Plundervolt

Murdock, Kit; Oswald, David; Garcia, Flavio; Van Bulck, Jo; Gruss, Daniel; Piessens, Frank

DOI:
10.1109/SP40000.2020.00057

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
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 [7, 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.

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,

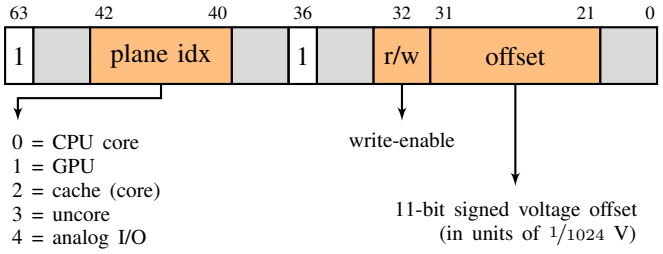

Fig. 1. Layout of the undocumented undervolting MSR with address `0x150`.

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-Step [71]. Since we only exploit software-accessible interfaces, our attacks can be mounted by remote adversaries who gained 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 I).

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 I).

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 link called "Serial VID" or "SVID", comprised of the pins VIDSOUT, VIDSCK, 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 slew rate *i.e.*, the time taken for a specific voltage change to occur (fastest rate in [30] is given as $80\,\mathrm{mV}/\mu\mathrm{s}$), 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.

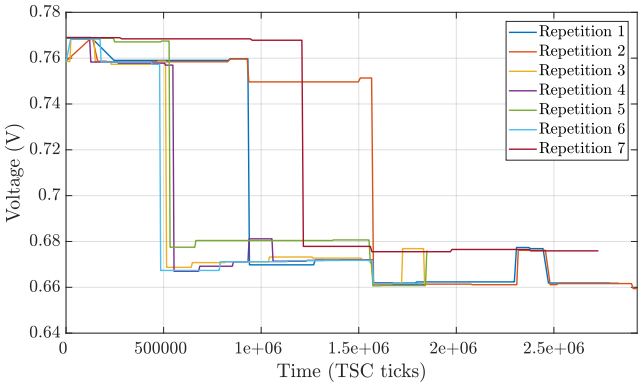

Fig. 2. Voltage decline over time for Intel i3-7100U-C, repeating a -100 mV undervolting seven times and measuring actual voltage in MSR `0x198`.

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
PROCESSORS USED FOR THE EXPERIMENTS IN THIS PAPER. WHEN MULTIPLE CPUS WITH THE SAME MODEL NUMBER WERE TESTED, WE APPEND UPPERCASE LETTERS (-A, -B ETC).

Code name	Model no.	Microcode	Frequency	Vulnerable	SGX
Broadwell	E5-1630V4	0xb000036	N/A	✗	✗
Skylake	i7-6700K	0xcc	2 GHz	✓	✓
Kaby Lake	i7-7700HQ	0x48	2.0 GHz	✓	✓
	i3-7100U-A	0xb4	1.0 GHz	✓	✓
	i3-7100U-B	0xb4	2.0 GHz	✓	✓
	i3-7100U-C	0xb4	2.0 GHz	✓	✓
Kaby Lake-R	i7-8650U-A	0xb4	1.9 GHz	✓	✓
	i7-8650U-B	0xb4	1.9 GHz	✓	✓
	i7-8550U	0x96	2.6 GHz	✓	✓
Coffee Lake-R	i9-9900U	0xa0	3.6 GHz	✓	✓

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 hypercall 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:

```
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.

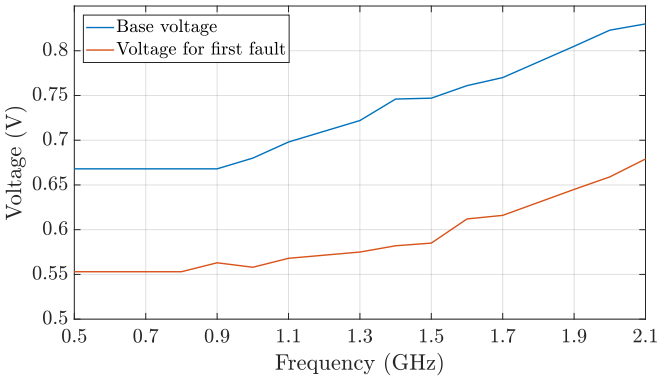

Fig. 3. Base voltage (blue) and voltage for first fault (orange) vs. CPU frequency for the i3-7100U-A

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:

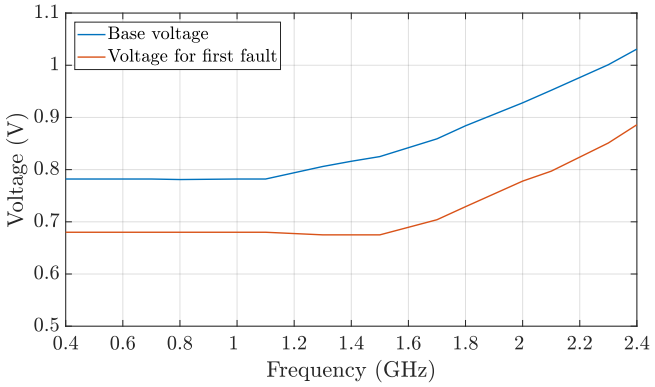

Fig. 4. Base voltage (blue) and voltage for first fault (orange) vs. CPU frequency for the i7-8650U-A

```
/* drop voltage */

do {
    i++;
    var = start_value * multiplier;
} 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
EXAMPLES OF FAULTED MULTIPLICATIONS ON I3-7100U-B AT 2 GHZ

Start value	Multiplier	Faulty result	Flipped bits
0x080004	0x0008	0xfffffffff0400020	0xfffffffff0000000
0xa7fccc	0x0335	0x000000020abdba3c	0x0000000010000000
0x9fff4f	0x00b2	0x000000004f3f84ee	0x0000000020000000
0xacff13	0x00ee	0x000000009ed523aa	0x000000003e000000
0x2bffc0	0x0008	0x00000000005ffe00	0x0000000001000000
0x2bffc0	0x0008	0xfffffffff15ffe00	0xfffffffff0000000
0x2bffc0	0x0008	0x00000100115ffe00	0x0000010010000000

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 III
NUMBER OF ITERATIONS UNTIL A FAULT OCCURS FOR THE MULTIPLICATION (`0xAE0000 * 0x18`) VS. NECESSARY UNDERVOLTING ON I3-7100U-B AT 2 GHZ.

Iterations	Undervolting
1,000,000,000	-130mV
100,000,000	-131mV
10,000,000	-132mV
1,000,000	-141mV
500,000	-146mV
100,000	crash at -161mV

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 [33], 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. To this end, we showcase practical fault attacks on minimalist benchmark enclaves using off-the-shelf cryptographic libraries.

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 [34].

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 recombined 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 $x_p^{d_p} \pmod p$ and given the respective faulty output y', one can factor the modulus n using the Bellcore attack as:

$$q = \gcd\left(y - y',\, n\right),\; p = {}^{n}\!/_{q}$$

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

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[1]. AES-NI is widely used in cryptographic libraries, including SGX's `tcrypto` 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 `(v)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:

```
[Enclave] plaintext: 697DBA24B0885D4E120FFCAB82DDEC25
[Enclave] round key: F8BD0C43844E4B4F28A6D3539F3A73E5
[Enclave] ciphertext1: C9210B59333A07A922DE59788D7AA1A7
[Enclave] ciphertext2: C9230B59333A07A922DE59788D7AA1A7
[Enclave] plaintext: 4C96DD4E44B4278E6F49FCFC8FCFF5C9
[Enclave] round key: BE7ED6DB9171EBBF9EA51569425D6DDE
[Enclave] ciphertext1: 0D42753C23026D11884385F373EAC66C
[Enclave] ciphertext2: 0D40753C23026D11884385F373EAC66C
```

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[2] and ran it in an enclave with undervolting as before. Unsurprisingly, the probability of a fault hitting a particular round instruction is approx. 1/10, 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: 5ABB97CCFE5081A4598A90E1CEF1BC39
[Enclave] CT1: DE49E9284A625F72DB87B4A559E814C4 <- faulty
[Enclave] CT2: BDFADCE3333976AD53BB1D718DFC4D5A <- correct

input to round 10:
[Enclave]   1: CD58F457 A9F61565 2880132E 14C32401
[Enclave]   2: AEEBC19C D0AD3CBA A0BCBAFA C0D77D9F

input to round  9:
[Enclave]   1: 6F6356F9 26F8071F 9D90C6B2 E6884534
[Enclave]   2: 6F6356C7 26F8D01F 9DF7C6B2 A4884534

input to round  8:
[Enclave]   1: 1C274B5B 2DFD8544 1D8AEAC0 643E70A1
[Enclave]   2: 1C274B5B 2DFD8544 1D8AEAC0 646670A1
```

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 Jovanovic[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 $2^{32}+256$ encryptions on average. We have run this attack in practice and it only took a couple of minutes to extract the full AES key from 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 D.

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 [14] 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 objectives (the attestation will

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=233&text=_mm_aesenc_si128

[2] https://gist.github.com/acapola/d5b940da024080dfaf5f

[3] 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 [1, 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 undervolting showed that Plundervolt can reliably fault such SGX key derivations. We provide several samples of incorrectly derived keys in Appendix E. 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 E.

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 `KeyID` 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 `tcrypto` 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,

[4] https://github.com/intel/linux-sgx/blob/master/sdk/simulation/tinst/deriv.cpp#L90

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 $\leq$ `0xFFFF` without crashing the CPU (cf. Section V-B). Hence, here we only consider cases in this section where `sizeof(elem_t)` $\neq 2^x$ 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 [35]. For completeness, we refer to Appendix G 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 `OWNEREPOCH` 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 G, this array access compiles to an `imul $0x21, %rdi, %rdi` instruction, which calculates the required offset to be added to the array base address afterwards. In order to reliably fault ② 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

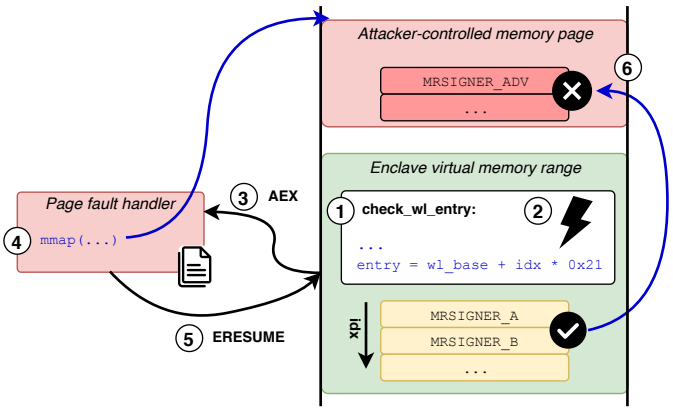

Fig. 5. Example scenario of a custom launch enclave where erroneous multiplication bitflips allow to redirect a trusted white list array lookup to attacker-controlled memory outside the enclave.

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` $\star$ `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 ③ 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 [78, 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 G) providing the same information without noise.

Finally, after the bogus white list entry has been constructed in untrusted memory, the adversary merely resumes ⑤ the victim enclave. The latter will now proceed and unknowingly dereferences ⑥ 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:

```
void vuln_ecall([in, count=cnt] struct_foo_t *arr,
                size_t cnt, size_t offset)
{
    if (offset >= cnt) return;

    arr[offset].foo1 = 0xdeadbeef;
}
```

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

```
...
size_t _tmp_cnt = ms->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_arr` 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_arr` 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, 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[5]:

```
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`

[5] https://github.com/lsds/sgx-lkl-musl/blob/db8c09/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:

```
if (n_elements != 0) {
   req = n_elements * elem_size;
   if (((n_elements | elem_size) & ~(size_t)0xffff)
       && (req / n_elements != elem_size))
     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 `0xFFFF`. Producing faults where both operands are $\leq$ `0xFFFF` (cf. Section III) is possible, e.g., we got a fault for $\texttt{0x97b5} * \texttt{0x40}$ 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 $>$ `0xFFFF`) 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, 46, 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 enclaved 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 enclaved 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 enclaved 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

voltage and frequency scaling functionality in modern Intel x86 processors, we are concerned that this proposed solution is still rather ad-hoc and does not cover the root cause for Plundervolt. That is, other yet undiscovered vectors for software-based fault injection through power and clock management features might exist and would need to be disabled in a similar manner.

Ultimately, even if all software-accessible interfaces have been disabled, adversaries with physical access to the CPU are also within Intel SGX's threat model. Especially disturbing in this respect is that the SerialVID bus between the CPU and voltage regulator appear to be unauthenticated [30, 31]. Hence adversaries might be able to physically connect to this bus and overwrite the requested voltage directly at the hardware level. 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 enclaved 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 [18].

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 `mul` and `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 `free()`. For comparison, the trusted runtime part of Microsoft's OpenEnclave SDK [48] contains 203 multiplications, while Graphene-SGX's [67] `libpal-Linux-SGX.so` features 71 `mul`/`imul` instructions.

Certain standard library functions like `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:

"After carefully reviewing the CPU voltage setting modification, Intel is mitigating the issue in two parts, a BIOS patch to disable the overclocking mailbox interface configuration. Secondly, a microcode update will be released that reflects the mailbox enablement status as part of SGX TCB [Trusted Computing Base] attestation. The Intel Attestation Service (IAS) and the Platform Certificate Retrieval Service will be updated with new keys in due course. The IAS users will receive a 'CONFIGURATION_NEEDED' message from platforms that do not disable the overclocking mailbox interface."

We note that Intel's strategy to disable MSR `0x150` (*i.e.*, said "mailbox interface") corresponds to our recommended mitigation outlined in Section VII-A. However, this strategy may not cover the root cause for Plundervolt. Other, yet undiscovered, avenues for fault injection through power and clock management features might exist (and would have to be disabled in a similar manner). Finally, we want to stress that, similiar to previous high-profile SGX attacks like Foreshadow [69], Intel's mitigation plan for Plundervolt requires trusted computing base recovery [1, 14]. That is, after the microcode update, different sealing and attestation keys will be derived depending on whether or not the undervolting interface at MSR `0x150` has been disabled at boot time. This allows remote verifiers to re-establish trust after resealing all existing enclave secrets with the new key material.

VIII. Conclusion

In this paper we have identified a new, powerful attack surface of Intel SGX. We have shown how voltage scaling can be reliably abused by privileged adversaries to corrupt both integrity and confidentiality of SGX enclaved computations. To the best of our knowledge, this represents the first practical attack that directly breaches the integrity guarantees in the Intel SGX security architecture. We have proven that this attack vector is realistic and practical with full key recovery PoC attacks against RSA-CRT and AES-NI. Furthermore, we have provided evidence that other micro-instructions can be faulted as well. Some of these instructions, like `EGETKEY` and `EREPORT`, are the basic building blocks that underpin the security of the whole SGX ecosystem.

We have shown that Plundervolt attacks are not limited to cryptographic primitives, but also enable more subtle memory safety violations. We have exploited multiplication faults in fundamental programming constructs such as array indexing, and shown their relevance for widespread memory allocation functionality in Intel SGX-SDK `edger8r`-generated code and in the SGX-LKL runtime. In conclusion, our work provides further evidence that the enclaved execution promise of outsourcing sensitive computations to untrusted remote platforms creates new and unexpected attack surfaces that continue to be relevant and need to be studied further.

Acknowledgments

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).

References

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[2] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In *CHES 2002*, 2002.

[3] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 2016.

[4] 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.

[5] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 – Combining Fault and Logical Attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *CARDIS '10*, pages 148–163, Berlin, Heidelberg, 2010. Springer.

[6] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[7] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology – CRYPTO'97*, pages 513 – 525, 1997.

[8] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, Baltimore, MD, 2018. USENIX Association.

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

[10] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[12] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[13] Jonathan Corbet. Defending against Rowhammer in the Kernel, October 2016. online, accessed 2019-08-01: https://lwn.net/Articles/704920/.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. 2016.

[15] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016*, pages 369–395, Berlin, Heidelberg, 2016. Springer.

[16] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.

[17] Junfeng Fan and Ingrid Verbauwhede. *An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost*, pages 265–282. Springer, Berlin, Heidelberg, 2012.

[18] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118, Aug 2013.

[19] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT '12*, pages 305–321, Berlin, Heidelberg, 2012. Springer.

[20] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *S&P 2003*, pages 154–165, May 2003.

[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[24] Shay Gueron. A memory encryption engine suitable for general purpose processors. ePrint 2016/204, 2016.

[25] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018.

[26] Hagai Bar-El Hamid, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In *Proceedings of the IEEE*, volume 94, 2006.

[27] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[28] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *ACM SIGPLAN Notices*, volume 50, pages 339–353. ACM, 2015.

[29] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

[30] Infineon. OptiMOS IPOL DC-DC converter single-input voltage, 30 A buck regulators with SVID. online, accessed 2019-07-29: https://www.infineon.com/cms/de/product/power/dc-dc-converter/integrated-dc-dc-pol-converters/ir38163m/, 2019.

[31] Intel. Voltage Regulator-Down 11.1: Processor Power Delivery Design Guide. online, accessed 2019-07-29: https://www.intel.com/content/www/us/en/power-management/voltage-regulator-down-11-1-processor-power-delivery-guidelines.html, 2009.

[32] Intel. L1 Terminal Fault SA-00161, August 2018.

[33] Intel. 7th Generation Intel Processor Families for U/Y Platforms and 8th Generation Intel Processor Family for U Quad-Core and Y Dual Core Platforms. Datasheet, Volume 1 of 2, revision 006, Jan 2019.

[34] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography – Example of Using RSA Primitive Functions. online, accessed 2019-07-29: https://software.intel.com/en-us/ipp-crypto-reference-2019-example-of-using-rsa-primitive-functions, 2019.

[35] Intel. Get Started with the SDK. online, accessed 2019-05-10: https://software.intel.com/en-us/sgx/sdk, 2019.

[36] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual; Volume 2 (2A, 2B, 2C & 2D)*. Number 325383-070US. May 2019.

[37] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. ePrint 2016/1196, 2017.

[38] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[39] Simon Johnson. An update on 3rd Party Attestation. online, accessed 2019-07-30: https://software.intel.com/en-us/blogs/2018/12/09/an-update-on-3rd-party-attestation, December 2018.

[40] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. Magic: Malicious aging in circuits/cores. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1), 2015.

[41] 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.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[43] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[44] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[45] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security*

17), pages 523–539, Vancouver, BC, 2017. USENIX Association.

[46] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.

[47] Francis X McKeen, Carlos V Rozas, Uday R Savagaonkar, Simon P Johnson, Vincent Scarlata, Michael A Goldsmith, Ernie Brickell, Jiang Tao Li, Howard C Herbert, Prashant Dewan, et al. Method and apparatus to provide secure application execution, July 21 2015. US Patent 9,087,200.

[48] Microsoft. Open Enclave SDK. online, accessed 2019-05-10: https://openenclave.io/sdk/, 2019.

[49] Miha Elerščič. Guide to Linux undervolting for Haswell and never Intel CPUs. online, accessed 2019-07-29: https://github.com/mihic/linux-intel-undervolt, 2019.

[50] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, 2018.

[51] NotebookReview. The ThrottleStop Guide. online, accessed 2019-07-29: http://forum.notebookreview.com/threads/the-throttlestop-guide.531329/, 2019.

[52] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[53] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[54] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[55] 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 Symposium*, 2016.

[56] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[57] RightMark Gathering. RMClock Utility. online, accessed 2019-07-29: http://cpu.rightmark.org/products/rmclock.shtml, 2019.

[58] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks . In *DIMVA*, 2017.

[59] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, November 2019.

[60] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*, 2015.

[61] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*, 2017.

[62] Sergei P. Skorobogatov. Copy Protection in Modern Microcontrollers, 2001. online; accessed 2019-07-29: http://www.cl.cam.ac.uk/~sps32/mcu_lock.html.

[63] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *CHES 2002*, pages 2–12, Berlin, Heidelberg, 2003. Springer.

[64] Adrian Tang. *Security Engineering of Hardware-Software Interface*. PhD thesis, Columbia University, 2018.

[65] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.

[66] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 19–34. IEEE, 2017.

[67] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[68] 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, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[69] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[70] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, November 2019.

[71] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, page 4. ACM, 2017. https://github.com/jovanbulck/sgx-step.

[72] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*. ACM, October 2018.

[73] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, August 2017.

[74] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[75] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, May 2019.

[76] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (ATC16)*, pages 167–179, 2016.

[77] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*, 2016.

[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[79] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.

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
ED41043A440D6B6F07454F51FB8DFBAAF035C02AB61EA4
8CEEB6FCD4876ED520D60E1EC4619719D8A5B8B807FAFB
8E0A3DFC737723EE6B4B7D93A2584EE6A649D060953748
834B2454598394EE0AAB12D7B61A51F527A9A41F6C1687
FE2537298CA2A8F5946F8E5FD091DBDCB

e = 0x11

d = 0xA5DAFC5341FAF289C4B988DB30C1CDF83F31251E0668B42784813801579641B29410B3C7998D6BC465745E5C392669D6870DA2C082A939E37FDCB82EC93EDAC97FF3AD5950ACCFBC111C76F1A9529444E56AAF68C56C092CD38DC3BEF5D20A939926ED4F74A13EDDFBE1A1CECC4894AF9428C2B7B8883FE4463A4BC85B1CB3C1

The following ciphertext x decrypts to $y = x^d \pmod{n}$:

x = 0x1253E04DC0A5397BB44A7AB87E9BF2A039A33D1E996FC82A94CCD30074C95DF763722017069E5268DA5D1C0B4F872CF653C11DF82314A67968DFEAE28DEF04BB6D84B1C31D654A1970E5783BD6EB96A024C2CA2F4A90FE9F2EF5C9C140E5BB48DA9536AD8700C84FC9130ADEA74E558D51A74DDF85D8B50DE96838D6063E0955

y = 0xEB7A19ACE9E3006350E329504B45E2CA82310B26DCD87D5C68F1EEA8F55267C31B2E8BB4251F84D7E0B2C04626F5AFF93EDCFB25C9C2B3FF8AE10E839A2DDB4CDCFE4FF47728B4A1B7C1362BAAD29AB48D2869D50241214358115 91BE392F982FB3E87D095AEB40448DB972F3AC14F7BC275195281CE32D2F1B76D4D353E2D

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 obtained in one of our experiments:

y' = 0xAA105EAFB6BDD9E5A15443729670B70F042889103E023428F37B1CEFFAECC91292772652E2016AA5955DFDA6FD5B685AE062A32DEA9C9E99F516370BE2ED4EF48A3C3513E4026E5DE3647267A83C9C245A72EA9F4D8C2B373A8CE70047C922A108807197A6BC15A1DF31E06FCD5521AA00ECC0B3A2A5BCDDE5A8B7B5AAD3015F

Appendix C
Further Examples for AES-NI aesenc Faults

```
[Enclave] plaintext: 4C96DD4E44B4278E6F49FCFC8FCFF5C9
[Enclave] round key: BE7ED6DB9171EBBF9EA51569425D6DDE
[Enclave] ciphertext1: 0D42753C23026D11884385F373EAC66C
[Enclave] ciphertext2: 0D40753C23026D11884385F373EAC66C

[Enclave] plaintext: 2A89F789FAE690774FB2FC04DC8EB7BE
[Enclave] round key: E420AFB5B6ECE976B7A55812705DC2A7
[Enclave] ciphertext1: A2A556F8BBE848CA125E110507DC2E0E
[Enclave] ciphertext2: A2A756F8BBE848CA125E110507DC2E0E

[Enclave] plaintext: D15DBCAA47A8D62B281FFCF9CEF49F5D
[Enclave] round key: FF27B41E3A0F2D9215F4AF61F394C3E8
[Enclave] ciphertext1: 2203E7B64DEE0F3133FBE61E451F43FD
[Enclave] ciphertext2: 2201E7B64DEE0F3133FBE61E451F43FD

[Enclave] plaintext: A67DBE59F885B1AD4F20FE212A2F1767
[Enclave] round key: A4A28B5577F4D771C19B20A90B0CFA98
[Enclave] ciphertext1: 70E2C1040C009C78D64952B4F5B2777A
[Enclave] ciphertext2: 70E0C1040C009C78D64952B4F5B2777A

[Enclave] plaintext: 7815CBC04D8FB2A3B464946A9E9B5596
[Enclave] round key: 596FA60CC6496FD3E9E2B41DF701BA3D
[Enclave] ciphertext1: 19C386B99889F93DC16C0D8E3FE3804A
[Enclave] ciphertext2: 1DC386B99889F93DC16C0D8E3FE3804A
```

Appendix D
Running DFA against AES-NI

Based on the fault described in Section IV-C, the input file `fault.txt` to the DFA implementation from https://github.com/Daeinar/dfa-aes should contain the following line:

```
BDFADCE3333976AD53BB1D718DFC4D5A DE49E9284A625F72DB87B4A559E814C4
```

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 fault is in byte one as follows:

```
./dfa 4 1 fault.txt
```

This yields 595 key candidates for this particular example, including the correct secret key value `0x000102030405060708090a0b0c0d0e0f`.

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.

```
KEY_ID    = 1966dd54d49f568111ae77074bf14522
            860942817065d0cebc7370bd9e5d9549
KEY_OK    = 9ed9a757b4bfe29e90833f4b40df4fb7
KEY_FAULT = 745a2d0054b0f7e2542c1bcd502f7ad5

KEY_ID    = 728210dc53f1f22b24e79be5fc375f42
            421f9dcb67cb6bac29a7caf9aad94cb6
KEY_OK    = 6049723afc45f7eb1728cd7eb1b7ea66
KEY_FAULT = 43e1ed22d58729db1e4def53a882a3f9
```

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 occured in one experiment.

```
KEY_ID    = a4717110f732e75fa4f021ae3fbb6da8
            bbb55e1a8b38dc74e4554749b7ad141f
KEY_OK    = 11fea22c14125fd11de205ca3df643be
KEY_FAULT = 760e5d0a50d4ce4a6b5859f58c42b62c

KEY_ID    = a4717110f732e75fa4f021ae3fbb6da8
            bbb55e1a8b38dc74e4554749b7ad141f
KEY_OK    = 11fea22c14125fd11de205ca3df643be
KEY_FAULT = 37535d4ff210de92917cc931a1fe7c08
```

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.

```
=== Local attestation REPORT: 'REPORT_OK' ===
    CPU_SVN:     0x0809ffffff020000000000000000000000
    MISC_SEL:    0x0
    MRENCLAVE:   0x144833bab7d6d1a98154da987f2634b1
                   682311384613dc08d7334e53291eb524
    MRSIGNER:    0xf088eb845e3f5fd691e807942a423dc6
                   5f421c35d79d5a60c019367a72e38170
    PROD_ID/SVN: 0x0/0x0
    DATA:        0x4141414141414141414141414141414141
                   4141414141414141414141414141414141
                   4141414141414141414141414141414141
                   4141414141414141414141414141414141
    KEY_ID:      0xe754cdfebe332605944c1813fa2416ed
                   00000000000000000000000000000000
    MAC:         0x5ab280f46073878588ce8e537888caaa

=== Local attestation REPORT: 'REPORT_FAULT' ===
    CPU_SVN:     0x0809ffffff020000000000000000000000
    MISC_SEL:    0x0
    MRENCLAVE:   0x144833bab7d6d1a98154da987f2634b1
                   682311384613dc08d7334e53291eb524
    MRSIGNER:    0xf088eb845e3f5fd691e807942a423dc6
```

```
                5f421c35d79d5a60c019367a72e38170
PROD_ID/SVN:    0x0/0x0
DATA:           0x41414141414141414141414141414141
                  41414141414141414141414141414141
                  41414141414141414141414141414141
                  41414141414141414141414141414141
KEY_ID:         0xe754cdfebe332605944c1813fa2416ed
                  00000000000000000000000000000000
MAC:            0xb58acd215557ed3bddb7f648173d8bde
```

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)
{
    /*
     * XXX the following array index compiles to a
     * multiplication that can be faulted..
     */
    ref_le_white_list_entry_t *current_entry = &
    g_ref_le_white_list_cache[idx];

    /*
     * Our exemplary launch policy requires that the
     * enclave author is white listed, plus is optionally
     * allowed access to the platform provisioning key.
     */
    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;
        }

        /* NOTE: we explicitly leak the loop iteration
         * here for simplicity; real-world adversaries
         * could use a #PF side-channel or count
         * instructions w precise single-stepping
         */
        *it = i;
    }

    /* 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    %edx,%ebp
    mov    $0x20,%edx
    sub    $0x8,%rsp
    add    %rdi,%rbx
    mov    %rbx,%rdi
    callq  memcmp
    xor    %edx,%edx
    test   %eax,%eax
    jne    1f
    test   %ebp,%ebp
    mov    $0x1,%edx
    je     1f
    movzbl 0x20(%rbx),%edx
1:
    mov    %edx,%eax
    pop    %rdx
    pop    %rbx
    pop    %rbp
    retq
```