# Plundervolt: How a little bit of undervolting can create a lot of trouble

**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

## Introduction

Two-thirds of the world's population now own a personal computing device in the form of a smart phone. These devices store vast amounts of privacy-sensitive data along with a large number of user applications. Trusted Execution Environments (TEEs) were created out of the need to protect our private and valuable data from other, possibly malicious, applications and even the operating system itself.

But it is not just mobile phones that store valuable data—our personal computers often carry copies of our passwords. We use our computers for online banking and this is where it's desirable that no adversary can tamper with the data, *even if the computer's operating system is compromised*.

For these reasons, Intel processors (from 2015 onward) include Software Guard Extensions (SGX), which allows an application to self-quarantine sensitive data and functions in an *enclave* using dedicated CPU instructions. Intuitively, SGX enclaves represent a secure vault or fortress in the processor, which cannot be read or modified by any other software, *including the privileged operating system.* Intel SGX was purposely designed to protect against the most advanced types of adversaries who have unrestricted physical access to the host machine, e.g., untrusted cloud providers under the jurisdiction of foreign nation states. SGX therefore includes state-of-the-art memory encryption technology [4] that protects the confidentiality, integrity and freshness of all enclave memory while it resides in untrusted off-chip DRAM.

## Performance vs. security

More and more is being demanded of our computers: faster response times to render complex graphics, multiple programs being run at once and constantly switching applications. These demands increase power consumption and raise the temperature of already over-worked computers. To manage this, CPU manufacturers have introduced various software interfaces to dynamically adjust the processor's operating voltage and frequency.But, as we will see, putting this power at a user's finger tips comes with a cost. *With great power comes great responsibility*.

Hardware is being optimized to meet the growing need for performance. The aim: to maximize performance whilst keeping functional correctness. Modern processors cannot continuously run at maximum clock frequency—they would simply get too hot. And, in mobile devices, the battery would drain too quickly.

In an electrical circuit, voltage and frequency can be thought of as two sides of the same coin: higher clock frequencies require higher voltages for electrical signals to arrive in time, and, likewise, lower voltages require the processor and memory to operate at a slower rate.

Power management jargon therefore specifies optimal "frequency/voltage pairs" for different use cases. Hence, the question arises: *if frequency and voltage are changed independently, what will happen?* As we discuss below, this is the question that security researchers have been exploring when attempting to deliberately induce faulty computations and take advantage of the resulting errors.

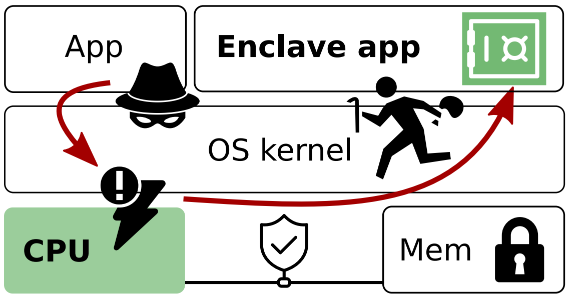

**Figure 1.** Plundervolt circumvents SGX's memory encryption engine protection boundaries by abusing an undocumented voltage scaling interface which allows privileged software adversaries to induce predictable computation faults within the processor itself.

## Software-based fault attacks

Since the early days of computing, researchers recognized that software computation results may be affected when hitting the physical limits of the underlying hardware, e.g., after adjusting the voltage, glitching the clock, overheating or cooling the operating temperature, or even focusing a laser at a chip [2]. Apart from apparent safety concerns, for instance in the avionics or space industry, these fault injections have also been extensively studied from a security perspective. That is, fault attacks may deliberately corrupt calculations in order to bypass security mechanisms such as sophisticated *"You shall not pass"* functions. For a long time, such advanced fault attacks were considered to be of limited importance as they required physical access to the target device, e.g., a smart card.

This all changed, however, in 2014 with the discovery of the Rowhammer [7] effect, which causes bits flips in memory—entirely from software. Underlying this attack is the physical layout of DRAM, which consists of capacitors storing very small voltage charges for '1's and '0's. Security researchers observed that DRAM memory cells can leak their charges into nearby memory rows when they are accessed at high frequency—causing memory corruption and bit flips. In other words, Rowhammer fundamentally changed the threat of fault attacks. It is no longer just adversaries with physical access, attacks can now be mounted by remotely executing code to modify specific data structures and escalate privileges.

Rowhammer remained, for some time, the only known purely software-based fault attack on x86 systems. However, Intel ultimately considers main memory as an *untrusted* storage facility in the design of SGX [4]. When researchers tried to attack SGX with Rowhammer, they merely discovered a denial-of-service effect because the memory encryption engine produced an integrity check error, halting the entire system. Intel SGX enclaves were hence considered immune to such fault attacks!

Initially, researchers were only interested in attackers who were unprivileged, e.g., in a sandboxed environment like JavaScript. However, with the creation of TEEs such as Intel SGX, ARM TrustZone, and AMD SEV, threat models changed once again. In the newly emerging TEE landscape, it suddenly becomes vital to protect against attackers who have gained root privileges. In 2017, Tang et al. [11] presented a privileged software fault attack called CLKscrew. They discovered that ARM processors permitted changing the frequency and voltage from system software. And this is where the story really starts. CLKscrew showed that overclocking features can be abused to jeopardize the integrity of computations for privileged adversaries in the ARM Trustzone TEE. This attack has been demonstrated to defeat RSA signature checks and extract full cryptographic keys from the TrustZone of a Nexus 6 mobile phone.

## Why Plundervolt is different

With our new attack, Plundervolt, we demonstrate the first ever software-based fault injection attack against Intel SGX enclaves. Plundervolt abuses undocumented power management interfaces present in all recent Intel Core processors. We use these interfaces to lower the voltage and cause *predictable* faults in secure enclave computations. Our attack is able to steal secrets—even in the presence of state-of-the-art memory encryption technology (Fig. 1). In contrast to prior high-profile attacks on Intel SGX, which abused microarchitectural design flaws to break confidentiality of enclave secrets [13, 14], we are the first to demonstrate that even the *integrity* of seemingly secure enclave computations cannot be trusted anymore. But we didn't only break crypto code. We show that an attacker can induce memory-misbehavior in secure, bug-free code without any enclave software vulnerabilities [15].

For a more technical description, we point interested readers to our original paper [9], on which this article is based.

### Current status and concurrent discoveries

After we responsibly disclosed our findings and Intel prepared a microcode patch, Plundervolt was disclosed to the public on December 10, 2019. Intel confirmed that we were the first to report this issue. However, during the embargo period, two other research teams independently investigated undervolting security implications. One of these, known as V0LTpwn [6] outlines a similar attack on Intel SGX enclaves where undervolting is used, in combination with additional stress from a sibling logical processor, to study the fault behavior of x86 vector instructions. Also, another group of researchers developed the VoltJockey [10] attack against ARM processors. VoltJockey continued the CLKscrew saga by showing that secure TrustZone computations can also be faulted through voltage changes. This attack was later also demonstrated on Intel SGX processors, by faulting a proof-of-concept software-based AES implementation.

## The Plundervolt effect

Before we discuss undervolting, we need to talk about overclocking. CPUs have official maximum clock frequency limits, but gamers often want to speed up their machines by pushing the clock frequency *over* the recommended values.

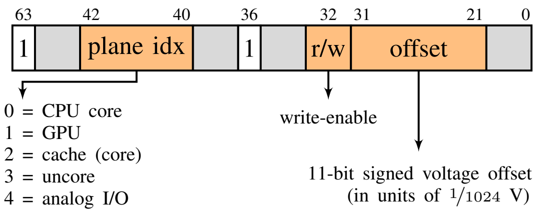

**Figure 2.** Layout of the undocumented MSR `0x150` for undervolting.

This can be tricky because integrated circuits have strict timing requirements. The electrical signals need to pass through the circuitry within one clock cycle before the next signals arrive. If the clock is too fast, computation results may not arrive in time: leading to bit flips in the expected output. Similarly, the lower the voltage, the longer it takes to propagate input signals throughout the circuitry. So, if the voltage is too low (for a specific frequency) the input signals may not traverse the circuitry before the next clock tick.

Intel processors have features that enable the modification of both clock frequency and CPU voltage from privileged software. These are controlled through undocumented Model Specific Registers (MSRs). We focus on MSR `0x150` which is responsible for voltage. Figure 2 shows how the 64-bit value in MSR `0x150` can be decomposed into a *plane index* and a *voltage offset*. By specifying the plane index, system software can select which components will have their voltage changed. The CPU core and cache share the same voltage plane on all machines we tested and the higher voltage will be applied to both. The voltage offset is encoded as an 11-bit signed integer relative to the core's base voltage in units of approximately 1 mV.

This feature can be abused to inject faults into secure SGX computations. For starters, we configured the CPU to run at a fixed frequency. Then, undervolting is applied by writing to the concealed MSR `0x150` just before entering the code in the victim enclave. After returning from the enclave, the host program immediately returns to a stable operating voltage.

One of the hardest parts of this research was finding good parameters to work with. Too much undervolting and the system repeatedly crashes, too little and no faults are injected. We experimented by reducing the voltage in small steps of 1 mV until a fault occurs but before the dreaded kernel panic or system freeze. In practice, we found that it is sufficient to undervolt for short periods of time (<100 ms) by -100 mV to -260 mV, depending on the specific CPU, frequency, and temperature.

### Tested processors

We tested different SGX-enabled processors from Skylake onwards , cf. Table 1. We had multiple CPUs with the same model numbers and, surprisingly, we found they can respond very differently when undervolted. We list each individual processor with a letter appended. All our tests were run on Ubuntu 16.04 or 18.04 with stock Linux v4.15 and v4.18 kernels.

**Table 1. Processors used for the experiments in this paper.**

| Code name | Model no. | $\mu$-code | Frequency |
|---|---|---|---|
| Skylake | i7-6700K | 0xcc | 2.0 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 |

## Inducing the first enclave fault

We tried undervolting various x86 instructions. We observed that multiplications (e.g., `imul`) and other complex instructions such as the AES-NI extensions can be most easily faulted. We do not definitively know why these specific instructions, but we can put forward a conjecture: these instructions will have longer critical paths compared to simpler operations. Not only that, they will have been more aggressively optimized. When lowering the voltage, electrical signals may not have enough time to propagate through the circuitry before the next clock tick.

Consider the following enclave multiplication proof-of-concept (the code compiles to assembly with `imul` instructions):

```
uint64_t multiplier = 0x1122334455667788;
uint64_t correct = 0xdeadbeef * multiplier;
uint64_t var = 0xdeadbeef * multiplier;

while (var == correct)
{
   var = 0xdeadbeef * multiplier;
}
uint64_t flipped_bits = var ^ correct;
```

Clearly, this is an infinite loop—it should never terminate. But undervolting leads to a bit-flip in `var`, typically in byte 3 (counting from the least-significant byte as byte 0). This forces the enclave program to erroneously exit the loop. The XOR on the last line highlights only the flipped bit(s). In this configuration, the output is always `0x04 00 00 00`. This is worth emphasising: the loop *always exits with the same bit flipped.*

### In-depth analysis of undervolting effects

To better understand what was happening, we undervolted and measured the core voltage using the fully documented MSR `0x198 (MSR_PERF _STATUS)`. For different clock frequencies, we recorded both the base voltage and the voltage when the first faulty result appeared. The results for the i3-7100U-A are shown in Figure 3.

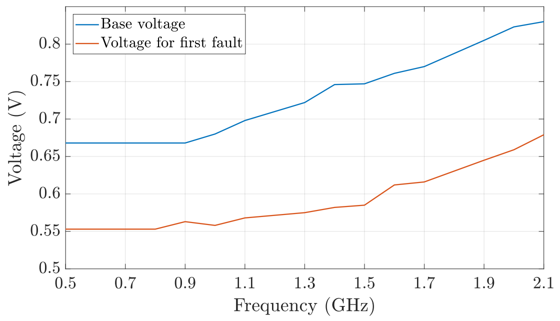

**Figure 3.** Base voltage (blue) and voltage for first fault (orange) vs. CPU frequency for the i3-7100U-A

We induced thousands of faulted multiplications and were able to draw up some conclusions. The faulty results, see Table 2 for selected examples, generally fell into the following categories: (i) one to five (contiguous) bits flip or (ii) all most-significant bits flip. And, *very occasionally* we observed faulty states in between. From this, we can summarise:

- 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`;
- the order of the operands is important: for example, `0x4 * 0x80000` *never* faulted in our experiments.

**Table 2. Faulted multiplications on i3-7100U-B at 2 GHz**

| Start | Mult | 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 | 0xffffffff15ffe00 | 0xfffffffff0000000 |
| 0x2bffc0 | 0x0008 | 0x00000100115ffe00 | 0x0000010010000000 |

The probability of a fault increases with the undervolting: on the i3-7100U-B, we had to repeat `0xae0000 * 0x18` around 1,000,000,000 times to fault at -130 mV, while 500,000 repetitions were sufficient at -146 mV.

## From faults to enclave key extraction

Having shown that Plundervolt can practically fault in-enclave computations, let us see how that translates to actual attacks against widely-used cryptographic algorithms that secure our everyday communications.

### Factoring RSA keys with one fault

We wrote a proof-of-concept application for RSA signature generation that would run inside an enclave. We used Intel's example code which uses the Chinese Remainder Theorem (CRT) optimization. Given an RSA public key $(n, e)$ and the corresponding private key $(d, p, q)$, RSA-CRT makes the computation time of $y = x^d \pmod{n}$ up to four times faster.

RSA-CRT private key operations (decryption and signature) are well-known to be vulnerable to the famous Bellcore attack, one of the first published fault attacks [3]. This requires a fault in one of the two exponentiations of the core RSA operations. And if we can do that—we only need *one single* faulty signature to be able to factor the modulus $n$:

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

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

To make sure we only hit one exponentiation we undervolted for roughly the first third of the enclave computation. The obtained faults could then be used to factor the 2048-bit RSA modulus using the Lenstra and Bellcore attacks—thus recovering *the full key*.

### Breaking AES-NI

Intel's AES New Instructions (AES-NI) provide efficient hardware implementations for key schedule and round computation. These instructions are widely used in the Intel SGX-SDK to implement crucial operations like sealing and unsealing, which refers to the encryption and decryption of enclave secrets so that they can be persistently stored outside the enclave, e.g., on the untrusted hard drive [1]. Other SGX crypto libraries (e.g., `mbedtls` in Microsoft OpenEnclave) similarly rely on 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).

We ran a canonical implementation using AES-NI instructions in an enclave with undervolting as before. By repeating the attack a few times, we got a fault in round 8:

```
plaintext: 5ABB97CCFE5081A4598A90E1CEF1BC39
CT1: DE49E9284A625F72DB87B4A559E814C4 <- faulty
CT2: BDFADCE3333976AD53BB1D718DFC4D5A <- correct

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

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

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

Now we apply the differential fault analysis technique by Tunstall et al. [12], which, given a pair of correct and faulty ciphertexts on the same plaintext, recovers the full 128-bit AES key with a computational complexity of only $2^{32} + 256$

encryptions on average. In practice it takes a few minutes to extract the full AES key from the enclave, including both fault injection and key recovery phases. It is worth noting that the attacks we are using were first discovered in embedded systems. These twenty-year-old fault attacks can now be leveraged against CPUs on non-embedded devices such as consumer laptops and company servers.

### Other faults in crypto

Besides key extractions from RSA-CRT and AES-NI, we were able to inject faults into SGX-provided crypto functions: the MAC used in AES-GCM, elliptic curve signatures and key exchange. We also looked at the SGX-provided instructions for key derivation and attestation [1]. The `EGETKEY` instruction derives an enclave-specific 128-bit symmetric key from a hardware-level master secret, which is never directly exposed to software. The key derivation uses AES-CMAC with a software-provided `KeyID` and the calling enclave's identity. Our experiments on the i3-7100U-C running at 2 GHz with -134 mV undervolting showed that Plundervolt can reliably fault such key derivations. Interestingly, we noticed that key derivation faults appear to be largely *deterministic*: for a fixed `KeyID`, the same wrong key seems to be produced most of the time when undervolting, even across reboots.

SGX supports local attestation through the `EREPORT` primitive to create a measurement report for another target enclave on the same platform. `EREPORT` first performs an internal key derivation to establish a secret key that can only be derived by the intended target enclave on the same processor. This key is then used 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. As with the `EGETKEY` experiments above, we noticed that the faulty MACs appear to be deterministic—but they do change across reboots, because `EREPORT` generates an internal random `KeyID` on every processor power cycle.

This does not directly break SGX's security objectives (attestation will simply fail), but faulty key derivations may reveal information about the processor's long-term key material that should never be exposed. We leave further exploration and cryptanalysis of the above faults as future work.

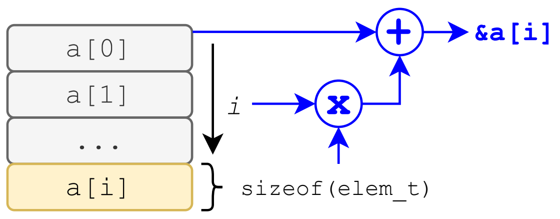

**Figure 4.** The address of element `a[i]` in an array is computed as `&a[0]+i*sizeof(elem_t)`.

## Beyond crypto

From our previous examples it would be logical to assume that only cryptographic code is vulnerable to Plundervolt. However, we were able to attack standard code—and this is where things get really interesting.

We know that compilers rely on multiplication results for pointer arithmetic and memory allocation. These multiplications themselves are not visible at the source-code level—but they are generated *under the hood*. Consequently, if we can fault one of these compiler-generated multiplications, we can introduce memory-safety issues in code that is entirely bug-free. As an example, Fig. 4 illustrates how the pervasive code pattern of indexing into an array may cause the compiler to use a multiplication to dynamically compute the address of element `a[i]`. Crucially, unexpected out-of-bounds accesses will occur if an attacker can fault such compiler-generated multiplications to produce incorrect addresses. In other words, Plundervolt ultimately breaks the processor's architectural instruction specification, thereby violating the hardware-software contract expected by the compiler.

We explore two scenarios where faulty multiplications break memory safety in seemingly secure code. We first present a case-study enclave application where a trusted in-enclave array pointer is flipped to untrusted, attacker-controlled memory outside the enclave. Next, we look at memory allocations where Plundervolt may cause heap corruption.

## Faulting pointer arithmetics

Let us revisit the array indexing example of Fig. 4, where a multiplication is used to calculate the effective memory address of the $i$-th element in an array. Intuitively, all an attacker has to do is undervolt whilst the multiplication is being performed and unexpected addresses will be produced. However, there are some limitations. When the type `elem_t` has a size that is a power of two, compilers will use left bitshifts instead of explicit `imul` instructions. We also found it difficult to consistently produce multiplication faults where both operands are $\leq$ `0xFFFF`. We were able to fault with smaller operands—but we crashed the computer a lot more. Therefore we only consider cases where:

$$\texttt{sizeof(elem\_t)} \neq 2^x \text{ and } i > 2^{16}.$$

*An example scenario*

To demonstrate that our attack is realistic and can be exploited in compiler-generated enclave code, we constructed a small case-study application. Consider an enclave that holds a relatively large amount of data in an array of `struct` elements. This could, for example, be a long list of biometric features in a fingerprint template.

We assume that the enclave loads secret data into this array, e.g., the user's fingerprint template decrypted from permanent storage. The code might look like this:

```
// Get offset to feature in large array
// with around 500k elements
fingerprint_feature_t *f = &features[idx];
// Store some secret data into array entry
f->data = some_secret_feature;
```

Figure 5 overviews the attack procedure. During normal execution, only trusted memory inside the enclave will ever be referenced. When undervolting ① during the `imul` used for computing the pointer `f`, however, the higher bits of the product may flip. This effectively causes the result to become a large *negative* offset, relative to the trusted array base address. Crucially, after adding this corrupted offset, the resulting address suddenly points into the untrusted address space *outside* the enclave. Now, the victim enclave unknowingly dereferences the outside pointer as if it was in-enclave memory. As the referenced

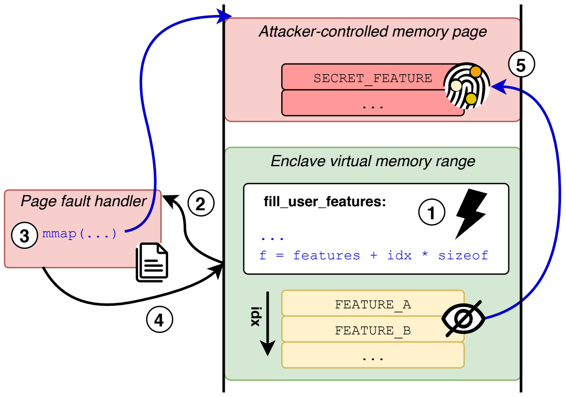

**Figure 5.** Example scenario of an application enclave where erroneous multiplication bitflips allow to redirect a trusted fingerprint array lookup to attacker-controlled memory outside the enclave.

address is most likely not currently mapped, this access causes a page fault ② which invokes the untrusted operating system. We installed a custom page fault handler ③ that maps the required untrusted memory page on demand. The attacker can now simply resume ④ the enclave. It will unknowingly ⑤ write `some_secret_feature` into untrusted, attacker-controlled memory. Plundervolt has succeeded in breaking perfectly secure, bug-free code.

## Faulting memory allocations

Another example for fault-induced vulnerabilities are size computations for dynamic memory allocations. These are very common and (again) rely on multiplications. For example, a large array of `struct` elements might be allocated using:

```
// Compute size
size_t size = count * sizeof(elem_t);
// Allocate array
elem_t *array = malloc(size);
// ... use array ...
```

But we showed that Plundervolt breaks the processor's architectural guarantees as `imul` can be faulted to produce erroneous results that are *smaller* than the expected value. If a multiplication fault occurs during calculation of the `size` variable, a smaller buffer than expected will be allocated. Because Plundervolt corrupts multiplications silently, without failing the `malloc()` call, the subsequent code has no means of determining the actual size of the allocated buffer. Subsequent writes or reads to the allocated buffer will assume a larger buffer and hence read or write out of bounds, corrupting the trusted enclave heap—Plundervolt has again induced a memory-safety issue in memory-safe code.

### The bigger picture

The ideas presented here have implications beyond SGX and Plundervolt. Many researchers have studied the use of faults to break cryptographic algorithms. Less attention has been paid to fault injection for inducing memory-safety issues into safe code. But *any* code, whether it is running on a small embedded device or inside an enclave on a complex processor, is, in principle, vulnerable to this type of attack—the only requirement is that some vector for fault injection exists. This is a substantial shift in the risk potential for at least two reasons.

Firstly, now all software, not just cryptographic implementations, needs protection against fault attacks, forming a much bigger pool of attack targets than previously anticipated.

Secondly, code execution for software-based fault attacks is often easier to obtain than hooking up an oscilloscope and glitching equipment to a specific victim machine. Thus, inducing faults via (remote) code execution may be a much more realistic threat and, at the same time, affect substantially more users.

## Countermeasures and counterattacks

Due to SGX's threat model, countermeasures cannot be implemented at the level of the untrusted OS or in the untrusted runtime components (which the attacker controls). Instead, unsafe undervolting can only be prevented in the CPU hardware or microcode.

Alternatively, the trusted in-enclave code itself can be hardened against faults. One approach to do that would be to *detect* faulty computation results. Such a defense could leverage ideas from multi-variant execution techniques. Specifically: one could execute enclaved computations twice in parallel on two different cores or hyperthreads and halt if executions diverge.

Many fault injection countermeasures have been proposed for cryptographic algorithms, including the use of (generic) temporal redundancy (*i.e.*, compute-twice-and-compare) as well as more algorithm-specific approaches. For instance in the RSA-CRT case, the signature could be verified. In the AES-NI case the encryption can be verified with a subsequent decryption, and so on. However, this would incur substantial performance overheads.

For non-cryptographic code the situation is complicated—the exact results of a fault injection will vary. Mitigations like address space layout randomization (which changes the location of the program in memory each time it runs) make exploits harder but still do not remove the root cause.

Removing the undervolting interface (MSR `0x150`) via microcode or in hardware is a rather radical solution and will certainly mitigate our specific attack. Following the responsible disclosure (embargoed from June 7, 2019 to December 10, 2019), Intel informed us that their countermeasure is exactly this—they included an option to disable MSR `0x150`. The fact that an enclave runs on a "protected" machine, *i.e.*, without software-controlled undervolting, is verifiable through remote attestation. Similiar to previous high-profile SGX attacks like Foreshadow [13] and LVI [14], Intel's mitigation for Plundervolt requires trusted computing base recovery [1]. After the microcode update, different sealing and attestation keys will be derived depending on whether or not the undervolting interface has been disabled at boot time. This allows remote verifiers to restore trust after re-encrypting all existing enclave secrets with the new key material.

However, we consider this to be an ad-hoc mitigation which does not address the root cause for Plundervolt. Other undiscovered vectors for software-based fault injection through power or clock management features might exist and would need to be similarly disabled. Ultimately, even without any software-accessible interfaces, adversaries with physical access to the CPU are also within Intel SGX's threat model. The CPU requests a specific voltage from the mainboard's voltage regulator via the SerialVID bus. But this bus appears to be completely unauthenticated. So an attacker could physically connect to this SerialVID bus and overwrite the requested voltage directly.

## Lessons learned

SGX has brought flexible, trusted execution onto laptops, desktops and servers. Unfortunately, building a high-assurance SGX "fortress" on weak foundations (like the complex and general-purpose x86 microarchitecture), seems unlikely to succeed. Over and over again, attacks like Foreshadow [13], Spectre [8], and LVI [14] have shown that microarchitectural optimisations prove catastrophic to SGX's security.

Some of these attacks, like LVI and Spectre, are somewhat similar in spirit to our work, as they too "inject" faulty computations and cause the program to deviate from its intended execution path

Crucially, however, these techniques manifest entirely at the microarchitectural level: the faulty computations are only "speculatively" executed and are never persisted to the architectural state. Plundervolt goes one step further and induces *persistent* architectural faults by exploiting fundamental physical properties of the CPU—namely the need for a stable supply voltage. In this, our work once again shows that abstraction levels are only relative in the eyes of attackers. Plundervolt, for the first time, has extended the attack surface of SGX from the "high-level" microarchitectural design to the underlying physical properties of the electronic circuitry itself. We can only expect more, yet-undiscovered physical effects to be exploited in the future.

The smartcard industry has spent decades defending much less complex chips (typically constrained 8-bit, 16-bit, or 32-bit microcontrollers) against side channels, power glitching and other fault attacks. This has led to countermeasures with substantial overheads. For example, Infineon smartcard chips include the "Integrity Guard" technology [5] in which the same code is executed by two identical CPUs in parallel. The two CPUs constantly cross-check their results to detect fault injection.

The chip layout itself is carefully designed with special meshes to avoid attackers connecting to the internal data lines and stealing or tampering with chip-internal secrets. Third-party labs carry out extensive and expensive tests (e.g. under Common Criteria) to check and certify that the countermeasures are effective.

These overheads and costs may be acceptable for smartcards that protect high-value data in narrow use cases like bank cards or passports. But for general-purpose consumer-grade processors, doubling the size of the whole CPU core would be absolutely prohibitive. So it remains to be seen if Intel and others can learn from the smartcard experience and strike a balance between performance, functionality and security. After all, having a TEE properly secured against physical attacks would open up many fantastic, new applications.

## In summary

With Plundervolt, we created a new and powerful attack that breaks the integrity and (indirectly) confidentiality of SGX. We demonstrated realistic and practical attacks against RSA and AES. Fault injection is not limited to small embedded devices—it is applicable to large scale CPUs, and this opens up the landscape of attacks.

Excitingly, we also show that fault attacks are not limited to cryptographic operations—we introduced controlled memory corruptions, e.g., flipping bits in pointer arithmetic so as to redirect enclave secrets to be written to untrusted memory outside the enclave. As Plundervolt and other fault attacks ultimately break the processor's instruction set specification, even formally verified and bug-free code can be successfully attacked.

PLUNDER VOLT

https://plundervolt.com/

## 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. 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.
3. Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Proceedings of Eurocrypt'97*, pages 37 – 51, 1997.
4. Shay Gueron. A memory encryption engine suitable for general purpose processors. ePrint 2016/204, 2016.
5. Infineon. Integrity guard. online, accessed 2020-04-05: https://www.infineon.com/dgdl/Infineon-Integrity_Guard_The_smartest_digital_security_technology_in_the_industry_06.18-WP-v01_01-EN.pdf?fileId=5546d46255dd933d0155e31c46fa03fb, 2018.
6. Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. *arXiv preprint arXiv:1912.04870*, 2019.
7. 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.
8. 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.
9. Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
10. Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 195–209, New York, NY, USA, 2019. ACM.
11. Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.
12. 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.
13. 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.
14. Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
15. 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.