel fault attack based on briefly delayed memory/cache writes.

5.1 Reproducing Plundervolt Proof-of-Concepts

The authors of Plundervolt provide several PoCs on their Github repository [16]. We mainly focused on two of their PoCs, namely faulting integer multiplications (in userspace) and CRT-RSA decryption/signature (running inside an SGX enclave). We made this choice to (*i*) give examples both for SGX and non-SGX code and (*ii*) compare the behaviour of the well-documented faults on `imul` when using hardware-based undervolting (with various parameters shown in Table 2). Additionally, we also successfully reproduced Plundervolt's PoC for AES-NI.

Faulting Multiplications We integrated Plundervolt's `faulting_multiplications` PoC into our experimental setup as shown in Listing 1. This code segment is similar to the one used in Plundervolt: two multiply operations (compiled to `imul`) are executed with the same input in a tight loop and the result of the calculation is compared after each operation. However, before entering the loop, a trigger is generated to start the hardware-based undervolting using VoltPillager.

```
1 TRIGGER_SET // Set trigger
2
3 do {
```

```
  i++;
  correct_a = operand1 * operand2;
  correct_b = operand1 * operand2;
  if (correct_a != correct_b) {
    faulty = 1;
  }
} while (faulty == 0 && i < iterations);

TRIGGER_RST // Reset trigger
// ... fault check omitted ...
```

Listing 1: Simplified C code used for demonstrating hardware-based fault injection into multiplication

Setting first the operands of `imul` to `0xAE0000` and `0x18`, respectively, we obtained the same faulty result (`0xC500000` instead of `0x10500000`) on all our test systems, using the parameters from Table 3.

Cryptographic Operations inside SGX We then adapted Plundervolt's PoC `sgx_crt_rsa` to our setup. This program computes an RSA signature/decryption inside an SGX enclave, using the standard `ipps` library functions. Again, with the parameters shown in Table 3, we successfully obtained faulty signatures and confirmed that these faulty values can be used to factor the RSA modulus and recover the private key using the Lenstra attack [6]. Crucially, this attack also succeeded when the software-undervolting interface was disabled on i3-7100U-NUC through the respective BIOS option.

End-to-end Attack To demonstrate the real-world implications of successful fault injection, we developed an end-to-end attack on the mbed TLS library as used in Microsoft Open Enclave [35]. As a first step, we targeted mbed's `mbedtls_aesni_crypt_ecb()` API function, which internally is accelerated using `aesni`. We confirmed that we can mount a VoltPillager attack to inject a single-byte fault on i3-7100-AZ170 into the 8th round of AES and then perform a Differential Fault Analysis (DFA) to extract the full key [52].

Based on this, we developed an attack on an unmodified enclave, namely the `file-encryptor` sample from Open Enclave[1]. This enclave exposes `ecalls` to encrypt/decrypt files using AES in Cipher Block Chaining (CBC) mode with an in-enclave secret key. The enclave uses the respective mbed TLS function, that internally calls `mbedtls_aesni_crypt_ecb()`. For simplicity, we focus on encryption using a 128-bit key, but note that DFA can be extended to larger key sizes and decryption [31]. A challenge when attacking more complex modes of operations such as CBC is that the DFA requires both the correct and a faulty ciphertext for a given plaintext. However, the `file-encryptor` sample allows us to invoke the encryption `ecall` multiple times without resetting the CBC chaining. Hence, we can first invoke the `ecall` with a chosen plaintext p_0 and obtain:

$$c_0 = \mathrm{AES}_k\left(p_0 \oplus \mathrm{iv}\right)$$

where iv is a fixed value. Then, we can repeatedly invoke the `ecall` again to encrypt the value $p_0 \oplus \mathrm{iv} \oplus c_0$. By construction of CBC, this will repeatedly yield the same ciphertext c_0. We can now inject faults into the computation until we obtain a faulty ciphertext c_0', and then use DFA to recover the in-enclave key from c_0 and c_0' in ≈ 2 min of computation on a 16-core CPU.

We practically implemented this attack against the unmodified `file-encryptor` enclave. We verified on i3-7100U-NUC and i3-7100-AZ170 that we can successfully inject the desired faults and recover the secret key. On i3-7100U-NUC (with Plundervolt patches disabled), we used software-undervolting by -272 mV and invoked the respective `ecall` for a maximum of 100,000 times. On i3-7100-AZ170 we used hardware undervolting with VoltPillager and successfully injected faults while encrypting 10 blocks of data repeatedly in a loop of 700 iterations. After starting the program, it took less than 15 s to obtain a fault using the parameters $V_p = 0.7\,V$, $T_p = 30\,\mu\mathrm{s}$, $V_f = 0.64\,V$, $T_f = 35\,\mu\mathrm{s}$, $V_n = 0.83\,V$, $T_d = 600\,\mu\mathrm{s}$, and $N = 1$. In both cases, triggering was performed outside the enclave from untrusted code.

5.2 Comparison with Software-based Undervolting

In this section, we compare VoltPillager to software-based undervolting through MSR `0x150` and discuss the advantages of our hardware-based approach. First of all, crucially, attacks with VoltPillager are not prevented by the mitigations deployed in response to CVE-2019-11157 (cf. Section 2), and hence can be mounted on systems with the most recent, patched µCode and BIOS. This undermines the common assumption that SGX can protect against an attacker with physical access (e.g., a malicious cloud provider). Furthermore, as discussed in detail in Section 7, mitigating this issue will require, at least in the short term, substantial changes to enclave code to detect fault injections.

Timing Precision The authors of [37] reported that they required more than 100,000 iterations to fault an `imul`. Furthermore, the fault cannot target

[1] `https://github.com/openenclave/openenclave/tree/98b71a/samples/file-encryptor`

		Multiplication				RSA-CRT			
Device	Clock	V_p	V_f	T_f	Temp.	V_{cc}	V_f	T_f	Temp.
i3-7100-AZ170	2 GHz	0.83 V	0.64 V	29 µs	23° C	0.83 V	0.63 V	29 µs	24° C
i3-9100-MZ370	3.4 GHz	1.050 V	0.81 V	83 µs	26° C	1.050 V	0.81 V	43 µs	27° C
i3-7100U-NUC	2 GHz	0.94 V	0.71 V	8 µs	32° C	0.94 V	0.75 V	9 µs	22° C

Table 3: Parameters for successful fault injection into the Plundervolt PoCs for userspace multiplication and SGX RSA-CRT. We also record the clock frequency and the actual CPU temperature when our program starts. All experiments are conducted with $N = 1$. For experiments with i3-7100-AZ170 and i3-9100-MZ370, $V_n = V_p$, for experiments with i3-7100U-NUC, $V_n = 1.05\,V$. See Appendix A for full glitch details.

a particular loop iteration. VoltPillager can overcome both limitations: first, setting $T_d = 0$ µs, $T_p = 26$ µs, $V_p = 0.615\,V$, and $V_f = V_n = V_{cc} = 0.830\,V$, we were able to introduce a fault with as little as $i = 1{,}680$ iterations (for Listing 1) on i3-7100-AZ170.

To evaluate the precision of VoltPillager when targeting a particular loop iteration, we re-ran the multiplication experiment several times with fixed parameters and observed in which loop iteration (*i.e.*, at which value for `i` in Listing 1) the fault occurred. We conducted the following experiment on i3-7100-AZ170 at 2 GHz. Core 1 was isolated with the kernel parameter `isolcpus=1` and used solely for running the target code. We then recorded the faulted loop iterations for the following undervolting parameters, repeating the experiment 60 times: $T_d = 10$ µs, $N = 1$, $V_p = V_n = 0.830\,V$, $T_p = 35$ µs, $V_f = 0.635\,V$ and $T_f = 24$ µs. Out of the 60 runs, we observed a fault in 53. Within these successful faults, the median value for the faulted iteration was 14,634, with 21 faults within iterations 14,562 and 14,729. In fact, 75% of all faults occurred within iteration 14,634 ± 300 as shown in Figure 12.

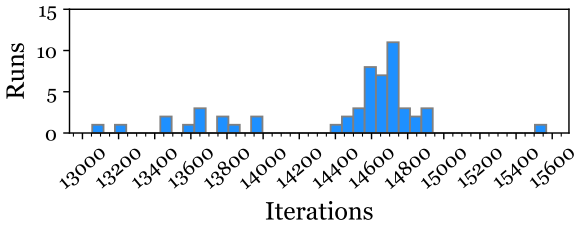

Figure 12: Histogram over affected loop iteration for 53 successful fault injections into multiplication on i3-7100-AZ170 at 2 GHz.

To compare the jitter of the different triggering methods, we used the following setup: we periodically toggle the trigger signal in a fixed-period loop on the controlling PC and let VoltPillager generate a pulse on a pin when it detects the trigger. To ensure consistent timing of the loop, we created a delay with `nanosleep()` and ran the program on a single core at priority 99 and with SCHED_RR policy. Furthermore, we set the following kernel parameters: `intel_pstate=disabled`, `intel_idle.max_cstate=0`, `isolcpus=1`.

Without any jitter, the period of the waveform would be constant. Any jitter added by the controlling PC, the interface, and the Teensy will lead to the period varying. Note that while the delay loop itself might introduce some jitter, this would be present for both triggering methods and hence still allows for relative comparison. Figure 13 shows the distribution (over 100 trigger period measures) for both `DTR` and USB trigger, with the loop period on the controlling Personal Computer (PC) set to 400 µs. The average deviation from the ideal period value, *i.e.*, the jitter, was measured as 4.521 µs (for `DTR`) and 54.442 µs (for USB). Clearly, the `DTR` trigger exhibits substantially less jitter than USB.

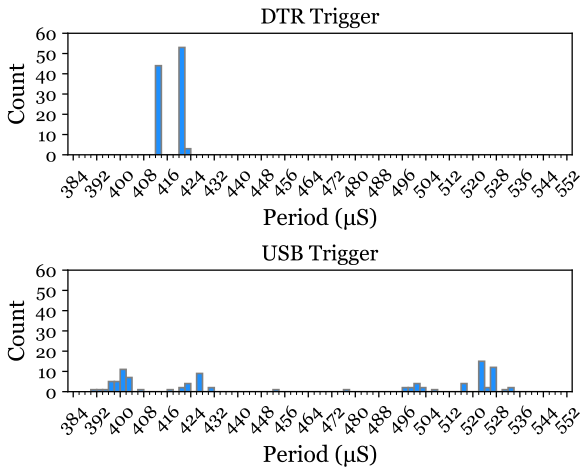

Figure 13: Histogram over 100 trigger period measured on i3-9100-MZ370 (2 GHz) with program running at priority 99 and SCHED_RR policy on core 1.

Glitch Width In software-based fault attacks, a long delay was observed between the MSR write and the actual voltage change. This limits the ability to generate short and potentially "deeper" glitches. In contrast, using VoltPillager, the only limitation on the glitch width is the slew rate SR: given SR, V_p, V_f and V_n, the minimal glitch width T_{min} is:

$$T_{min} = \frac{|V_p - V_f| + |V_n - V_f|}{SR}$$

Assuming $V_p = V_n = V_{cc}$, a typical fault voltage $V_f = V_{cc} - 200\,\text{mV}$, and typical $SR = 20\ \text{mV/µs}$, the minimal glitch width is hence $T_{min} = 20\,\text{µs}$.

The VR on some systems, e.g., i3-7100U-NUC, further supports a higher slew rate of $SR = 40\,\text{mV/µs}$. Thus, in this case, T_{min} can be further reduced to 10µs. However, during practical experiments with i3-7100U-NUC, we noticed that typically 2 µs after the voltage has reached V_f, the CPU emits an SVID packet, which negatively affects the ability to inject short glitches. The injected SVID packet to immediately reset the voltage from V_f to V_n has a high probability of colliding with the CPU's command, leading to an ineffective packet injection and the voltage cannot be increased to V_n until the next SVID set voltage packet.

6 Delayed-Write Fault Attacks through Undervolting

In this section, we describe a novel class of undervolting-induced faults not reported in prior research. Specifically, we observed that undervolting appears to briefly delay memory writes to the cache, so adjacent instructions still read the previous value. We initially observed these faults with hardware-based undervolting using VoltPillager, however, could also later reproduce them on a i7-7700HQ in a Dell XPS15 9560 laptop (without the CVE-2019-11157 mitigations installed) through software-based undervolting. We refer to this system as i7-7700HQ-XPS in the following. For our first PoC, we compare two integers for inequality, as further explained in the following Section 6.1.

6.1 Initial Proof-of-Concept

The C code used for our PoC is shown in Listing 2. We developed this initial PoC in userspace and later verified that a realistic exploit also works when running inside an SGX enclave, cf. Section 6.2.

```
1 int i = faulty = 0;
2 int operand1 = ...;
3 int operand2 = operand1;
4
5 do {
6   if(operand1 != operand2)  {
7         faulty = 1;
8   }
9   operand1++;
10  operand2++;
11  i++;
12 } while(faulty == 0 && i < iterations);
13 // ... trigger code and fault check omitted ...
```

Listing 2: Simplified C code used for demonstrating fault injection into memory accesses

Note that the code from Listing 2, under normal non-faulty execution, never sets `faulty` to 1.This tight loop of memory writes (for incrementing) and reads (for comparison) was essential to discover this novel effect. After a pre-set number of executions (or if `faulty` is set), the loop terminates and proceeds to check whether a fault was injected (*i.e.*, `faulty` has been set). To avoid faulting adjacent instructions after the detection of a successful fault, we also inserted a group of `nop` instructions as a buffer between the loop and subsequent code.

If a fault has occurred, we output the values of both operands to ensure that the actual operand values have not changed (or e.g., the increment has been faulted). With this design, we can reduce the actual assembly instructions that can possibly be affected by a fault to the instructions on Lines 1, 3, 5, 9, 10 shown in Listing 3. Note that even though Listing 3 uses the 32-bit register `%eax`, all our code is compiled and runs in 64-bit mode. We have not observed any difference between 32 and 64-bit instructions.

```
1 mov    -0x18(%rbp),%eax
2 // compare operand1 (%eax) and operand2
3 cmp    -0x14(%rbp),%eax
4 // continue at no_fault if equal
5 je     no_fault
6 // else set faulty = 1
7 movl   $0x1,0x20290f(%rip)
8 // Increment operands and counter
9 no_fault: addl   $0x1,-0x18(%rbp)
10 addl   $0x1,-0x14(%rbp)
11 addl   $0x1,-0x1c(%rbp)
```

Listing 3: Assembly compiled from Listing 2 with Lines 1, 3, 5, 9, 10 presumably affected by fault injection (AT&T syntax)

In other words, a fault injection that would lead to `faulty` being set could only affect one of the following common instructions: (*i*) a load from memory into a register, (*ii*) increment/write to memory, (*iii*) a comparison of a memory location with a register, or (*iv*) a jump-if-equal operation.

Initial Experiments We ran the target code described in Section 6.1 on the i3-7100-AZ170 at a clock frequency of 3 GHz and observed several successful fault injections (*i.e.*, `faulty` being set to 1) at 24° C with $V_f = 0.76\,V$, $V_p = 0.95\,V$, and $T_f = 29\,\text{µs}$ as shown in Figure 14. In all cases, the operands

printed after the detection of the fault were equal, *i.e.*, the undervolting could have only affected the assembly instructions pointed out above.

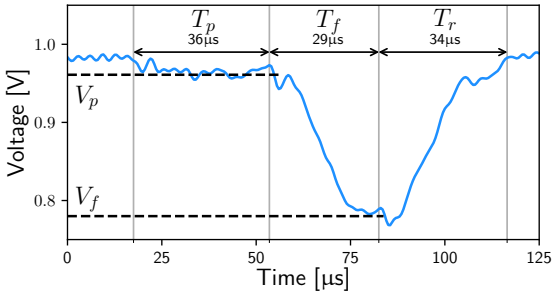

Figure 14: Oscilloscope capture of the CPU voltage during a successful fault injection. Signal low-pass filtered (Gaussian filter with $\sigma = 2$) for clarity. Actual V_f different to the value specified by VoltPillager due to physical effects. T_r indicates recovery time from V_f to V_n.

We found the parameters for successful fault injection through manual tuning combined with an exhaustive search in a specified region. Note that our experiments required a relatively low voltage, which in turn can affect system stability.

As reported by earlier work [37, 24], the CPU temperature affects the success rate and system stability. For faulting operations with (presumably) short critical paths such as memory accesses, we found this effect even more pronounced than for multiplications or AES-NI rounds (with longer critical path) on the i3-7100-AZ170. Hence, we ensured that we kept the core temperature at around 24° C with standard cooling. Subsequently, we successfully reproduced the same experiment on i7-7700HQ-XPS at 2 GHz with software-based undervolting through MSR `0x150`. The attack succeeded at normal core temperature of approximately 50° C without any additional cooling.

Further Analysis By modifying the code from Section 6.1, we obtained further insights into the likely cause of the fault (*i.e.*, the "operands not equal" branch being taken). To this end, we replaced the increment of `operand2` on Line 10 with a decrement of `operand1`, *i.e.*, `operand1/2` should be equal and constant throughout the loop, as shown in Listing 4:

```
1 do {
2   if(operand1 != operand2) {
3       faulty = 1;
4   }
5   operand1++;
6   operand1--;
7   i++;
8 } while(faulty == 0 && i < iterations);
```

Listing 4: Modified C code used for demonstrating fault injection into memory accesses

However, in this case, fault injection led to `operand1` being only decremented, with the increment on Line 5 seemingly ignored. Conversely, when swapping the order and first decrementing followed by incrementing, `operand1` took the value `operand2 + 1` when undervolted. Furthermore, we modified the assembly code from Listing 3 to store the value of `%eax` used for the comparison on Line 3 when the "operands not equal" branch had been taken. We observed that, e.g., when a fault was found for `operand1` = `operand2` = 3, the value loaded into `%eax` was 2 *i.e.*, the increment had not taken effect.

From these observations, we conjecture that the most likely explanation for the observed faults is that *recent* memory (cache) writes are delayed and thus ignored in adjacent reads of the modified location. This suggests that the fault affects the load-store queue logic of the CPU, causing writes to be delayed for a few cycles while the execution of dependent instructions progresses with old values. For example, for the (undervolted) code sequence `operand1++; operand1--;` the decrement operates on the previous value of `operand1`, ignoring the update through the preceding increment.

6.2 Practical Exploitation Scenario

We now consider a realistic scenario to exploit the effects from Section 6.1. The experiments described in this subsection were all performed inside an SGX enclave. We show how a delayed-write fault can be used to trigger out-of-bounds accesses in memory-safe code. To this end, the PoC code shown in Listing 5 initializes elements of an array to a fixed value.

```
1 uint32_t array[8] = { 0 };
2 // Attacker-supplied out-of-bounds size
3 int copy_size = 7;
4
5 // Ensure we stay within bounds
6 if(copy_size >= 5)
7   copy_size = 4;
8
9 // overwrite elements 4, 3, 2, 1
10 while(copy_size >= 1)  {
11   array[copy_size] = 0xabababab;
12   copy_size--;
13 }
```

Listing 5: Proof-of-concept to demonstrate out-of-bounds memory accesses due to undervolting

In Listing 5, `array[]` holds eight `uint32_t` elements all initially set to zero. The code then first ensures that the (potentially adversary-controlled) upper bound `copy_size` is ≤ 4, using a common

code pattern that effectively implements `min(4, copy_size)`. It then proceeds to write `0xABABABAB` to array elements 4 to 1, leaving the other elements (0 and 5 . . . 7) at their initial value of zero.

We are intentionally only writing to part of the eight-element array in this PoC to avoid triggering actual stack corruptions (and hence crashes). However, note that all experiments also apply to a real scenario, where the attacker would write beyond array bounds and corrupt the enclave stack, thus gaining control over the program counter and applying traditional exploitation techniques afterwards [30].

We undervolted the CPU whilst executing the above code in a loop within an `ecall` handler. The experiments were run on i7-7700HQ-XPS at a frequency of 2 GHz, undervolting by -170 mV. The core temperature reported by the CPU varied between 44° C and 49° C. We observed two distinct effects induced by the fault (cf. Appendix B), as illustrated in Figure 15: (*i*) in addition to elements 4 to 1, element 0 was also overwritten (*i.e.*, an underflow) and (*ii*) the upper bound was not limited to 4 but stayed at 7, *i.e.*, an overflow into elements 5 . . . 7 occurred. In both cases, out-of-bounds accesses take

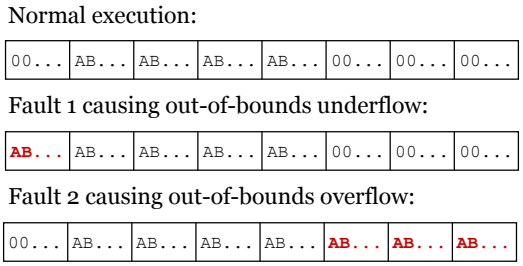

Figure 15: State of `array[]` after normal execution of Listing 5 and out-of-bounds under/overflow when undervolted. Faulty values in red bold font.

place, leading to potential memory corruption and enabling further exploitation with traditional techniques, e.g., through stack overflows. We describe the two observed fault types in the following.

Case 1: Out-of-Bounds Underflow As shown in Figure 15, undervolting caused `array[0]` to be incorrectly overwritten. Our analysis showed that this is due to a fault affecting the code responsible for decrementing and directly afterwards comparing the loop counter on Lines 12 and 10 in Listing 5, which translates to the assembly code shown in Listing 6.

```
1 // check for copy_size >= 1
2 copy_loop: cmpq   $0x0,-0x28(%rbp)
3 jle    exit_loop
4 // move copy_size into rax
5 mov    -0x28(%rbp),%rax
6 // move 0xabababab into array[copy_size]
7 movl   $0xabababab,-0x20(%rbp,%rax,4)
8 // copy_size--
9 subq   $0x1,-0x28(%rbp)
10 jmp    copy_loop
11 exit_loop: // ...
```

Listing 6: Assembly affected by underflow

When undervolting, we observed the decrement of the loop counter on Line 9 in Listing 6 had not been committed by the time the comparison on Line 2 occurs. Thus, the loop performs one additional iteration for `copy_size` = 0. We found that the decrement *does* come into effect on the subsequent read into `%rax` on Line 5, which is the index into the array, hence overwriting `array[0]`.

Case 2: Out-of-Bounds Overflow In the second observed fault, elements 5–7 are incorrectly overwritten. In this case, we concluded that the fault affects the initialisation of the upper limit on Lines 6 and 7 in Listing 5. The respective assembly snippet is shown in Listing 7.

```
1 movq   $0x7,-0x28(%rbp)
2 cmpq   $0x4,-0x28(%rbp)
3 // jump if copy_size less than or equal to 4
4 // THIS JUMP SHOULD NEVER BE TAKEN
5 jle    cont
6 // set copy_size = 4
7 movq   $0x4,-0x28(%rbp)
8 cont: // ...
```

Listing 7: Assembly affected by overflow

As with the previous fault, we conclude that the operation `copy_size` = 7 on Line 1 has not completed by the time the compare statement on Line 2 is reached. Consequently, `copy_size` is not limited to 4 but remains at the higher value of 7, triggering writes beyond the upper limit of 4. Note that in this example the initial value 7 is loaded as a constant, but it could equivalently be loaded from an attacker-controlled parameter, e.g., an untrusted length field passed to an `ecall`.

7 Implications and Countermeasures

To the best of our knowledge, this paper presents the first practical attack that directly breaches integrity guarantees in the Intel SGX security architecture through a hardware-based attack. We show that the fix currently deployed by Intel—disabling the software undervolting interface— is insufficient when taking hardware-based attacks with physical access into account.

We also believe that these attacks might have implications for non-SGX programs, because the fault injection in principle does not require code execution on the CPU (in contrast to the software-based fault

attacks [37, 40, 24]). It is conceivable that it could enable attacks on e.g., locked computers using disk encryption, similar to the attacker model for Direct Memory Access (DMA) attacks [33], where an adversary with physical access is able to bypass security mechanisms. However, an adversary mounting attacks would face substantial challenges, including proper triggering to fault the desired program while keeping the system stable, and the need to open the case and connect to SVID without powering down.

It is worth noting that the type of attacks described in this paper could be applied to CPUs by other vendors: AMD uses a similar design with a VR connected to the CPU though their SVI bus [44].

Countermeasures against the attacks described in this paper can be implemented at the level of (*i*) the SVID protocol, (*ii*) the CPU hardware or μCode, (*iii*) the enclave code itself. In the following, we discuss mitigations in detail. Note that countermeasures cannot be implemented in the BIOS or in components outside the CPU package, because SGX regards the BIOS and external hardware as untrusted.

Mitigation through Changes to SVID In our opinion, the issue of voltage glitching with physical access cannot be effectively addressed by e.g., adding cryptographic authentication to the SVID protocol. As explained in Section 3, the VR essentially converts the SVID commands to a PWM-modulated waveform, which controls the transistors generating the actual core voltage. Hence, instead of injecting into SVID, an attacker could disconnect these control outputs and supply their own (malicious) PWM signal, bypassing any authentication of SVID. A well-resourced attacker could even completely replace the VR with a custom voltage glitcher, and a malicious cloud provider could use custom motherboards with built-in glitching functionality.

Mitigation in CPU Hardware or μCode To detect the adversary's injected SVID packets, the CPU could monitor the bus for packets that were not generated by itself. On detecting packet injection, the CPU could raise an exception and abort execution. But, as pointed out in Section 4, an adversary could split the connection between CPU and VR and act as man-in-the-middle, hiding malicious packets from the CPU. Hence, this countermeasure would only protect against basic SVID injection attacks with VoltPillager in parallel to the bus.

Secondly, the CPU could continuously monitor its own supply voltage and abort if the voltage falls below a safe threshold. When using existing functionality such as measuring the core voltage through the Running Average Power Limit (RAPL) interface and MSR `0x198` [22, 25], it should be taken into account that such interfaces have a low sample rate in the order of 1 kHz. Hence, they would not be fast enough to detect glitches shorter than the sampling window.

Therefore, future CPU generations could include dedicated hardware countermeasures [23], including e.g., voltage monitoring circuitry as commonly found in smartcards [42]. One could also consider running critical code paths on multiple cores and detect deviations through additional CPU logic as e.g., implemented by certain Infineon smartcard ICs [19]. However, such countermeasures would amount to substantial hardware changes and incur overheads. Another option would be the use of a Fully Integrated Voltage Regulator (FIVR), *i.e.*, the VR integrated within the CPU package, as used in 4th generation Intel CPUs, but later abandoned in newer generations [7]. However, as the input voltage for the FIVR is still supplied externally, such circuitry would have to appropriately handle malicious modification of that input voltage.

Mitigation in Enclave Code For SGX enclaves that require immediate protection against fault injection, countermeasures can be implemented within the enclave code. According to our experience, it is highly unlikely to produce the same fault twice in adjacent or nearby instructions; however, this warrants further detailed investigation. Enclaves could duplicate potentially vulnerable instructions and compare the results. While manual insertion of such countermeasures might be feasible for comparatively small pieces of critical code (e.g., crypto functions or memory management), fully protecting an existing codebase would require excessive effort. However, prior research [32, 4] shows that automatic instruction duplication at the compiler level is feasible. Hence, porting such techniques to x86 architectures and in particular SGX poses an interesting problem for future work. Note that SGX enclaves cannot rely on mitigations based on measuring CPU voltage, as SGX does not offer a trusted way to access MSRs, so any such countermeasure could be bypassed by a compromised operating system.

8 Conclusions

In this paper we identified a novel and powerful attack surface of Intel CPUs. We have shown how the SVID interface can be leveraged by adversaries with physical access to gain full control over the voltage regulator. We then demonstrate that dynamic voltage scaling can be reliably exploited to mount

fault-injection attacks against the CPU. To the best of our knowledge, this represents the first hardware-based fault-injection attack against a fully-fledged CPU and also the first one that directly breaches the integrity and confidentiality of SGX enclaved computations on a fully patched system.

We have proven that this attack vector is practical by recovering RSA keys from an enclaved application, and have shown that other fundamental operations such as multiplication and memory/cache writes can be faulted as well. These lead to novel memory safety vulnerabilities within SGX, which are not detected by SGX's memory protection mechanisms. The results in this paper, together with the manufacturer's decision to not mitigate this type of attack, prompt us to reconsider whether the widely believed enclaved execution promise of outsourcing sensitive computations to an untrusted, remote platform is still viable.

Acknowledgments

This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1, EP/V000454/1, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM), and by the Paul and Yuanbi Ramsay Endowment Fund.

References

[1] Arduino. Arduino IDE. `https://www.arduino.cc/en/Main/Software`.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Usenix OSDI '16*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[3] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[4] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *CS2 '16*, page 1–6. ACM, 2016.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Usenix OSDI '14*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[6] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Eurocrypt'97*, pages 37 – 51, 1997.

[7] Edward Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William Lambert, Kaladhar Radhakrishnan, and Michael Hill. FIVR — Fully integrated voltage regulators on 4th generation Intel Core SoCs. In *IEEE APEC '14*, pages 432–439, 03 2014.

[8] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC '17*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[9] Ang Cui and Rick Housley. BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. In *Usenix WOOT '17*, Vancouver, BC, August 2017. USENIX Association.

[10] DreamSource Technology Co., Ltd. DSLogic Plus. accessed June 2, 2020.

[11] Enarx. Threat model. accessed June 17, 2020, revision 678e2c2. `https://github.com/enarx/enarx/wiki/Threat-Model`.

[12] Fortanix. intel-oc-mbox. accessed September 21, 2020. `https://github.com/fortanix/intel-oc-mbox/tree/jb/initial`.

[13] Fortanix. Intel SGX FAQ. accessed June 2, 2020. `https://fortanix.com/intel-sgx/`.

[14] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020.

[15] Github. intel-undervolt issue 43. accessed June 18, 2020. `https://github.com/kitsunyan/intel-undervolt/issues/43#issuecomment-619373836`.

[16] Github. Plundervolt. accessed June 18, 2020, commit 3bb0295. `https://github.com/KitMurdock/plundervolt`.

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P '18*, pages 245–261, 2018.

[18] Infineon. IR35204 3+1 Dual Output Digital Multi-Phase Controller datasheet, 2016.

[19] Infineon. Integrity guard. online, accessed 2020-04-05: `https://www.infineon.com/dgdl/Infineon-Integrity_Guard_The_smartest_digital_security_technology_in_the_industry_06.18-WP-v01_01-EN.pdf?fileId=5546d46255dd933d0155e31c46fa03fb`, 2018.

[20] Intel. Intel SGX Technical Details for INTEL-SA-00289 and INTEL-SA-00334. accessed June 5, 2020. `https://cdrdv2.intel.com/v1/dl/getContent/619320`.

[21] Intel. 4th Generation i7 Datasheet Vol. 1, 2014.

[22] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers, May 2019.

[23] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[24] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security '20*, Boston, August 2020. USENIX Association.

[25] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ToMPECS*, 2018.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[27] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. TensorSCONE: A Secure TensorFlow Framework using Intel SGX, 2019. arXiv 1902.04413.

[28] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P '20*, 2020.

[29] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca A. Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security '20*, Boston, August 2020. USENIX Association.

[30] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security '17*, pages 523–539, 2017.

[31] Yifan Lu. Attacking hardware AES with DFA. *arXiv preprint arXiv:1902.08693*, 2019.

[32] Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter, and Koen De Bosschere. Mitigating smart card fault injection with link-time code rewriting: A feasibility study. In *Financial Cryptography*, 2013.

[33] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS '19*, 2019.

[34] Microsoft. Azure confidential computing. version of Dec 6, 2019 retrieved from archive.org. `https://web.archive.org/web/20191206233429/https://azure.microsoft.com/en-gb/solutions/confidential-compute/`.

[35] Microsoft. Open Enclave SDK. accessed September 23, 2020. `https://github.com/openenclave/openenclave`.

[36] Microsoft. Azure confidential computing, 2020.

[37] 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 *S&P '20*, 2020.

[38] PJRC. Teensy 4 development board. accessed June 2, 2020. `https://www.pjrc.com/store/teensy40.html`.

[39] PJRC. Teensyduino arduino library. https://www.pjrc.com/teensy/td_download.html.

[40] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *AsianHOST '19*, pages 1–6, 2019.

[41] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS '19*, pages 195–209. ACM, 2019.

[42] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. Wiley, 4th edition, 2010.

[43] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security '16*, pages 1–18, Austin, August 2016. USENIX Association.

[44] Renesas. ISL95712 Datasheet, 2015. available at https://www.renesas.com/eu/en/www/doc/datasheet/isl95712.pdf.

[45] Rigol. DS1074Z 70MHz Digital Oscilloscope.

[46] Thomas Roth. TrustZone-M(eh): Breaking ARMv8-M's security—Hardware attacks on the latest generation of ARM Cortex-M processors. presentation at 36C3, 2019.

[47] Synaptics. Lenovo, Intel, PayPal and Synaptics announce collaboration to bring FIDO authentication to laptops. accessed June 4, 2020.

[48] System Management Interface Forum, Inc. System Management Bus (SMBus) Specification Version 3.0. accessed June 9, 2020. http://smbus.org/specs/SMBus_3_0_20141220.pdf.

[49] System Management Interface Forum, Inc. PMBus^{TM} Power System Management Protocol Specification, Part I, Revision 1.2X. accessed June 9, 2020, September 2010.

[50] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security '17*, pages 1057–1074, Vancouver, BC, August 2017. USENIX Association.

[51] Texas Instruments. SN74LVC1G07 Single Buffer/Driver With Open-Drain Output, 2016.

[52] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. Springer, 2011.

[53] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Hardware and Systems Security*, 2(2):111–130, 2018.

[54] ZEROPLUS. Protocol Analyzer SVID 1.04.00. http://www.zeroplus.com.tw/logic-analyzer_en/news_detail.php?news_id=1755, 2014.

A Glitch Configuration and Results for Multiplication and CRT-RSA

```
0x3c, 0xf7, 0x21, 0x56, 0xe7, 0x59, 0x69, 0x06,
0x08, 0x06, 0x01, 0x69, 0xf0, 0xa3, 0x0c, 0xb9,
0x0d, 0x3b, 0x75, 0xe9, 0x02, 0xb3, 0xe0, 0x05,
0xef, 0x59, 0xbf, 0x05, 0x54, 0x0f, 0xec, 0xc3,
0xc8, 0x90, 0x7b, 0x45, 0x90, 0x9c, 0x4b, 0x4e,
0xfc, 0x8d, 0xed, 0x0f, 0x31, 0xaa, 0xad, 0xae,
0x40, 0x0d, 0xf3, 0xc4, 0x6c, 0x00, 0x3b, 0xdd,
0x7a, 0xf6, 0x22, 0x61, 0x53, 0x2a, 0xcc, 0xf2,
0x16, 0xb9, 0xa7, 0x3e, 0x98, 0xbb, 0x8f, 0x56,
0xad, 0x4c, 0x35, 0xa2, 0x6e, 0x47, 0xd8, 0x80,
0x36, 0x4c, 0x9a, 0x2b, 0xab, 0x25, 0x08, 0x63,
0x93, 0x28, 0x6b, 0x98, 0xad, 0xda, 0x74, 0xab,
0x8b, 0xd2, 0x04, 0xeb, 0x4e, 0x76, 0xc5, 0x09,
0xe7, 0xd8, 0x5f, 0x97, 0xf3, 0x13, 0x75, 0x29,
0xd3, 0xa6, 0x07, 0xb5, 0x1f, 0x9f, 0x07, 0xfc,
0x82, 0x19, 0x70, 0x04, 0xda, 0x12, 0x71, 0x3e
```

Listing 8: Example faulty result obtained on i3-7100-AZ170 for CRT-RSA

```
0x41, 0xbf, 0xa9, 0x4d, 0x96, 0xae, 0x2d, 0x35,
0xe4, 0xa8, 0xc7, 0x24, 0xaa, 0x8c, 0xc2, 0x05,
0x0f, 0x32, 0x56, 0xe5, 0x37, 0x56, 0x5d, 0x94,
0x31, 0x82, 0x62, 0xd8, 0xbc, 0x32, 0x34, 0xc0,
0x70, 0xdb, 0xfe, 0x98, 0xcc, 0x6e, 0x26, 0x75,
0x58, 0xa8, 0x2a, 0x84, 0xe7, 0x14, 0xe2, 0x4a,
0x93, 0x3b, 0xc2, 0x4d, 0xe9, 0xcb, 0xa2, 0x61,
0x07, 0x62, 0x88, 0xcb, 0x01, 0x36, 0x58, 0x1d,
0x8d, 0x09, 0x9b, 0x0a, 0x0b, 0x7e, 0x42, 0xd0,
0x68, 0xbb, 0x16, 0x28, 0x60, 0x14, 0x78, 0x3d,
0x73, 0x0a, 0xf5, 0x62, 0x2d, 0xbd, 0x22, 0xf0,
0x59, 0x96, 0x39, 0x5c, 0xbc, 0xe1, 0x46, 0x0b,
0x99, 0x3e, 0x04, 0x4a, 0x69, 0xbc, 0xdf, 0xc0,
0x5b, 0xb3, 0x98, 0x11, 0x56, 0xea, 0x03, 0xa2,
0x3a, 0x80, 0xc9, 0xd3, 0xe0, 0x7c, 0x55, 0xe6,
0x5c, 0x20, 0x13, 0x86, 0x7b, 0xba, 0x87, 0x6d
```

Listing 9: Example faulty result obtained on i3-9100-MZ370 for CRT-RSA

```
0x6e, 0x35, 0xea, 0x8c, 0xac, 0xe4, 0xe8, 0x1d,
0xc0, 0x3f, 0x52, 0xe7, 0xf8, 0x27, 0x21, 0xd1,
0x75, 0x86, 0x1e, 0x30, 0xe7, 0xe6, 0x90, 0x07,
0x5a, 0xc6, 0xed, 0x97, 0x21, 0x59, 0xad, 0x4d,
0x61, 0x64, 0x43, 0x5f, 0x70, 0x78, 0xbc, 0x78,
0x1a, 0x82, 0x1e, 0x0d, 0x8f, 0xd3, 0x6d, 0x27,
```

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Iteration	Temp.
i3-7100-AZ170	2 GHz	1000 µs	1	0.83 V	35 µs	0.64 V	29 µs	0.83 V	90236	23° C
i3-9100-MZ370	3.4 GHz	300 µs	1	1.050 V	35 µs	0.81 V	83 µs	1.05 V	126490	26° C
i3-7100U-NUC	2 GHz	200 µs	1	0.94 V	35 µs	0.71 V	8 µs	1.05 V	41827	32° C

Table 4: Parameters for successful fault injection with the Plundervolt PoC for userspace multiplication

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.
i3-7100-AZ170	2 GHz	100 µs	1	0.83 V	35 µs	0.63 V	29 µs	0.83 V	24° C
i3-9100-MZ370	3.4 GHz	10 µs	1	1.050 V	10 µs	0.81 V	43 µs	1.05 V	27° C
i3-7100U-NUC	2 GHz	10 µs	1	0.94 V	35 µs	0.75 V	9 µs	1.05 V	22° C

Table 5: Parameters for successful fault injection with the Plundervolt PoC for CRT-RSA

Run	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.	Iteration
1	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	24° C	97702
2	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	92286
3	500 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	174087

Table 6: Parameters for fault injection into the initial memory access PoC on i3-7100-AZ170 at 3 GHz

```
0x78, 0x28, 0x72, 0x6f, 0xf9, 0x63, 0x6e, 0x8e,
0x92, 0x98, 0x40, 0x96, 0x2e, 0xde, 0x28, 0x0a,
0x14, 0x1d, 0xc0, 0xc3, 0x27, 0xf3, 0x44, 0xa8,
0x8d, 0xf5, 0xb5, 0xe5, 0x1c, 0x96, 0xed, 0xe4,
0xf6, 0x11, 0xa4, 0xa6, 0x26, 0x7f, 0xf1, 0x82,
0xaf, 0x33, 0x85, 0x24, 0xc5, 0x3d, 0x67, 0x2a,
0x55, 0x69, 0xd9, 0xc3, 0x9b, 0xcb, 0x25, 0xfc,
0xa4, 0x9a, 0x2a, 0x5d, 0x6e, 0xa6, 0x92, 0x97,
0xf0, 0x14, 0x3f, 0x8e, 0x91, 0x33, 0x65, 0xa1,
0x61, 0x0f, 0x75, 0xbf, 0xc1, 0x08, 0xec, 0x61
```

Listing 10: Example faulty result obtained on i3-7100U-NUC for CRT-RSA

B Example Results for Faults during Memory Accesses

The following out-of-bounds overflow fault happened at iteration 769170 with -172 mV undervolting and the CPU running at 2 GHz on i7-7700HQ-XPS during computation inside SGX.

```
[Enclave] FAULT: array[00]:    0x00000000
[Enclave] FAULT: array[01]:    0xabababab
[Enclave] FAULT: array[02]:    0xabababab
[Enclave] FAULT: array[03]:    0xabababab
[Enclave] FAULT: array[04]:    0xabababab
[Enclave] FAULT: array[05]:    0xabababab
[Enclave] FAULT: array[06]:    0xabababab
[Enclave] FAULT: array[07]:    0xabababab
```

Listing 11: Overflow on i7-7700HQ-XPS

The following out-of-bounds underflow happened at iteration 210612 with -175 mV undervolting on the same system during computation inside SGX.

```
[Enclave] FAULT: array[00]:    0xabababab
[Enclave] FAULT: array[01]:    0xabababab
[Enclave] FAULT: array[02]:    0xabababab
[Enclave] FAULT: array[03]:    0xabababab
[Enclave] FAULT: array[04]:    0xabababab
[Enclave] FAULT: array[05]:    0x00000000
[Enclave] FAULT: array[06]:    0x00000000
[Enclave] FAULT: array[07]:    0x00000000
```

Listing 12: Underflow on i7-7700HQ-XPS

C Details of SVID

The 1-byte VID value for a target voltage U (in volt) is computed as:

$$\text{VID} = \left\lfloor \frac{U - 0.245}{0.005} \right\rfloor$$

SVID commands are 5 bit. We used SetVID-Fast (`0x01`) for setting the voltage. We also discovered other commands shown in Table 7 with the help of a screenshot of an SVID protocol analyzer [54].

Command name	Value
Extended	0x00
SetVID-Fast	0x01
SetVID-Slow	0x02
SetVID-Decay	0x03
SetPS	0x04
SetRegADR	0x05
SetRegDAT	0x06

Table 7: 5-bit SVID commands based on [54]