Plundervolt: Software-based Fault Injection Attacks against Intel SGX

Kit Murdock*, David Oswald*, Flavio D. Garcia*, Jo Van Bulck†, Daniel Gruss‡, and Frank Piessens‡

*University of Birmingham, UK
d.kxm663@cs.bham.ac.uk, d.f.oswald@bham.ac.uk, f.garcia@bham.ac.uk
†Graz University of Technology, Austria
daniel.gruss@iaik.tugraz.at
‡imec-DistriNet, KU Leuven, Belgium
jo.vanbulck@cs.kuleuven.be, frank.piessens@cs.kuleuven.be

Abstract—Dynamic frequency and voltage scaling features have been introduced to manage ever-growing heat and power consumption in modern processors. Design restrictions ensure frequency and voltage are adjusted as a pair, based on the current load, because for each frequency there is only a certain voltage range where the processor can operate correctly. For this purpose, many processors (including the widespread Intel Core series) expose privileged software interfaces to dynamically regulate processor frequency and operating voltage.

In this paper, we demonstrate that these privileged interfaces can be reliably exploited to undermine the system’s security. We present the Plundervolt attack, in which a privileged software adversary abuses an undocumented Intel Core voltage scaling interface to corrupt the integrity of Intel SGX enclave computations. Plundervolt carefully controls the processor’s supply voltage during an enclave computation, inducing predictable faults within the processor package. Consequently, even Intel SGX’s memory encryption/authentication technology cannot protect against Plundervolt. In multiple case studies, we show how the induced faults in enclave computations can be leveraged in real-world attacks to recover keys from cryptographic algorithms (including the AES-NI instruction set extension) or to induce memory safety vulnerabilities into bug-free enclave code. We finally discuss why mitigating Plundervolt is not trivial, requiring trusted computing base recovery through microcode updates or hardware changes.

I. INTRODUCTION

The security of modern systems builds on abstractions of the underlying hardware. However, hardware is subject to physical effects and is increasingly optimized to meet the ever-growing need for performance and efficiency. Modern CPUs are highly optimized such that performance and efficiency are maximized while maintaining functional correctness under specified working conditions.

In fact, many modern processors cannot permanently run at their maximum clock frequencies because it would consume significant power that, in turn, produces too much heat (e.g., in a data center). Additionally, in mobile devices, high power consumption drains the battery quickly.

This voltage and frequency dependency of the (dynamic) power consumption $P_{dyn}$ of a CMOS circuit is expressed as: $P_{dyn} \propto f \cdot V^2$, i.e., the dynamic power consumption is proportional to the clock frequency $f$ and to the square of the supply voltage $V$.

Because of this relationship (and other factors), modern processors keep the clock frequency and supply voltage as low as possible—only dynamically scaling up when necessary. Higher frequencies require higher voltages for the processor to function correctly, so they should not be changed independently. Additionally, there are other types of power consumption that influence the best choice of a frequency/voltage pair for specific situations.

Lowering the supply voltage was also important in the development of the last generations of DRAM. The supply voltage has been gradually reduced, resulting in smaller charges in the actual capacitors storing the single bits—this led to the well-known Rowhammer [41] effect. Exploiting this, a long line of research has mounted practical attacks, e.g., for privilege escalation [60, 22, 77, 74], injecting faults into cryptographic primitives [55, 6], or reading otherwise inaccessible memory locations [44]. While fault attacks have been extensively studied for adversaries with physical access to embedded devices [71, 62, 2, 26], Rowhammer remains, to date, the only known purely software-based fault attack on x86-based systems. Hence, both the scientific community and industry have put significant effort in developing Rowhammer mitigations [41, 37, 27, 3, 23, 53, 12, 79, 13, 74, 22, 10]. This has reached a point where Intel ultimately considers main memory as an untrusted storage facility and fully encrypts and authenticates all memory within the Intel SGX enclave security architecture [24]. But is authentication and encryption of memory enough to safeguard the integrity of general-purpose computations?

To answer this question, we investigate interfaces for supply voltage optimizations on x86 Intel Core CPUs. With shrinking process technology, the processor supply voltages have gradually been reduced to make systems more efficient. At the same time, voltage margins (the stable voltage ranges for each frequency) have shrunk. The actual voltage margin is strongly influenced by imperfections in the manufacturing process and also the specific system setup, including the voltage regulator on the main board. Since these dynamic voltage and frequency scaling features are undocumented and only exposed to privileged system software, they have been scarcely studied from a security perspective. However, this is
very relevant in the context of SGX. Intel SGX enclaves are currently considered immune to fault attacks. In particular, Rowhammer, the only software-based fault attack known to work on x86 processors, simply causes the integrity check of the Memory Encryption Engine (MEE) to fail [21, 38], halting the entire system.

A. Related Work on Software-based Fault Attacks

A fault attack manipulates the computations of a device with the purpose of bypassing its security mechanisms or leaking its secrets. With this aim, the attacker manipulates the environment to influence the target device’s computations. Typically such fault-inducing environments are at the border of (or beyond) the specified operational range of the target device. Different environment manipulations have been investigated [26], such as: exposure to voltage and clock glitching [2] [62], extreme temperatures [29] or laser/UV light [63].

Software-based fault attacks shift the threat model from a local attacker (with physical access to the target device) to a potentially remote attacker with only local code execution.

Initially, these attacks were interesting in scenarios where the attacker is unprivileged or even sandboxed. However, with secure execution technologies, such as: Intel SGX, ARM TrustZone and AMD SEV, privileged attackers must also be considered as they are part of the corresponding threat models.

In 2017, Tang et al. [65] discovered a software-based fault attack, dubbed CLKscrew. They discovered that ARM processors allow configuration of the dynamic frequency scaling feature, *i.e.*, overclocking, by system software. Tang et al. show that overclocking features may be abused to jeopardize the integrity of computations for privileged adversaries in a Trusted Execution Environment (TEE). Based on this observation, they were able to attack cryptographic code running in TrustZone. They used their attack to extract cryptographic keys from a custom AES software implementation and to overcome RSA signature checks and subsequently execute their own program in the TrustZone of the System-on-Chip (SoC) on a Nexus 6 device.

However, their attack is specific to TrustZone on a certain ARM SoC and not directly applicable to SGX on Intel processors. In fact, it is unclear whether similar effects exist on x86-based computers, whether they are exploitable, and whether the processor package or SGX has protections against this type of attack, *e.g.*, machine-check errors on the system level, or data integrity validation in SGX enclaves. Furthermore, CLKscrew is based on changing the frequency, while in this paper we focus on voltage manipulations. Finally, the question arises whether faults are limited to software implementations of cryptographic algorithms (as in CLKscrew), or can also be used to exploit hardware implementations (like AES-NI) or generic (non-crypto) code.

B. Our Contribution

In this paper, we present Plundervolt, a novel attack against Intel SGX to reliably corrupt enclave computations by abusing privileged dynamic voltage scaling interfaces. Our work builds on reverse engineering efforts that revealed which Model-Specific Registers (MSRs) are used to control the dynamic voltage scaling from software [64, 57, 49]. The respective MSRs exist on all Intel Core processors. Using this interface to *very briefly* decrease the CPU voltage during a computation in a victim SGX enclave, we show that a privileged adversary is able to inject faults into protected enclave computations. Crucially, since the faults happen *within* the processor package, *i.e.*, before the results are committed to memory, Intel SGX’s memory integrity protection fails to defend against our attacks. To the best of our knowledge, we are the first to practically showcase an attack that directly breaches SGX’s integrity guarantees.

In summary, our main contributions are:

1) We present Plundervolt, a novel software-based fault attack on Intel Core x86 processors. For the first time, we bypass Intel SGX’s integrity guarantees by directly injecting faults *within* the processor package.

2) We demonstrate the effectiveness of our attacks by injecting faults into Intel’s RSA-CRT and AES-NI implementations running in an SGX enclave, and we reconstruct full cryptographic keys with negligible computational efforts.

3) We explore the use of Plundervolt to induce memory safety errors into bug-free enclave code. Through various case studies, we show how in-enclave pointers can be redirected into untrusted memory and how Plundervolt may cause heap overflows in widespread SGX runtimes.

4) Finally, we discuss countermeasures and why fully mitigating Plundervolt may be challenging in practice.

C. Responsible Disclosure

We have responsibly disclosed our findings to Intel on June 7, 2019. Intel has reproduced and confirmed the vulnerabilities which they are tracking under CVE-2019-11157. Intel’s mitigation is provided in Section VII-C.

Our current results indicate that the Plundervolt attack affects all SGX-enabled Intel Core processors from Skylake onward. We have also experimentally confirmed the existence of the undervolting interface on pre-SGX Intel Core processors. However, for such non-SGX processors, Plundervolt does not currently represent a security threat in our assessment, because the interface is exclusively available to privileged users. Furthermore, in virtualized environments, hypervisors should never allow untrusted guest VMs to read from or write to undocumented MSRs.

We have made our PoC attack code available at: [https://github.com/KitMurdock/plundervolt](https://github.com/KitMurdock/plundervolt)

D. Structure of the Paper

Section II presents the attacker model, our experimental setup and the tested CPUs. In Section III, we present the basic working principle of the Plundervolt attack when targeting multiplications, with a detailed analysis of the fault characteristics in Section III-A. Section IV shows how Plundervolt can be used to recover cryptographic keys from RSA and AES-NI implementations running inside an SGX enclave. In Section V...
we discuss how Plundervolt can be used to induce memory safety vulnerabilities into bug-free code. In Section [VI], we discuss the Plundervolt attack w.r.t. related work, while Section [VII] considers different mitigation strategies. Section [VIII] concludes this paper.

II. EXPERIMENTAL SETUP

A. Attacker Model

We assume the standard Intel SGX adversary model where the attacker has full control over all software running outside the enclave (including privileged system software such as operating system and BIOS). Crucial for our attacks is the ability for a root adversary to read/write MSRs, e.g., through a malicious ring 0 kernel module or an attack framework like SGX-erasing system and BIOS). Crucial for our attacks is the ability for arbitrary kernel code execution, but without physical access to the target machine. At the hardware level, we assume a recent Intel Core processor with (i) Intel SGX enclave technology, and (ii) dynamic voltage scaling technology. In practice, we found these requirements to be fulfilled by all Intel Core processors we tested from Skylake onward (cf. Table 1).

B. Voltage Scaling on Intel Core Processors

We build on the reverse engineering efforts of [64, 49, 57] that revealed the existence of an undocumented MSR to adjust operating voltage on Intel Core CPUs. To ensure reproducibility of our findings, we document this concealed interface in detail. All results were experimentally confirmed on our test platforms (cf. Table 1).

Figure 1 shows how the 64-bit value in MSR 0x150 can be decomposed into a plane index and a voltage offset. Firstly, by specifying a valid plane index, system software can select to which CPU components the under- or overvolting should be applied. The CPU core and cache share the same voltage plane on all machines we tested and the higher voltage of both will be applied to the shared plane. Secondly, the requested voltage scaling offset is encoded as an 11-bit signed integer relative to the core’s base operating voltage. This value is expressed in units of 1/1024 V (about 1 mV), thus allowing a maximum voltage offset of ±1 V.

After software has successfully submitted a voltage scaling request, it takes some time before the actual voltage transition is physically applied. The current operating voltage can be queried from the documented MSR 0x198 (IA32_PERF_STATUS). We experimentally verified that all physical CPUs share the same voltage plane (i.e., scaling voltage on one core also adjusts all the other physical CPU cores).

From Skylake onwards, the voltage regulator is external to the CPU as a separate chip on the main board. The CPU requests a supply voltage change, which is then transferred to and executed by the regulator chip. In Intel systems, this is implemented as follows (based on datasheets for respective voltage regulator chips [30] and older, public Intel documentation [31]):

1) The CPU outputs an 8-bit value “VID”, encoding the currently requested voltage, to the voltage regulator on the mainboard. Based on CPU datasheets (Table 6-11 in [33]), it appears this value is transferred over a three-wire serial line called “Serial VID” or “SVID”, comprised of the pins VIDSORT, VIDSCCK, and VIDALERT#. Presumably, the offset in MSR 0x150 is subtracted from the base value within the CPU logic before outputting a VID code; however it is unclear why MSR 0x150 is in steps of 1/1024 V, while the 8-bit VID allegedly uses steps of 5 mV [30].

2) Based on the VID, the voltage regulator chip adjusts the voltage supplied via the core voltage pins (VCC) to the CPU. Note that there are configuration options for the slow rate i.e., the time taken for a specific voltage change to occur (fastest rate in [30] is given as 80 mV/μs), as well as limits on overshoot and undershoot.

C. Configuring Voltage and Frequency

In order to reliably find a faulty frequency/voltage pair, we configured the CPU to run at a fixed frequency. This step can be easily executed using documented Intel frequency scaling interfaces, e.g., through the script given in Appendix A.

The undervolting is applied by writing to the concealed MSR 0x150 (e.g., using the msr Linux kernel module) just before entering the victim enclave through an ECALL in the untrusted host program. After returning from the enclave, the host program immediately reverts to a stable operating voltage. Note that, apart from the msr kernel module, attackers can also rely on more precise methods to control undervolting, e.g., if configuration latency should be minimized. For this, we have extended the SGX-Step [71] enclave execution control framework with x86 interrupt and call gate functionality so as to be able to execute the privileged rdmsr and wrmsr instructions directly before entering a victim enclave.

One challenge for a successful Plundervolt attack is to establish the correct undervolting parameter such that the processor produces incorrect results for certain instructions, while still allowing the remaining code base to function normally. That is, undervolting too far leads to system crashes and freezes, while undervolting too little does not produce any faults. Finding the right undervolting value therefore requires some experimentation by carefully reducing the core voltage in small decrements (e.g., by 1 mV per step) until a fault occurs, but before the system crashes. In practice, we found
that it suffices to undervolt for short periods of time by -100 to -260 mV, depending on the specific CPU, frequency and temperature (see Section III-A for a more precise analysis).

D. Undervolting Decline Micro-benchmark

To study how quickly writes to MSR 0x150 manifest in actual changes to the core voltage, we performed a micro-benchmark where we continuously read the reported current CPU voltage from MSR 0x198 (IA32_PERF_STATUS). We executed the micro-benchmark code by means of a privileged x86 interrupt gate that first applies -100 mV undervolting and then immediately executes a tight loop of 300 iterations to collect pairs of measurements of the current processor voltage and the associated Time Stamp Counter (TSC) value.

![Figure 2. Voltage decline over time for Intel i3-7100U-C, repeating a -100 mV undervolting seven times and measuring actual voltage in MSR 0x198.](image)

Figure 2 displays the measurement results for seven repetitions of a -100 mV drop. It is immediately evident that there is a substantial delay (between 500k and 1M TSC ticks) between the MSR change and the actual undervolting being applied. While some of this delay might be due to the software-based measurement via MSR 0x198, our benchmark primarily reveals that voltage changes incur a non-negligible overhead. We will come back to this point in Section VII when devising countermeasures because this delay means returning to normal voltage when entering enclave mode may incur substantial overhead. Furthermore, when comparing the repetitions, it becomes apparent that voltage scaling behaves non-deterministically, i.e., the actual voltage drop occurs at different times after writing to MSR 0x150. However, from an attacker’s perspective, our micro-benchmark also shows that it is possible to precisely delay entry into a victim enclave by continuously measuring current operating voltage until the desired threshold is reached.

E. Tested Processors

For our experiments, we used different SGX-enabled processors from Skylake onwards, cf. Table I. We also had access to multiple CPUs with the same model numbers in some cases. Because we found that different chips with the same model number can behave differently when undervolted (cf. Section III-A), we list those separately and refer to them with a letter appended to the model number, e.g., i3-7100U-A, i3-7100U-B, etc. We carried out all experiments using Ubuntu 16.04 or 18.04 with stock Linux v4.15 and v4.18 kernels.

We attempted to undervolt a Xeon processor (Broadwell-EP E5-1630V4), however, found that in this case the MSR 0x150 does not seem to affect the core voltage.

**TABLE I**

<table>
<thead>
<tr>
<th>Code name</th>
<th>Model no.</th>
<th>Microcode</th>
<th>Frequency</th>
<th>Vulnerable</th>
<th>SGX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadwell</td>
<td>E5-1630V4</td>
<td>0xb000036</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skylake</td>
<td>i7-6700K</td>
<td>0xcc</td>
<td>2.0 GHz</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Kaby Lake</td>
<td>i7-7700HQ</td>
<td>0x48</td>
<td>2.0 GHz</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>i3-7100U-A</td>
<td>0xb4</td>
<td>1.0 GHz</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>i3-7100U-B</td>
<td>0xb4</td>
<td>2.0 GHz</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>i3-7100U-C</td>
<td>0xb4</td>
<td>2.0 GHz</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Kaby Lake-R</td>
<td>i7-8650U-A</td>
<td>0xb4</td>
<td>1.9 GHz</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>i7-8650U-B</td>
<td>0xb4</td>
<td>1.9 GHz</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>i7-8550U</td>
<td>0x96</td>
<td>2.6 GHz</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Coffee Lake-R</td>
<td>i9-9900U</td>
<td>0x0a</td>
<td>3.6 GHz</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

F. Implications for Older Processors

We verified that software-controlled undervolting is possible on older CPUs, e.g., on the Haswell i5-4590, Haswell i7-4790 and the Core 2 Duo T9550. In fact, it has been possible for system software to undervolt the processor from the first generation of Intel Core processors [51]. However, to the best of our understanding, this has no direct impact on security because SGX is not available and the attacker requires root permissions to write to the MSRs. The attack might nevertheless be relevant in a hypervisor or cloud setting, where an untrusted virtual machine can undervolt the CPU just before a hypcall and/or context switch to another VM. This attack scenario would require the hypervisor to be configured to allow the untrusted virtual machine to directly access undocumented MSRs (e.g., 0x150) and we did not find this in any real-world configurations. Consequently, for the lack of plausible attack targets, we did not extensively study the possibility of fault induction on these processors. Our initial undervolting testing yielded a voltage-dependent segmentation fault on the Haswell i5-4590 and Haswell i7-4790 for the simple test program described in Section III.

III. FAULTING IN-ENCLAVE MULTIPLICATIONS

As a first step towards practical fault injection into SGX enclaves, we analyzed a number of x86 assembly instructions in isolation. While we could not fault simple arithmetic (like addition and subtraction) or bit-wise instructions (like shifts and OR/XOR/AND), we found that multiplications can be faulted. This might be explained by the fact that, on the one hand, multipliers typically have a longer critical path compared to adders or other simple operations, and, on the other hand, that multiplications are likely to be most aggressively optimized due to their prevalence in real-world code. This conjecture is supported by the fact that we also
observed faults for other instructions with presumably complex circuitry behind them, in particular the AES-NI extensions (cf. Section IV-C).

Consider the following proof-of-concept implementation, which runs a simple multiplication (the given code compiles to assembly with `imul` instructions) in a loop inside an ECALL handler:

```c
uint64_t multiplier = 0x1122334455667788;
uint64_t var = 0xdeadbeef * multiplier;
while (var == 0xdeadbeef * multiplier) {
    var = 0xdeadbeef;
    var *= multiplier;
} var ^= 0xdeadbeef * multiplier;
```

Clearly, this program should not terminate. However, our experiments show that undervolting the CPU just before switching to the enclave leads to a bit-flip in `var`, typically in byte 3 (counting from the least-significant byte as byte 0). This allows the enclave program to terminate. The output is the XOR with the desired value, to highlight only the faulty bit(s). We observe that in this specific configuration the output is always `0x04 00 00 00`.

### A. Analysis of Undervolting Effects on Multiplications

Using MSR `0x198 (IA32_PERF_STATUS)`, we were able to read the voltage in normal operating mode and also record the voltage when a faulty result was computed. While we are aware that the measurements in this register might not be precise in absolute terms, they reflect the relative undervolting precisely. Figure 3 and Figure 4 show the measured relation between frequency, normal voltage (blue), and the necessary undervolting to trigger a faulty multiplication inside an SGX enclave (orange) for the i3-7100U-A and an i7-8650U-A, respectively.

We conducted further investigations from normal (non-SGX) code, as we found that these faults were identical to those inside the SGX enclave. We wrote the following code to enable the first operand (`start_value`) and the second operand (`multiplier`) to be tested:

```c
/* drop voltage */
do {
    var = start_value * multiplier;
    i++;
} while (var == correct && i < iterations);
/* return voltage */
```

We then performed a search over different values for both operands. The faulty results (see Table II for selected examples) generally fell into the following categories:

- One to five (contiguous) bits flip, or
- all most-significant bits flip.

Additionally, we also *rarely* observed faulty states in between, cf. the last entry in Table II and the fault used in Section V-A. From those results, we noted:

- The smallest first operand to fault was `0x89af`;
- the smallest second operand to fault was `0x1`;
- the smallest faulted product was `0x80000 * 0x4`, resulting in `0x200000`; and
- the order of the operands is important when attempting to produce a fault: For example, `0x4 * 0x80000` never faulted in our experiments.

### Table II

tabular data

<table>
<thead>
<tr>
<th>Start value</th>
<th>Multiplier</th>
<th>Faulty result</th>
<th>Flipped bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800004</td>
<td>0x0008</td>
<td>0xfffffffff0400020</td>
<td>0xfffffffff0000000</td>
</tr>
<tr>
<td>0x27fccc</td>
<td>0x0335</td>
<td>0x0000000204dbb43c</td>
<td>0x000000001000000</td>
</tr>
<tr>
<td>0x95ff4f</td>
<td>0x00b2</td>
<td>0xfffffffff0000000</td>
<td>0xfffff000000000</td>
</tr>
<tr>
<td>0x2bffc0</td>
<td>0x0008</td>
<td>0xfffffffff0000000</td>
<td>0xfffffffff0000000</td>
</tr>
<tr>
<td>0x9f13</td>
<td>0x0008</td>
<td>0xfffffffff0000000</td>
<td>0xfffffffff0000000</td>
</tr>
<tr>
<td>0x2bff00</td>
<td>0x0008</td>
<td>0xfffffffff0000000</td>
<td>0xfffffffff0000000</td>
</tr>
</tbody>
</table>

We also investigated the iterations and undervolting required to produce faults (cf. Table III) on the i3-7100U-B at 2 GHz. A higher number of iterations will fault with less undervolting, *i.e.*, the probability of a fault is lower with less undervolting. For a small number of iterations, it is very difficult to induce a fault, as the undervolting required caused the CPU to freeze.
before a fault was observed. For the experiments in Fig. 3 and Fig. 4, we used a large number of 100,000,000 iterations, so faults occur with relatively low undervolting already.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Undervolting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000,000</td>
<td>-130mV</td>
</tr>
<tr>
<td>100,000,000</td>
<td>-131mV</td>
</tr>
<tr>
<td>10,000,000</td>
<td>-132mV</td>
</tr>
<tr>
<td>1,000,000</td>
<td>-141mV</td>
</tr>
<tr>
<td>500,000</td>
<td>-146mV</td>
</tr>
<tr>
<td>100,000</td>
<td>crash at -161mV</td>
</tr>
</tbody>
</table>

B. Differences between CPUs with Same Model Number and Temperature Dependencies

Another interesting observation is that the amount of undervolting can differ between CPUs with the same model number. We observed that the i3-7100U in an Intel NUC7i3BNH: i3-7100U-A had a base voltage of 0.78 V at 1 GHz, and we observed the first fault at 0.68 V (over 100,000,000 iterations). In contrast, two other (presumably slightly newer) CPUs i3-7100U-B and i3-7100U-C had a base voltage of approximately 0.69 V at the same frequency and began to fault at 0.6 V.

However, the processor with the higher base voltage tolerated more undervolting overall: the system was stable undervolting up to approximately -250 mV, while the other CPUs crashed at around -160 mV. This indicates that for certain CPUs, a higher base voltage is configured (potentially in the factory based on internal testing).

Finally, we observed that the required undervolting to reach a faulty state depends (as expected) on the CPU temperature. For example, while the i3-7100U-A reliably faulted at approximately -250 mV with a CPU temperature of 47°C, an undervolting of -270 mV was required to obtain the same fault at 39°C. While we have not investigated this behaviour in detail, we note that the temperature dependency and possible differences in “stability” of the fault warrant further investigation. All our attacks were performed at room temperature and caused no impediments.

C. Overvolting

The VID interface specification limits the maximum voltage to 1.52 V. According to the CPU datasheets, this voltage is within the normal operating region. We experimentally confirmed that we could not increase the voltage beyond 1.516 V (even with a higher value in the MSR), and we did not observe any faults at 1.516 V at any frequency on i3-7100U-A.

IV. FROM FAULTS TO ENCLAVE KEY EXTRACTION

Having demonstrated the feasibility of fault injection into SGX enclaves in Section III, we apply the undervolting techniques to cryptographic libraries used in real-world enclaves.

A. Corrupting OpenSSL Signatures

We first developed a simple proof-of-concept application using OpenSSL in userspace. This application runs the multiplication loop from Section III until the first fault occurs (to make sure the system is in a semi-stable state) and then invokes OpenSSL as follows:

```
system("openssl dgst -sha256 -sign private.pem test.c | openssl base64 >> log.txt");
```

Running at the standard voltage, this proof-of-concept outputs a constant signature. Running with undervolting (on the i3-7100U-A at 1 GHz, -230 mV was sufficient), this generated incorrect, apparently randomly changing signatures. While we have not exploited this fault to factor the RSA key, this motivating example shows that undervolting can successfully inject faults into complex cryptographic computations, without affecting overall system stability.

B. Full Key Extraction from RSA-CRT Decryption/Signature in SGX using IPP Crypto

The tcrypto API of the Intel SGX-SDK only exposes a limited number of cryptographic primitives. However, the developer can also directly call IPP Crypto functions when additional functionality is needed. One function that is available through this API is decryption or signature generation using RSA with the frequently used Chinese Remainder Theorem (CRT) optimization. In the terminology of IPP Crypto, this is referred to as “type 2” keys initialized through ippsRSA_InitPrivateKeyType2(). We developed a proof-of-concept enclave based on Intel example code 32.

Given an RSA public key \((n, e)\) and the corresponding private key \((d, p, q)\), RSA-CRT can speedup the computation of \(y = x^d \pmod{n}\) by a factor of around four. Internally, RSA-CRT makes use of two sub-exponentiations, which are recomposed as:

\[
y = [q \cdot c_p] \cdot x_p^{d_p} + [p \cdot c_q] \cdot x_q^{d_q} \pmod{n}
\]

where \(d_p = d \pmod{p-1}\), \(d_q = d \pmod{q-1}\), \(x_p = x \pmod{p}\), \(x_q = x \pmod{q}\), and \(c_p, c_q\) are pre-computed constants.

RSA-CRT private key operations (decryption and signature) are well-known to be vulnerable to the Bellcore and Lenstra fault-injection attacks [9], which simply require a fault in exactly one of the two exponentiations of the core RSA operation without further requirements to the nature or location of the fault. Assuming that a fault only affects one of the two sub-exponentiations \(x_p^{d_p}\) and given the respective faulty output \(y'\), one can factor the modulus \(n\) using the Bellcore attack as:

\[
q = \gcd(y - y', n), p = n/q
\]
The Lenstra method removes the necessity to obtain both correct and faulty output for the same input $x$ by computing $q = \gcd((x')^e - y, n)$ instead.

As a first step to practically demonstrate this attack for SGX, we successfully injected faults into the `ippsRSA_Decrypt()` function running within an SGX enclave on the i3-7100U-A, undervolting by -225 mV for the whole duration of the RSA operation. However, this resulted in non-exploitable faults, presumably since both sub-exponentiations had been faulted. We therefore introduced a second thread (in the untrusted code) that resets the voltage to a stable value after one third of the overall duration of the targeted ECALL. With this approach, the obtained faults could be used to factor the 2048-bit RSA modulus using the Lenstra and Bellcore attacks, and hence to recover the full key with a single faulty decryption or signature and negligible computational effort. An example for faulty RSA-CRT inputs and outputs is given in Appendix [8]

C. Differential Fault Analysis of AES-NI in SGX

Having demonstrated the feasibility of enclave key-extraction attacks for RSA-CRT, we turn our attention to Intel AES New Instructions (AES-NI). This set of processor instructions provide very efficient hardware implementations for AES key schedule and round computation. For instance, on the Skylake architecture, an AES round instruction has a latency of only four clock cycles and a throughput of one cycle per instruction. AES-NI is widely used in cryptographic libraries, including SGX’s `crypto` API, which exposes functions for AES in Galois Counter Mode (GCM), normal counter mode, and in the CMAC construction. These crypto primitives are then used throughout the Intel SGX SDK, including crucial operations like sealing and unsealing of enclave data. Other SGX crypto libraries (e.g., mbedtls in Microsoft OpenEnclave) also make use of the AES-NI instructions.

Our experiments show that the AES-NI encryption round instruction `aesenc` is vulnerable to Plundervolt attacks: we observed faults on the i7-8650U-A with -195 mV undervolting and on the i3-7100U-A with -232 mV undervolting.

The faults were always a single bit-flip on the leftmost two bytes of the round function’s output. Such single bit-flip faults are ideally suited for Differential Fault Analysis (DFA). Examples of correct and faulty output are:

Next, we use these single-round faults to build an enclave key-recovery attack against the full AES. We took a canonical AES implementation using AES-NI instructions [3] and ran it in an enclave with undervolting as before. Unsurprisingly, the probability of a fault hitting a particular round instruction is approx. $\frac{1}{128}$, which suggests a uniform distribution over each of the ten AES rounds. By repeating the operation often enough (5 times on average) we get a fault in round 8. An example output for this (using the key $0x000102030405060708090a0b0c0d0e0f$) is the following:

| [Enclave] plaintext: 5ABB97CCF5081A4596A0E1CEF1BC39 |
| [Enclave] CT1: DE49E9204625F7D98764595E814C4 <- faulty |
| [Enclave] CT2: BDFACDE333976AD558B1D718DFC405A <- correct |

In order to understand the fault (the following profiling is not part of the actual attack and only needs to be done once), we took both correct and faulty ciphertexts and decrypted them round-by-round while comparing the intermediate states. The result can be seen in the above output: Observe that byte one (counting from the left in the rightmost word) in round 8 has changed from $0x66$ to $0x3E$. This faulty byte is actually caused by an XOR with $0x02$ (i.e., a single-bit flip) for state byte one after SubBytes in round 8. We established this by simulating the AES invocation and trying different fault masks. Equipped with this fault in round 8, we were able to apply the differential fault analysis technique by Tunstall et al. [68] as implemented by Jovanović [3].

Given a pair of correct and faulty ciphertext on the same plaintext, this attack is able to recover the full 128-bit AES key with a computational complexity of only $2^{32}+2^{56}$ encryptions on average. We have run this attack in practice and it took only a couple of minutes to extract the full key of the enclave, including both fault injection and key computation phases. The steps to reproduce this attack with the above pair of correct and faulty ciphertexts are given in Appendix [1].

D. Faulting Intel SGX’s Key Derivation Primitives

Finally, we investigated whether we can successfully apply our undervolting techniques to inject faults in Intel SGX’s hardware-level key derivation instructions. These primitives form the basis for local and remote attestation, as well as sealing, and are indispensable to bootstrap trust in the SGX ecosystem [1]. As with most of SGX’s trusted computing base, complex key derivation functionality is implemented in microcode [13], and, according to an Intel patent [47], may leverage the processor’s native AES-NI instructions to accelerate some of the cryptographic operations. Hence, our hypothesis is that we can produce incorrect key derivations through an Plundervolt attack. While this in itself does not directly break SGX’s security features (the attestation will

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=233&text=mm_aesenc_si128

https://gist.github.com/acapola/d5b940da024080dfaf5f

https://github.com/Daeinar/dfa-aes
simply fail), faulty key derivations may, in turn, reveal information about the processor's long-term key material that should never be exposed to software. In this section, we merely want to show that even complex microcode instructions can be successfully faulted. We leave further exploration and cryptanalysis of such faults as future work.

a) Faulting EGETKEY: Enclaves can make use of SGX’s key derivation facility by means of a dedicated EGETKEY instruction [11] [14]. This instruction derives an enclave-specific 128-bit symmetric key based on a hardware-level master secret, which is burned into efuses during the processor manufacturing process and never directly exposed to software. The exact key derivation algorithm implemented in the microcode is largely undocumented, but one Intel patent [47] reveals that AES-CMAC is used with a derivative string specifying, among others, a software-provided KeyID and the calling enclave’s identity. We furthermore confirmed that Intel’s official SGX software simulator [4] indeed relies on AES-CMAC with a fixed 128-bit secret for key derivations.

Our experimental setup consists of a minimal attacker-controlled enclave that first prepares a fixed key request and thereafter repeatedly derives the expected cryptographic key using the EGETKEY assembly instruction. We expect the derived key to be constant, since we made sure to always supply the exact same KeyID meta data. However, our experiments on the i3-7100U-C running at 2 GHz with -134 mV under-volting showed that Plundervolt can reliably fault such SGX key derivations. We provide several samples of incorrectly derived keys in Appendix F. Interestingly, we noticed that key derivation faults appear to be largely deterministic. That is, for a fixed KeyID, the same (wrong) key seems to be produced most of the time when undervolting, even across reboots. However, we also observed, at least once, that two different faulty keys can be produced for the same KeyID, cf. Appendix F.

b) Faulting EREPORT: SGX supports local attestation through the EREPORT primitive. This instruction can be invoked by a client enclave to create a tagged measurement report destined for another target enclave residing on the same platform. For this, EREPORT first performs an internal key derivation to establish a secret key that can only be derived by the intended target enclave executing on the same processor. This key is thereafter used in the EREPORT microcode to create a 128-bit AES-CMAC that authenticates the report data. We experimentally confirmed that Plundervolt can indeed reliably induce faults in local attestation report MACs. We provide a few samples of faulty report MACs in Appendix F. As with the EGETKEY experiments above, we noticed that the faulty MACs appear to be deterministic. However, faulty MACs do change across reboots as EREPORT generates an internal random Key ID on every processor cycle.

E. Faulting Other Intel IPP Crypto Primitives in SGX

In addition to the above key extractions from RSA-CRT and AES-NI, we applied the undervolting technique to a number of enclaves using other crypto APIs. We successfully injected faults into the following primitives among others:

AES-GCM In certain cases, faults in sgx_rijndael128GCM_encrypt() only affect the MAC, aside from our results on AES-NI in Section IV-C. Note that DFA is not directly applicable to AES in GCM mode, since it is not possible (if used correctly) to get two encryptions with the same nonce and plaintext.

Elliptic Curves We also observed faults in elliptic curve signatures (sgx_ecdsa_sign()) and key exchange (sgx_ecc256_compute_shared_dhkey()).

This list of cryptographic fault targets is certainly not exhaustive. We leave the examination of fault targets for Plundervolt, as well as the evaluation of their practical exploitability for future work, which requires pinpointing the fault location and debugging IPP crypto implementations. There is a large body of work regarding the use of faults for key recovery that could be applicable once the effect of the fault for each implementation has been precisely understood. Fan et al. [17] provide an overview of fault attacks against elliptic curves, while other researchers [18] [15] discuss faults in nonce-based encryption modes like AES-GCM.

V. MEMORY SAFETY VIOLATIONS DUE TO FAULTS

In addition to the extraction of cryptographic keys, we show that Plundervolt can also cause memory safety misbehavior in certain situations. The key idea is to abuse the fact that compilers often rely on correct multiplication results for pointer arithmetic and memory allocation sizes. One example for this would be indexing into an array a of type elem_t: according to the C standard, accessing element a[i] requires calculating the address at offset i*sizeof(elem_t). Clearly, out-of-bounds accesses arise if an attacker can fault such multiplications to produce address offsets that are larger or smaller than the architecturally defined result (cf. Section III). Note that Plundervolt ultimately breaks the processor’s ISA-level guarantees, i.e., we assume perfectly secure code that has been guarded against both traditional buffer overflows [16] as well as state-of-the-art Spectre-style [42] transient execution attacks.

In this section, we explore two distinct scenarios where faulty multiplications impair memory safety guarantees in seemingly secure code. First, we fault imul instructions transparently emitted by the compiler to reliably produce out-of-bounds array accesses. Next, we analyze trusted SGX runtime libraries and locate several sensitive multiplications in allocation size computations that could lead to heap corruption by allocating insufficient memory.

A. Faulting Array Index Addresses

We first focus on the case where a multiplication is used for computing the effective memory address of an array element as follows: &a[i] = &a[0] + i*sizeof(elem_t). However, we found that, in most cases, when the respective type has a size that is a power of two, compilers will use left bitshifts instead of explicit imul instructions. Furthermore,

as concluded from the micro-benchmark analysis presented in Section III we found it difficult (though not impossible) to consistently produce multiplication faults where both operands are **0xFFFF** without crashing the CPU (cf. Section V-B). Hence, here we only consider cases in this section where sizeof(elem_t) ≠ 2\(^{i}\) and \(i > 2^{16}\).

a) Launch Enclave Application Scenario: To illustrate that our attack requirements can be realistically met and exploited in compiler-generated enclave code, we constructed an example enclave application. Our application scenario takes advantage of the “flexible launch control” [39] features in the latest SGX processors via a custom Launch Enclave that decides which other enclaves are allowed to be loaded on the platform. We loosely based our implementation on the open-source reference launch enclave code (psw/ae/ref_le) provided by Intel as part of its SGX SDK [33]. For completeness, we refer to Appendix C for full source code and disassembly of the relevant functions.

Our custom Launch Enclave maintains a global fixed-length array of white-listed enclave authors. Each element in this array is a composite struct data type specifying the white-listed enclave author’s MRSIGNER hash, plus whether or not her enclaves are allowed access to the special platform “provisioning key”. The latter restriction relates to CPU tracking privacy concerns [14]. Specifically, the provisioning key is the only SGX key which is directly derived from a long-term platform-specific cryptographic secret, without first including an internal OWNREEPOCH register. Hence, the provisioning key remains constant as a processor changes owners, and is normally only available to privileged architectural enclaves that establish long-term platform attestation key material.

The security objective of our example scenario is to enforce a simple launch control policy: only enclave authors whose MRSIGNER value is present in the global white list are allowed to run production enclaves on the system (potentially with an additional restriction on the provisioning key attribute). For this, our sample Launch Enclave repeatedly calls a check_wl_entry() function in a for loop to look up and compare to every element in the global white list array. Note that our sample Launch Enclave merely returns a non-zero value if access is allowed, as we omitted the actual computation of the cryptographic launch token for simplicity. Evidently, after the global white list has been initialized to all zeroes, our Launch Enclave should never return 1 when looking up the adversary’s non-zero MRSIGNER value.

b) Launch Enclave Exploitation: Figure 5 visualizes the high-level attack flow in our application scenario. For exploitation, we first turn our attention to the check_wl_entry() function 1 which indexes into the global white list array. As evident from Appendix C this array access compiles to an lmul 50x21, %rdi, %rdi instruction, which calculates the required offset to be added to the array base address afterwards. In order to reliably fault 3 this product, the array index specified in the %rdi parameter needs to be sufficiently large (cf. Section III). Specifically, we experimentally established that a white list of about 530,000 entries suffices to reliably induce predictable faults in the multiplication result (the victim Launch Enclave almost always hits an exploitable fault in under 100 invocations). We noticed that Plundervolt frequently causes the higher-order bits to be flipped in the faulty product. For example, 0x80D36 * 0x21 = 0x109b3f6 predictably faults to 0xfffffffffe109b417 at 2 GHz and undervolting of -118 mV on the i3-7100U-C. Notice that flipping the most-significant bits effectively causes the resulting product to be interpreted as a large negative offset (in the order of magnitude of the computed number) relative to the trusted array base address. Hence, after adding the faulty product, the resulting address now points in the large untrusted 48-bit address space outside the enclave.

In the next stage of the attack, the victim enclave unknowingly dereferences the erroneous white list element pointer as if it was in-enclave memory. Since the virtual memory region before the enclave base has not been allocated, this access causes a page fault 5 to be delivered to the untrusted operating system. We installed a custom signal handler 4 which subsequently mmap()s the required memory page on demand. At this point, the adversary can setup a bogus white list entry with her own MRSIGNER hash and provisioning key attribute in attacker-controlled memory outside the enclave. Note, however, that SGX clears the least significant 12 bits in the reported page fault address to limit page fault side-channel exposure [78]. Hence, for a successful attack, we should still determine at which offset within the allocated page to place the bogus MRSIGNER value. To overcome this challenge, we observe that the multiplication bit-flips induced by Plundervolt are often very deterministic and predictable. That is, we noticed that frequently the exact same bits are flipped by applying a constant XOR mask to the expected product (e.g., 0xfffffffffe00007e1 in the above example). We thus conveniently pre-compute the expected fault mask by running identical code in our own debug enclave, so that we can afterwards accurately predict the page offset of the faulty address by XORing the precomputed mask with the correct product. As a final challenge for this to work, we still require
knowledge of the correct product, *i.e.*, the architecturally expected array offset depending on the current loop iteration in the enclave. However, this is actually a standard scenario in classical side-channel attack works, which accurately reconstruct enclave control flow by, for instance, monitoring page table access patterns [28] [73], cache accesses [58] or interrupt counts [71] [72]. To improve reproducibility, we disclose the current loop iteration in our current proof-of-concept attack code (Appendix E) providing the same information without noise.

Finally, after the bogus white list entry has been constructed in untrusted memory, the adversary merely resums the victim enclave. The latter will now proceed and unknowingly dereferences the attacker-controlled memory page instead of the trusted white list entry in enclave memory. The attack is successfully concluded when the benign Launch Enclave eventually returns one after the adversary’s MRSIGNER and provisioning key values were successfully matched.

### B. Faulting Memory Allocation Sizes

Apart from array indices, we identified size computations for dynamic memory allocations as another common programming pattern that relies on correct multiplication results. We showed in Section III that *imul* can also be faulted to produce results that are smaller than the correct value. Clearly, heap corruption may arise when such a faulty multiplication result is used to allocate a contiguous chunk of heap memory that is smaller than the expected size. Since Plundervolt corrupts multiplications silently, *i.e.*, without failing the respective *malloc* library call, the client code has no means of determining the actual size of the allocated buffer and will subsequently read or write out-of-bounds.

- **a) edger8r-generated Code:** To ease secure enclave development, the official Intel SGX-SDK comes with a dedicated edger8r tool that generates trusted proxy bridge code to transparently copy user arguments to and from enclave private heap memory [35] [70]. The tool automatically generates C code based on the ECALL function’s prototype and explicit programmer annotations that specify pointer directions and sizes. Consider the following (simplified) example enclave code, where the [in,count] attributes are used to specify that *arr* is an input array with cnt elements:

```c
void vuln_ecall([in, count=cnt] struct foo_t *arr, size_t cnt, size_t offset) {
  if (offset >= cnt) return;
  arr[offset].foo0 = 0xdeadbeef;
}
```

The edger8r tool will generate the following (simplified) trusted wrapper code for parameter checking and marshalling:

```c
... size_t _tmp_cnt = m * ms_cnt;
size_t _len_arr = _tmp_cnt + sizeof(struct foo_t);
... _in_arr = (struct foo_t*)malloc(_len_arr);
... vuln_ecall(_in_arr, _tmp_cnt);
```

The above code first computes the expected size _len of the input array, allocates sufficient space on the enclave heap, and finally copies the input array into the enclave before invoking the programmer’s *vuln_ecall()* function. Crucially, if a multiplication fault occurs during calculation of the _len variable, a potentially smaller buffer will be allocated and passed on to the actual ECALL function. Any subsequent writes or reads to the allocated buffer may cause inadvertent enclave heap corruption or disclosure. For example, the above *vuln_ecall()* implementation is safeguarded against overflows in a classical sense, but can trigger a heap overflow when the above multiplication is faulted and *arr* is smaller than expected.

For the type used in this example, we have sizeof(struct foo_t) = 0x64. We performed initial testing based on our micro-benchmark from Section III, which established a predictable fault for this parameter, and verified that the enclave indeed corrupts trusted heap memory when computing on a buffer with the faulty size. Specifically, we found that the multiplication 0x08b864 * 0x64 = 0x3680710 reliably faults to a smaller result 0x1680710 with an undervolting of -250 mV on our i3-7100U-A system.

For convenience during exploit development, we artificially injected the same fault at compile time by changing the generated edger8r code from the Makefile.

- **b) calloc() in SGX Runtime Libraries:** Another possible target for fault injection is the hidden multiplication involved in calls to the prevalent *calloc()* function in the standard C library. This function is commonly used to allocate memory for an array where the number of elements and the size of each element are provided as separate arguments. According to the *calloc()* specification, the resulting buffer will have a total size equal to the product of both arguments if the allocation succeeds. Note that optimizations of power-of-two sizes to shifts are not applicable in this case, since the multiplication happens with generic function parameters.

Consider the following *calloc()* implementation from musl-libc, an integral part of the SGX-LKL [54] library OS for running unmodified C applications inside enclaves:

```c
void *calloc(size_t m, size_t n) {
  if (n && m > (size_t)-1/n) {
    errno = ENOMEM;
    return 0;
  }
  n *= m;
  void *p = malloc(n);
  ...
}
```

In this case, if the product n * m can be faulted to produce a smaller result, subsequent code may trigger a heap overflow, eventually leading to memory leakage, corruption, or possibly even control flow redirection when neighbouring heap chunks contain function pointers *e.g.*, in a vtable. Based on practical experiments with the i3-7100U-A, we artificially injected a realistic fault for the product 0x2bffc0 * 0x8 = 0x15ffe00

[https://github.com/lsds/sgx-lkl-musl/blob/d8c09/src/malloc/malloc.c#L352](https://github.com/lsds/sgx-lkl-musl/blob/d8c09/src/malloc/malloc.c#L352)
via code rewriting in SGX-LKL’s musl-libc to cause an insufficient allocation of 0x5ffe00 bytes and a subsequent heap overflow in a test enclave.

We also investigated calloc() implementations in Intel’s SGX SDK [35] and Microsoft’s OpenEnclave [48], but interestingly found that their implementations are hardened against (traditional) integer overflows as follows:

```c
if (n_elements != 0) {
    req = n_elements * elem_size;
    if (((n_elements | elem_size) & ~(size_t)0xffff))
        req = MAX_SIZE_T; /* force downstream failure on overflow */
}
```

Note how the above code triggers a division (that would detect the faulty product) if at least one of n_elements and elem_size is larger than 0xffffffff. Producing faults where both operands are \( \leq 0xffffffff \) (cf. Section III) is possible, e.g., we got a fault for 0x97b5 * 0x40 on the i3-7100U-A. However, in the majority of attempts, this leads to a crash because the CPU has to be undervolted to the point of becoming unstable. The above check (without the restriction on only being active for at least operand being > 0xffffffff) serves as an example of possible software hardening countermeasures, as discussed in Section VII.

VI. DISCUSSION AND RELATED WORK

Compared to widely studied fault injection attacks in cryptographic algorithms, memory safety implications of faulty instruction results have received comparatively little attention. In the context of physically injected faults, Govindavajhala et al. [20] demonstrated how a single-bit memory error can be exploited to achieve code execution in the Java/.NET VM, using a lightbulb to overheat the memory chip. Barbu et al. [5] used laser fault injection to bypass a type check on a Javacard and load a malicious applet afterwards. In the context of software-based Rowhammer attacks, on the other hand, Seaborn and Dullien [60] showed how to flip operand bits in x86 instruction streams to escape a Native Client sandbox, and more recently Gruss et al. [21] flipped opcode bits to bypass authentication checks in a privileged victim binary. While flipping bits in instruction opcodes enables the application control flow to be illegally redirected, none of these approaches directly produce incorrect computation results. Furthermore, Rowhammer attacks originate outside the CPU package and are hence fully mitigated through SGX’s memory integrity protection [24], which reliably halts the system if an integrity check fails [21, 38].

To the best of our knowledge, we are the first to explore the memory safety implications of faulty multiplications in compiler-generated code. Compared to prior work [65] that demonstrated frequency scaling fault injection attacks against ARM TrustZone cryptographic implementations, we show that undervolting is not exclusively a concern for cryptographic algorithms. As explored in the following Section VII this observation has profound consequences for reasoning about secure enclave code, i.e., merely relying on fault-resistant cryptographic primitives is insufficient to protect against Plundervolt adversaries at the software level.

While there is a long line of work on dismantling SGX’s confidentiality guarantees [69, 11, 40, 73, 50, 25, 72] as well as exploiting classical memory safety vulnerabilities in enclaves [45, 8, 70], Plundervolt represents the first attack that directly violates SGX’s integrity guarantees for functionally correct enclave software. By directly breaking ISA-level processor semantics, Plundervolt ultimately undermines even relaxed “transparent enclave execution” paradigms [66] that solely require integrity of enclave computations while assuming unbounded side-channel leakage.

The differences and ramifications of violating integrity vs. confidentiality guarantees for enclave computations can often be rather subtle. For instance, the authors of the Foreshadow [69] attack extracted enclave private sealing keys (confidentiality breach), which subsequently allowed an active man-in-the-middle position to be established - enabling all traffic to be read and modified from an enclave (integrity breach). Likewise, we showed that faulty multiplications or encryptions can lead to unintended disclosure of enclave secrets. Our Launch Enclave application scenario of Section V-A is another instance of the tension between confidentiality and integrity. That is, the aforementioned Foreshadow attack showed how to bypass enclave launch control by extracting the platform’s “launch key” needed to authenticate launch tokens, whereas our attack intervened much more directly with the integrity of the enclave execution by faulting pointer arithmetics and redirecting the trusted white list into attacker-controlled memory.

VII. COUNTERMEASURES

In Intel SGX’s threat model, the operating system is considered untrusted. However, we showed that while an enclave is running, privileged adversaries can manipulate MSR 0x150 and reliably fault in-enclave computations. Hence, countermeasures cannot be implemented at the level of the untrusted OS or in the untrusted runtime components. Instead, two possible approaches to mitigating Plundervolt are possible: preventing unsafe undervolting directly at the level of the CPU hardware and microcode, or hardening the trusted in-enclave code itself against faults. Respective methods can be used separately or—to increase the level of protection—in combination, as is common practice for high-security embedded devices like smartcards.

In the following, we first overview potential approaches to mitigate Plundervolt attacks at the hardware and software levels. Next, we conclude this section by summarizing the specific mitigation strategy adopted by Intel.

A. Hardware-Level and Microcode-Level Countermeasures

a) Disabling MSR Interface: Given the impact of our findings, we recommend initiating SGX trusted computing base recovery by applying microcode updates that completely disable the software voltage scaling interface exposed via MSR 0x150. However, given the apparent complexity of dynamic
Completely removing this feature might incur substantial computing and for other end user applications like gaming. It is clear that properties that are highly desirable in data centers, for mobile voltting the CPU creates significant power and heat reductions; however, this does not come without a substantial overhead. TSC cycles, cf. Fig. 2). Therefore, removing and restoring a substantial overhead.

Alternatively, advanced adversaries could even replace the voltage regulator completely with a dedicated voltage glitcher (although this may be technically non-trivial given the required large currents).

b) Scaling Back Voltage during Enclave Operation: Plundervolt relies on the property that CPU voltage changes outside of enclave mode persist during enclave execution. A straw man defense strategy could be to automatically scale back any applied undervolting when the processor enters enclave mode. Interestingly, we noticed that Intel seems to have already followed this path for its (considerably older) TXT trusted computing extensions. In particular, the documentation of the according SENTER instruction mentions that [36, 6-21]: “Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. […] the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner.”

However, we make the crucial observation that this defense strategy does not suffice to fully safeguard Intel SGX enclaves. That is, in contrast to Intel TXT which transfers control to a measured trusted environment, SGX features a more dynamic design where attacker code and trusted enclaves are interfaced at runtime. Hence, while one core is in enclave mode, another physical core could attempt to trigger the undervolting for the shared voltage plane in parallel after entering the victim enclave. Therefore, such checks would need to be continuously enforced every time any core is in enclave mode. This defense strategy is further complicated by the observation that the time between a write to MSR 0x150 and the actual voltage change manifesting is relatively large (order of magnitude of 500k TSC cycles, cf. Fig. 2). Therefore, removing and restoring undervolting on each enclave entry and exit would likely add a substantial overhead.

c) Limiting to Known Good Values: Even slightly undervolting the CPU creates significant power and heat reductions; properties that are highly desirable in data centers, for mobile computing and for other end user applications like gaming. Completely removing this feature might incur substantial limitations in practice. As an alternative solution, the exposed software interface could be adjusted to limit the amount of permitted undervolting to known “safe” values whitelisted by the processor. However, this mitigation strategy is further complicated by our observations that safe voltage levels depend on the current operating frequency and temperature and may even differ between CPUs of the same model (cf. Section III-A). Hence, establishing such safe values would require a substantial amount of additional per-chip testing at each frequency. Even then, circuit-aging effects can affect safe values as the processor gets older [40].

d) Multi-variant Enclave Execution: A perpendicular approach, instead of trying to prevent undervolting faults directly, would be to modify processors to reliably detect faulty computation results. Such a defense may, for instance, leverage ideas from multi-variant execution [28, 76, 43] software hardening techniques. Specifically: processors could execute enclave computations twice in parallel on two different cores and halt once the executions diverge. To limit the performance penalty of such an approach, we propose leveraging commodity HyperThreading [36] features in Intel CPUs and turn them from a security concern into a security feature for fault resistancy. After a long list of SGX attacks [69, 75, 59, 50] demonstrated how enclave secrets can be reconstructed from a sibling CPU core, Intel officially recommended disabling hyperthreading when using SGX enclaves [32]. However, this also imposes a significant performance impact on any non-SGX workloads.

A well known solution to fault injection attacks is redundancy [4], either in hardware, by duplicating potentially targeted circuits, or in software by duplicating potentially targeted parts of the instruction stream, and frequently checking for mismatches in both cases. For instance, Oh et al. [52] and later Reis et al. [56] proposed duplicating the instruction stream to produce software that is tolerant against hardware-induced faults. In the case of SGX, such a solution might also be applied at the microarchitectural level. The processor would simply run the duplicated instructions in parallel on the two hyperthreads of a core. Faults would be reliably detected if the probability that the attacker induced the exact same fault in two immediately subsequent executions of the same instructions is significantly lower than the probability of observing a single fault at some point in time.

B. Software-Level Hardening  

a) Fault-Resistant Cryptographic Primitives: There is a large body of work regarding fault injection countermeasures for cryptographic algorithms, including (generic) temporal and/or spatial redundancy [26] and algorithm-specific approaches such as performing the inverse operation or more advanced techniques like ineffective computation [19].

For the example of RSA-CRT signature/decryption (cf. Section IV-B), the result could be verified before outputting by performing a (in the case of RSA with small public exponent) cheap signature verification/encryption operation. Indeed, such a check is present by default in some cryptographic libraries, e.g., mbedtls. However, for the Intel SGX-SDK this might require changes to the API specification of tcrypto, as the public key is currently not supplied as a parameter to private key operations.
For AES-NI (cf. Section IV-C), an encryption operation could be followed by a decryption to verify that the plaintext remains unchanged. However, this would incur substantial performance overhead, doubling the runtime of an encryption. Trade-offs like storing the intermediate state after \( k \) rounds and then only performing \( 10 - k \) inverse rounds (for AES-128) can defeat DFA but might still be susceptible to statistical attacks [13].

b) Application and Compiler Hardening: It is important to note that SGX supports general-purpose, non-cryptographic code that can also be successfully exploited with Plundervolt, as demonstrated in Section V. To further complicate matters, typical enclave runtime libraries contain numerous, potentially exploitable \texttt{mul} and \texttt{imul} instructions. For instance, we found that the trusted runtime code for a minimalistic enclave using the Intel SGX-SDK [35] contains 23 multiplications, with many in standard library functions like \texttt{free()}. For comparison, the trusted runtime part of Microsoft’s OpenEnclave SDK [48] contains 203 multiplications, while Graphene-SGX’s [67] \texttt{libpal-Linux-SGX.so} features 71 \texttt{mul/imul} instructions.

Certain standard library functions like \texttt{calloc()} could be hardened manually by inserting checks for the correctness of a multiplication, e.g., through a subsequent division, as already implemented in the Intel SGX-SDK (see Section V-B). However, in functions where many “faultable” multiplications are being used (e.g., public-key cryptography, signal processing, or machine learning algorithms), this would incur significant overhead. Furthermore, each case of a problematic instruction needs to be analyzed separately, often at the assembly level to understand the exact consequences of a successful fault injection. Finally, it should be noted that while we have focused on multiplications in our analysis, defenses should also take into account the possibility of faulting other high-latency instructions.

c) Traditional Memory Safety Hardening: As a final consideration, we recommend applying more general countermeasures known from traditional memory safety application hardening [16] in an enclave setting. One approach to hinder Plundervolt-induced memory safety exploitation would be to randomize the enclave memory layout using systems like SGX-Shield [61]. Yet, it is important to note that these techniques can only raise the bar for actual exploitation, without removing the actual root cause of the attack.

C. Intel’s Mitigation Plan

Following the responsible disclosure, Intel’s Product Security Incident Response Team informed us of their mitigation plans with the following statement:

“This research is partially funded by the Research Fund KU Leuven, and by the Agency for Innovation and Entrepreneurship (Flanders). Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1, and by the European Union’s Horizon 2020 research and innovation programme under grant agreements No. 779391 (FutureTPM) and No. 681402 (SOPHIA).”
APPENDIX A
SCRIPT FOR CONFIGURING CPU FREQUENCY

#!/bin/bash
if [ $# -ne 1 ]; then
echo "incorrect number of arguments" >&2
echo "Usage: $0 <frequency>" >&2
echo "Example: $0 1.6GHz" >&2
exit
fi
sudo cpupower -c all frequency-set -u $1
sudo cpupower -c all frequency-set -d $1

APPENDIX B
EXAMPLE FAULT FOR RSA-CRT

The following 2048-bit RSA key was taken from the Intel example code:

n = 0xBBF82F090682CE9C2338AC2B9DA871F7368D07E
6D41043A40D6BFF745F51FBBDBFA0F3C02A61E4A
8BCEBE6FC4D876ED520D0E1EC4619719D8A5BBBFB7FAF
8E0A3DFC773723E6B4B7D93A2584EEA649D060953748
834B2454598394EEAAB12DB76A51F5279A41F6C1687
FE2537298CA2AFF5946F65F8DF091DBDCB

\[ n = (p-1)(q-1) + \lambda(p,q) \times \text{ecdf} + \text{gcd} + \text{gcd}^{-1} \times \text{gcd} \]

\[ \text{gcd}(p,q) = \text{gcd}(p - q, q - p) \]
\[ e = 0x11 \]
\[ d = Ax5dAFC5341FAF289C4B988BB3C1DF83F31251E4B6B4A7813801579461B29410B37C9989D6BC45745E5C932696B7D07AA20849E7FDB82EC93EDAC9FF3A5D95ACCBF11C76FA1925494E56A5AFA68C56C902D380BC3BEF5D2D0A939926EDF74A13EDDFBE1E1ACEC4894AF9428C87B883EFE463348C85B1C3C1

The following ciphertext \( x \) decrypts to \( y = x^d \) (mod \( n \)): 
\[ x = 0x1253E04DC0A5397BBA47AB87E9BF2A039A33D1E996FC82A949CD03704C3DF6732107695265DA1C0B4F782FCF653C11DF82314A6796DF8AEEA28DEF04BBDB84B3C1D651A9170E583B6DE9B0A242C2AFA90FEF2EF59C140E5BB84D9A536AD8700C84F19330ADE7A45E58D15A74DFDB58B68DE68383D636E9055
\]
\[ y = 0xEB7A19AECE9E03635E62954B45E2CA23B106D8D75C68F1EEA8F55267C31BE2B84251F7E0B2C046265F5AFF93EDCBFB252923BF8EA10E839A2DDB4CDCF4EF4F7728B8A1B7C11E2A29A84B82D869D504212143S1811591E392F82FB3EB7DD95AE4B448DB9723F3AC4F7BC725195281C3E2DF1B76D353ED32E

Injecting a fault during the first half of the RSA-CRT computation on the i3-7100U-A at 1 GHz with -225 mV undervolting, the following faulty \( y \) was one of our experiments:
\[ y = 0xAA105EAFB6BDD99E5A1544729670B7F042889103E0232428F37B1CEFFAEECC912972762562016EA5955DFDAA6FD5B685A0E2A32D9E99F51637078ED24EF48A3C313E40265DE6347267A83C92C45A72EA9F4DF8C2B378BCE70047C922A108807197A68BC1A5DF31E06FCD552A
\]

APPENDIX C
FURTHER EXAMPLES FOR AES-NI EASCU FAULTS

[Enclave] plaintext: 4C96DD4E44B4278E6F49FCFC8FCFF5C9
[Enclave] ciphertext 2: A2A756F8BBE848CA125E110507DC2E0E
[Enclave] ciphertext 1: A2A556F8BBE848CA125E110507DC2E0E
[Enclave] ciphertext 2: 0D40753C23026D11884385F373EAC66C
[Enclave] round key: BE7ED6DB9171EBBF9EA51569425D6DDE
[Enclave] plaintext: 03E023428F37B1CEFFAECC912972762562016EAA5955DF

Based on the fault described in Section IV-C, the input file data.txt should contain the following line:

BDFADEC33337976A533BDBD178DFC454DAE49E9284A625F72DB87B84A559E814C4

This fault was obtained on the i7-8650U-A with -195 mV undervolting at 1.9 GHz.

We ran the DFA implementation on four cores, knowing that the DFA implementation in byte one as follows:

```plaintext
./dfa 4 fault.txt
```

This yields 595 key candidates for this particular example, including the correct secret key value \(0x000012030405060908090a0b0cd0edef\).

APPENDIX E
EXAMPLES OF EGETKEY FAULTS

All samples in this appendix were collected on the i3-7100U-C running at 2 GHz with -134 mV of undervolting.

<table>
<thead>
<tr>
<th>KEY_ID</th>
<th>1966dd5d49f56811ae77074bf4522</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_OK</td>
<td>9edad757b4bbef29e0833f4fd40df8f7</td>
</tr>
<tr>
<td>KEY_ID</td>
<td>745a2d054b5bfe72542c1bdc502f7a5d</td>
</tr>
<tr>
<td>KEY_OK</td>
<td>726210dc53f6f22ba24e79be5f375f42</td>
</tr>
<tr>
<td>KEY_ID</td>
<td>421f9db6c76bb6ca297aefc9d49c0b6</td>
</tr>
<tr>
<td>KEY_OK</td>
<td>6049723ac4f57eb7128cedeb17be6a6</td>
</tr>
<tr>
<td>KEY_ID</td>
<td>3ae1ed2d58729db1e4edf53a882a3f9</td>
</tr>
</tbody>
</table>

The following correct/faulty values were obtained for a fixed key ID (a47171...). The first faulty key (760e5d...) was observed numerous times over different runs, while the second faulty value (37535d...) only occurred in one experiment.

<table>
<thead>
<tr>
<th>KEY_ID</th>
<th>a4717170f327e5f4a4f021a3eb3b6a8</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_OK</td>
<td>11feac2c14125fd1d1e205ca3d643be</td>
</tr>
<tr>
<td>KEY_ID</td>
<td>760e5d0a5d4e4ea6b585f58c42b62c</td>
</tr>
<tr>
<td>KEY_OK</td>
<td>a4717170f327e5f4a4f021a3eb3b6a8</td>
</tr>
<tr>
<td>KEY_ID</td>
<td>11feac2c14125fd1d1e205ca3d643be</td>
</tr>
<tr>
<td>KEY_OK</td>
<td>375354d1f213d9291cc931af7c08</td>
</tr>
</tbody>
</table>

APPENDIX F
EXAMPLES OF EREPORT FAULTS

All samples in this appendix were collected on the i3-7100U-C running at 2 GHz with -134 mV of undervolting.

```plaintext
=== Local attestation REPORT: 'REPORT_FAULT' ===
KEY_FAULT = 37535d4ff210de92917cc931a1fe7c08
KEY_ID = a4717170f327e5f4a4f021a3eb3b6a8
=== Local attestation REPORT: 'REPORT_OK' ===
KEY_OK = 9ed9a757b4bbef29e0833f4fd40df8f7
KEY_ID = 745a2d054b5bfe72542c1bdc502f7a5d |
```

The correct key ID (760e5d...) was observed numerous times over different runs, while the second faulty value (37535d...) only occurred in one experiment.
APPENDIX G
REFERENCE LAUNCH ENCLAVE IMPLEMENTATION

In this appendix, we provide the full C source code and compiled assembly for the minimalist launch enclave application scenario presented in Section V-A. We loosely based our implementation on the open-source reference launch enclave code (psw/ae/ref_le) provided by Intel as part of its SGX SDK [35]. Our custom launch enclave enforces a simple launch control policy by only returning valid launch tokens for known enclave authors. Specifically, the enclave maintains a global fixed-length array of known enclave authors (identified by the respective MRSIGNER values) plus whether or not they are allowed access to the long-term platform provisioning key. After the global white list has been initialized to all zeroes, our implementation should never return 1.

/* Minimal example implementation based on <https://github.com/intel/linux-sgx/blob/master/psw/ae/ref_le/ref_le.cpp#L47> */

typedef struct _ref_le_white_list_entry_t {
  sgx_measurement_t mr_signer;
  uint8_t provision_key;
} ref_le_white_list_entry_t;

#define REF_LE_WL_SIZE 0x8D1EE

ref_le_white_list_entry_t g_ref_le_white_list_cache[REF_LE_WL_SIZE] = { 0 };

void init_wl(void)
{
  memset(g_ref_le_white_list_cache, 0x00, sizeof(ref_le_white_list_entry_t) * REF_LE_WL_SIZE);
}

int check_wl_entry(size_t idx, sgx_measurement_t *mrsigner, int provision)
{
  ref_le_white_list_entry_t *current_entry = &g_ref_le_white_list_cache[idx];
  if (memcmp(&(current_entry->mr_signer), mrsigner, sizeof(sgx_measurement_t)) == 0)
  {
    return (provision ? current_entry->provision_key : 1);
  }
  return 0;
}

int get_launch_token(size_t *it, sgx_measurement_t mrsigner, int provision)
{
  for (size_t i = 0; i < REF_LE_WL_SIZE; i++)
  {
    if (check_wl_entry(i, &mrsigner, provision))
    {
      return 1;
    }
  }
  /* For simplicity, we only return true or false and do not compute the actual launch token. */
  return 0;
}

For completeness, we also provide a disassembled version of the relevant check_wl_entry function, as compiled with gcc v7.4.0 (optimization level -Os):

check_wl_entry:
  imul $0x21,%rdi,%rdi
  push %rbp
  push %rbx
  lea g_ref_le_white_list_cache(%rip),%rbx
  mov $0x20,%edx
  sub $0x8,%rsp
  add %rdi,%rbx
  mov %rbx,%rdi
  callq memcmp
  xor %edx,%edx
  test %eax,%eax
  jne 1f
  test %edx,%edx
  jne 1f
  movzbl 0x20(%rbx),%edx
  1:
  mov %edx,%eax
  pop %rdi
  pop %rbx
  pop %rbp
  retq