Finding and exploiting faults in hardware and software

By

Kit Murdock

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
November 2022

ABSTRACT

Computers are constantly being enhanced to improve their speed, size, security, and energy consumption. Dynamic Voltage and Frequency Scaling (DVFS) improves energy efficiency by enabling a procesor to upscale its power as needed, thus using little energy when idle. And, more recently, hardware-based trusted execution environments such as Software Guard Extensions (SGX) have been created with the promise of securely executing sensitive processes—thus protecting the data and running computations from a root adversary.

In the first part of this thesis, we show how the attempt to make computers more efficient by dynamically responding to their energy needs has created a new attack surface. Specifically, we are able to retrieve keys from both an AES and a RSA cryptographic process running inside an SGX enclave by lowering the operating voltage. We further investigate the undervolting effect and are able to improve the attack to create an out-of-bounds under/overflow.

Meanwhile, fault injection attacks (such as our software undervolting one) represent a major threat to Internet-of-Things and embedded devices. As of today, evaluating to what extent a device is susceptible to fault injection is a mostly manual process, requiring significant expert knowledge and often expensive, complex lab equipment. In addition, even if a fault can be induced, it is often unclear which effect caused the incorrect output. In the second part of this thesis, we address this difficulty by designing and building a performant, exhaustive fault injection tool. We compare our software with three others and demonstrate it out-performs on features and speed.

Events During my PhD

- 3 September 2018: I start my PhD.
- 25 November 2018: After more than 18 months of negotiations, EU leaders endorse the UK Brexit withdrawal agreement.
- 31 March 2019: The e-petition calling on the UK Government to revoke Article 50 reaches 6,000,000 signatures.
- 24 May 2019: Prime Minister Theresa May announces her resignation as Conservative Party leader.
- 23 July 2019: Boris Johnson is elected Conservative leader and becomes prime minister. He prorogues Parliament (which the Supreme Court later ruled unlawful). After failing to win parliamentary support for his Northern Ireland Protocol, he calls a snap election.
- 22 October 2019: Abortion is decriminalised and same-sex marriage is legalised in Northern Ireland.
- 12 December 2019: Johnson leads the Conservative Party to a general election victory with 43.6 per cent of the vote.
- 31 December 2019: WHO is informed of a cluster of cases of pneumonia of unknown cause detected in Wuhan City, China.
- 21 January 2020: The USA reports its first confirmed case of the novel Coronavirus.
- 29 January 2020: The UK's first two patients test positive for Coronavirus.
- 23 March 2020: The UK Government announces the first nationwide lockdown.
- 25 May 25 2020: Derek Chauvin murders George Floyd, a 46-year-old black man. Chauvin kneels on Floyd's neck for over nine minutes.

- 14 September 2020: Indoor and outdoor social gatherings in England are limited to six people.
- 31 October 2020: The Prime Minister announces a second lockdown in England to prevent a ‘medical and moral disaster’ for the NHS.
- 6 January 2021: England enters its third national lockdown.
- 6 January 2021: Following Donald Trump’s defeat in the 2020 presidential election, a mob of his supporters attack the Capitol Building in Washington.
- 3 March 2021: 33-year-old Sarah Everard is kidnapped in South London. Metropolitan Police officer Wayne Couzens tells Everard that he is arresting her for having breached COVID-19 regulations. He rapes and strangles her, then burns her body and disposes of her remains in a nearby pond.
- 8 March 2021: England begins a phased exit from lockdown.
- 19 July 2021: In England, legal limits on social contact are removed.
- 1 October 2021: Scotland Yard chief Cressida Dick tells women to ‘wave a bus down’ if they don’t trust a male officer.
- 24 February 2022: Russia invades Ukraine. The invasion causes Europe’s largest refugee crisis since World War 2 and causes global food shortages.
- 11 May 2022: My dad falls at home and is taken to hospital by paramedics. They wait ten hours in the back of an ambulance because no beds are available. He remains there for six weeks.
- 24 June 2022: The US Supreme Court overturns Roe v Wade, the landmark 1973 Supreme Court decision that affirmed the constitutional right to abortion.
- 7 July 2022: Boris Johnson resigns as party leader following a series of scandals and controversies.

- 20 July 2022: Britain's temperatures break the 40°C barrier for first time ever.
- 28 July 2022: Birmingham hosts the Commonwealth Games.
- 6 September 2022: Liz Truss becomes the Prime Minister of the United Kingdom.
- 8 September 2022: Queen Elizabeth II, the UK's longest-serving monarch, dies at Balmoral aged 96, after reigning for 70 years.
- 16 September 2022: A 22-year-old Iranian, Mahsa Amini, is arrested by the Iranian morality police for not wearing a hijab correctly. She dies later in hospital due to having been severely beaten by the police.
- 7 October 2022: My dad has another fall and returns to hospital.
- 16 October 2022: Louise Casey's report into how Scotland Yard deals with officers accused of sexual misconduct and domestic abuse is released. The report uncovers systemic failings including the shocking statistic that: more than half of the Met officers found guilty of sexual misconduct from 2016 to 2020 kept their jobs.
- 20 October 2022: Liz Truss resigns as Prime Minister of the United Kingdom
- 25 October 2022: Rushi Sunak becomes Prime Minister immediately reversing most of Liz Truss's policies.
- 27 October 2022: The UN environment agency releases a report which states there is 'no credible pathway to 1.5C in place', and the failure to reduce carbon emissions means the world is facing 'irreversible' climate breakdown.
- 2 November 2022: I submit my PhD.

ACKNOWLEDGMENTS

A grateful thanks to the sponsors of this degree, I hope to repay the debt in my future work.

My research would not have been possible without the phenomenal, David Oswald—an outstanding person who is empathetic, technical and humorous. These traits rarely appear in one person. Huge thanks to Flavio Garcia for giving me this opportunity, your ability to sprinkle magic into dull academic papers is inspirational.

Thank you to Chris A and Steve H for their unique nudges every month, you left me looking at my research with new eyes.

My experience at the University was immeasurably richer for the following people: Richard Thomas, Tom Chothia, Jesse Spielman, Jo van den Herrewegen and Chris McMahon-Stone.

Incredible thanks to Andreea-Ina Radu for being the *other-woman*! Thank you for letting me moan and rant... and being the female PhD role model I needed. I wouldn't have returned to do a PhD if I hadn't seen you doing it first.

And where would I be without late night conversations about quirky C programs with Mike Stimpson! No one knows more about C than you and it's endearing and annoying in equal measure.

And finally, eternal thanks to my Mum, Sue Murdock and my Dad, Tony Murdock. Neither of you went to university, but you both showed me that I was just as entitled to aspire as anyone else. And when a 10 year-old girl wanted a Commodore 64 for Christmas... you said 'yes'.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

AVX	Advanced Vector Extensions.
DEP	Data Execution Protection.
DRAM	Dynamic Random Access Memory.
DVFS	Dynamic Voltage and Frequency Scaling.
IR	Intermediate Representation.
MEE	Memory Encryption Engine.
MMU	Memory Management Unit.
MSR	Model Specific Register.
OCD	On-Chip Debug.
PHI	Power Hungry Instructions.
PoC	Proof-of-Concept.
RAPL	Running Average Power Limit.
SGX	Software Guard Extension.

SIMD Single Instruction, Multiple Data.

SMBus System Management Bus.

SMT Simultaneous Multithreading.

SVID Serial Voltage Identification.

TCB Trusted Compute Base.

TEE Trusted Execution Environments.

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.

VHF Very High Frequency.

VLF Very Low Frequency.

VR Voltage Regulator.

WLAN Wireless Local Area Network.

XOR Exclusive OR.

Chapter 1

Introduction

From pacemakers to phones and doorbells to cuddly toys, we are increasingly putting embedded processors at the centre of our lives. Statista has predicted that there will be over 30 billion connected devices by 2025 [195]. It is our growing reliance upon these devices that increases their need to be secure, reliable and trustworthy. Many devices are considered safe because of mathematically secure cryptographic algorithms. However these algorithms can only be relied upon under correct operating conditions—when something goes wrong the mathematical properties may no longer hold and the device may leak sensitive information.

1.1 Faults, Fault Injection and Fault Attacks

The *something goes wrong* is known as a *fault* and can be produced by anything from ageing components to power-cuts, from design deficiencies to unexpected inputs. One small fault can propagate through a system to create an incorrect state or output: the fault has then created an *error*. In 1982, software engineers at Digital Equipment Corporation (DEC) [191] coined the term *fault injection*—its purpose, as originally conceived, was to test whether a system could withstand faults that occurred during runtime. Initially, these faults were simulated hardware failures that corrupted data or prevented normal operations.

In 1997, Boneh et al. evolved the concept of fault injection by producing one of the first ever *fault attacks* (Bellcore), against a CRT-RSA signature [21]. By faulting a single calculation they were able to retrieve the entire private key. And, in 2011, a single byte fault was shown to be sufficient to extract the key from an AES encryption[203]. As a result of these initial results, a profusion of attack vectors sprung up. Researchers attempted to fault the computations of a device by directly modifying its environment or hardware. Typically, such fault-inducing environmental changes are at the border of (or beyond) the specified operational range of the target device. The research into fault injection has been surprisingly creative: Optical fault attacks [194], clock glitches [9, 193], voltage glitches [225], electromagnetic pulses [42], electromagnetic fault attacks [41], focused photon injection using laser beams [40, 178] and extreme temperatures [87]. The aim of a fault attack has also expanded, it now includes: retrieving cryptographic keys, privilege escalation [186, 69, 217, 211], bypassing firmware security mechanism [210] and reading otherwise inaccessible memory locations [122].

But not all fault attacks require physical intervention, in 2015, Rowhammer [111], created the first purely software-based fault attack against x86-based systems. Rowhammer was able to cause bit flips in DRAM memory by constantly writing to a target row of memory. Nearby rows (that were not accessed) were affected because the memory cells interact electrically by leaking their charges. Several authors [172, 68, 122] furthered the work with new applications, variations, and improvements of the original attack, including the successful bypass of countermeasures in recent DDR4 DRAM chips [58]. Google's Project Zero created two privilege escalation attacks against the x86-64 architecture [186], demonstrating the severity of the attack. Because Rowhammer could flip bits from *software only*, the mitigations needed to be profound. Consequently, the scientific community and industry have put significant effort in developing Rowhammer mitigations [111, 100, 80, 10, 70, 163, 33, 221, 36, 211, 69, 25]. Interestingly, Rowhammer (and software-based fault attacks generally) now cause the threat model to shift from a local attacker (with physical access to the target device) to a, potentially, remote attacker with only local code execution. There is little doubt that if an attacker does not need direct access, the risks are raised.

1.2 Trusted Execution Environments

Before continuing, we need to take a quick pit-stop to look at how computer chip manufacturers have attempted to manage the ever-growing need for *trust* with our devices.

As far back as 1985, the term Trusted Compute Base (TCB) was coined [179] to refer to the small quantity of software and hardware that security replies upon e.g., in the event of a kernel driver coding error the system security could be fully compromised. And, in the last three decades, software security has increasingly been directed toward security functions in both software and hardware. An operating system has to manage multiple applications and resources and, to ensure safety from potentially malicious applications, modern operating systems employ a number of protection mechanisms. For example, vendors have introduced the NX bit to enforce Data Execution Protection (DEP). Similarly, protections rings were introduced employing hierarchical levels of privilege (usually hardware enforced). Ring 0 (Kernel) is the level with the most privileges and has access to control the physical hardware such as the CPU and memory. Ring 3 is usually referred to as *userspace* and applications that running in userspace will need to request services from the kernel.

Additionally, memory isolation is managed by a dedicated Memory Management Unit (MMU) which is responsible for mapping physical memory addresses to virtual addresses by consulting a page-table tree structure. The MMU is configured and managed by the Operating

System. Memory isolation ensures that processes running in userspace cannot access the physical memory assigned to a different process and provides the foundations for software security. Theoretically, one process cannot access the memory of another.

In the last thirty years, large amounts of research have pushed the security of operating systems, mostly through micro-kernels [113]. But, as the features and environments grew, so did the attacks. Computers and mobile devices became *feature-rich* leading to more and more attack surfaces.

The complexities of creating operating system security and memory security grabbed industry and academia's attention and, as mobile phones turned into smart-phones, the desire for hardware security grew. From this, a large line of work [155, 222, 139, 4, 106, 5], lead to the creation of hardware-based Trusted Execution Environmentss (TEEs). A TEE is an area on the main processor of a device with minimal TCB and separate from the operating system. Data is stored, processed, and protected in this secure environment. They provide protection by enabling an isolated area with end-to-end security—which includes executing code. The memory is encrypted with integrity protection *i.e.*, an injected bit-flip would be trapped and reported. Additionally, TEEs use attestation to demonstrate that the software is properly instantiated on a platform, this is usually done with the aid of an attestation primitive to cryptographically verify that a specific enclave has been loaded on a genuine TEE processor. At a human level, 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.

For these reasons, Intel processors (from 2015 onward) include Software Guard Extension (SGX) (a TEE) which allows an application to self-quarantine sensitive data and functions in an *enclave* using dedicated CPU instructions. These 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 [74] that protects the confidentiality, integrity and freshness of all enclave memory while it resides in untrusted off-chip DRAM. Indeed Intel's own threat model considers main memory as an *untrusted* storage facility and fully encrypts and authenticates all memory within the Intel SGX enclave security architecture [74].

Meanwhile, in 2004, ARM proposed a system-wide hardware isolation execution environment [4], marketed as *TrustZone Technology* as its Trusted Execution Environment. This has since become widespread in Android mobile devices mostly used for: (*i*) Storing keys (*ii*) Cryptographic operations (*iii*) Trusted boot (*iv*) Digital signatures.

Although this thesis is focused on fault injection, as we will see, protecting a TEE against a privileged adversary opens up a whole new set of challenges.

1.3 Energy Management Systems

In order to continue, we need to take a moment to examine Energy Management Systems—in later chapters we will combine all of these knowledge areas.

Hardware is being aggressively optimised to meet the growing need for fast response times. The aim: to maximise performance and energy savings 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. Because of this relationship, modern processors keep the clock frequency and supply voltage as low as possible—only 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.

Modern computers have, therefore, introduced the energy management architectural mechanism: Dynamic Voltage and Frequency Scaling (DVFS) to reactively control the physical operating conditions of the underlying hardware. At the time of writing, DVFS was available on ARM, AMD and Intel CPUs. DVFS can be used to offer high performance for intensive mathematical calculations—dubbed Power Hungry Instructions (PHI). But DVFS is not just for performance, it is also a safety feature to protect the physical board: when thermal meters warn of approaching unsafe values, the processor will quickly and temporarily reduce the power and/or frequency: this is known as *throttling*.

Memory mapped registers were introduced by Intel to support these dynamic hardware changes. Starting with the Pentium processor, Intel created a pair of instructions (`RDMSR` and `WRMSR`) to access current and future Model Specific Register (MSR), as well as the `CPUID` instruction to determine which features are present on a particular model. Simply put: a model-specific register is an x86 instruction control register used for debugging, program execution tracing, computer performance monitoring, and enabling/disabling CPU features.

As we saw in the previous section, the TCB is the set of components critical to the security of the system, it is vital therefore that these software interfaces cannot be used maliciously. In 2017, Tang et al. [198] created a fault attack against ARM's TEE to create a new class of software-based fault attack—those using the energy management systems. They demonstrated that ARM processors allow configuration of the dynamic frequency scaling feature, *i.e.*, overclocking, by system software. Using this, Tang et al. showed that overclocking features can be abused to jeopardise the integrity of computations for privileged adversaries in a 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.

1.4 What Just Happened?

Given the wide range of faults attacks, it is not surprising that the research in this field shows no sign of waning. However, despite extensive research, carrying out practical attacks and thus evaluation of countermeasures remains challenging. Firstly, the parameter space for fault injection is usually large—the physical parameters of the fault must be carefully chosen, e.g., regarding fault intensity, duration, trigger point, and so on.

Then, even when a faulty output or program behaviour has been discovered, it often remains unclear *how* the fault was produced (*i.e.*, which fault model affected which instruction in the executed program code). Second, we must have a thorough knowledge of the architectures, memory and executables being used. This can take hours of painstaking and time-consuming examination into the particular device and software. And, throughout all this, we run the risk of breaking the device under test.

As of today, evaluating to what extent a device is susceptible to fault injection is mostly a manual process requiring significant expert knowledge and often expensive, complex lab equipment. In addition, even if a fault can be induced, it is often unclear which effect caused the incorrect output. The evaluator only observes a faulty output and faces the Herculean task of working out *What just happened?*

One approach explored in the research community to address this problem is the use of formal methods (like symbolic execution) to model the behaviour of software programs [167, 65] or hardware circuits [174] under fault injection. An alternative approach is to simulate the injection of various fault types into every target instruction using an emulator for the target CPU architecture. While the approach of *fault simulation* has disadvantages compared to formal methods (e.g., that the results depend on the concrete input values), it also offers benefits: it covers various CPU architectures as long as a basic emulator is available and readily applies to arbitrary firmware binaries. In this thesis, we thus focus on the latter approach and in particular improve on the performance compared to earlier tools for fault simulation, inspired by ideas from the related area of *fuzzing*.

1.5 Dissertation Scope

In Chapter 2 we begin by continuing the work introduced by CLKscrew [198] and use Intel's energy management systems to create a novel software fault injection attack against SGX—this work was presented at S&P in 2020 [149] and also in IEEE Security and Privacy, 2020 [150]. We further investigate its faults using the more refined hardware undervolting in Chapter 3, this work was published as part of Voltpillager at USENIX in 2021 [32]. We then attempt to address the previously identified question: *What just happened?* by designing, implementing and testing a performant application called Faultfinder which is detailed in Chapter 4. And finally, we investigate leakage from Intel's power management interfaces in Chapter 5.

1.6 Published Papers

- Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. 2020. [149]
- Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Frank Piessens, and Daniel Gruss. "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble". *IEEE Security and Privacy, special hardware edition.* 2020. [150]
- Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface" . *30th USENIX Security Symposium (USENIX Security 21).* [32]

1.7 Conference Presentations

- "Plundervolt: Flipping Bits from Software without Rowhammer" [71] at *36th CCC*. December 2019.
- "Plundervolt: Pillaging and plundering SGX with Software-based Fault Injection Attacks" [147] at *Redhat*. January 2020.
- "Plundervolt: Software-Based Fault Injection Attacks against Intel SGX" [148] online presentation at *S&P*. March 2020
- "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble" [72] at *Blackhat*. August 2020.

Chapter 2

Plundervolt

2.1 Introduction

This chapter is based on the author's published work at S&P 2020 [149] and in IEEE Security and Privacy, 2020 [150].

Energy management systems in computers efficiently control operating voltages and frequencies, scaling up for high short bursts of intense activity, then returning to lower values to protect hardware components. Aggressive energy optimisations have resulted in the stable voltage margins shrinking. 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.

In this chapter, we investigate Intel's energy management systems and the memory mapped registers used to control the dynamic voltage and frequency scaling from software. Although, we introduced this concept in Chapter 1, we did not highlight that these features are often undocumented. What is more, they are only exposed to privileged system software—

consequently they have been scarcely studied from a security perspective. In this chapter we focus on one specific memory mapped register: `MSR_OC_MAILBOX (0x150)` which allows core voltage to be modified—both up and down—and even to the point of full kernel panic and crash. We also saw in Section 1.2 that Intel's SGX enclaves are 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 [68, 102], halting the entire system.

2.1.1 Related Work on Software-based Fault Attacks

Whilst CLKscrew [198] was able to exploit these energy management systems on an ARM processor, it is unclear whether similar effects exist on x86-based computers and, if so, whether they are exploitable or not. It is unknown if SGX has has protections against this type of attack, e.g., machine-check errors on the system level, or SGX enclave data integrity. Furthermore, CLKscrew is based on changing the frequency, we focus on manipulating the voltage, because, as we will see, the granularity is finer on Intel-based machines. Finally, the question arises: are faults limited to software implementations of cryptographic algorithms (as in CLKscrew), or can the faults also be used to exploit hardware implementations (like AES-NI) or generic (non-cryptographic) code.

2.1.2 Our Contribution

Here 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 [197, 175, 144]. 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:

- 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. We provide a thorough analysis of the fault characteristics on recent Intel processors.

- 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.

- In both cases, we recover the full key with a single successful fault injection and negligible computational efforts. We are able to recover the full 128-bit AES key within a couple of minutes.

- 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.

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

2.1.3 Responsible Disclosure

We responsibly disclosed our findings to Intel on June 7, 2019. Intel reproduced and confirmed the vulnerabilities which they are tracked under CVE-2019-11157. Intel's mitigation is provided in Section 2.7.3.

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 virtualised environments, hypervisors should never allow untrusted guest VMs to read from or write to undocumented MSRs.

The PoC attack code is available at: https://github.com/KitMurdock/plundervolt.

2.2 Experimental Setup

2.2.1 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 [208]. 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 2.1).

2.2.2 Voltage Scaling on Intel Core Processors

We build on the reverse engineering efforts of [197, 144, 175] 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 2.1).

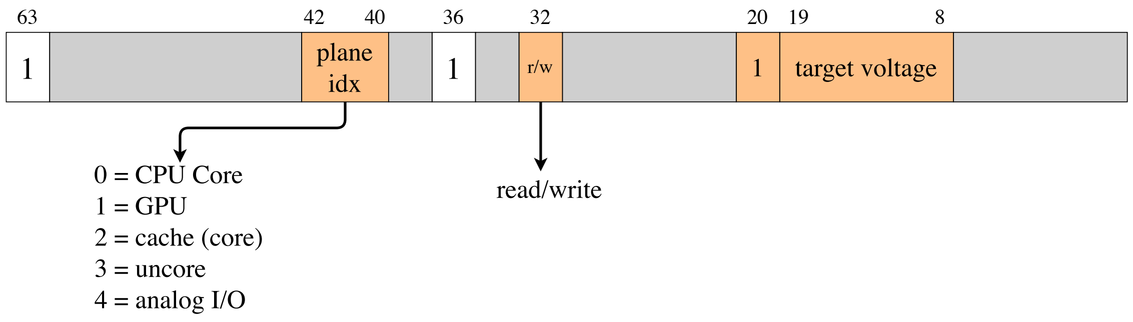


Figure 2.1: Layout of the undocumented MSR `0x150` (for set voltage)

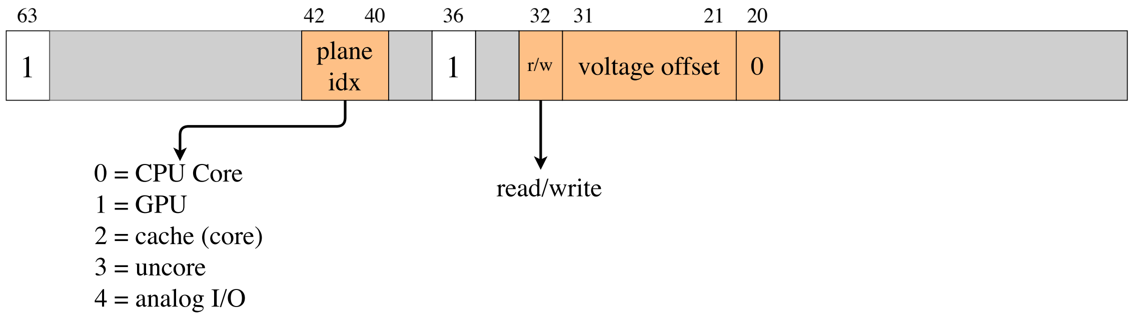


Figure 2.2: Layout of the undocumented MSR `0x150` (for undervolting)

Figures 2.1 and 2.2 show how the 64-bit MSR value can be decomposed into a *plane index (idx)* and a *voltage offset* or *target voltage*. Firstly, by specifying a valid plane index, system software can select to which CPU components the voltage change should be applied. The CPU core and cache share the same voltage plane on all machines we tested and the higher

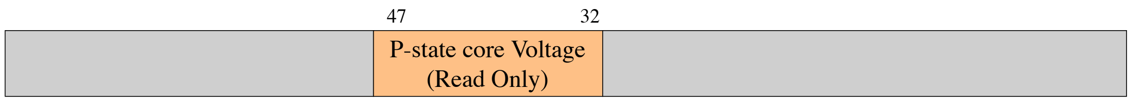


Figure 2.3: Layout of the documented MSR `0x198` - reading core voltage

voltage of both will be applied to the shared plane. We focused our efforts on undervolting always starting with small steps to help ensure the stability of the device under test. 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`), (Figure 2.3) using the calculation below.

$$Voltage(volts) = \frac{IA32\_PERF\_STATUS[47:32]}{2^{13}}$$

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 [90] and older, public Intel documentation [98]).

2.2.3 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.1.

The undervolting is applied by writing to 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 minimised.

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 2.3.1 for a more precise analysis).

2.2.4 Undervolting Decline Micro-benchmark

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

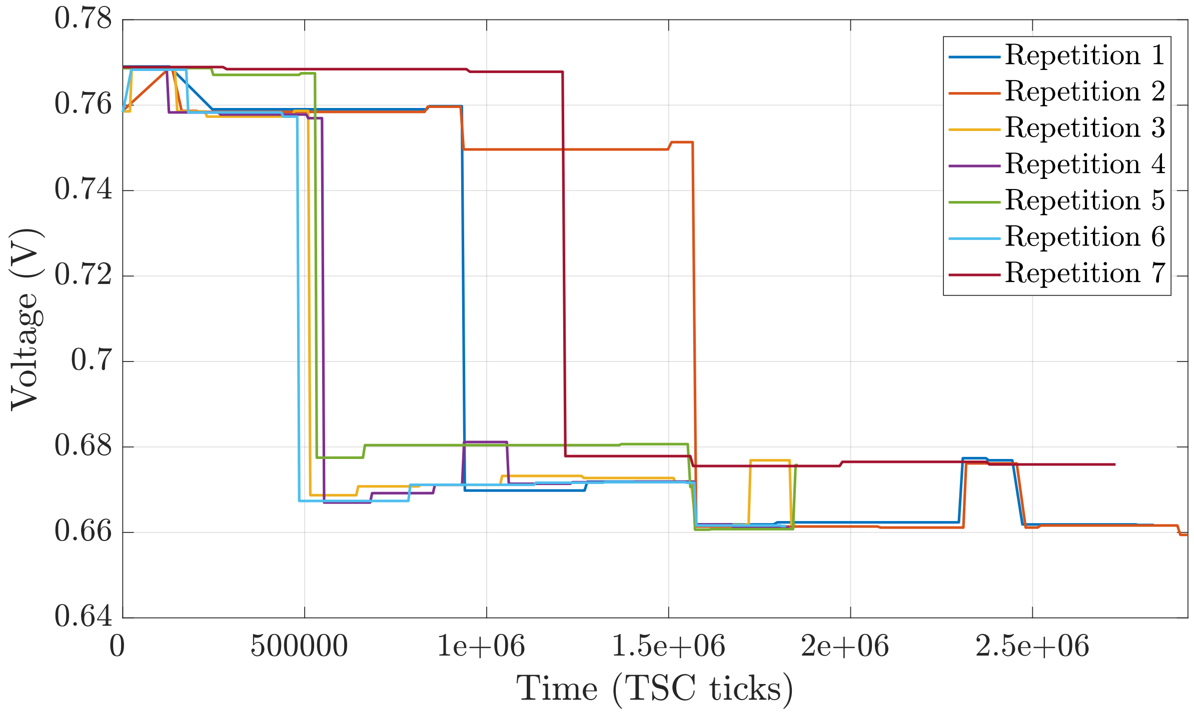


Figure 2.4: Voltage decline over time for Intel i3-7100U-C, repeating a -100 mV undervolting seven times and measuring actual voltage in MSR `0x198`.

The measurement results for seven repetitions of a -100 mV drop are displayed in Figure 2.4. 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 2.7 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.

2.2.5 Tested Processors

For our experiments, we used different SGX-enabled processors from Skylake onwards, cf. Table 2.1. 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 2.3.1), 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.

2.2.6 Ambient Temperature

All experiments were conducted in a university office environment with a typical UK room temperature.

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	✓	✓

Table 2.1: Processors used for the experiments in this chapter. When multiple CPUs with the same model number were tested, we append uppercase letters (-A, -B etc).

2.2.7 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 [156]. 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 2.3.1.

2.3 Faulting In-Enclave Multiplications

As a first step towards practical fault injection into SGX enclaves, we analysed 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 optimised 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 2.4.3).

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`.

2.3.1 Analysis of Undervolting Effects on Multiplications

Using MSR `0x198` (`MSR_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 2.5 and Figure 2.6 show the measured relation between frequency, normal voltage (blue), and the necessary undervolting to trigger a faulty multiplication inside an SGX enclave (orange) for the i3-7100U-A and an i7-8650U-A, respectively.

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

```
/* 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 2.2 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 2.2 and the fault used in Section 2.5.1. From those results, we noted:

- The smallest first operand to fault was `0x89af`;

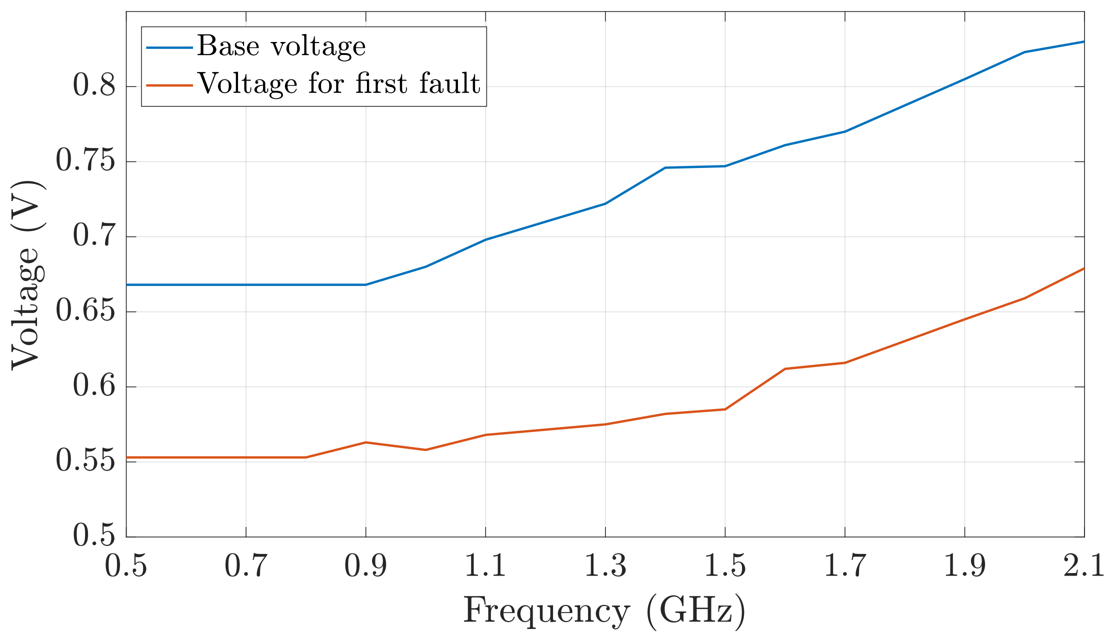


Figure 2.5: Base voltage (blue) and voltage for first fault (orange) vs. CPU frequency for the i3-7100U-A

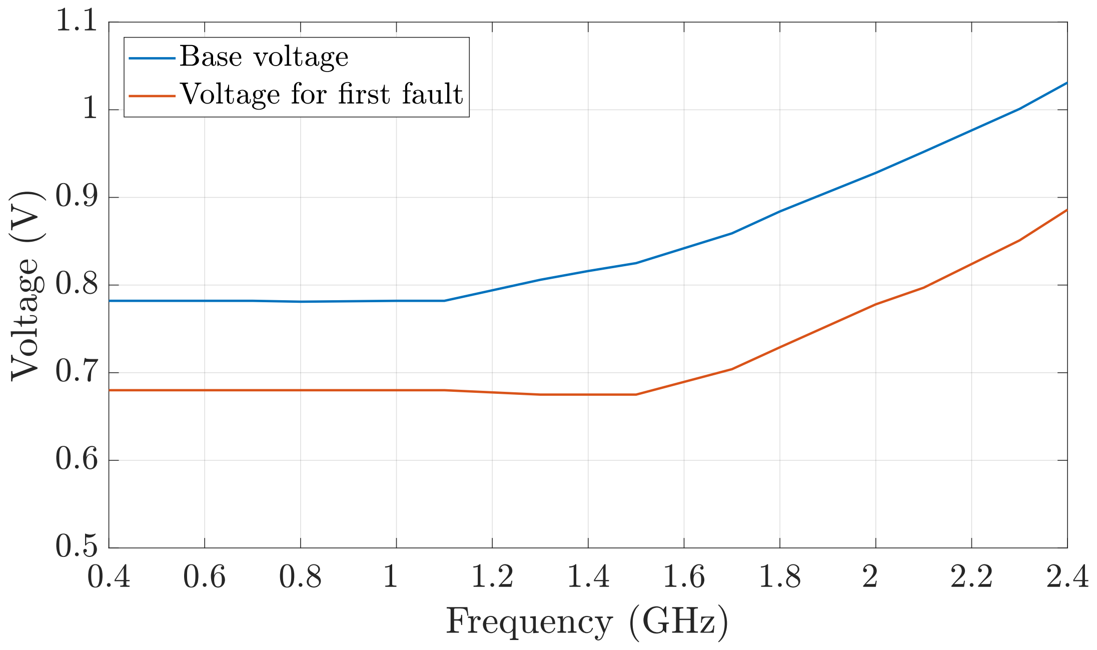


Figure 2.6: Base voltage (blue) and voltage for first fault (orange) vs. CPU frequency for the i7-8650U-A

- 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.

Start value	**Multiplier**	**Faulty result**	**Flipped bits**
0x080004	0x0008	0xffffffff0400020	0xffffffff0000000
0xa7fccc	0x0335	0x000000020abdba3c	0x0000000010000000
0x9fff4f	0x00b2	0x000000004f3f84ee	0x0000000020000000
0xacff13	0x00ee	0x000000009ed523aa	0x000000003e000000
0x2bffc0	0x0008	0x00000000005ffe00	0x0000000001000000
0x2bffc0	0x0008	0xffffffff15ffe00	0xffffffff0000000
0x2bffc0	0x0008	0x00000100115ffe00	0x0000010010000000

Table 2.2: Examples of faulted multiplications on i3-7100U-B at 2 GHz

We also investigated the iterations and undervolting required to produce faults (cf. Table 2.3) 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 Figure 2.5 and Figure 2.6, we used a large number of 100,000,000 iterations, so faults occur with relatively low undervolting already.

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

Table 2.3: Number of iterations until a fault occurs for the multiplication `(0xae0000 * 0x18)` vs. necessary undervolting on i3-7100U-B at 2 GHz.

2.3.2 Differences between CPUs with Same Model Number

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).

2.3.3 Temperature Dependencies

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 noted that on very hot UK Summer days, putting the machines into a refrigerator or air vent decreased the likelihood that the computer would crash before producing a fault. Differences in "stability" depending upon the temperature of the fault warrant further investigation. However, all our attacks were performed at room temperature and caused no impediments.

2.3.4 Overvolting

The VID interface specification limits the maximum voltage to 1.52 V. According to the CPU datasheets [91], 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.

2.4 From Faults to Enclave Key Extraction

Having demonstrated the feasibility of fault injection into SGX enclaves in Section 2.3.1, 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.

2.4.1 Corrupting OpenSSL Signatures

We first developed a simple proof-of-concept application using OpenSSL in userspace. This application runs the multiplication loop from Section 2.3.1 until the first fault occurs (to make sure the system is in a semi-stable state) and then invokes OpenSSL as follows:

```
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.

2.4.2 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 [92].

Given an RSA public key (n, e), the corresponding private key $(d, p\,q)$ and the encrypted value x, RSA-CRT can speedup the decryption 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 [21], 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 A.2.

2.4.3 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 approximately $^1/_{10}$, 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:

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

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

```
[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. [203] 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 A.4.

[3]https://github.com/Daeinar/dfa-aes

2.4.4 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 2.4.3. 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. [51] provide an overview of fault attacks against elliptic curves, while other researchers [59, 44] discuss faults in nonce-based encryption modes like AES-GCM.

2.5 Memory Safety Violations due to Faults

In addition to the extraction of cryptographic keys, we show that Plundervolt can also cause memory safety misbehaviour 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 2.3.1). 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 [50] as well as state-of-the-art Spectre-style [114] 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 analyse trusted SGX runtime libraries and locate several sensitive multiplications in allocation size computations that could lead to heap corruption by allocating insufficient memory.

2.5.1 Faulting Array Index Addresses

We first focus on the case where a multiplication is used for computing the effective memory address of an array element as follows: `&a[i] = &a[0] + i * sizeof(elem_t)`. However, we found that, in most cases, when the respective type has a size that is a power of two, compilers will use left bitshifts instead of explicit `imul` instructions. Furthermore, as concluded from the micro-benchmark analysis presented in Section 2.3.1, we found it difficult (though not impossible) to consistently produce multiplication faults where both operands are $\leq$ `0xFFFF` without crashing the CPU (cf. Section 2.5.2). Hence, here we only consider cases in this section where `sizeof(elem_t)` $\neq 2^x$ and $i > 2^{16}$.

2.5.2 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 2.3.1 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.

`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 [93, 206]. 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 2.3.1, 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

`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 [168] library OS for running unmodified C applications inside enclaves[4]:

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

In this case, if the product `n *= m` can be faulted to produce a smaller result, subsequent code may trigger a heap overflow, eventually leading to memory leakage, corruption, or possibly even control flow redirection when neighbouring heap chunks contain function pointers e.g., in a vtable. Based on practical experiments with the i3-7100U-A, we artificially injected a realistic fault for the product `0x2bffc0 * 0x8 = 0x15ffe00` 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 [93] and Microsoft's OpenEnclave [143], but interestingly found that their implementations are hardened against (traditional) integer overflows as follows:

[4]https://github.com/lsds/sgx-lkl-musl/blob/db8c09/src/malloc/malloc.c#L352

```
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 ≤ `0xFFFF` (cf. Section 2.3.1) is possible, e.g., we got a fault for `0x97b5 * 0x40` on the i3-7100U-A. However, in the majority of attempts, this leads to a crash because the CPU has to be undervolted to the point of becoming unstable. The above check (without the restriction on only being active for at least operand being > `0xFFFF`) serves as an example of possible software hardening countermeasures, as discussed in Section 2.7.

2.6 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. [66] 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. [12] 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 [186] showed how to flip operand bits in x86 instruction streams to escape a Native Client sandbox, and more recently Gruss et al. [68] 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 [74], which reliably halts the system if an integrity check fails [68, 102].

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 [198] 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 2.7, 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 [204, 30, 126, 209, 146, 75, 207] as well as exploiting classical memory safety vulnerabilities in enclaves [125, 19, 206], 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 [201] 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 [204] 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 2.5.1 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.

2.7 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 summarising the specific mitigation strategy adopted by Intel.

2.7.1 Hardware-Level and Microcode-Level Countermeasures

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 [90, 98]. 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).

Scaling Back Voltage during Enclave Operation Plundervoltrelies 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 [96, pp. 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 defence 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 defence 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. Figure 2.4). Therefore, removing and restoring undervolting on each enclave entry and exit would likely add a substantial overhead.

Limiting to Known Good Values Even slightly undervolting the CPU creates significant power and heat reductions; properties that are highly desirable in data centres, 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 white-listed 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 2.3.1). 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 [108].

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 [85, 212, 118] 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 [96] 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 [204, 181, 184, 146] demonstrated how enclave secrets can be reconstructed from a sibling CPU core, Intel officially recommended disabling hyperthreading when using SGX enclaves [97]. However, this also imposes a significant performance impact on any non-SGX workloads.

A well known solution to fault injection attacks is redundancy [78], 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. [160] and later Reis et al. [173] 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.

2.7.2 Software-Level Hardening

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 [78] and algorithm-specific approaches such as performing the inverse operation or more advanced techniques like ineffective computation [62].

For the example of RSA-CRT signature/decryption (cf. Section 2.4.2), 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 2.4.3), 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 [59].

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 2.5. 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 [93] contains 23 multiplications, with many in standard library functions like `free()`. For comparison, the trusted runtime part of Microsoft's OpenEnclave SDK [143] contains 203 multiplications, while Graphene-SGX's [202] `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 2.5.2). 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, defences should also take into account the possibility of faulting other high-latency instructions.

Traditional Memory Safety Hardening As a final consideration, we recommend applying more general countermeasures known from traditional memory safety application hardening [50] in an enclave setting. One approach to hinder Plundervolt-induced memory safety exploitation would be to randomise the enclave memory layout using systems like SGX-Shield [187]. 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.

2.7.3 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 2.7.1. 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 [204], Intel's mitigation plan for Plundervoltrequires trusted computing base recovery [5, 39]. 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.

2.8 Conclusion

In this chapter 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.

We have proven that this attack vector is realistic and practical with full key recovery PoC attacks against RSA-CRT and AES-NI. 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.

Chapter 3

VoltPillager

3.1 Introduction

The work included in this section represents the author's main contribution to the publication, "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface" by Chen et al. at USENIX 2021 [32].

Intel addressed the vulnerability disclosed in Chapter 2 by providing features to disable software undervolting through MSR `0x150`. Because SGX was compromised, Intel initiated Trusted Computing Base recovery and modified remote attestation to verify that software-based undervolting is disabled. This requires microcode and BIOS updates. However, hardware fault injection considers a different adversarial model where the adversary has physical access to the device under attack. When targeting an SGX enclave running on a fully patched system (with the latest microcode and BIOS updates), software-based fault attacks have been fully mitigated and that is where hardware-based attacks become relevant.

Fault attacks induce a computation fault in the target processor, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability or any code execution privileges. Voltage fault injection (aka, glitching) in particular has the advantage of being very powerful whilst not requiring expensive lab equipment.

Colleagues at Birmingham University (Zitai Chen et al.[32]) analysed the voltage scaling feature of the x86 system at the hardware level. They found that a three-wire bus, Serial Voltage Identification (SVID), is used to send the currently required voltage to an external Voltage Regulator (VR) chip on the motherboard. The VR then adjusts the voltage supplied to the CPU. They reverse-engineered the communication protocol of SVID and developed a small microcontroller-based board that can be connected to the SVID bus. As there is no cryptographic authentication of the SVID packets, they were able to inject their own commands to control the CPU voltage. With this, they reproduced the Plundervolt Chapter 2 attacks, including against code running inside an SGX enclave. Because the software interface MSR `0x150` is not used, Intel's countermeasures do not prevent this attack. They were the first to practically showcase a hardware-based attack that directly breaches SGX's integrity guarantees. They demonstrated this practically with end-to-end secret-key recovery attacks against mbed TLS and the unmodified `file-encryptor` sample enclave from Microsoft Open Enclave.

3.2 Undervolting Controls

One of the advantages of controlling the SVID externally is that we can change the timing of undervolting with a finer granularity. We discovered a number of new faults that we had not been able to create with *software-only* undervolting. We tested a range of new scenarios and discovered some new fault-injections, these are detailed in the next sections.

3.3 New Undervolting Attacks

Consider the code in Listing 3.1 (`operand1` and `operand2` were originally set to identical random values).

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

Listing 3.1: Modified C code used for demonstrating fault injection into memory accesses

Throughout the running of this code both operands should be equal and constant. However, fault injection occasionally led to `operand1` being only decremented, with the increment on Line 5 seemingly ignored. Conversely, when swapping the order and first decrementing followed by incrementing, `operand1` took the value `operand2 + 1` when undervolted *i.e.*, the decrement had not taken effect.

We disassembled lines 5 and 6 in Listing 3.1 as shown in Listing 3.2. From these observations, we conjecture that the most likely explanation for the observed faults is that *recent* memory (cache) writes are delayed and thus ignored in adjacent reads of the modified location. This suggests that the fault affects the load-store queue logic of the cpu, causing writes to be delayed for a few cycles while the execution of dependent instructions progresses with old values.

```
      83 45 dc 01             add    DWORD PTR [rbp-0x24],0x1
      83 6d dc 01             sub    DWORD PTR [rbp-0x24],0x1
```

Listing 3.2: Assembly code for lines 5-6 from Listing 3.1

```c
uint32_t array[8] = { 0 };
// Attacker-supplied out-of-bounds size
int copy_size = 7;

// Ensure we stay within bounds
if(copy_size >= 5)
  copy_size = 4;

// overwrite elements 4, 3, 2, 1
while(copy_size >= 1)  {
  array[copy_size] = 0xabababab;
  copy_size--;
}
```

Listing 3.3: Proof-of-concept to demonstrate out-of-bounds memory accesses due to undervolting

For example, for the (undervolted) assembly code sequence `add DWORD PTR [rbp-0x24],0x1; sub DWORD PTR [rbp-0x24],0x1` the sub operates on the previous value, ignoring the update through the preceding increment.

3.4 Practical Exploitation Scenario

We now consider a realistic scenario to exploit these effects. The experiments described in this subsection were all performed inside an SGX enclave. We show how a delayed-write fault can be used to trigger out-of-bounds accesses in memory-safe code. To this end, the Proof-of-Concept (PoC) code shown in Listing 3.3 initialises elements of an array to a fixed value. We then use the following code to write `0xabababab` to elements 4 to 1.

In Listing 3.3, `array[]` holds eight `uint32_t` elements all initially set to zero. The code first ensures that the (potentially adversary-controlled) upper bound `copy_size` is ≤ 4, using a code pattern that effectively implements `min(4, copy_size)`. It then proceeds to write `0xABABABAB` to array elements 4 to 1, leaving the other elements at their initial value of zero.

We are intentionally only writing to part of the eight-element array in this PoC to avoid triggering actual stack corruptions (and hence crashes). However, note that all experiments also apply to a real scenario, where the attacker would write beyond array bounds and corrupt the enclave stack, thus gaining control over the program counter and applying traditional exploitation techniques afterwards [125].

We undervolted the CPU whilst executing the above code in a loop within an `ecall` handler. The experiments were run on i7-7700HQ-XPSat a frequency of 2 GHz, undervolting by -170 mV. The core temperature reported by the CPU varied between 44° C and 49° C. We observed two distinct effects induced by the fault (cf. Appendix B.1), as illustrated in Figure 3.1: (*i*) in addition to elements 4 to 1, element 0 was also overwritten (*i.e.*, an underflow) and (*ii*) the upper bound was not limited to 4 but stayed at 7, *i.e.*, an overflow into elements 5 ... 7 occurred. In both cases, out-of-bounds accesses take place, leading to

Normal execution:

00...	AB...	AB...	AB...	AB...	00...	00...	00...

Fault 1 causing out-of-bounds underflow:

AB...	AB...	AB...	AB...	AB...	00...	00...	00...

Fault 2 causing out-of-bounds overflow:

00...	AB...	AB...	AB...	AB...	**AB...**	**AB...**	**AB...**

Figure 3.1: State of `array[]` after normal execution of Listing 3.3 and out-of-bounds under/overflow when undervolted. Faulty values in red bold font.

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

Case 1: Out-of-Bounds Underflow As shown in Figure 3.1, undervolting caused `array[0]` to be incorrectly overwritten. Our analysis showed that this is due to a fault affecting the code responsible for decrementing and directly afterwards comparing the loop counter on

Lines 12 and 10 in Listing 3.3, which translates to the assembly code shown in Listing 3.4.

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

Listing 3.4: Assembly affected by underflow

When undervolting, we observed the decrement of the loop counter on Line 9 in Listing 3.4 had not been committed by the time the comparison on Line 2 occurs. Thus, the loop performs one additional iteration for `copy_size` = 0. We found that the decrement *does* come into effect on the subsequent read into `%rax` on Line 5, which is the index into the array, hence overwriting `array[0]`.

Case 2: Out-of-Bounds Overflow In the second observed fault, elements 5–7 are incorrectly overwritten. In this case, we concluded that the fault affects the initialisation of the upper limit on Lines 6 and 7 in Listing 3.3. The respective assembly snippet is shown in Listing 3.5.

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

Listing 3.5: Assembly affected by overflow

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

3.5 Conclusion

Taking control of the SVID enabled undervolting to be more finely controlled, this allowed us to refine our understanding of the injected faults. Through this investigation we discovered potentially more powerful attacks creating buffer overflow and underflow exploits. Again, using undervolting, this work successfully recovered RSA keys from an enclaved application.

Chapter 4

Faultfinder

4.1 Introduction

As we saw in Chapter 1, researchers in both industry and academia use fault injection to evaluate the reliability and resilience of systems and applications. Faults can be injected intentionally (maliciously, for example) or unintentionally (environmental factors, for example)—both fault-reasons need to be protected against.

Fault injection can help evaluate the reliability of an application—simulating different fault scenarios can help pin-point weak spots in code. However, where a binary is needed to behave correctly under *all* possible conditions, formal methods are often used [167, 65]. Formal logic can provide a guarantee of correctness—this approach is highly effective in identifying errors but can be time-consuming and expensive to implement. Consequently, formal methods are well-suited for testing small, critical parts of a binary that require a high level of confidence in their correctness and reliability, such as those used in aviation or medical applications.

Alternatively, exhaustive fault injection simulation, involves systematically injecting faults into a binary. This approach can be automated and may identify vulnerabilities that will be missed by other testing methods. Simulation scales well and can be used to test large and complex systems, making it suited for those that require a more comprehensive and practical approach.

And finally, manual fault injection, using a highly skilled and knowledgeable person is incredibly slow. Firstly, the parameter space is huge, even with the source code available (which it often is not). Historically, identifying a good fault-target in a binary can take weeks: analysing the binary, finding potential promising targets and then prioritising them. And even if a target injection site is not found, that does not mean it does not exist. Exhaustive fault-injection enables millions of faults to be injected and inspected without any need for a human to initially suggest a 'potential target site'.

In the last two decades many attempts have been made to fill the much needed gap-in-the-research-area hole. We discuss these in the next section and then proceed to provide a solution which is fast and works across a plethora of processors.

4.1.1 Related Work

To simulate fault injection on different architectures, most fault simulation tools are based on generic emulators: In 2012, QEmu Fault Injection (QEFI) [34] was presented as one of the first fault simulation frameworks. It is based on QEMU, an open-source emulator and uses the GNU Debugger (GDB) for the fault injection. QEFI allows fault to be injected into registers and memory, in addition to faults on storage devicess and network interterfaces. QEFI focuses on the ARM architecture and uses a Python-based script API as the main user interface. QEFI implements its fault injection using QEMU'S TCG (Tiny Code Generator) plugin, and the authors note that this negatively impacts the performance.

At the same time as QEFI, XEMU was published [16]. XEMU is a framework for mutation-based testing of binaries. The authors also extended QEMU with the TCG to inject faults at runtime. The fault injections are based on a mutation table generated from the Control Flow Graph (CFG) of the disassembled code. The authors of XEMU also limited their efforts to ARM binaries, specifically ones for vehicle engine management. XEMU uses a *golden run* (*i.e.*, an execution trace with no faults) and compares each mutated run to the golden run. If the mutated run writes a single character differently to the golden run, execution stops to save unnecessary overhead. The output of XEMU is a report detailing instruction coverage, mutation coverage, and how many mutants were discovered.

In 2015, Ferraretto et al. [53] presented fault injection simulation using QEMU, considering three different types of faults: *stuck-at*, *transition*, and *bit flip*. They demonstrated their tool with three benchmark binaries: `btrees`, `mandelbrot` and `dhrystone`. They reported the results as one of four effects: *loop*, *safe*, *error*, and *crash*.

ARMory [84] bucked the QEMU-trend by presenting a a fault simulation tool for the ARM-M architecture based on the "M-ulator" emulator—which the authors wrote themselves. ARMory is security-focused in its design approach and includes an *exploitability model*, *i.e.*, the conditions under which the fault would pose a security issue. ARMory is of particular interest to us because the authors used Unicorn for runtime comparisons. ARMory also demonstrated that implementing countermeasures for one specific fault can in turn make another fault more likely to occur. The authors argue that individual faults viewed in isolation should not be used as the basis for countermeasures—instead, a holistic approach has to be taken.

FiSim is an open-source deterministic fault attack simulator prototype for exploring the effects of fault injection attacks [177]. It is built using the Unicorn Engine [153] and the Capstone disassembler [152]. FiSim is a tool built by Riscure [176] that originally supported their ‘Designing Secure Bootloaders’ course, the application is focused on modern bootloader attacks and their countermeasures. Whilst this software has been released under the GNU General Public License, Riscure also have a newer application, TrueCode, however the application is proprietary and not available for testing or comparison. According to Riscure, TrueCode is the only solution completely focused on embedded software security. They state that TrueCode uses blackbox dynamic checks by using fault injection simulation on the target architecture.

And, in 2022, the most recent addition to the exhaustive fault simulation body-of-work is ARCHIE [79]. ARCHIE is another QEMU-based tool (also using the TCG plugin) to simulate transient and permanent instruction and data faults in RAM, flash and processor registers. In Section 4.6 we attempt to reproduce the work from FiSim, ARCHIE and ARMory using Faultfinder.

Related to the area of simulating fault injection into binaries is the software testing technique of *fuzzing*, where a binary is presented with many (cleverly chosen) inputs to uncover vulnerabilities in the code. Fuzzing has received very substantial attention from the research community (e.g., https://wcventure.github.io/FuzzingPaper/) for an overview), and been implemented in advanced open-source tools like AFL++ [54]. Unlike fuzzing, where the binary remains constant and the inputs are muted, in fault simulation, the binary is mutated while the inputs are kept constant. Still, ideas from fuzzing can be applied to fault simulation, e.g., the work of Xu et al. that uses snapshots (or “checkpoints”) to optimise the performance of individual program runs [218].

4.1.2 The Contribution of this Work

In this chapter we present Faultfinder, a fast, fault injection simulation tool. Faultfinder supports exhaustive, easy-to-create fault models which can be applied to any firmware binary, from bootloaders to cryptographic implementations. Our main contributions are:

- An optimised tool to simulate millions of faults injected into an arbitrary binary.
- Built upon the Unicorn CPU emulator framework [153], Faultfinder supports ARM, AArch64, M68K, MIPS, SPARC, PowerPC, RISC-V, S390x, and x86 architectures.
- We have tested Faultfinder on ARM, x86_64, RISC-V, Tricore and included code for MIPS and PowerPC. To date, we have not seen any such software for Tricore architectures.
- Faultfinder provides a highly configurable and easy-to-use fault model, written in a plain text file. Changing fault models requires no code compilation.
- Faultfinder can fault up to 128 different registers (depending upon the architecture) - all other tools only permit faulting R0-R15.
- We employ three optimisations: register-instruction bitmaps, equivalent states and the use of checkpoints, to heavily optimise the simulation process. Compared to state-of-the-art tools like ARCHIE [79] and FiSim [177], we obtain speed-ups of one or several orders of magnitude.
- Faultfinder is available at https://github.com/KitMurdock/faultfinder_public

4.2 Design and Implementation of Faultfinder

In this section, we first address the design rationale behind Faultfinder and then explain specific implementation choices.

4.2.1 Terminology

In this chapter we use the following terms:

- The *golden run* is the full execution of the binary-under-test without faults injected.
- A *fault* is the injection of single change in the state of a program e.g., a bit flip.
- A *fault model* is the definition of multiple different faults that we wish to inject.
- A *campaign* takes a number of fault models and simulates them, usually with a specific goal (e.g., whether if a specific code section is vulnerable to certain fault models).

4.2.2 Process Overview

The high-level end-to-end process for successfully faulting a binary is given as:

- *Examine binary*: The analyst will need to find and define the relevant start and end addresses along with the location for output analysis e.g., register, memory location.
- *Create binary details file*: Using the information obtained in the previous step the analyst will create a JSON-formatted binary details file.
- *Complete a golden run*: Faultfinder uses the binary details file to complete a golden run—this should confirm that all address locations and outputs are correctly defined.

- *Define fault model*: The analyst now needs to decide which types of fault she wishes to inject (e.g., register bit flips, instruction skips, flag sets)—these are configured in a text file.
- *Define campaign*: the newly defined fault model and binary are now brought together to define the campaign.
- *Run the campaign*: All the information has been prepared to now run the exhaustive fault injection campaign. The campaign run can be optimised to the specific host machine by, for example, using all cores available.
- *Analyse Results*: Whilst this is not specifically part of Faultfinder, we provide a Python script to convert our plaintext output files into an SQL database file for ease of result evaluation.

4.2.3 Why Unicorn Engine?

The Unicorn emulator [153] is based on QEMU 5 and is built for performance with many optimisations (including being thread-safe). The authors have stripped out any subsystems that do not involved CPU emulation, consequently, Unicorn is ten times smaller in size and memory consumption than QEMU.

One feature that is important is that Unicorn is a CPU emulator that is architecture aware, *i.e.*, if a fault injects incorrect code or attempts to write outside of the mapped memory then the error will be raised.

4.2.4 Integration with Unicorn

Faultfinder uses *unicorn hooks* to control the fault injection—a hook is created for the start and end fault address and counting takes place in between these. Because an instruction may be called multiple times within the faulting range, a counter is used to identify the specific instruction.

An additional *unicorn hook* is created at the faulting address. When the instruction count reaches the specified instruction, the fault hook injects the fault and the program runs to completion (if possible). Injecting a fault may create an infinite loop or a read from an accessible memory location—in these instances the binary-under-test will fault and not complete a run. Faultfinder records the result status of each run. Possible end run statuses are:

- faulted and reached end address
- faulted and error produced
- reached hard stop address
- maximum instructions reached

We found the two most common 'faulted and error produced' reasons were reading or writing to unmapped memory and invalid opcodes. In these instances the code ends with the error written to output. However, this behaviour is undefined and might produce different outputs on different machines. The 'maximum instructions reached' status was most frequently seen when an injected fault produced an infinite loop.

4.2.5 Golden Run

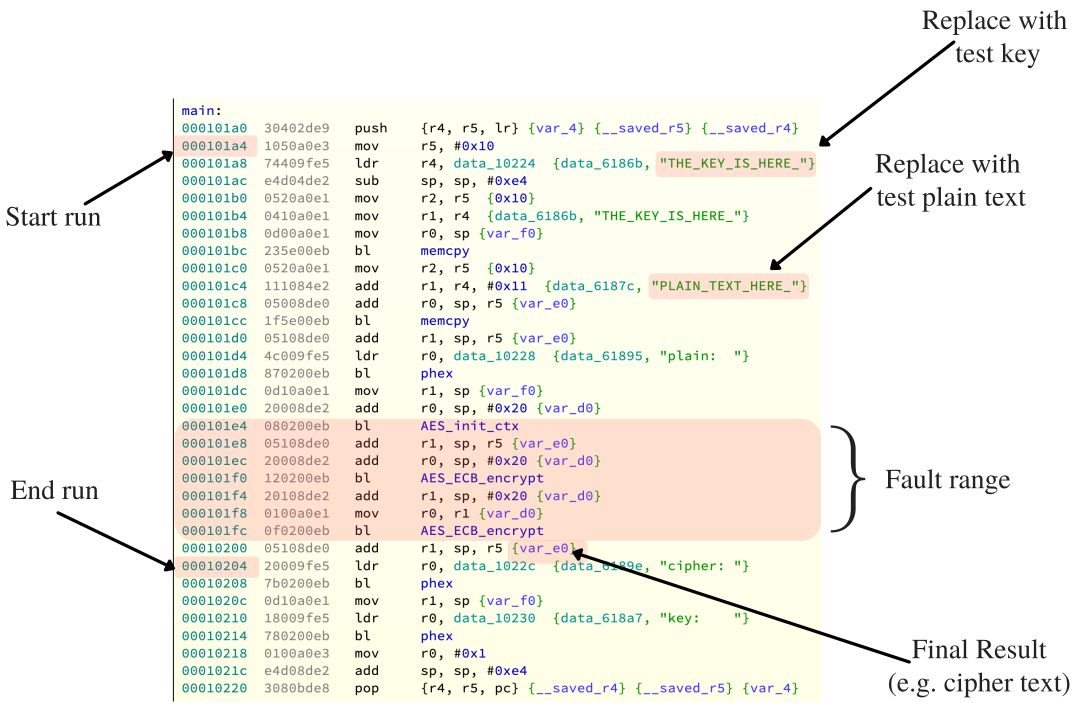
Faultfinder creates fault simulations at the binary level, we simulate faults on the machine code that would be flashed onto a microprocessor. Inevitably we needed to do some manual analysis prior to being able to emulate the code. A variety of address locations need to be identified in order to emulate the code (Figure 4.1). We categorise the concept of a *golden run* (Listing 4.1) as the perfect execution without faults—this can be used during the setup phase to ensure that all input and outputs are correctly determined. Memory locations can be overwritten to test other values but it is not required.

```
---- Running the program to display all instructions in faulting range -------
    00000001 hit:0001. 0x00402417 e8 ad fd ff ff    : call 0xdb2
    00000002 hit:0001. 0x004021c9 f3 0f 1e fa       : endbr64
    00000003 hit:0001. 0x004021cd e9 63 fb ff ff    : jmp 0xb68
    00000004 hit:0001. 0x00401d35 31 c0             : xor eax, eax
    00000005 hit:0001. 0x00401d37 8a 14 06          : mov dl, byte ptr [rsi + rax]
    00000006 hit:0001. 0x00401d3a 88 14 07          : mov byte ptr [rdi + rax], dl
    00000007 hit:0001. 0x00401d3d 8a 54 06 01       : mov dl, byte ptr [rsi + rax + 1]
    00000008 hit:0001. 0x00401d41 88 54 07 01       : mov byte ptr [rdi + rax + 1], dl
    00000009 hit:0001. 0x00401d45 8a 54 06 02       : mov dl, byte ptr [rsi + rax + 2]
    00000010 hit:0001. 0x00401d49 88 54 07 02       : mov byte ptr [rdi + rax + 2], dl
    00000011 hit:0001. 0x00401d4d 8a 54 06 03       : mov dl, byte ptr [rsi + rax + 3]
    00000012 hit:0001. 0x00401d51 88 54 07 03       : mov byte ptr [rdi + rax + 3], dl
    00000013 hit:0001. 0x00401d55 48 83 c0 04       : add rax, 4
    00000014 hit:0001. 0x00401d59 48 83 f8 10       : cmp rax, 0x10
```

Listing 4.1: Example output from golden run.

4.2.6 Fault Models and Injection

We use the instruction count (at runtime) as our index, this enables us to identify individual instructions. This is important because a single instruction at a specific address, may be called multiple times—the *golden run* helpfully provides a hit counter for each address (particularly useful when searching for the 8th round in a cryptographic implementation of AES, for example). Faultfinder can exhaustively inject faults based upon an easy-to-define *fault model.* The fault can be applied to every possible position in a *register* or *instruction.*



```
main:
000101a0  30402de9  push  {r4, r5, lr} {var_4} {__saved_r5} {__saved_r4}
000101a4  1050a0e3  mov   r5, #0x10
000101a8  74409fe5  ldr   r4, data_10224  {data_6186b, "THE_KEY_IS_HERE_"}
000101ac  e4d04de2  sub   sp, sp, #0xe4
000101b0  0520a0e1  mov   r2, r5  {0x10}
000101b4  0410a0e1  mov   r1, r4  {data_6186b, "THE_KEY_IS_HERE_"}
000101b8  0d00a0e1  mov   r0, sp {var_f0}
000101bc  235e00eb  bl    memcpy
000101c0  0520a0e1  mov   r2, r5  {0x10}
000101c4  111084e2  add   r1, r4, #0x11  {data_6187c, "PLAIN_TEXT_HERE_"}
000101c8  05008de0  add   r0, sp, r5 {var_e0}
000101cc  1f5e00eb  bl    memcpy
000101d0  05108de0  add   r1, sp, r5 {var_e0}
000101d4  4c009fe5  ldr   r0, data_10228  {data_61895, "plain:  "}
000101d8  870200eb  bl    phex
000101dc  0d10a0e1  mov   r1, sp {var_f0}
000101e0  20008de2  add   r0, sp, #0x20 {var_d0}
000101e4  080200eb  bl    AES_init_ctx
000101e8  05108de0  add   r1, sp, r5 {var_e0}
000101ec  20008de2  add   r0, sp, #0x20 {var_d0}
000101f0  120200eb  bl    AES_ECB_encrypt
000101f4  20108de2  add   r1, sp, #0x20 {var_d0}
000101f8  0100a0e1  mov   r0, r1 {var_d0}
000101fc  0f0200eb  bl    AES_ECB_encrypt
00010200  05108de0  add   r1, sp, r5 {var_e0}
00010204  20009fe5  ldr   r0, data_1022c  {data_6189e, "cipher: "}
00010208  7b0200eb  bl    phex
0001020c  0d10a0e1  mov   r1, sp {var_f0}
00010210  18009fe5  ldr   r0, data_10230  {data_618a7, "key:    "}
00010214  780200eb  bl    phex
00010218  0100a0e3  mov   r0, #0x1
0001021c  e4d08de2  add   sp, sp, #0xe4
00010220  3080bde8  pop   {r4, r5, pc} {__saved_r4} {__saved_r5} {var_4}
```

Figure 4.1: Addresses required by Faultfinder for the golden run.

Faults can be created using any of the following binary operations: `XOR` (*bit flips*), `AND` (*clear bits*), `OR` (*set bits*). Additionally, `SET` can be used to fully replace the target with a new value. Fault injection mostly commonly results in a bit flip and so we included a shorthand for bit shifts. `Mask: 3<0<4` translates to: *start with the mask 3 (0x11) and shift it left 0-3 times*. The full mask list would become:

- `0b11, 0b110, 0b1100, 0b11000`; or in hex:
- `0x3, 0x6, 0xc, 0x18`

The *lifespan* configuration enables faults to be transient or persistent. Where the lifespan is transient, it can be configured to be reverted after any number of instructions. An example of a fault model is shown in Listing 4.2.

```
# Faulting S-box access. 0x800101a-0x8001032
Instructions: 4937-4942
    Instruction:
        Op_codes: ALL
            Lifespan: 2, revert
                Operations: OR
                    Masks:0x4,0x400
    Registers: 2,3
        Op_codes: imul
            Lifespan: 0
                Operations: XOR
                    Masks:1<0<16
```

Listing 4.2: Fault model example in Faultfinder.

4.2.7 Fault Injection Campaign

A campaign consists of simulating multiple faults with the purpose of finding a usable fault e.g., avoiding bootloader protections or mathematical faults enabling cryptographic keys to be leaked. Faultfinder can automatically run a campaign if provided with three input files Figure 4.2. The *binary details* and *fault model* were introduced in Section 4.2.6, and finally we have the JSON-formatted *campaign details* file. The campaign details brings together the binary and the fault models. The campaign also enables optimisations to be enabled tailoring the run to the specifications of the underlying machines. Some prior work attempts to categorise outputs for each campaign however we have left the output-data-processing as a post-campaign exercise. This has the advantage of not unnecessarily impacting the pure fault injection runtime. In addition we have included *hard-stop addresses* for scenarios where we regard a "success" as having redirected the flow of a program to an ordinarily non-reachable address. For example, when attempting to circumvent a secure bootloader, any execution with a fault that reaches the bootloader address without having supplied the correct hash would be considered 'success' and would be configured as a *hard-stop address.*

Examples of each of the required campaign files are given in Appendix C.1.

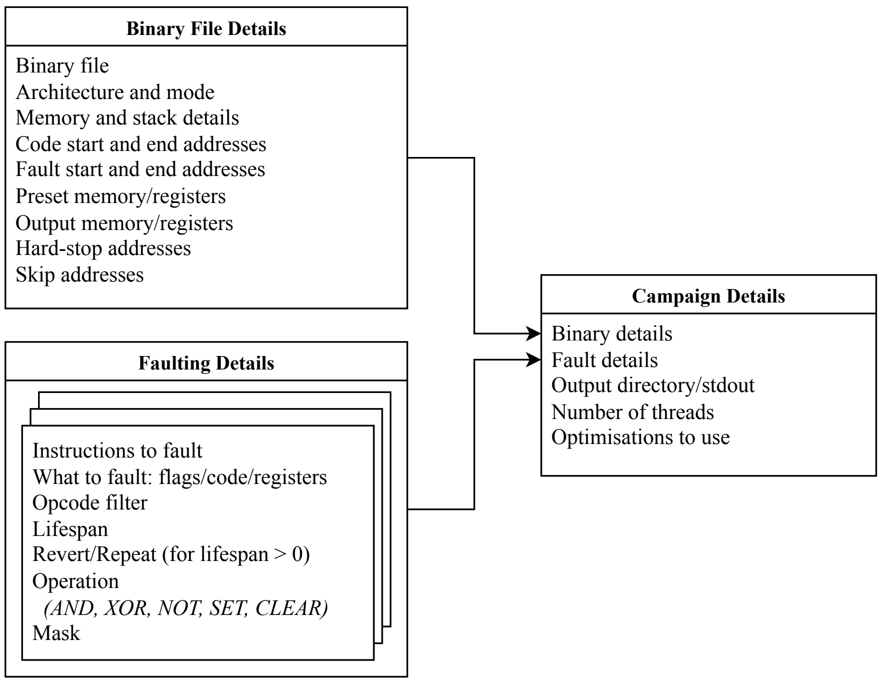


Figure 4.2: Configuration files required to run a fault injection campaign.

4.2.8 Result Format

To create as little overhead as possible, the outputs are written to a text file. Afterwards, we developed a program to import the results into a database for easy analysis (also available in the repository). Given the campaign details, Faultfinder exhaustively simulates the faults supplied from the fault models. Each new fault to be injected begins with a header-line for ease of searching. Each individual fault is also logged. An example of a single faulted run is shown in Listing 4.3. While we considered processing the output, we decided that the raw data is more useful since it can be analysed for a variety of purposes later on. Capstone was used to disassemble the instructions, and can be disabled to speed up performance if required.

```
*** Starting new run. Address: 0x800106a. Hit: 8. Lifespan: 2, revert.
   Instruction: 00004406. Faulting Target: Instruction.
    Mask: 0x000000000000d600. Operation: OR.  ***

 >> Writing input #0 directly to address: 0x0000000020000000:
    2b7e151628aed2a6abf7158809cf4...6abf7158809cf4f3c00127a0000127a00
 >> Original instruction          :  8b 42            : cmp r3, r1
 >> Mask                          :  0xd600
 >> Updated instruction           :  8b d6            : bvs #0xf1a
 >> Lifespan countdown: 2. (0x800106c)
 >> Lifespan countdown: 1. (0x8001142)
 >> Reverting instruction         :  8b 42
 >> Run result: reached end address after faulting.
 >>> Output from address (0x20000000): ed20342c0f4ee201c69d712b3596c2f2
```

Listing 4.3: Output example for a faulted instruction.

4.3 Optimisation Techniques

Faultfinder initially runs the binary twice to gather information. The first golden run counts the number of instructions executed within the faulting range. The second *state-saving run* stores the registers used at each instruction as a bitmap. Faults are only injected into registers if the register is referenced by that instruction. This prevents repeating unnecessary duplicated faults. Additionally, if the checkpoint optimisation is used, a full state snapshot at that instruction is stored.

4.3.1 Registers/Instruction Bitmap

The register/instruction bitmap stores up to 128 different registers (other software tested only supports 16 registers). When faulting, if a register is not referenced at the line—the fault is not injected, this prevents unnecessarily repeating faults that will have no distinct effect. We use the Capstone disassembler[152] to check if a register is referenced. This happens only once: in the *Golden Run*.

However, we encountered a problem when using Faultfinder with the Tricore architecture—Capstone does not support Tricore. If Faultfinder is not provided with a Capstone architecture and mode, the register/instruction bitmap is filled with `'1's` and provides no optimisation or register checking.

It is also worth noting that if a user specifies that they wish to fault the register `ax` this will fault all lines containing 'ax'—hence, the fault will also be injected at `rax` and `eax`. However the reverse is not true, if the user specified they wish to fault the register `rax`, it will not fault lines only containing `ax`. We consider this to be the correct and acceptable way of working.

Additionally, if a user wants the register to be faulted, even though it is not referenced at that instruction - they can choose to override it by using the `Registers-force` code in the fault model file.

4.3.2 Checkpoints

When a program is long, a significant amount of time is spent running the program to reach the fault instruction. To reduce runtime, we use *checkpoints* (or snapshots), inspired by Xu et al. [218]. In the second *state-saving run*, Faultfinder saves the full state of the program, *i.e.*, the context (registers, flags and stack) and all memory regions. Each instruction is then associated with its nearest checkpoint. To fault a specific instruction, Faultfinder can thus directly jump to the nearest checkpoint and continue the run from there.

We determined that weighting the number of checkpoints (to include more checkpoints towards the end of the execution) using the following (empirically selected) function yields good performance results:

$$y = 1 - ((1 - x)^{1.25})$$

Where x is the percentage through the program (for a four checkpoints this would be: 0.2, 0.4, 0.6, 0.8) and y is the new percentage (0.24, 0.47, 0.46, 0.87) this ensures the checkpoints are created where the most time savings can be made. To give a concrete example: if a user requests 8 checkpoints for a 100 instruction program, then the checkpoints would be at instructions 14, 27, 40, 52, 64, 75, 85, and 94. The main advantage of having more checkpoints towards the end of the program is as follows: at the beginning, the cost of taking and restoring checkpoints outweighs the cost to simply emulate the program to that point, while at the end, this situation is inverted.

Additionally, we ensure that Faultfinder does not create unnecessary checkpoints—if the campaign file requests 100 checkpoints but is only faulting 50 instructions—then only 50 checkpoints will be created. We further evaluate the use of checkpoints in Section 4.4.2.

4.3.3 Multithreading

Unicorn, as the foundation for Faultfinder, is thread-safe by design. Faultfinder automatically employs all available threads set in the *campaign details* by splitting the work over individual instructions. Each thread has access to the checkpoints from the state-saving run. Each thread consumes the next instruction and injects faults according to the model specifications.

4.3.4 Equivalences

Creating an exhaustive application has the disadvantage of generating a large number of outputs. Furthermore, many will be identical states. To counteract the issue of hundreds of results we have created the concept of an *equivalence*. After each fault has been injected (and before moving onto the next instruction) Faultfinder creates a hash from every byte of *registers, flags, stack and memory locations.* This hash is compared with all others for that *specific instruction count*. If there is a match, we can suggest with a high degree of certainly that we have a fault that creates an identical program state. In this scenario, the run is terminated and the new fault added to the list. Faultfinder displays the equivalences when all the faults for the instruction are complete (Listing 4.4). Hashing is necessary because the total amount of memory for a binary could be huge. Comparing each byte of memory (and storing them) for multiple states would cause the host computer to run out of memory (and possibly crash!)

While we have placed *equivalences* in the optimisation section, as we see, this cannot cleanly be defined as a performance optimisation. Turning equivalences on may have deleterious effect on the speed of Faultfinder, yet, reduces the number of outputs to simplify interpretation of the results. On the other hand, runtime reductions can be achieved where many identical states occur.

```
****** Equivalence faults for instruction: 4371 ******
 = Equivalence =
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xeb. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xec. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xed. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xee. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xef. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xa0. Operation: XOR.
 = Equivalence =
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xfb. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xfc. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xfd. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xfe. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xff. Operation: OR.
   === Instruction: 00004371. Faulting: Register. Reg#: r1. Mask: 0xb0. Operation: XOR.
```

Listing 4.4: Example output for found equivalences.

This is particularly useful because the aim of Faultfinder is to help identify exactly what fault was injected. This ensures that any output can be mapped back to one of a number of potential individual faults, thus saving the analyst a substantial amount of time and manual work. In order to pinpoint the fault more accurately, it would be possible to run Faultfinder with different inputs and look for faults common to both runs. Output from a run with *equivalences* is shown in Listing 4.5—note the additional line displaying the *state hash*. As

```
##### Starting new run. Address: 0x8001024. Hit: 128. Lifespan: 0.
      Instruction: 00004371. Faulting Target: Register. Reg#: r1.
      Mask: 0x000000000000005f. Operation: OR.  ###

>> Writing input #0 directly to address: 0x0000000020000000: 2
b7e151628aed2a6abf7158809cf4f3c2b7e151628aed2a6abf7158809cf4f3c00127a0000127a00
>> Original register           : 0x000000000000004f
>> Mask                        : 0x000000000000005f
>> Updated                     : 0x000000000000005f
>> State hash                  : 55dff839e50b52302cbff229d596d1d9
>> Run result: Run ended early - equivalence found.
```

Listing 4.5: Example output with equivalence hashes.

far as we are aware, Faultfinder is the first fault injection tool to highlight identically injected faults. We further evaluate the use of checkpoints in Section 4.4.3.

4.4 Performance Impact of Optimisations

In this section, we evaluate how the described optimisations impact the performance of Faultfinder. The experiments were performed on an Intel i7 8-core laptop running at 2.6 GHz. We used tinyAES [116] as the binary-under-test, compiled for x86_64.

4.4.1 Multithreading

As would be expected, using as many threads as cores creates the optimum conditions. Table 4.1 demonstrates that the performance Faultfinder scales proportionally with the number of threads (until the number of CPU cores is reached).

4.4.2 Checkpoints

Substantial overhead is due to running the program from the beginning each time. Checkpoints are most useful for programs with many instructions. However, there is a trade-off:

Number of Threads	*Runtime*
0	87m 50s
2	45m 13s
4	29m 19s
8	21m 41s
16	21m 46s

Table 4.1: Time to inject 625,550 faults in x86 tinyAES binary with 200,000 instructions, running on an 8-core Intel i7 @ 2.6 GHz (no checkpoints used).

the saving and restoring creates additional time-overheads, which will vary depending upon the size of the memory regions created. In reality we were surprised to see that very few checkpoints were required to create significant time savings. 50 checkpoints were sufficient to halve the total campaign time (Table 4.2). More checkpoints require more memory, and the time to create and store the checkpoints seems to have a negative time impact beyond 50.

The run times of 625,000 faults with 10 checkpoints is shown in Figure 4.3—the checkpoints are visible in the sudden steps of decrease in run time. It is interesting to note the wide range of timings for each instruction address. This graph clearly substantiates that checkpoints can significantly reduce runtime in programs with a large number of instructions.

# Checkpoints	*Memory used*	*Runtime*
0	0 MB	21m 41s
10	26.54 MB	12m 37s
25	41.59 MB	12m 29s
50	66.69 MB	11m 50s
100	116.89 MB	12m 00s
1000	1020.40 MB	12m 16s

Table 4.2: Runtime to inject 625,550 faults into an x86 tinyAES binary with 200,000 instructions, running on an Intel i7 @ 2.6 GHz with 8 threads.

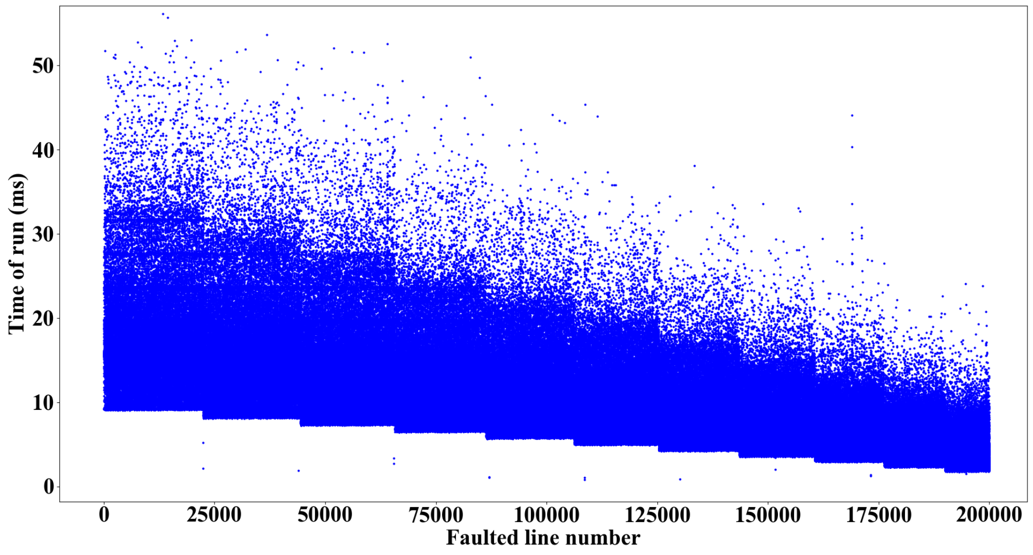


Figure 4.3: Time of runs within a 200000 instruction program with 10 checkpoints.

4.4.3 Equivalences

We do *not* consider equivalences to be a runtime optimisation, nevertheless, we verified their impact. We used a 200,000 instruction tinyAES program and faulted a set of 10,000 instructions in the *centre* of the run. This was to ensure the timings wold not be impacted by the position of the faulting. We were also aware that the timing impact will be lessened if many equivalences are found. Thus we ran the experiments twice: once with *different faults* and once with *identical faults* (cf. Table 4.3 and Table 4.4). It should be noted that when we used different faults (Table 4.3), we did occasionally accidentally create a small amount of equivalent states due to the properties of the binary-under-test. We were, again, surprised by the results—the time-overhead of checking the state *against all others for that instruction* was much lower than expected. As expected, where there were many identical states, a runtime-saving was observed (cf. Table 4.4).

# Faults per instr.	*Total faults*	*With equivalences*	*Without equivalences*
4	23,880	1m 10s	1m 10s
8	47,760	2m 39s	2m 41s
16	95,520	5m 16s	5m 24s
32	191,040	10m 16s	9m 59s
64	382,080	21m 30s	22m 21s
128	764,160	43m 43s	43m 27s

Table 4.3: Comparison of runtimes injecting different faults in an x86 tinyAES binary with equivalences turned on/off on an Intel i7 @ 2.6 GHz with 8 threads.

# Faults per instr.	*Total faults*	*With equivalences*	*Without equivalences*
4	23,880	1m 18s	1m 10s
8	47,760	2m 16s	2m 41s
16	95,520	4m 31s	5m 24s
32	191,040	8m 52s	9m 59s
64	382,080	17m 48s	22m 21s
128	764,160	35m 03s	43m 27s

Table 4.4: Comparison of runtimes injecting identical faults in an x86 tinyAES binary with equivalences turned on/off on an Intel i7 @ 2.6 GHz with 8 threads.

4.5 Multi-architectural Validation

We wanted to ensure that Faultfinder was producing identical results for different architectures, so we compiled a tinyAES [116] implementations for: ARM, RISC-V, Tricore and x86_64. We analysed the different binaries using Ghidra to find the Round Key being retrieved. Each architecture used different registers and slightly different methods for loading the data. After this analysis we crafted a fault model for each that would inject identical faults.

4.5.1 Method

The process for each architecture was similar, specifically:

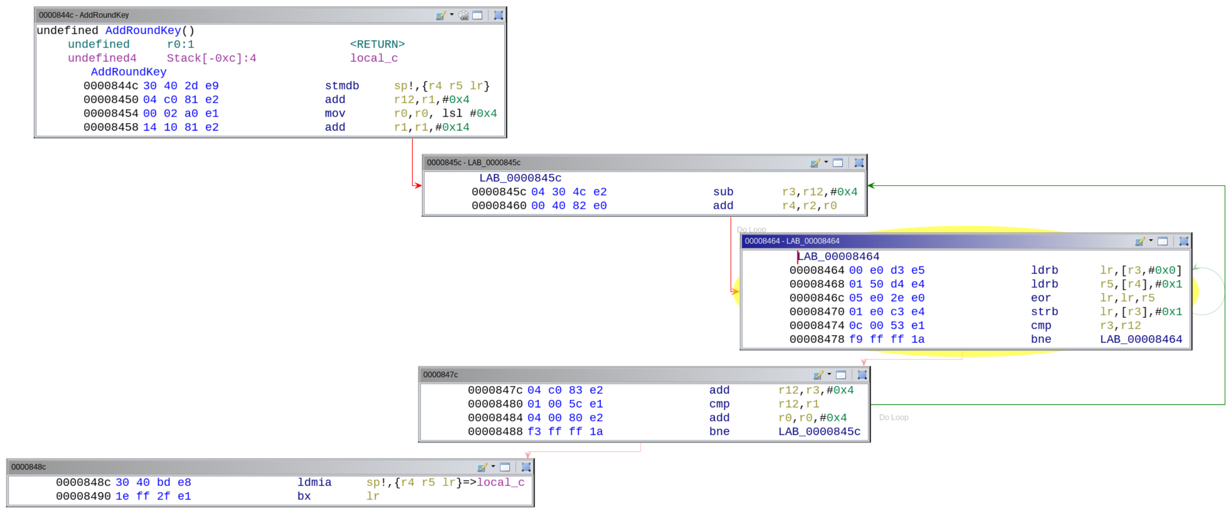
- Open the binary in Ghidra
- Search for the function `AddRoundKey`
- Identify the exact line that loads the data into a register
  - Note the register
  - Note the address immediately after *i.e.*, the address to be faulted
- Complete a *golden run* - and search for the faulted address
  - For simplicity of comparing results we decided to fault only one occurrence of this address (arbitrarily chosen as the 128th)
- Create a fault model using this instruction number and the previously noted register. An example fault model is show in Listing 4.6.
- Record the total number of unique outputs and their counts

```
Instructions: 4375-4375
    Registers-force: X10
        Op_codes: ALL
            Lifespan: 0
                Operations: XOR
                    Masks:1<0<32
```

Listing 4.6: Example fault model for Tricore flipping bits in register X10 at instruction 4375

4.5.2 ARM

We analysed Figure 4.4 and concluded that line `ldrb r5,[r4],#0x01` retrieves the Round Key data and therefore the address to fault is: `0x0000846c` (the one directly after it) and the register to fault is: `r5`.



```
undefined AddRoundKey()
     undefined      r0:1                        <RETURN>
     undefined4     Stack[-0xc]:4               local_c
        AddRoundKey
       0000844c 30 40 2d e9                 stmdb      sp!,{r4 r5 lr}
       00008450 04 c0 81 e2                 add        r12,r1,#0x4
       00008454 00 02 a0 e1                 mov        r0,r0, lsl #0x4
       00008458 14 10 81 e2                 add        r1,r1,#0x14
```

```
        LAB_0000845c
       0000845c 04 30 4c e2                 sub        r3,r12,#0x4
       00008460 00 40 82 e0                 add        r4,r2,r0
```

```
        LAB_00008464
       00008464 00 e0 d3 e5                 ldrb       lr,[r3,#0x0]
       00008468 01 50 d4 e4                 ldrb       r5,[r4],#0x1
       0000846c 05 e0 2e e0                 eor        lr,lr,r5
       00008470 01 e0 c3 e4                 strb       lr,[r3],#0x1
       00008474 0c 00 53 e1                 cmp        r3,r12
       00008478 f9 ff ff 1a                 bne        LAB_00008464
```

```
       0000847c 04 c0 83 e2                 add        r12,r3,#0x4
       00008480 01 00 5c e1                 cmp        r12,r1
       00008484 04 00 80 e2                 add        r0,r0,#0x4
       00008488 f3 ff ff 1a                 bne        LAB_0000845c
```

```
       0000848c 30 40 bd e8                 ldmia      sp!,{r4 r5 lr}=>local_c
       00008490 1e ff 2f e1                 bx         lr
```

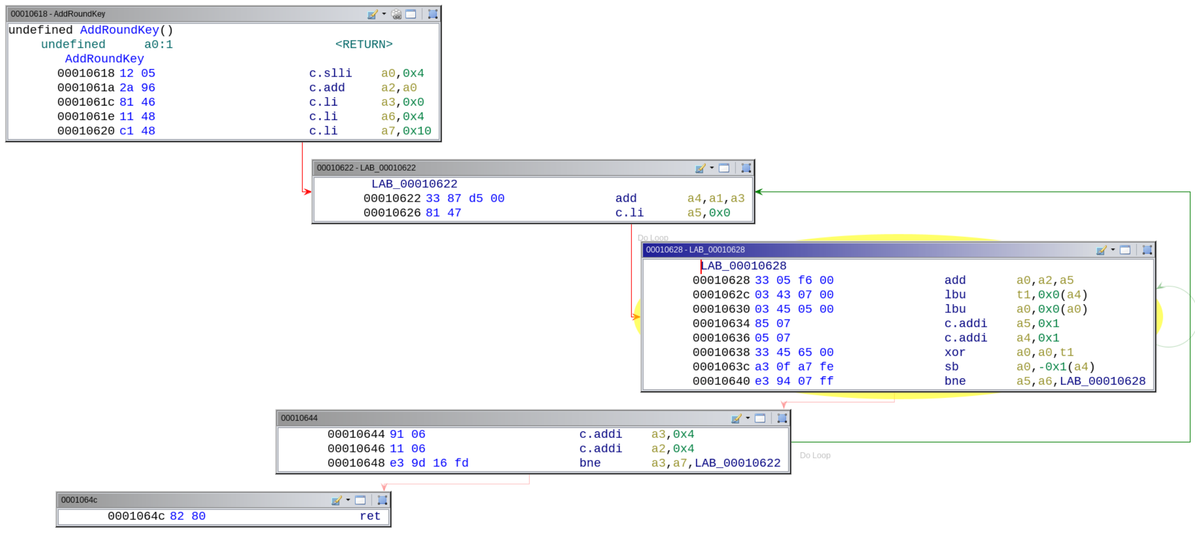
Figure 4.4: Ghidra code snippet from tinyAES compiled for ARM.

Output from Faultfinder	*Count of output*
`Output from address (0x00080f20) in register (r1): 3ad77bb40d7a3660a89ecaf32466ef97`	24
`Output from address (0x00080f20) in register (r1): 2c45ea42d60971c93149eb4250e0c821`	1
`Output from address (0x00080f20) in register (r1): 2eee3a41b0a6b0f734468e0ddd823ad3`	1
`Output from address (0x00080f20) in register (r1): 333c88dd35d593ba7ba369531c80cbdc`	1
`Output from address (0x00080f20) in register (r1): 4d3b20152f1718a5fc2eeca51eb27886`	1
`Output from address (0x00080f20) in register (r1): 65d2181dd9ff076b30feff3c9910da1f`	1
`Output from address (0x00080f20) in register (r1): 88789c84b26fd72833b10bd36fbe1340`	1
`Output from address (0x00080f20) in register (r1): c1c29e2068fe1b52d764e522bac1b842`	1
`Output from address (0x00080f20) in register (r1): fd7c33ade091744c44940e9f0077aeac`	1

Table 4.5: Outputs from running Faultfinder to flip each bit in r1 at instruction 3864 in an ARM compiled tinyAES binary

4.5.3 RISC-V

We analysed Figure 4.5 and concluded that line `lbu a0,0x0(a0)` retrieves the Round Key data and therefore the address to fault is: `0x0080ffb8` (the one directly after it) and the register to fault is: `X18`.



```
undefined AddRoundKey()
    undefined     a0:1                          <RETURN>
      AddRoundKey
        00010618 12 05                     c.slli    a0,0x4
        0001061a 2a 96                     c.add     a2,a0
        0001061c 81 46                     c.li      a3,0x0
        0001061e 11 48                     c.li      a6,0x4
        00010620 c1 48                     c.li      a7,0x10

      LAB_00010622
        00010622 33 87 d5 00               add       a4,a1,a3
        00010626 81 47                     c.li      a5,0x0

      LAB_00010628
        00010628 33 05 f6 00               add       a0,a2,a5
        0001062c 03 43 07 00               lbu       t1,0x0(a4)
        00010630 03 45 05 00               lbu       a0,0x0(a0)
        00010634 85 07                     c.addi    a5,0x1
        00010636 05 07                     c.addi    a4,0x1
        00010638 33 45 65 00               xor       a0,a0,t1
        0001063c a3 0f a7 fe               sb        a0,-0x1(a4)
        00010640 e3 94 07 ff               bne       a5,a6,LAB_00010628

        00010644 91 06                     c.addi    a3,0x4
        00010646 11 06                     c.addi    a2,0x4
        00010648 e3 9d 16 fd               bne       a3,a7,LAB_00010622

        0001064c 82 80                     ret
```

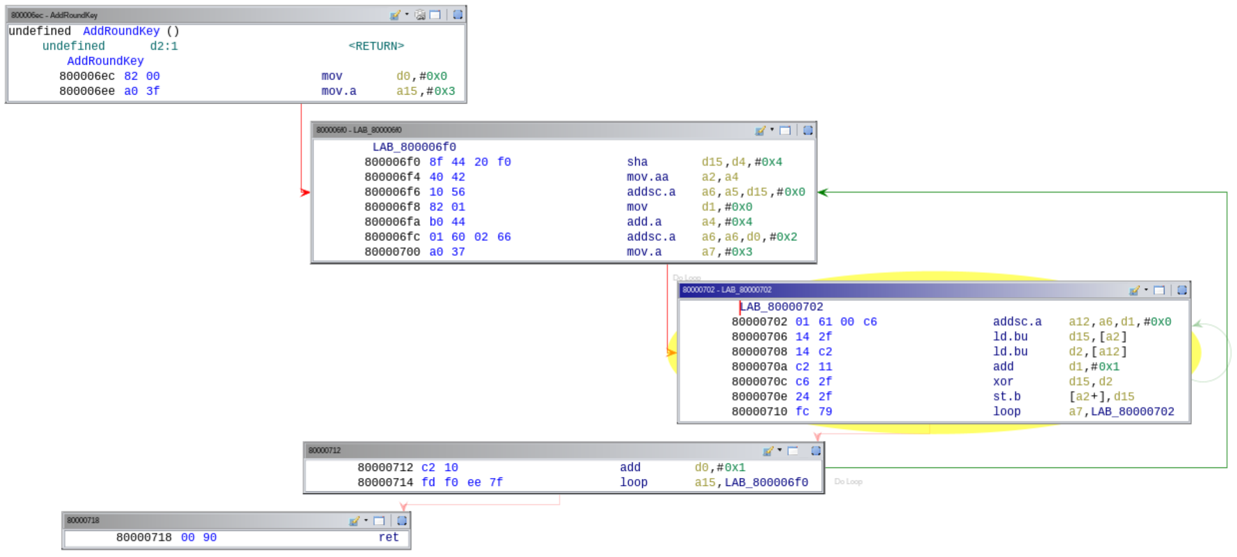
Figure 4.5: Ghidra code snippet from tinyAES compiled for RISC-V.

Output from Faultfinder	*Count of output*
`Output from address (0x0080ffb8) in register (X18): 3ad77bb40d7a3660a89ecaf32466ef97`	24
`Output from address (0x0080ffb8) in register (X18): 2c45ea42d60971c93149eb4250e0c821`	1
`Output from address (0x0080ffb8) in register (X18): 2eee3a41b0a6b0f734468e0ddd823ad3`	1
`Output from address (0x0080ffb8) in register (X18): 333c88dd35d593ba7ba369531c80cbdc`	1
`Output from address (0x0080ffb8) in register (X18): 4d3b20152f1718a5fc2eeca51eb27886`	1
`Output from address (0x0080ffb8) in register (X18): 65d2181dd9ff076b30feff3c9910da1f`	1
`Output from address (0x0080ffb8) in register (X18): 88789c84b26fd72833b10bd36fbe1340`	1
`Output from address (0x0080ffb8) in register (X18): c1c29e2068fe1b52d764e522bac1b842`	1
`Output from address (0x0080ffb8) in register (X18): fd7c33ade091744c44940e9f0077aeac`	1

Table 4.6: Outputs from running Faultfinder to flip each bit in X18 at instruction 4375 in an RISC-V compiled tinyAES binary

4.5.4 Tricore

We analysed Figure 4.5 and concluded that line `ld.bu d2,[a12]` retrieves the Round Key data and therefore the address to fault is: `0x70008f30` (the one directly after it) and the register to fault is: `A5`.



```
800006ec - AddRoundKey
undefined  AddRoundKey ()
     undefined        d2:1                               <RETURN>
        AddRoundKey
          800006ec 82 00                         mov        d0,#0x0
          800006ee a0 3f                         mov.a      a15,#0x3
```

```
800006f0 - LAB_800006f0
          LAB_800006f0
     800006f0 8f 44 20 f0                  sha        d15,d4,#0x4
     800006f4 40 42                        mov.aa     a2,a4
     800006f6 10 56                        addsc.a    a6,a5,d15,#0x0
     800006f8 82 01                        mov        d1,#0x0
     800006fa b0 44                        add.a      a4,#0x4
     800006fc 01 60 02 66                  addsc.a    a6,a6,d0,#0x2
     80000700 a0 37                        mov.a      a7,#0x3
```

```
80000702 - LAB_80000702
          LAB_80000702
     80000702 01 61 00 c6                  addsc.a    a12,a6,d1,#0x0
     80000706 14 2f                        ld.bu      d15,[a2]
     80000708 14 c2                        ld.bu      d2,[a12]
     8000070a c2 11                        add        d1,#0x1
     8000070c c6 2f                        xor        d15,d2
     8000070e 24 2f                        st.b       [a2+],d15
     80000710 fc 79                        loop       a7,LAB_80000702
```

```
80000712
     80000712 c2 10                        add        d0,#0x1
     80000714 fd f0 ee 7f                  loop       a15,LAB_800006f0
```

```
80000718
     80000718 00 90                        ret
```

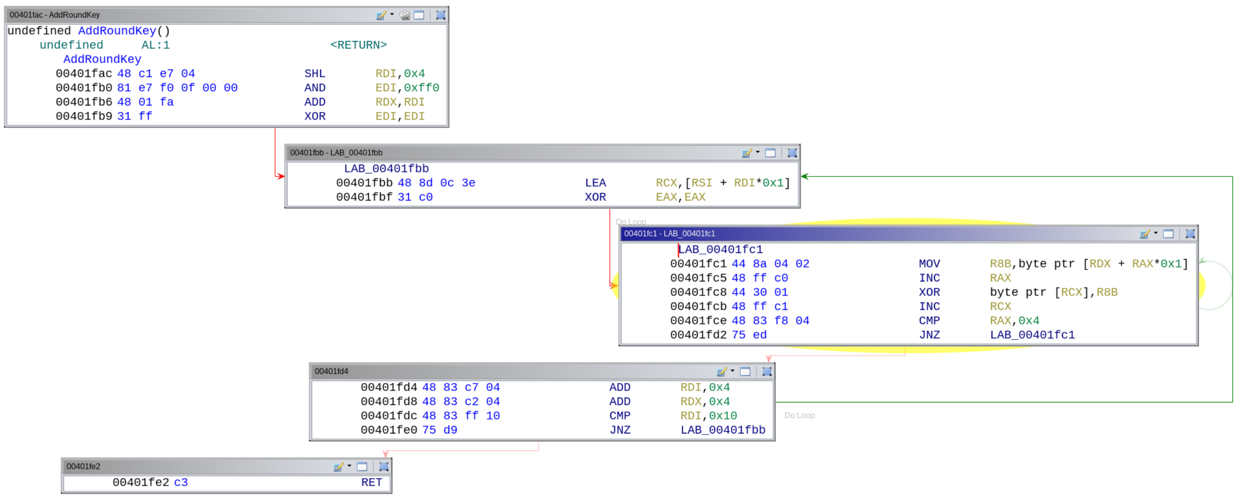
Figure 4.6: Ghidra code snippet from tinyAES compiled for Tricore.

Output from Faultfinder	*Count of output*
`Output from address (0x70008f30) in register (A5): 3ad77bb40d7a3660a89ecaf32466ef97`	24
`Output from address (0x70008f30) in register (A5): 2c45ea42d60971c93149eb4250e0c821`	1
`Output from address (0x70008f30) in register (A5): 2eee3a41b0a6b0f734468e0ddd823ad3`	1
`Output from address (0x70008f30) in register (A5): 333c88dd35d593ba7ba369531c80cbdc`	1
`Output from address (0x70008f30) in register (A5): 4d3b20152f1718a5fc2eeca51eb27886`	1
`Output from address (0x70008f30) in register (A5): 65d2181dd9ff076b30feff3c9910da1f`	1
`Output from address (0x70008f30) in register (A5): 88789c84b26fd72833b10bd36fbe1340`	1
`Output from address (0x70008f30) in register (A5): c1c29e2068fe1b52d764e522bac1b842`	1
`Output from address (0x70008f30) in register (A5): fd7c33ade091744c44940e9f0077aeac`	1

Table 4.7: Outputs from running Faultfinder to flip each bit in A5 at instruction 5069 in a Tricore compiled tinyAES binary

4.5.5 x86_64

We analysed Figure 4.5 and concluded that line `mov r8B, byte ptr [rdx + rax*0x1]` retrieves the Round Key data and therefore the address to fault is: `0x08000fd0` (the one directly after it) and the register to fault is: `rsi`.



```
00401fac - AddRoundKey
undefined AddRoundKey()
    undefined       AL:1                        <RETURN>
      AddRoundKey
        00401fac 48 c1 e7 04              SHL        RDI,0x4
        00401fb0 81 e7 f0 0f 00 00        AND        EDI,0xff0
        00401fb6 48 01 fa                 ADD        RDX,RDI
        00401fb9 31 ff                    XOR        EDI,EDI

00401fbb - LAB_00401fbb
          LAB_00401fbb
        00401fbb 48 8d 0c 3e              LEA        RCX,[RSI + RDI*0x1]
        00401fbf 31 c0                    XOR        EAX,EAX

00401fc1 - LAB_00401fc1
          LAB_00401fc1
        00401fc1 44 8a 04 02              MOV        R8B,byte ptr [RDX + RAX*0x1]
        00401fc5 48 ff c0                 INC        RAX
        00401fc8 44 30 01                 XOR        byte ptr [RCX],R8B
        00401fcb 48 ff c1                 INC        RCX
        00401fce 48 83 f8 04              CMP        RAX,0x4
        00401fd2 75 ed                    JNZ        LAB_00401fc1

00401fd4
        00401fd4 48 83 c7 04              ADD        RDI,0x4
        00401fd8 48 83 c2 04              ADD        RDX,0x4
        00401fdc 48 83 ff 10              CMP        RDI,0x10
        00401fe0 75 d9                    JNZ        LAB_00401fbb

00401fe2
        00401fe2 c3                       RET
```

Figure 4.7: Ghidra code snippet from tinyAES compiled for x86_64.

Output from Faultfinder	*Count of output*
`Output from address (0x08000fd0) in register (rsi): 3ad77bb40d7a3660a89ecaf32466ef97`	24
`Output from address (0x08000fd0) in register (rsi): 2c45ea42d60971c93149eb4250e0c821`	1
`Output from address (0x08000fd0) in register (rsi): 2eee3a41b0a6b0f734468e0ddd823ad3`	1
`Output from address (0x08000fd0) in register (rsi): 333c88dd35d593ba7ba369531c80cbdc`	1
`Output from address (0x08000fd0) in register (rsi): 4d3b20152f1718a5fc2eeca51eb27886`	1
`Output from address (0x08000fd0) in register (rsi): 65d2181dd9ff076b30feff3c9910da1f`	1
`Output from address (0x08000fd0) in register (rsi): 88789c84b26fd72833b10bd36fbe1340`	1
`Output from address (0x08000fd0) in register (rsi): c1c29e2068fe1b52d764e522bac1b842`	1
`Output from address (0x08000fd0) in register (rsi): fd7c33ade091744c44940e9f0077aeac`	1

Table 4.8: Outputs from running Faultfinder to flip each bit in rsi at instruction 3977 in a x86_64 compiled tinyAES binary

4.5.6 Summary

Analysing four binaries compiled for different architectures was educational, each architectures has subtle differences e.g., Tricore produced the most instructions (`6457`) and ARM, the least: (`5233`). Ultimately, the cryptographic outputs were identical and we were able to demonstrate that faultfinder works smoothly across three different Unicorn supported architectures.

4.6 Comparison to Existing Fault Simulation Tools

In order to confirm the accuracy and performance of Faultfinder we picked three of the more recent tools in the research area. We chose ARCHIE [79] because of its excellent documentation and well-sourced example repository [200]. ARCHIE faults are injected into a cryptographic algorithm (tinyAES) and the outputs stored in an HDF5 files. Secondly, we selected FiSim [177] for its focus on a non-cryptographic secure boot application. Thirdly, we used ARMory [84] because it also had a well documented repository injecting faults into an AES implementation. It is evident that these scenarios are relevant to the real-life application of fault attacks and, additionally, are well-known research scenarios. ARCHIE and ARMory use a linux command-line interface, ARCHIE is built upon QEMU and ARMory upon M-ulator (which the authors also wrote). FiSim runs on Windows only, using a compiled version of the Unicorn engine. We believe that these projects represent a good reference to compare against. Our approach when reproducing prior work was to leave the binaries-under-test untouched and to adjust as few parameters as possible within the tools. We built a set of Python programs to import code into a database from: (*i*) ARCHIE's HDF5 files (*ii*) Faultfinder's output directory (*iii*) FiSim's raw output window. (*iv*) ARMory's raw output to screen This enabled us to precisely evaluate the results. However, we did have to modify the code from ARMory to be able to create a usable output (see Section 4.6.3).

4.6.1 Comparison with ARCHIE

Differential Fault Analysis (DFA) is a well known attack against an AES cipher. In 2009, Saha et al. demonstrated that the entire key can be leaked from AES if a fault can be injected into one of the four diagonals of the state matrix in the 8th round [180]. To demonstrate this, the ARCHIE fault model was configured to inject faults into the S-box and to log the resulting ciphertexts.

Fault model ARCHIE's fault model differs from Faultfinder by using a *trigger address* with a *trigger counter*. We can see the advantage of this—it enables the analyst to focus on a specific address and only fault on its xth call. For example, one of their faults triggers is shown below:

```
"fault_address"   : [134221926, 134221932, 1],
"trigger_address"  : [-1],
"trigger_counter"  : [8, 10, 1]
```

Listing 4.7: ARCHIE fault model snippet.

The first fault in this model is: *trigger and fault addresses: 0x8001066, 8th occurrence.* To replicate this in Faultfinder we use the *golden run* to find the instruction count on its eighth occurrence. From the output snippet in Listing 4.8, we can see that Faultfinder would reference this instruction using the instruction counter 4404.

ARCHIE can fault any address from any trigger address. Faultfinder does not have this feature. Faultfinder only faults the current instruction prior to execution, *i.e.*, effectively faulting the instruction fetch register or the memory bus. The fault models for ARCHIE and Faultfinder are given in Appendix C.2.1 and Appendix C.2.2 respectively.

```
00004400 hit:0008. 0x0800105e 82 70     : strb r2, [r0, #2]
00004401 hit:0008. 0x08001060 c5 70     : strb r5, [r0, #3]
00004402 hit:0008. 0x08001062 c1 72     : strb r1, [r0, #0xb]
00004403 hit:0008. 0x08001064 c3 71     : strb r3, [r0, #7]
00004404 hit:0008. 0x08001066 03 9b     : ldr r3, [sp, #0xc]
00004405 hit:0008. 0x08001068 01 99     : ldr r1, [sp, #4]
```

Listing 4.8: Faultfinder golden run snippet with hitcount

Reproducibility Of the 8278 faults injected, 8258 faults were identical with only 20 faulty outputs differing. Faultfinder reports the outputs as `unmapped memory write` or `unmapped memory read`. When we compared the individual faulted instruction they were identical, hence, it is likely that the faults might differ depending upon how the CPU (emulator) manages such unmapped memory read/writes.

Timings The AES implementation used for the ARCHIE test only injects 8278 faults into a ≈20000 instruction program—due to the fairly low number of instructions, we did not use any checkpoints for our timings when running the comparisons. Had there been more faults to inject, we would have turned on checkpoints and achieved even larger runtime savings (cf. the results in Section 4.3.2). As shown in Table 4.9, Faultfinder consistently outperforms ARCHIE by factors between 70 and 281, depending on the specific machine.

Machine	*Freq. (GHz)*	*Cores*	*Threads*	*ARCHIE*	*Faultfinder*	*Speed increase*
i7-4510U	2.00	4	4	25m 27s	22s	70x
i7-7700HQ	2.80	8	8	34m 59s	9s	233x
E-2244G	3.80	8	8	15m 58s	6s	160x
i9-10900K	3.70	20	20	4m 41s	1s	281x

Table 4.9: Runtime comparison between Faultfinder and ARCHIE when injecting 8278 faults into an ARM-compiled tinyAES binary (no checkpoints used for Faultfinder). All CPUs are Intel.

Discussion ARCHIE and Faultfinder represent fault models in a similar manner. However, while ARCHIE includes the ability to fault any address at any time, we do not consider this very specific fault model. Instead, we focus on data-in-use at any instruction, *i.e.*, registers and instructions. Additionally, ARCHIE makes use of an single HDF5 file for its outputs. We note that it took around 5 s to open the file and a further 10 s to view any single experiment.

4.6.2 Comparison with FiSim

FiSim's sample code replicates an embedded system attempting to employ a secure boot. The device loads the image, calculates the hash, and if it is correct, continues to boot to the next stage. For the Internet of Things (IoT), this is a common scenario—developers want to ensure that only their own firmware can be loaded onto their devices.

In this particular fault injection we are not looking for faulty output, but rather imagining an attacker who wants to skip or trick the verification process so that she can execute arbitrary code on the device (e.g., read private data). The results of FiSim were difficult to reproduce: the demo application runs on Windows only and has the fault models hardcoded into the sources. We thus dual-booted Windows and Linux to compare the tools on the same CPU.

Additionally, FiSim's software fully emulates the one-time programmable memory read that is performed when loading the image. Faultfinder was not able to reproduce this (and it is not designed to); therefore, we were not fully able to replicate every fault. With the FiSim sample code representing a bootloader, "success" is represented as reaching a specific location in the binary: `boot_next_stage`. For this, we made use of *hard-stop addresses*. If the program flow arrives at this function, the simulation stops and the result is recorded.

```
Instructions: 1-1299
  Instruction Pointer:
    Op_codes: ALL
      Lifespan: 0
        Operation: SKIP
  Instruction:
    Op_codes: ALL
      Lifespan: 1, revert
        Operations: XOR
         Masks: 1<0<32
```

Listing 4.9: Faultfinder's fault model used to reproduce FiSim.

Reproducibility Initially we intended to detail every single fault for comparison. However, FiSim outputs a single line per address where a fault has successfully resulted in side-stepping the security—we therefore decided to compare those addresses that each tool considers *vulnerable*. FiSim reports a total of 94 different *vulnerable addresses*, of these Faultfinder reports 72. Additionally, Faultfinder reported an extra 6 addresses as vulnerable that were not noted by FiSim.

Fault models In FiSim's demo, only two fault types are enabled:

- A *Transient Nop Instruction* replaces the instruction with a NOP but only once. The next time that instruction is called it will execute as normal.
- A *Transient Single Bit Flip Instruction* flips a bit in the instruction prior to execution, but does not persist.

Following our own rule of not modifying anything, we replicated these two faults only, and our fault model is show in Listing 4.9.

Timings The bootloader code used for the FiSim test is only 1200 instructions long therefore, we did not use checkpoints in Faultfinder. As evident from Table 4.10, Faultfinder outperforms FiSim by a factor of eight on the same machine. We did not compare the performance on other machines, as it would require dual-booting because FiSim runs on Windows only.

Machine	*Frequency*	*Cores*	*Threads*	*FiSim*	*Faultfinder*	*Speed increase*
Intel i7-4510U	2.00 GHz	4	1	9m 17s	1m 6s	8x

Table 4.10: Runtime comparison between Faultfinder and FiSim when injecting 22,000 faults into FiSim's demonstration bootloader ARM binary (no checkpoints used for Faultfinder).

Discussion When comparing the results we note that FiSim reports 22 additional addresses where a fault can be injected to bypass the bootloader. However we also note that Faultfinder produced 6 additional locations. We considered investigating further, however, FiSim has not been updated in two years and is running an older version of Unicorn. Additionally, FiSim is a teaching tool and does not make any claims about accuracy, in fact, they state that quick feedback is preferred for their purposes.

4.6.3 Comparison with ARMory

ARMory provides two case studies, along with the underlying code: (*i*) DFA on AES (*ii*) A secure bootloader. We choose to reproduce the AES implementation, because it has more runtime lines of code. The first problem is that ARMory does not output all results—only those that meet its *exploitability model.* This model is coded in C++ and only displays the fault, not the final cryptographic output. For the AES case study, an *exploitable fault* is coded as any cryptographic output with a single modified byte from the 8th to 10th rounds.

Reproducibility To be able to reproduce the work, we modified and recompiled the exploitability code to display the output, which we then had to reassembly with ARMory's fault outputs. We used this combined-information to create a Faultfinder fault model file.

```
Faulted  instruction: ldrb r6, [r4], #0x21
Cryptographic Output: 46388397e4e4d50364af54 24 0802922c
ARMory Description:   Transient Instruction Bit-Flip position=21 time=5316
```

Listing 4.10: Faulted output from ARMory for instruction time: 5316

This is not an ideal solution, as our file ended up being 30,000 lines long. However, we wanted to compare *identical faults* against an identical binary.

Issue	*Count*
Faulting XPSR register	2837
Error: Unable to disassemble instruction	68
Error: Reading from unmapped memory	55
Faulted and reached end address with no errors	775
Total faults producing different outputs	3715

Table 4.11: Breakdown of non-matching faults between Faultfinder and ARMory for an AES implementation on ARM

Of the 42,000 faults, Faultfinder matched 38,285 identical outputs. With 3715 different outputs—we break these down in Table 4.11. The XPSR (Special-purpose program status registers) did not reproduce identical outputs. The XPSR is constructed from three registers: Application PSR, Interrupt PSR and Execution PSR. Faultfinder reports different values for this register, we can only conclude that M-ulator and Unicorn are handling this register differently.

There are 775 different outputs where Faultfinder reports that the binary successfully faulted and reached the end address with no errors. An example of one such fault is shown in Listing 4.11. We see that the fault has produced an identical *new instruction* with both ARMory and Faultfinder. When we traced Faultfinder's data and code we could find no problems or inaccuracies. However, the output would match ARMory's *exploitability model* (one byte error). Currently, we are unable to explain these 775 different outputs.

```
Instruction:                5316
Address:                    0x808c
Fault injected:             XOR
Fault Mask:                 0x200000
Original instruction:       ldrb r6, [r4], #1
Faulted new instruction:    ldrb r6, [r4], #0x21
Faultfinder Result:         Reached end address after faulting
Cryptographic Output:       46388397e4e4d50364af54 a0 0802922c
```

Listing 4.11: Faulted output from Faultfinder for instruction time: 5316

Fault models ARMory supports five different categories of fault:

- *Instruction faults*: permanent (injected at the beginning of simulation and never reverted) or transient (active for a single instruction).
- *Register faults*: permanent (injected on all writes), until-overwrite (injects the fault once, and it remains until the register is overwritten), and transient (removes the fault after a single instruction).

With Faultfinder's plain text fault model file we are able to replicate each of these faults using the lifespan count. '0' creates a permanent instruction fault and an active until overwrite register fault. Meanwhile a "revert" lifespan of '1' replicates a transient fault for both instructions and registers. A "repeat" lifespan would correspond to the permanent register fault; however, in the case study we replicated, no faults of this type were reported. ARMory uses a time value which corresponds directly to Faultfinder's instruction number.

ARmory fault	Faultfinder	
	Lifespan mode	Lifespan count
Instruction Permanent	—	0
Instruction Transient	revert	1
Register Permanent	repeat	999
Register Transient	revert	1
Register Until-overwrite	—	0

Table 4.12: Faultfinder's lifespan configuration to correspond to ARMory's fault categories

Timings We only reproduced ARMory's *exploitable faults*, so the timing results are not like-for-like. Because the binary-under-test was very short, we did not use the checkpoint optimisation.

Machine	*Frequency*	*#Cores*	*#Threads*	*ARMory*	*Faultfinder*	*Speed increase*
Intel i7-7700HQ	2.80 GHz	8	8	7 secs	19 secs	2.7x slower

Table 4.13: Pseudo runtime comparison between Faultfinder and ARMory, 42000 faults were injected into an ARM AES implementation binary (no optimisations used).

Discussion We found it very time-consuming to reproduce ARMory, because code modification and compilation was required. However, we acknowledge that ARMory's authors achieve excellent performance for ARM emulation. Due to the short number of lines in the binary we did not use the checkpoint optimisation as it creates no improvement.

4.7 Faultfinder Summary

In this chapter, we presented Faultfinder, a fast, easy-to-use tool for exhaustive fault injection simulation. Faultfinder is built on the Unicorn engine with efficiency at the heart of its design. Two novel optimisations, checkpoints and equivalences, help Faultfinder to improve performance and usability compared to prior work. Checkpoints heavily reduce redundant computation during simulation and equivalences reduce the analysis work required after finishing the experimental phase. Compared to state-of-the-art tools like ARCHIE, FiSim and ARMory, we show Faultfinder gives performance improvements of one or several orders of magnitude, both for cryptographic code and more generic applications like secure bootloaders. Faultfinder is available as an open-source project and we hope that our work can be built upon by academia and industry to further advance the simulation of fault injection.

	Faultfinder	*ARMory*	*ARCHIE*	*FiSim*
Built using	Unicorn (QEMU)	M-ulator	QEMU with TCG	Unicorn (QEMU)
Tested architectures	ARM Tricore x86_64 RISC-V	ARM	ARM RISC-V	ARM
Speed v Faultfinder	—	2x faster (ARM only)	70x slower	8x slower
Fault model	Text file	Hardcoded	Text file	Hardcoded
Faultable registers	128	16	16	16
OS	Linux	Linux	Linux	Windows
Last updated	Jul 2022	Nov 2021	Mar 2022	Nov 2020

Table 4.14: Comparison of currently available fault injection simulation tools

Chapter 5

Leaky Throttling

5.1 Introduction

This chapter contains so-far unpublished results achieved between 2020 and 2021. Later, several papers, on this topic by other researchers have been published [129, 214]. This work was done independently of those.

As we previously detailed in Chapter 1, modern processors use DVFS and throttling to protect the physical CPU from damage. In addition to these mechanisms, Intel provides software access to a huge range of live sensor readings and event counters, to support improved performance, system reliability [82] and efficient power consumption [171, 35]. From Sandy Bridge architectures onwards, Intel introduced Running Average Power Limit (RAPL) instrumentation. These became available via MSRs (detailed in Section 5.2) and through the `sysfs` interface which enables the values to be read from a public file. And, to reiterate, the emergence of cloud computing and shared infrastructures underscores the need for strong hardware security. Multi-tenanted systems should ensure that processes cannot leak information between them.

In 2019 Paiva [162] used the Dynamic Random Access Memory (DRAM) energy telemetry to create a covert channel to transmit messages by modulating the DRAM power consumption. In 2017, Fusi [60] demonstrated that the Intel's RAPL energy telemetry interface leaks information. Lipp et al. in 2021 [127], used this RAPL interface to expose a low-resolution side channel attack which they dubbed: *PLATYPUS*.

We know that some instructions use more power (known as PHI)—it is no surprise that a 256-bit fused multiply-add (`FMA256`) uses more energy than a 32-bit register move (`mov`) [77]. More commonly used instructions would include Intel's Advanced Vector Extensions (AVX) which were introduced in 2008, these, again increase energy consumption [101]. A process is able to switch between a standard instruction and a PHI in only a few clock cycles. And, as we saw in Chapter 2, modifying the voltage takes a substantial amount of time (between 500k and 1 million clock cycles), to manage this huge imbalance ($\approx$5 vs. 500,000 clock cycles!), throttling can be employed to give the DVFS management systems time to address the power issues. In this chapter we create a covert process and arbitrarily run three different operations, *square root*, *multiply* and *array lookup*. We demonstrate that the energy consumption of each individual operands can be identified by their energy consumption at different frequencies. We then show that the much less granular *power throttling* also enables operand recognition.

5.2 Energy MSRs

Intel provides MSRs for four energy domains [37] as shown in Figure 5.1

- Package (PKG): domain measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components
- DRAM: measures the energy consumption of random access memory (RAM) attached to the integrated memory controller.
- Power Plane 0 (PP0): measures the energy consumption of all processor cores on the socket.
- Power Plane 1 (PP1): measures the energy consumption of processor graphics (only available on desktop models).

It should be noted that: $PP0 + PP1 <= PKG$ while DRAM is independent of the other three domains.

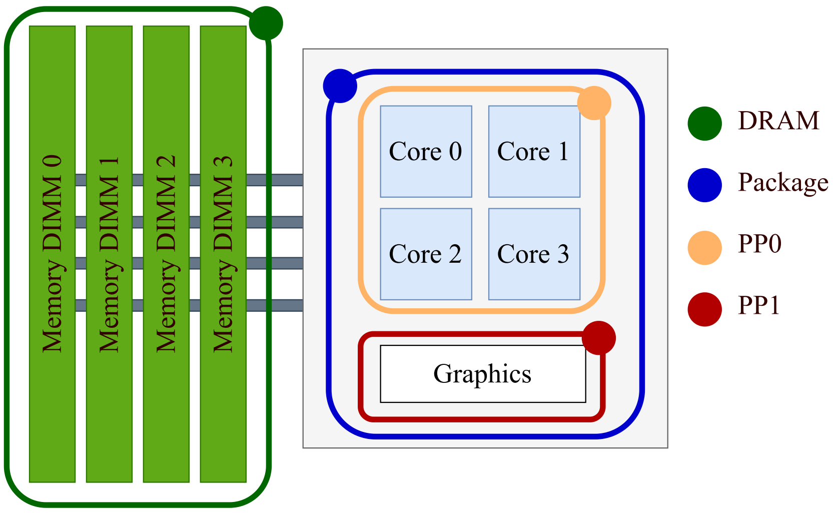


Figure 5.1: Layout of the power domains with MSRs availability

Intel allows a privileged user to set: power limits and domain policies and read energy usage, voltage and power limits. These MSRs are listed in Table 5.1. An additional MSR that we already used in Chapter 2 is `MSR_PERF_STATUS` - 0x198 which reports Core Voltage.

Domain	*Power Limit* *RW*	*Energy Status* *RO*	*Policy* *RW*	*Perf Status* *RO*	*Power Info* *RO*
PKG	PKG_POWER_LIMIT 0x610	PKG_ENERGY_STATUS 0x611	—	PKG_PERF_STATUS 0x613	PKG_POWER_INFO 0x614
DRAM	DRAM_POWER_LIMIT 0x618	DRAM_ENERGY_STATUS 0x619	—	DRAM_PERF_STATUS 0x61B	DRAM_POWER_INFO 0x61B
PP0	PP0_POWER_LIMIT 0x638	PP0_ENERGY_STATUS 0x639	PP0_POLICY 0x63A	PP0_PERF_STATUS 0x63B	—
PP1	PP1_POWER_LIMIT 0x640	PP1_ENERGY_STATUS 0x641	PP1_POLICY 0x642	—	—

Table 5.1: RAPL MSRs (each is prefixed with `MSR_`)

5.2.1 ENERGY_STATUS MSRs

The `ENERGY_STATUS` MSRs represent the accumulated energy consumed for each domain as shown in Figure 5.2. The units are specified in `MSR_RAPL_POWER_UNIT`, Figure 5.3.

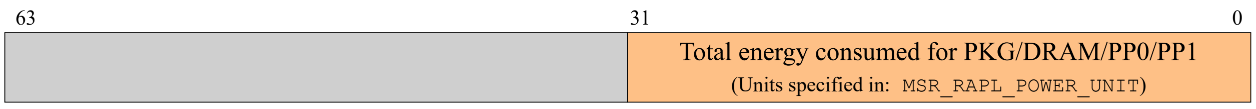

63
31
0
Total energy consumed for PKG/DRAM/PP0/PP1
(Units specified in: MSR_RAPL_POWER_UNIT)

Figure 5.2: Layout of MSRs `0x611, 0x619, 0x639, 0x641 ENERGY_STATUS` for `PKG, DRAM, PP0, PP1` respectively

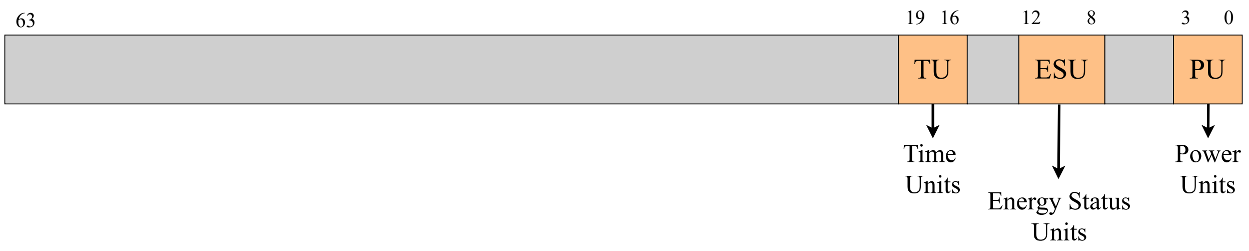


Figure 5.3: Layout of MSR `0x606 MSR_RAPL_POWER_UNIT`

$$Actual\ Power\ Units\ (Watts) = 2^{PU}$$
$$Actual\ Energy\ Units\ (Joules) = 2^{ESU}$$
$$Actual\ Time\ Units\ (Seconds) = 2^{TU}$$

5.3 Energy Usage Experiments

In this work, we further investigate Intel's documented telemetries in search of information leakage, including operand differentiation. We focused our efforts on power and energy. All experiments were performed with an Intel i7-7700HQ-XPS laptop. All graphs exclude readings outside of the 98 per cent quantile.

We began by creating a covert process with an 'off' mode where the process would sleep (represented by 0). And an 'on' mode where we test different functions and analyse the telemetry results. Intel reports that the MSRs update every 1ms, we therefore read the following MSRs every 2ms.

- Core voltage
- Package energy usage
- DRAM energy usage

We review the graphical results in the following subsections.

5.3.1 Square Root of Scalar Double

We began by testing the `sqrtsd` (Square Root of Scalar Double) function in a tight loop (Listing 5.1) for our covert channel's 'on' mode. We arbitrarily gave this function the value of '10' in our covert process. We record the telemetries in their raw state, as we were initially focused on correlation. We tested for two frequencies: 3.2GHz and 1.1GHz—CPU power consumption rises as the square of the core voltage with clock frequency rising in proportion to voltage. Consequently the energy readings will be significantly different. We did not change the loop iterations, and therefore, at a clock frequency of 1GHz, the time taken to complete is longer and more readings are taken. It is worth noting that, in this section, the y-axis (MSR readings) were calculated automatically and are not showing absolute values. We report the absolute values in the next section.

```
void asm_sqrtsd()
{
    double a = rand();
    double b = 0;
    for(int i = 0; i < ITERATIONS_SQRT; i++)
    {
        asm volatile(
            "movq %1, %%xmm0 \n"
            "sqrtsd %%xmm0, %%xmm1 \n"
            "movq %%xmm1, %0 \n"
            : "=r"(b)
            : "g"(a)
            : "xmm0", "xmm1", "memory"
        );
    }
}
```

Listing 5.1: Code for tight square root of scalar double loop for covert channel

From the graphs (Figures 5.4 and 5.5) it is clear that voltage and package energy correlate directly with the active operation `sqrtsd`. Interestingly, when the clock frequency is running at 3.2GHz, DRAM energy reduces when the `sqrtsd` of a scalar double is running. Running a mathematical calculation in a tight loop will not require access DRAM and it would be expected to see lower energy use. However at 1.1GHz the correlation cannot be clearly observed—indeed it the regularity of the rise and fall suggest another process is regularly accessing something in memory.

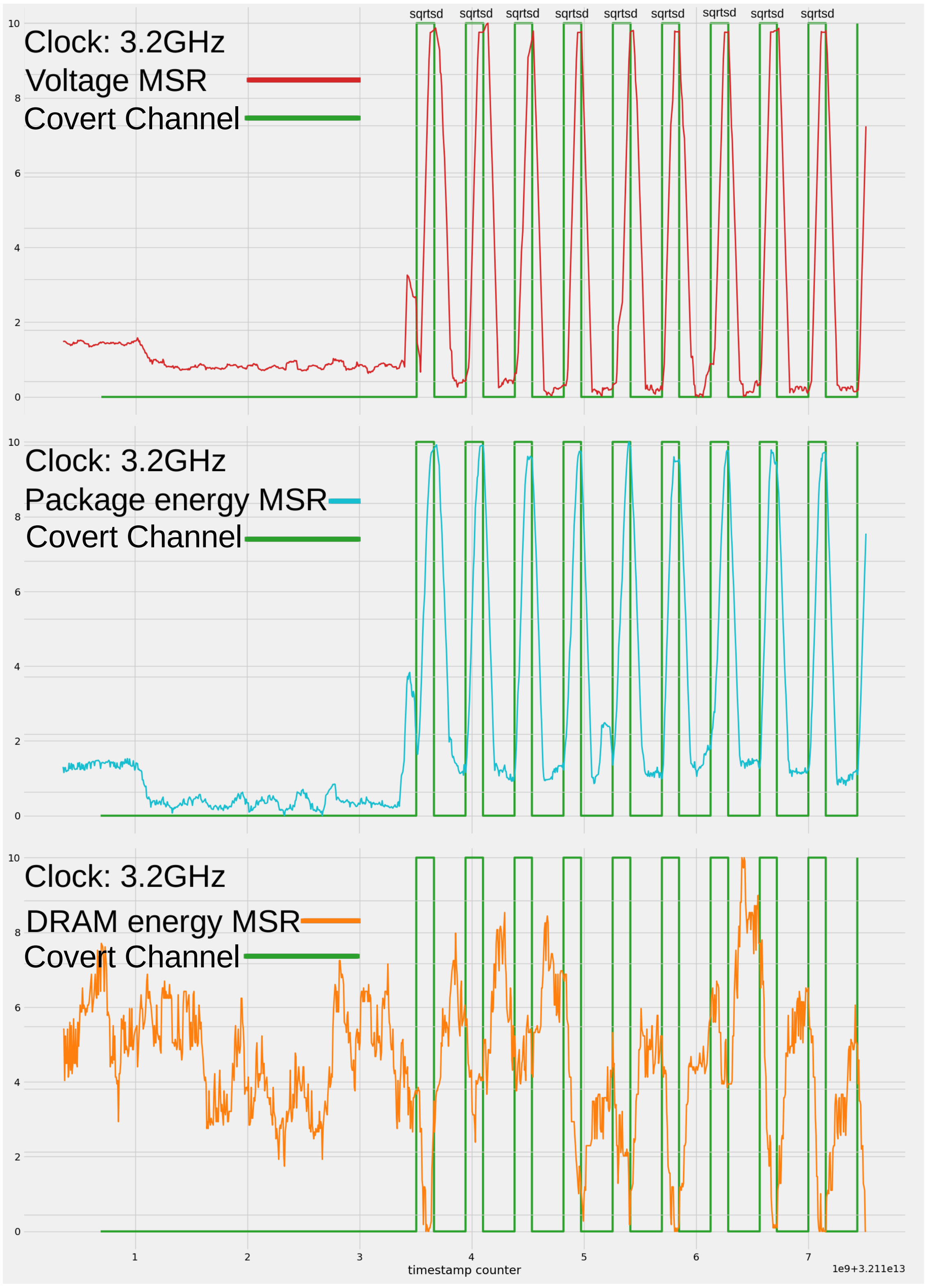


Figure 5.4: Covert channel running `sqrtsd` with MSR outputs from Package Energy, DRAM Energy and Core Voltage on an i7-7700HQ-XPS at 3.2GHz, using a rolling average of 15 points. Readings taken every 2ms.

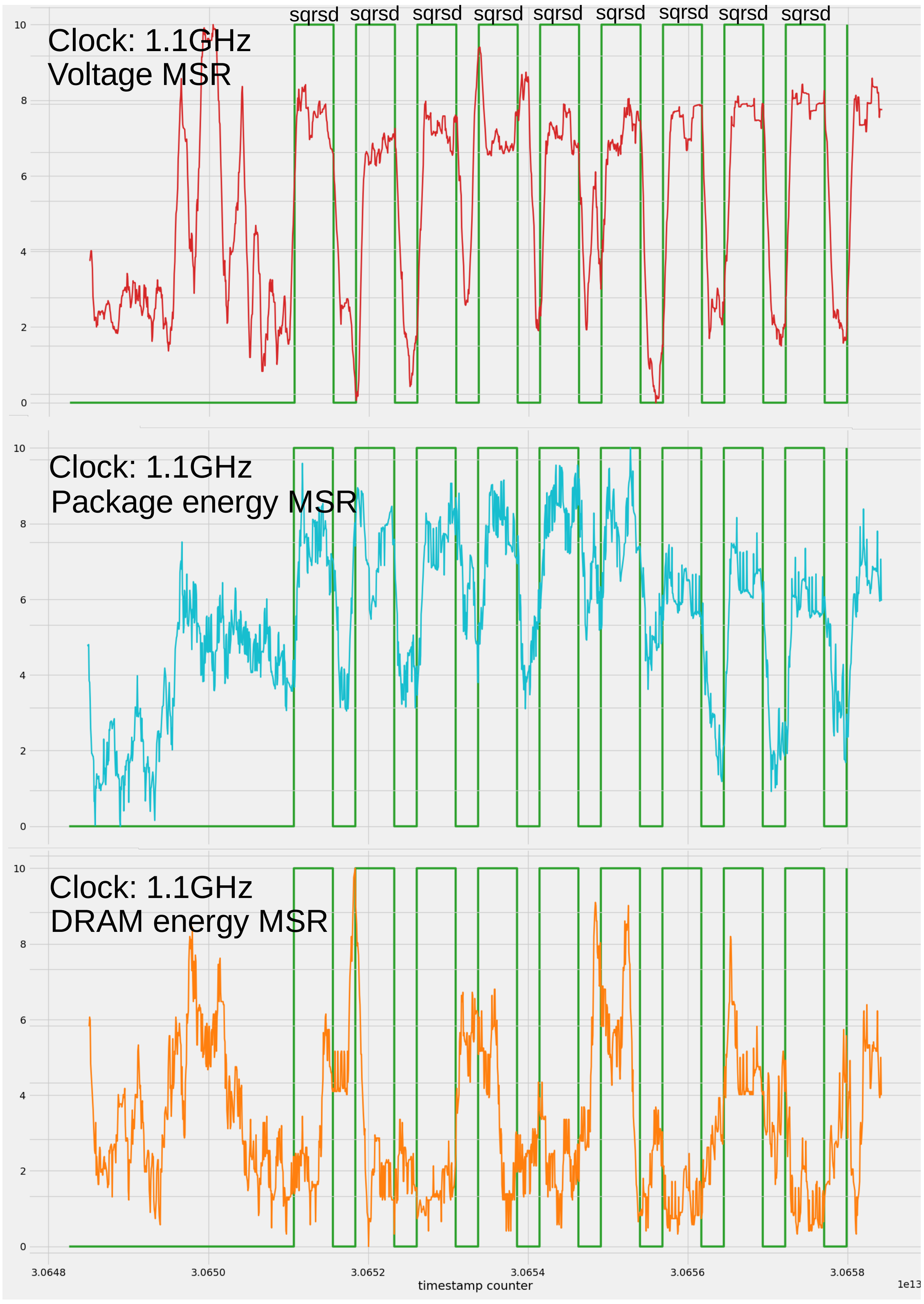


Figure 5.5: MSR outputs from Package Energy, DRAM Energy and Core Voltage on i7-7700HQ-XPS at 1.1GHz, using a rolling average of 15 points. Readings taken every 2ms.

5.3.2 Multiplication - imul

We now extend our covert channel to include the `imul` operation and assigned this the arbitrary value of '9' (Listing 5.2). For these sets of experiments we additionally include absolute values for greater comparison between different clock frequencies. As can be seen in Figure 5.6 at 3.2GHz, the `imul` uses more energy ($\approx 4mJ$) than the `sqrtsd` and the voltage is lower by $\approx 3mV$. Meanwhile at 1.1GHz (Figure 5.7) the voltage readings are lower for the `imul` by $< 1mV$, it is immediately possible to determine which operation was running based on the package energy readings. However, it is clear that higher frequencies produce greater differences in energy readings and the change in operation becomes obvious as a result.

```
void asm_imul()
{
    int input_val = rand();
    int output_val = rand();
    for (int i = 0; i < ITERATIONS_IMUL; i++)
    {
        asm volatile(
            "imul %1, %0 \n"
            : "+r"(output_val)
            : "rm"(input_val));
    }
}
```

Listing 5.2: Code for tight multiplication loop for covert channel

5.3.3 Array Lookup

And finally, we add in a third operation, an SBOX array lookup (Listing 5.3) which we assign the operation value '8'. We focus on package energy MSR readings because these were demonstrated to be the most reliable across clock frequencies. In Figure 5.8, we can clearly differentiate each of the covert operations. The `sqrtsd` consumes $\approx 27mJ$, `imul` $\approx 34mJ$ and `array lookup` $\approx 26mJ$.

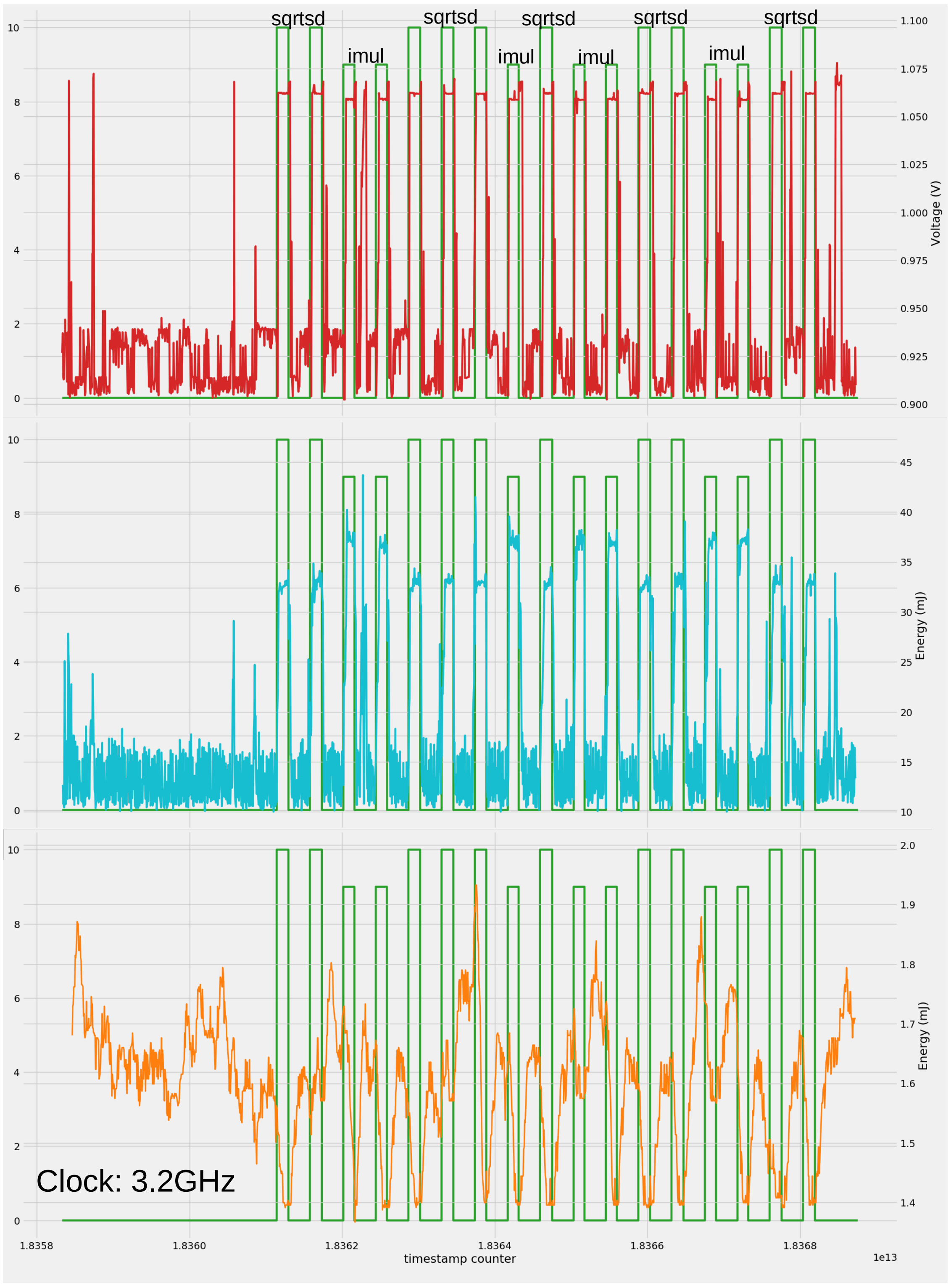


Figure 5.6: MSR outputs from Package Energy, DRAM Energy and Core Voltage on i7-7700HQ-XPS at 3.2GHz, using a rolling average of 10 points. Readings taken every 2ms.

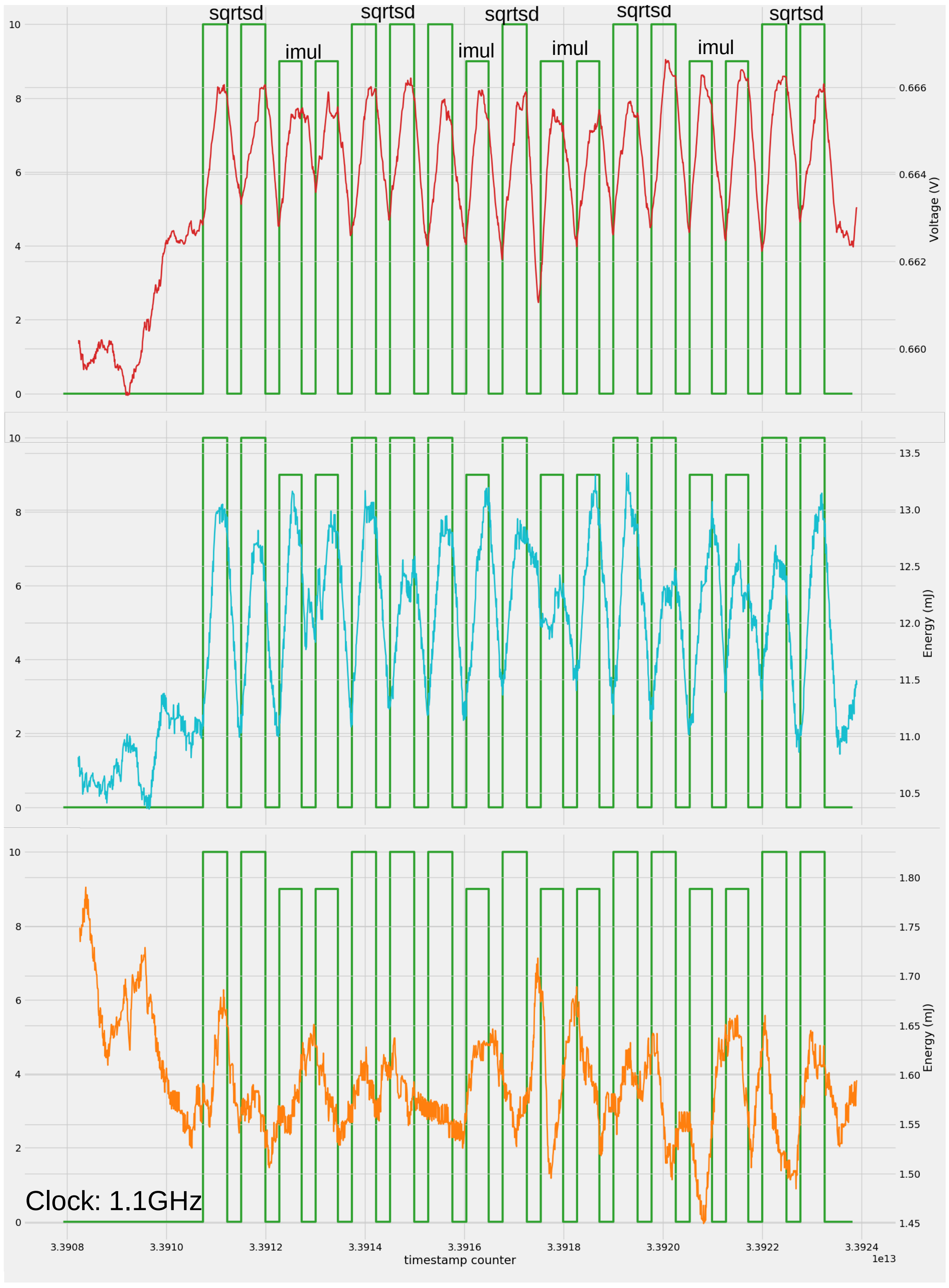


Figure 5.7: MSR outputs from Package Energy, DRAM Energy and Core Voltage on i7-7700HQ-XPS at 1.1GHz, using a rolling average of 10 points.

```
uint64_t run_array_memory_lookup()
{
    uint64_t v = rand() % 512;
    for (int i = 0; i < ITERATIONS_ARRAY_LOOKUP; i++)
    {
        asm volatile("mfence");
        v = SBOX[v];
        asm volatile("mfence");
    }
    return v;
}
```

Listing 5.3: Code running repeated array lookups covert channel

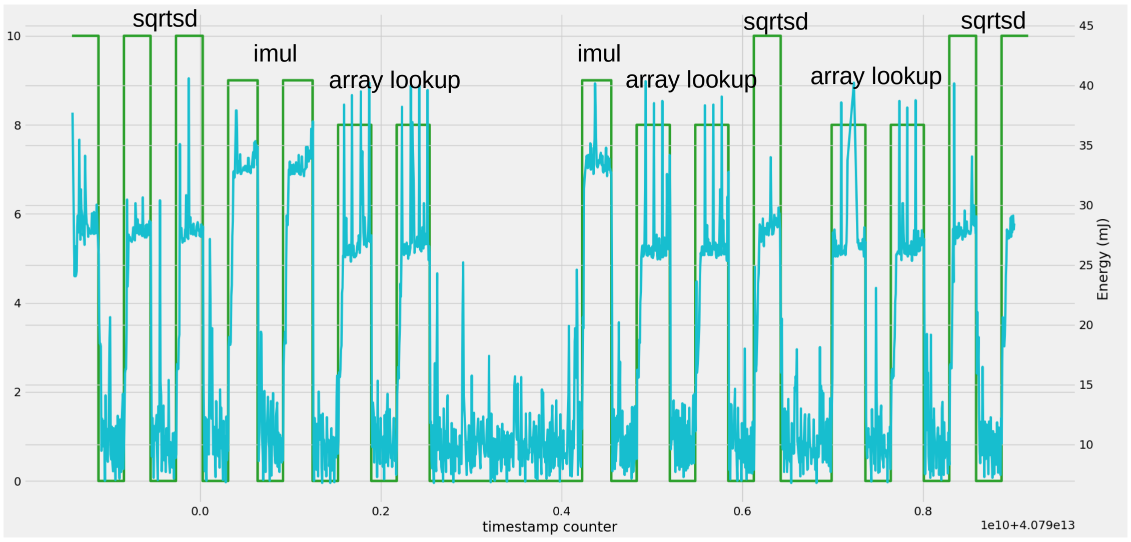


Figure 5.8: MSR outputs from Package Energy i7-7700HQ-XPS at 3.2GHz, no rolling average. Readings taken every 2ms.

We ran the covert operation 100 times, on i7-7700HQ-XPS at 2.7GHz with no other user processes running. We noted that the energy usage when `sleeping` fell into two distinct categories: 9.5-9.7mJ and 25.5-25.5mJ. The higher energy usage value is likely to be a kernel process running that was not visible to the user. We recorded the range in *package energy increase from sleep* in table Table 5.2. The range in package energy usage for each operation is surprisingly small and clearly identifiable.

	Sleeping Energy Usage	
	(9.6 ± 0.1 mJ)	*26 ± 0.5 mJ)*
Operation	*Increase in package energy usage*	
imul	90.4% ± 0.028	26.4% ± 0.002
sbox lookup	69.1% ± 0.014	19.5% ± 0.009
sqrtsd	62.0% ± 0.020	17.1% ± 0.008

Table 5.2: Average increase in package energy usage for covert operations across 100 experiments, frequency 2.7GHz.

5.4 Throttling

Previous research had investigated the leaking of information via DRAM energy telemetry[162] and package and pp0 energy [127, 73]. We therefore build upon our earlier tests to investigate leakage using power-limits and throttlings, which, at the time, had no substantial published research.

5.4.1 Setting Power Limits

RAPL also enables power consumption to be limited, such that: when the configured power limit is exceeded, the CPU will be throttled *i.e.*, forced to run at a lower frequency to maximise chip protection. Intel currently provides package-level power limits (PL1, PL2, PL3) and, from 10th generation onwards platform-level power limits (Psys). Users or applications at ring 0 can configure the running average time window τ and the power limit of each capability through the `POWER_LIMIT` MSRs as shown in Figure 5.9.

The formula for the time window (from Intel documentation [37]) is as below:

$$Time\ window\ (secs) = 2^{window\ y} + \left(1 + \frac{window f}{4}\right) * Actual\ Time\ Units \tag{5.1}$$

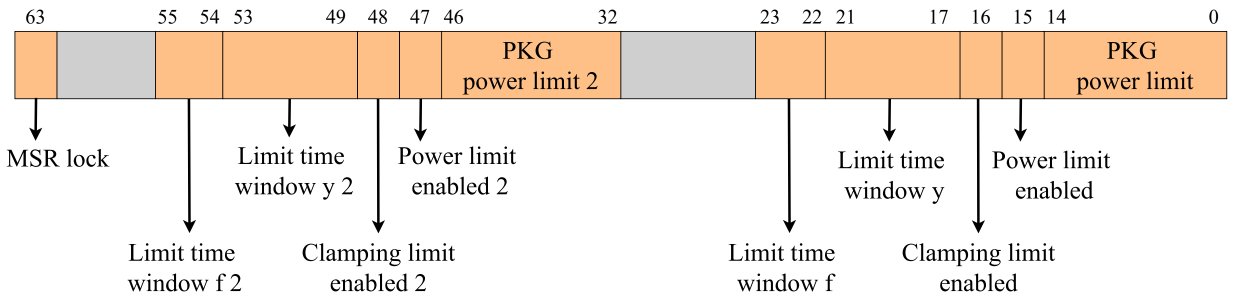


Figure 5.9: Layout of the documented MSR `0x610 - MSR_PKG_POWER_LIMIT`

The `PERF_STATUS` MSRs represent the accumulated time the domain has been throttled as shown in Figure 5.10.

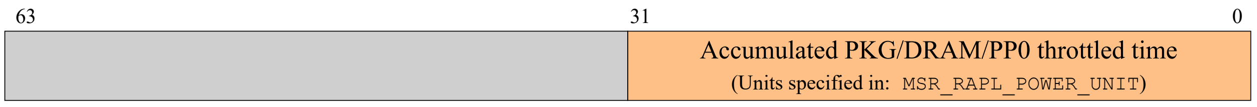

63
31
0
Accumulated PKG/DRAM/PP0 throttled time
(Units specified in: MSR_RAPL_POWER_UNIT)

Figure 5.10: Layout of MSRs `0x613/0x61B/0x63B PERF_STATUS` for `PKG/DRAM/PP0`

5.4.2 Threat model

The PLATYPUS attack read the RAPL values as both a privileged and unprivileged attacker. Unprivileged access was possible because, on Linux, the power capping framework `powercap` exposed the MSRs as files located in: `/sys/devices/virtual/powercap` to ring 3 users. However, in November 2020, a security update revoked this access and an unprivileged attacker can no longer read these power measurements.

Ring 3 Attacker

A Ring 3 attacker would have no access to modify any electrical or thermal limits. However, they might be able to effectively simulate those limits by using carefully tested stressor code. This was not an area that we investigated, but would be open for further research.

Privileged Attacker

Being able to modify the energy limits and time windows requires a privileged attacker, e.g., a kernel process. In this scenario, the victim process would be running inside SGX. As we saw in Section 2.2.1, SGX assumes that only the CPU package and application code are trustworthy. An adversary may, however, compromise other components including the operating system and other privileged system interfaces. For this reason we moved our covert processes inside an SGX enclave for the following tests.

5.4.3 Experiments

In the earlier experiments we observed greater granularity in changes at the higher frequencies, Therefore, for these experiments we picked two frequencies at the higher end of those available: 3GHz and 2.4GHz. We reduced the window to the smallest possible value and modified the power limits using `MSR_PKG_POWER_LIMIT`, on the test machine, `MSR_PP0_POWER_LIMIT` was not available.

5.4.4 Observations

In section 5.3 we recorded that package energy usage was highest for: `imul` and lowest for `array lookups`. Figures 5.11 and 5.12 align with the expectation that the operands consuming the greatest energy are those that will be throttled the most. We observed that the operand `imul` causes the throttling to kick-in quickly and continuously for both clock frequencies (3GHz and 2.4GHz). Meanwhile the throttling was not deployed for the `array lookup` when the clock frequency was set to 3GHz and only very briefly at 2.4GHz.

Furthermore, we witness that the enclave offers no protection from leaking information via

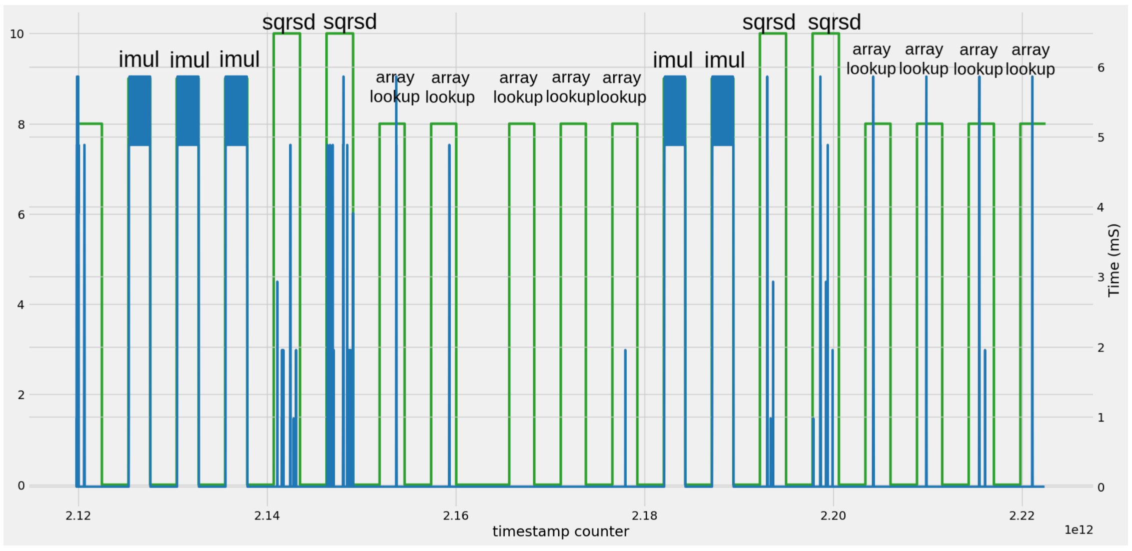


Figure 5.11: MSR outputs from Package Throttling i7-7700HQ-XPS at 2.4GHz with a power limit of 7.875 Watts and a window of 0.9765ms no rolling average. MSR readings taken every 4ms. Covert functions running inside SGX.

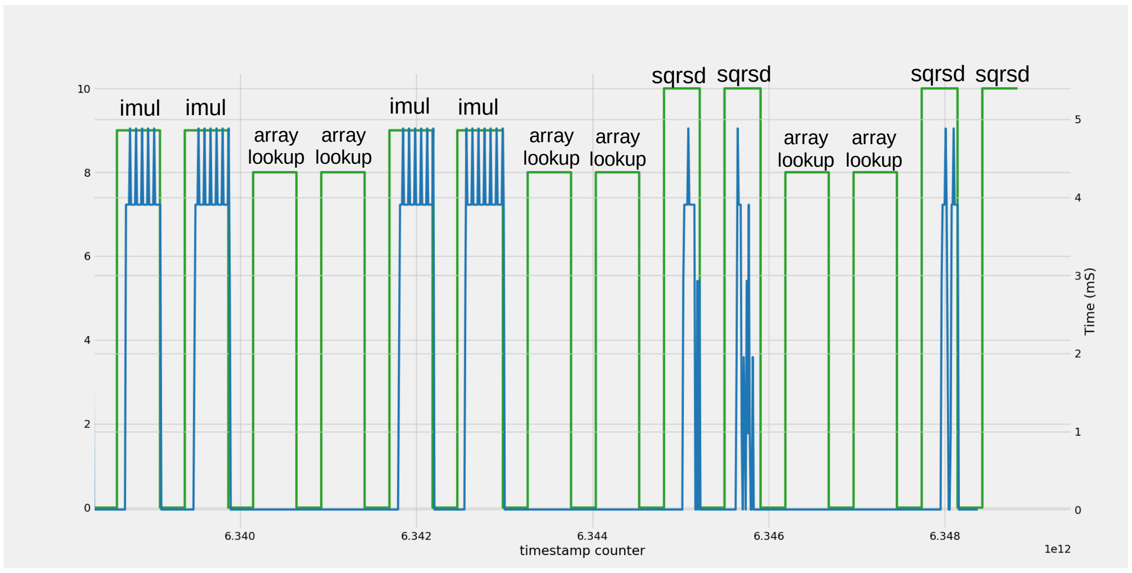


Figure 5.12: MSR outputs from Package Throttling i7-7700HQ-XPS at 3GHz with a power limit of 16 Watts and a window of 0.9765ms no rolling average. MSR readings taken every 4ms. Covert functions running inside SGX.

the power and energy readings. This is because the frequency and voltage regulators are oblivious to processes running inside an enclave, they serve the whole processor regardless of any secure code that may be running.

5.5 Non-continuing work

Our work clearly demonstrates that, if the power limits can be modified (or *manipulated* by other stressor code), there is leakage from the throttling telemetries, However, modifying the power limits requires a privileged access to the server—which only fits the SGX threat model. We felt that the work was going to be *another* paper attacking SGX, given the huge quantity of research published in this arena we did not continue the work.

5.6 Subsequent work

Having decided not to continue the research, three papers were subsequently published relating to throttling! In 2021, Haj-Yahya et al. [76] were the first to demonstrate that multi-level throttling side-effects can open up covert channels. In 2022, Intel [129] and Wan et al. [214] both demonstrated that forcing a machine to throttle changes the runtime of processes, including cryptographic *constant runtime implementations*. Their work demonstrated that current industry guidelines for how to write constant-time code are not sufficient to guarantee the execution time is actually constant—demonstrating the incredible complexity of writing secure cryptographic algorithms.

Chapter 6

Conclusions

The vast majority of the population unthinkingly put their private and personal information into physical devices and thus the need for trust is clear. TEEs such as Intel's SGX are promising concepts *i.e.*, isolated software components in the processor such that the underlying operating system does not need to be trusted. However SGX does not exist in a vacuum—it (at time of writing) lives on the same physical hardware and is bound by the same power management systems. Building a fortress on weak foundations does not make a good stronghold. SGX is built upon the dynamic execution x86 architecture—it was the out-of-order feature which lead directly to the infamous Foreshadow attack [204].

With Plundervolt we presented the first practical attack that directly breached the integrity guarantees in the Intel SGX security architecture. We were able retrieve keys by lowering the operating voltage but, interestingly, we also noted that the ambient temperature affected *how much undervolting* we were required to implement. Our work provided evidence that the enclaved execution promise of outsourcing sensitive computations to untrusted remote platforms creates new and unexpected attack surfaces. Consequently future additions to the Trusted Computing Base must also consider a enclave's: underlying architecture, system resources, physical dependencies and environment—a non-trivial ask!

Meanwhile, Voltpillager, was the first hardware-based fault injection attack against SGX that side-stepped the Plundervolt mitigations. However, when the issue was reported to Intel their response was, *"... opening the case and tampering of internal hardware to compromise SGX is out of scope for SGX threat model. Patches for CVE-2019-11157 (Plundervolt) were not designed to protect against hardware-based attacks as per the threat model"*. Intel has created a secure vault, designed with the specific purpose of securely holding and processing private information e.g., passwords and authentication keys. Denying that hardware attacks are within scope was a surprising response, particularly because a number of cloud providers Azure [142], Enarx [48] and Fortanix [55] all consider the host as untrusted in their threat model.

Several prominent companies are using SGX: Microsoft has integrated SGX into its Azure confidential computing platform [142] to provide secure environments for its customers and Alibaba has used SGX to secure its cloud-based blockchain platform [27]. Many other companies are likely using SGX but have not made public announcements due to confidentiality or security concerns.

In 2020 Nilsson et al. reviewed 24 attacks against SGX [154] and separated the attacks into seven different categories: (*i*) Controlled Channel Attack (*ii*) Cache-attack (*iii*) Branch Prediction Attacks (*iv*) Speculative Execution Attack (*v*) Rogue Data Cache Loads (*vi*) Microarchitectural Data Sampling (*vii*) Software-based Fault Injection Attacks. Additionally, when we searched google scholar for articles with the words 'SGX attack' it returned 7710 results. It is clear that the challenges to build a fully secure Trusted Execution Environment are immense—the enclave has to defend against *everything, always* and an attacker *only has to get lucky once.*

As with SGX, there has been extensive research into fault-injection attacks, this work has exploded with the proliferation of embedded devices. These devices have smaller amounts of memory and thus: older, less security-aware (but more efficient) programming languages are in the resurgence. Buffer overflows were partially documented fifty years ago [6] and `out-of-bounds write` was the number one most dangerous software weakness according to the Common Weakness Enumeration website [49]. Embedded security researchers have amassed a huge collective knowledge on fault injection attacks. However the journey to a deep understanding is long and only travelled by a few. When we consider that the number of embedded devices is growing rapidly then there has never been a greater need for automated tools. Such applications could (and should) test the weaknesses of binaries prior to being loaded onto the hardware. One of Faultfinder's strengths is its ability to run on any processor supported by QEMU, currently: ARM, ARM64 (ARMv8), M68K, MIPS, PowerPC, RISCV, SPARC, S390X, TriCore and X86 (16, 32, 64-bit). When submitting Faultfinder to conferences we received mostly excited feedback, a number of reviewers stated they would be keen to use the software and could see its benefit. However, even the enthusiastic reviewers viewed the tool as just™ useful to support fault-injection research. ARMory, [84] one of the tools for which we did a direct comparison, demonstrated that adding in a countermeasure to protect against one single type of fault can increase the binary's vulnerability to a different fault model. Researching hardware fault-injection during lockdown was challenging and we hope that in future work, we can build upon the preliminary examples in this thesis and colaborate with the embedded security community to demonstrate Faultfinder's strengths and versatility. The challenge of protecting binaries from fault injection attacks will continue for many, many years to come.

Appendix A

Appendix for Plundervolt

This appendix is from the author's work published in S&P 2020 [149] and IEEE 2020 [150]

A.1 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
```

A.2 Example Fault for RSA-CRT

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

n = 0xBBF82F090682CE9C2338AC2B9DA871F7368D07EED41043A440D6B6F07454F51FB8DFBAAF035C
02AB61EA48CEEB6FCD4876ED520D60E1EC4619719D8A5B8B807FAFB8E0A3DFC737723EE6B4B7D93A258
4EE6A649D060953748834B2454598394EE0AAB12D7B61A51F527A9A41F6C1687FE2537298CA2A8F5946
F8E5FD091DBDCB

e = 0x11

d = 0xA5DAFC5341FAF289C4B988DB30C1CDF83F31251E0668B42784813801579641B29410B3C7998D6
BC465745E5C392669D6870DA2C082A939E37FDCB82EC93EDAC97FF3AD5950ACCFBC111C76F1A9529444
E56AAF68C56C092CD38DC3BEF5D20A939926ED4F74A13EDDFBE1A1CECC4894AF9428C2B7B8883FE4463
A4BC85B1CB3C1

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

x = 0x1253E04DC0A5397BB44A7AB87E9BF2A039A33D1E996FC82A94CCD30074C95DF763722017069E
5268DA5D1C0B4F872CF653C11DF82314A67968DFEAE28DEF04BB6D84B1C31D654A1970E5783BD6EB96A
024C2CA2F4A90FE9F2EF5C9C140E5BB48DA9536AD8700C84FC9130ADEA74E558D51A74DDF85D8B50DE9
6838D6063E0955

y = 0xEB7A19ACE9E3006350E329504B45E2CA82310B26DCD87D5C68F1EEA8F55267C31B2E8BB4251F8
4D7E0B2C04626F5AFF93EDCFB25C9C2B3FF8AE10E839A2DDB4CDCFE4FF47728B4A1B7C1362BAAD29AB4
8D2869D5024121435811591BE392F982FB3E87D095AEB40448DB972F3AC14F7BC275195281CE32D2F1B
76D4D353E2D

Injecting a fault during the first half of the RSA-CRT computation on the i3-7100U-Aat 1 GHz with -225 mV undervolting, the following faulty y' was obtained in one of our experiments:

y' = 0xAA105EAFB6BDD9E5A15443729670B70F042889103E023428F37B1CEFFAECC91292772652E201
6AA5955DFDA6FD5B685AE062A32DEA9C9E99F516370BE2ED4EF48A3C3513E4026E5DE3647267A83C9C2

```
45A72EA9F4D8C2B373A8CE70047C922A108807197A6BC15A1DF31E06FCD5521AA00ECC0B3A2A5BCDDE5

A8B7B5AAD3015F
```

A.3 Further Examples for AES-NI AES Encryption 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
```

A.4 Running DFA against AES-NI

Based on the fault described in Section 2.4.3, 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-Awith -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`.

A.5 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 2.5.1. 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 [93]. 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 initialised 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
```

Appendix B

Appendix for Voltpillager

B.1 Example Results for Faults during Memory Accesses

This appendix is based upon the author's contribution to the published work in USENIX 2021 [32].

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

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

Listing B.1: Overflow on i7-7700HQ-XPS

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

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

Listing B.2: Underflow on i7-7700HQ-XPS

Appendix C

Appendix for Faultfinder

C.1 Setup Files

C.1.1 Example Binary Configuration File

```
{
    "comment"              :      "Everything in the comment is not used -
   other than by the humans",
    "binary filename"      :      "experiments/archie/bins/example-aes-archie.
   elf",
    "unicorn arch"         :      "arm",
    "unicorn mode"         :      "thumb",
    "cpu"                  :      "CORTEX_M7",
    "capstone arch"        :      "arm",
    "capstone mode"        :      "thumb",
    "memory address"       :      "0x08000000",
    "memory size"          :      "0x80000",
    "other memory"         :
    [
        {
            "address"      :    "0x20000000",
            "size"         :    "0x4000"
        }
    ],
    "code offset"    :      "0x10000",
    "stack address"  :   "0x82000000",
    "stack size"     :   "0x00001000",
    "code start"     :       "0x00dd",
    "code end"       :       "0x0122",
    "fault start"    :        "0x106",
    "fault end"      :        "0x122",
    "set memory"     :
    [{
            "type"            : "address",
            "format"          : "hex",
            "byte_array"      : "2b7e151628aed2...00127a0000127a00",
            "address"         : "0x20000000"
    }],
    "set registers"    :
    [
    ],
    "outputs"     :
    [{
            "comment"         : "sbox data",
            "location"        : "fixed address",
            "address"         : "0x20000000",
            "register"        : "",
            "length"          : "16",
            "offset"          : "not used"
    }],
    "skips":
    [{
            "address"         : "0xfe",
            "bytes"           : "4",
            "comment"         : "gpio_set"
        },
        {
            "address"         : "0x110",
            "bytes"           : "4",
            "comment"         : "gpio_clear"
    }],
    "hard stops":
    [{
            "address"         : "0x124c"
    }],
    "time out":"500000"
}
```

C.1.2 Example Fault Rules File

```
#Faulting sbox access. 0x800101a-0x8001032
Instructions: 4937-4942
 Instruction:
  Op_codes: ALL
   Lifespan: 2, revert
    Operations: OR
     Masks:0x4,0x400
```

C.1.3 Example Campaign File

```
{
    "binary json filename"   : "experiments/archie/jsons/binary-details.
   json",
    "mode"                   : "fault",
    "output directory name" : "experiments/archie/outputs/morphius/0
   _checkpoints",
    "threads"                : "8",
    "fault model filename"   : "experiments/archie/faultmodels/
   replicating_archie_arm.txt",
    "checkpoints"            : "no",
    "number of checkpoints" : "0",
    "equivalents"            : "no",
    "timeit"                 : "no",
    "max instructions"       : "300000",
    "display disassembly"    : "yes"
}
```

C.2 Fault Models

C.2.1 ARCHIE Fault Model

```
[{  "comment1" : "Sbox data fault in register r1",
    "fault_address"      : [1],
    "fault_type"         : "register",
    "fault_model"        : "set1",
    "fault_livespan"     : [4],
    "fault_mask"         : [1, 256, 1],
    "comment2"           : "Fault injected after values loaded from sbox 0
   x8001024, rounds 8 - 10",
    "trigger_address"    : [134221860],
    "trigger_counter"    : [128, 160, 1]
}],
[{  "comment1"           : "Fault instructions of sbox access. 0x800101a
   (134221850) - 0x8001032 (134221874)",
    "fault_address"      : [134221850, 134221874, 2],
    "fault_type"         : "instruction",
    "fault_model"        : "set1",
    "fault_livespan"     : [2],
    "fault_mask"         : [4],
    "trigger_address"    : [-1],
    "trigger_counter"    : [141]
}],
[{
    "comment1": "Fault last round check: 0x8001066(134221926) - 0x800106c
   (0x800106c)",
    "fault_address"      : [134221926, 134221932, 2],
    "fault_type"         : "instruction",
    "fault_model"        : "set1",
    "fault_livespan"     : [2],
    "fault_mask"         : { "type" : "shift" , "range" : [3, 0, 14]},
    "trigger_address"    : [-1],
    "trigger_counter"    : [8, 10, 1]
}]
```

C.2.2 Faultfinder Fault Model

```
Instructions: 4371-4371,4851-4851,4857-4857,4863-4863,4869-4869,4880-4880,
    4886-4886,4892-4892,4898-4898,4909-4909,4915-4915,4921-4921,4927-4927,
    4938-4938,4944-4944,4950-4950,4956-4956,5436-5436,5442-5442,5448-5448,
    5454-5454,5465-5465,5471-5471,5477-5477,5483-5483,5494-5494,5500-5500,
    5506-5506,5512-5512,5523-5523,5529-5529,5535-5535
 Registers: 1
   Op_codes: ALL
    Lifespan: 4,revert
     Operations: OR
     Masks:0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf,
      0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,
      0x1d,0x1e,0x1f,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,
      0x2a,0x2b,0x2c,0x2d,0x2e,0x2f,0x30,0x31,0x32,0x33,0x34,0x35,0x36,
      0x37,0x38,0x39,0x3a,0x3b,0x3c,0x3d,0x3e,0x3f,0x40,0x41,0x42,0x43,
      0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x4d,0x4e,0x4f,0x50,
      0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x5b,0x5c,0x5d,
      0x5e,0x5f,0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,
      0x6b,0x6c,0x6d,0x6e,0x6f,0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,
      0x78,0x79,0x7a,0x7b,0x7c,0x7d,0x7e,0x7f,0x80,0x81,0x82,0x83,0x84,
      0x85,0x86,0x87,0x88,0x89,0x8a,0x8b,0x8c,0x8d,0x8e,0x8f,0x90,0x91,
      0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0x9b,0x9c,0x9d,0x9e,
      0x9f,0xa0,0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xab,
      0xac,0xad,0xae,0xaf,0xb0,0xb1,0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,0xb8,
      0xb9,0xba,0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,
      0xc6,0xc7,0xc8,0xc9,0xca,0xcb,0xcc,0xcd,0xce,0xcf,0xd0,0xd1,0xd2,
      0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xdb,0xdc,0xdd,0xde,0xdf,
      0xe0,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xeb,0xec,
      0xed,0xee,0xef,0xf0,0xf1,0xf2,0f3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,
      0xfa,0xfb,0xfc,0xfd,0xfe,0xff
Instructions:937-4942
    Instruction:
        Op_codes: ALL
            Lifespan: 2,revert
```

```
                Operations: OR
                    Masks:0x4
Instructions: 4404-4406,4989-4991
    Instruction:
        Op_codes: ALL
            Lifespan: 2,revert
                Operations: OR
                    Masks:0x3<0<14
```

Appendix D

Appendix for Leaking Throttling

D.1 Covert Code

```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#define myprintf(...) printf("Covert> " __VA_ARGS__)

// bigger iterations for throttling
#define ITERATIONS_SQRT         20000000ul  // covert = 10 (sqrtsd)
#define ITERATIONS_IMUL         60000000ul  // covert = 9 (multply)
#define ITERATIONS_ARRAY_LOOKUP 5000000ul   // covert = 8 (array lookup)

static uint64_t SBOX[512] = {
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
 .
 .
 .
 0x142, 0x168, 0x141, 0x199, 0x12d, 0x10f, 0x1b0, 0x154, 0x1bb, 0x116};
```

```
typedef struct data_arrays_t
{
    uint64_t tsc;
    int mode;
} data_arrays;

uint64_t rdtsc()
{
    uint64_t a, d;
    asm volatile("mfence");
    asm volatile("rdtsc"
                 : "=a"(a), "=d"(d));

    a = (d << 32) | a ;

    asm volatile("mfence");
    return a;
}


void save_to_file(FILE *fptr, data_arrays *da, uint64_t counter)
{
    for (uint64_t i = 0; i < counter; i++)
    {
        fprintf(fptr, "%li,%i\n", da[i].tsc, da[i].mode);
    }
}


uint64_t run_array_memory_lookup()
{
    uint64_t v = rand() % 512;
    for (int i = 0; i < ITERATIONS_ARRAY_LOOKUP; i++)
    {
        asm volatile("mfence");
        v = SBOX[v];
        asm volatile("mfence");
    }
    return v;
}

void asm_imul()
{
    int input_val = rand();
    int output_val = rand();

    for (int i = 0; i < ITERATIONS_IMUL; i++)
    {
        asm volatile(
            "imul %1, %0 \n"
            : "+r"(output_val)
            : "rm"(input_val)
        );
    }
}

void asm_sqrt()
{
    double a = rand();
    double b = 0;
    for(int i = 0; i < ITERATIONS_SQRT; i++)
    {
        asm volatile(
            "movq %1, %%xmm0 \n"
            "sqrtsd %%xmm0, %%xmm1 \n"
            "movq %%xmm1, %0 \n"
            : "=r"(b)
            : "g"(a)
            : "xmm0", "xmm1", "memory"
        );
    }
}

void read_sending_details( int number_array[50])
{
    FILE *myFile;
    myFile = fopen("covert_sender_details.txt", "r");


    if (myFile == NULL){
        printf("Error Reading File\n");
        exit (0);
    }

    int i=0;
    int value=0;

    do
    {
        fscanf(myFile, "%d,", &value );
        number_array[i]=value;
        i++;
    } while (value != -99);

    fclose(myFile);
}

int main(int argc, const char **argv)
{
    /* Intializes random number generator */
    time_t t;
    srand((unsigned) time(&t));

    if (argc != 2 && argc != 3)
    {
        myprintf("Usage: %s  sleep between measurements (uS] [print to
screen 1 or 0]\n", argv[0]);
        exit(-1);
    }

    uint64_t period_us;
    period_us = strtol(argv[1], NULL, 10);

    int print_to_screen = 0;
    if (argc == 3)
        print_to_screen = atoi(argv[2]);

    char *filename = "data_covert.csv";
    FILE *fptr = fopen(filename, "w");

    myprintf("Starting %s.\n", argv[0]);
    myprintf("Period (uS) %li.\n", period_us);
    uint64_t return_val_sbox=0;

    int mode;
    int sending_it[100]={};
    read_sending_details(sending_it);

    int* send_it = sending_it;
    while (*send_it != -99)
    {
        mode = *send_it;
        send_it++;

        // start the timers
        fprintf(fptr, "%ld,%i\n", rdtsc(), mode);
        if (print_to_screen)
        {
            printf("%ld,%i\n", rdtsc(), mode);
        }
        switch (mode)
        {
        case 10:
            asm_sqrt();
            break;
        case 9:
            asm_imul();
            break;
        case 8:
            return_val_sbox = run_array_memory_lookup();
            break;
        default:
            usleep(period_us);
            break;
        }
        // end the timers
        fprintf(fptr, "%ld,%i\n", rdtsc(), mode);
        if (print_to_screen)
        {
            printf("%ld,%i\n", rdtsc(), mode);
        }
    }
    myprintf("Finished %s.\n", argv[0]);
    fclose(fptr);
    return 0;
}
```

D.2 Reading MSR Code

```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>
#include <pthread.h>

#define myprintf(...) printf("throttling_msr_read> " __VA_ARGS__)

FILE* fptr;
char* filename = "data_msr_read.csv";

typedef struct data_management_t
{
    int loop_counter;
    uint64_t sleep_u_secs;
    uint64_t max_loop_iterations;
} data_management;

typedef struct data_arrays_t
{
    uint64_t tsc_dram;
    uint64_t dram_throttle_val;
    uint64_t dram_energy_val;
    uint64_t tsc_pkg;
    uint64_t pkg_throttle_val;
    uint64_t pkg_energy_val;
    uint64_t tsc_voltage;
    uint64_t voltage;
} data_arrays;
```

```c
data_management dm;
data_arrays* da;


uint64_t rdtsc()
{
    uint64_t a, d;
    asm volatile ("mfence");
    asm volatile ("rdtsc" : "=a" (a), "=d" (d));
    a = (d << 32) |  a ;
    asm volatile ("mfence");
    return a;
}

uint64_t read_msr(int fd, int msr)
{
    uint64_t val;
    pread(fd, &val, sizeof(val), msr);
    return (uint64_t)val;
}
/*** DRAM ***/
uint64_t read_accumulated_dram_energy(int fd)
{
    return read_msr(fd,0x619);
}
uint64_t read_accumulated_dram_throttled_time(int fd)
{
    return read_msr(fd,0x61b); // dram_perf_status
}

/*** pkg ***/
uint64_t read_accumulated_pkg_energy(int fd)
{
    return read_msr(fd,0x611); //pkg_energy_status
}

uint64_t read_accumulated_pkg_throttled_time(int fd)
{
    return read_msr(fd,0x613);  // pkg_perf_status
}

uint64_t read_voltage(int fd)
{
    return read_msr(fd,0x198);
}

int max(int num1, int num2)
{
    return (num1 > num2 ) ? num1 : num2;
}
```

```
int main(int argc, const char **argv)
{
    if ( argc != 3 )
    {
        myprintf("Usage: %s loop_count sleep_usecs\n", argv[0]);
        exit (-1);
    }
    // Open the msr
    int fd = open("/dev/cpu/0/msr", O_RDWR);
    if(fd == -1)
    {
        myprintf("\nCould not open /dev/cpu/0/msr. \nDid you forget to use
    'sudo' ?\n\n");
        exit (-1);
    }
    dm.max_loop_iterations=strtol(argv[1],NULL,10);
    dm.sleep_u_secs=strtol(argv[2],NULL,10);

    FILE *fptr;
    char* filename = "data_msr_read.csv";
    fptr = fopen(filename,"w");

    myprintf ("Starting %s.\n",argv[0]);
    myprintf ("Running with max_loop_iterations of: %li \n",dm.
   max_loop_iterations);
    data_arrays da = {};
    uint64_t i;

    for (i=0;i<dm.max_loop_iterations;i++)
    {
        da.tsc_dram=rdtsc();
        da.dram_energy_val    = read_accumulated_dram_energy(fd);
        da.dram_throttle_val = read_accumulated_dram_throttled_time(fd);
        da.tsc_pkg=rdtsc();
        da.pkg_energy_val     = read_accumulated_pkg_energy(fd);
        da.pkg_throttle_val  = read_accumulated_pkg_throttled_time(fd);
        da.tsc_voltage=rdtsc();
        da.voltage   = read_voltage(fd);

        fprintf (fptr,"%li,%li,%li,%li,%li,%li,%li,%li\n",
            da.tsc_dram,
            da.dram_throttle_val,
            da.dram_energy_val,
            da.tsc_pkg,
            da.pkg_throttle_val,
            da.pkg_energy_val,
            da.tsc_voltage,
            da.voltage
            );

        usleep (dm.sleep_u_secs);
    }
    myprintf ("Finished %s.\n",argv[0]);
    return 0;
}
```

References

[1] Eltayeb S. Abuelyaman and Balasubramanian Devadoss. "Differential Fault Analysis Automation". In: *International Conference on Internet Computing* (2005), pp. 1–18.

[2] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. "When Clocks Fail: On Critical Paths and Clock Faults". In: *Smart Card Research and Advanced Application*. Ed. by Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny. Springer Berlin Heidelberg, 2010, pp. 182–193.

[3] Murugappan Alagappan, Jeyavijayan Rajendran, Miloš Doroslovački, and Guru Venkataramani. "DFS covert channels on multi-core platforms". In: *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2017, pp. 1–6. DOI: 10.1109/VLSI-SoC.2017.8203469.

[4] Tiago Alves. "Trustzone: Integrated hardware and software security". In: *White paper* (2004).

[5] 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*. Vol. 13. ACM New York, NY, USA. 2013.

[6] James P Anderson. *Computer Security Technology Planning Study*. Tech. rep. Fort Washington PA, 1972.

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

[8] Anjali Arora and Saibal Kumar Pal. "A Survey of Cryptanalytic Attacks on Lightweight Block Ciphers". In: *IRACST -International Journal of Computer Science and Information Technology & Security* 2.2 (2012), pp. 2249–9555.

[9] Christian Aumueller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures". In: *CHES*. 2002.

[10] 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". In: *ACM SIGPLAN Notices* (2016).

[11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. "An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs". In: *Proceedings - 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011* (2011), pp. 105–114.

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

[13] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769.

[14] Thierno Barry, Damien Couroussé, and Bruno Robisson. "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks". In: *CS2 '16*. ACM, 2016, pp. 1–6. DOI: 10.1145/2858930.2858931.

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

[16] Markus Becker, Daniel Baldin, Christoph Kuznik, Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. "XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software". In: *EMSOFT '12*. New York, NY, USA: ACM, 2012, pp. 33–42. DOI: 10.1145/2380356.2380368.

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

[18] 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*. 1997, pp. 513–525.

[19] 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)*. Baltimore, MD: USENIX Association, 2018, pp. 1213–1227.

[20] Dan Boneh. "On the Importance of Eliminating Errors in Cryptographic Computations". In: *Journal of Cryptology* (2001). DOI: 10.1007/S001450010016.

[21] Dan Boneh, RA Demillo, and RJ Lipton. "On the Importance of Checking Computations". In: July 1997 (1996), pp. 1–11.

[22] Pradip Bose and Saibal Mukhopadhyay. "Energy-Secure System Architectures (ESSA): A Workshop Report". In: *IEEE Micro* 39.4 (2019), pp. 27–34. DOI: 10.1109/mm.2019.2921508.

[23] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector". In: *S&P*. 2016.

[24] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.2 (Feb. 2019), pp. 199–224.

[25] 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: SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 117–130.

[26] Jakub Breier, Xiaolu Hou, and Yang Liu. "Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (May 2018), pp. 96–122. DOI: 10.13154/tches.v2018.i2.96-122. URL: https://tches.iacr.org/index.php/TCHES/article/view/876.

[27] *Build an SGX encrypted computing environment*. URL: https://www.alibabacloud.com/help/en/elastic-compute-service/latest/build-an-sgx-encrypted-computing-environment.

[28] Edward Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William Lambert, Kaladhar Radhakrishnan, and Michael Hill. "FIVR — Fully integrated voltage regulators on 4th generation Intel Core SoCs". In: *IEEE APEC '14*. Mar. 2014, pp. 432–439. DOI: 10.1109/APEC.2014.6803344.

[29] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. "Glitch It If You Can: Parameter Search Strategies for Successful

Fault Injection". In: *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. 2013, pp. 236–252.

[30] 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)*. IEEE. 2019, pp. 142–157.

[31] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races". In: *2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2018, pp. 178–194.

[32] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface". In: *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021.

[33] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. *Real time detection of cache-based side-channel attacks using Hardware Performance Counters*. ePrint 2015/1034. 2015.

[34] Sławomir Chyłek and Marcin Goliszewski. "QEMU-based fault injection framework". In: *Studia Informatica* 33.4 (2012), pp. 25–42.

[35] Maxime Colmant, Pascal Felber, Romain Rouvoy, and Lionel Seinturier. "WattsKit: Software-Defined Power Monitoring of Distributed Systems". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 514–523. DOI: 10.1109/CCGRID.2017.27.

[36] Jonathan Corbet. *Defending against Rowhammer in the Kernel.* Nov. 2016. URL: https://lwn.net/Articles/704920/. [Accessed: 1st August 2019].

[37] Intel Corporation. "Intel® 64 and IA-32 Architectures, System Programming Guide". In: *System* 3B (2016).

[38] Intel Corporation. "Volume 2". In: *System* 3.253665 (2011). DOI: 10.1109/MAHC.2010.22.

[39] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: (2016).

[40] Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. "Adjusting Laser Injections for Fully Controlled Faults". In: *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers.* Ed. by Emmanuel Prouff. Vol. 8622. Lecture Notes in Computer Science. Springer, 2014, pp. 229–242. DOI: 10.1007/978-3-319-10175-0_16.

[41] Ang Cui and Rick Housley. "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection". In: *Usenix WOOT '17.* Vancouver, BC: USENIX Association, Aug. 2017.

[42] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES". In: *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography.* 2012, pp. 7–15. DOI: 10.1109/FDTC.2012.15.

[43] Dell. *Support for XPS 15 9560.* URL: https://www.dell.com/support/home/uk/en/ukbsdt1/product-support/servicetag/djf1ph2/drivers. [Accessed: 4th July 2020].

[44] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. "Statistical Fault Attacks on Nonce-Based Authenticated Encryption

Schemes". In: *ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer, 2016, pp. 369–395.

[45] Christopher Domas. "Breaking the x86 ISA". In: *Black Hat* (2017).

[46] Jack Doweck, Wen Fu Kao, Allen Kuan Yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake". In: *IEEE Micro* (2017). DOI: 10.1109/MM.2017.38.

[47] João A. Durães and Henrique S. Madeira. "Emulation of software faults: A field data study and a practical approach". In: *IEEE Transactions on Software Engineering* (2006). DOI: 10.1109/TSE.2006.113.

[48] Enarx. *Threat model*. URL: https://github.com/enarx/enarx/wiki/Threat-Model. [Accessed: 17 June 2020, revision 678e2c2].

[49] Common Weakness Enumeration. *2022 CWE Top 25 Most Dangerous Software Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. [Accessed: 1st November 2022].

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

[51] Junfeng Fan and Ingrid Verbauwhede. "An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost". In: *Cryptography and Security: From Theory to Applications: Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*. Ed. by David Naccache. Berlin, Heidelberg: Springer, 2012, pp. 265–282. DOI: 10.1007/978-3-642-28368-0_18.

[52] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. “Security vulnerabilities of SGX and countermeasures: A survey”. In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–36.

[53] Davide Ferraretto and Graziano Pravadelli. “Efficient fault injection in QEMU”. In: *2015 16th Latin-American Test Symposium (LATS)*. 2015, pp. 1–6. DOI: 10.1109/LATW.2015.7102401.

[54] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *Usenix WOOT*. Aug. 2020.

[55] Fortanix. *Intel SGX FAQ*. URL: https://fortanix.com/intel-sgx/. [Accessed: 2 June 2020].

[56] Fortanix. *intel-oc-mbox*. URL: https://github.com/fortanix/intel-oc-mbox/tree/jb/initial. [Accessed: 21 September 2020].

[57] Apostolos P. Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. “Exploiting hardware vulnerabilities to attack embedded system devices: A survey of potent microarchitectural attacks”. In: *Electronics (Switzerland)* 6.3 (2017). DOI: 10.3390/electronics6030052.

[58] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “TRRespass: Exploiting the Many Sides of Target Row Refresh”. In: *S&P*. May 2020. DOI: 10.48550/ARXIV.2004.01807.

[59] 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*. Aug. 2013, pp. 108–118. DOI: 10.1109/FDTC.2013.18.

[60] Matteo Fusi. “Information-leakage analysis based on hardware performance counters”. MA thesis. 2017.

[61] Ruchi Gaba, Er Shobhit, Gupta Pg, and India Kurukshetra. *Fault Injection and Its Techniques*. Tech. rep. 4. 2014, p. 2277. URL: www.ijarcsse.com.

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

[63] Github. *intel-undervolt issue 43*. URL: https://github.com/kitsunyan/intel-undervolt/issues/43%5C#issuecomment-619373836. [Accessed: 18 June 2020].

[64] Github. *Plundervolt*. URL: https://github.com/KitMurdock/plundervolt. [Accessed: 18 June 2020].

[65] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. "The State of Fault Injection Vulnerability Detection". In: *VECoS '18*. Ed. by Mohamed Faouzi Atig, Saddek Bensalem, Simon Bliudze, and Bruno Monsuez. Springer, 2018, pp. 3–21.

[66] S. Govindavajhala and A.W. Appel. "Using memory errors to attack a virtual machine". In: *2003 Symposium on Security and Privacy (S&P)*. 2003, pp. 154–165.

[67] Daniel Gruss. "Software-based Microarchitectural Attacks". PhD thesis. Graz University of Technology, June 2017.

[68] 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: *Proceedings - IEEE Symposium on Security and Privacy (S&P)*. 2018.

[69] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A remote software-induced fault attack in JavaScript". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes*

in Bioinformatics). Vol. 9721. 2016. DOI: 10.1007/978-3-319-40667-1_15. eprint: 1507.06955.

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

[71] Daniel Gruss and Kit Murdock. "Plundervolt: Flipping Bits from Software without Rowhammer". In: Chaos Communication Congress (CCC). URL: https://media.ccc.de/v/36c3-10883-plundervolt_flipping_bits_from_software_without_rowhammer. Presented on: 27th December 2019.

[72] Daniel Gruss, David Oswald, and Kit Murdock. "Plundervolt: Flipping Bits from Software without Rowhammer". In: Blackhat. URL: https://www.youtube.com/watch?v=zKIliM-pHFs. Presented online on: 5th August 2020.

[73] Amina Guermouche and Anne-Cécile Orgerie. *Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints*. Research Report 2. Télécom Sud Paris, 2019.

[74] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. ePrint 2016/204. 2016.

[75] 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.

[76] J. Haj-Yahya, L. Orosa, J. S. Kim, J. Gomez Luna, A. Yaglikci, M. Alser, I. Puddu, and O. Mutlu. "IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2021, pp. 985–998. DOI: 10.1109/ISCA52012.2021.00081.

[77] Jawad Haj-Yihia, Yosi Ben Asher, Efraim Rotem, Ahmad Yasin, and Ran Ginosar. "Compiler-Directed Power Management for Superscalars". In: *ACM Trans. Archit. Code Optim.* 11.4 (Jan. 2015). DOI: 10.1145/2685393.

[78] 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*. Vol. 94. 2006. DOI: 10.1109/JPROC.2005.862424.

[79] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. "ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults". In: *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. 2021, pp. 20–30. DOI: 10.1109/FDTC53659.2021.00013.

[80] 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.

[81] Mikael Hirki, Zhonghong Ou, Kashif Nizam Khan, Jukka K. Nurminen, and Tapio Niemi. "Empirical Study of the Power Consumption of the x86-64 Instruction Decoder". In: *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)*. Santa Clara, CA: USENIX Association, Mar. 2016.

[82] Peter H. Hochschild, Paul Jack Turner, Jeffrey C. Mogul, Rama Krishna Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. "Cores that don't count". In: *Proceedings 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*. 2021.

[83] Matthew E. Hoekstra. *Intel® SGX for Dummies, Part 3.*

[84] Max Hoffmann, Falk Schellenberg, and Christof Paar. "ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries". In: *IEEE Transactions on Information Forensics and Security* 16 (2021).

[85] Petr Hosek and Cristian Cadar. "Varan the unbelievable: An efficient n-version execution framework". In: *ACM SIGPLAN Notices*. Vol. 50. 4. ACM. 2015, pp. 339–353.

[86] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. "Fully Automated Differential Fault Analysis on Software Implementations of Block Ciphers". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.3 (May 2019), pp. 1–29. DOI: 10.13154/tches.v2019.i3.1-29.

[87] Michael Hutter and Jörn-Marc Schmidt. "The temperature side channel and heating fault attacks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2014. DOI: 10.1007/978-3-319-08302-5_15.

[88] Aa Hwang, Ia Stefanovici, and Bianca Schroeder. "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design". In: *ACM SIGARCH Computer* 40.1 (2012), pp. 111–122. DOI: 10.1145/2189750.2150989.

[89] Infineon. *Integrity Guard*. 2018. URL: 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. [Accessed: 5th April 2020].

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

[91] 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.

[92] Intel. *Developer Reference for Intel Integrated Performance Primitives Cryptography – Example of Using RSA Primitive Functions*. 2019. URL: https://software.intel.com/en-us/ipp-crypto-reference-2019-example-of-using-rsa-primitive-functions. [Accessed: 29 July 2019].

[93] Intel. *Get Started with the SDK*. 2019. URL: https://software.intel.com/en-us/sgx/sdk. [Accessed: 10 May 2019].

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

[95] Intel. *Intel SGX Technical Details for INTEL-SA-00289 and INTEL-SA-00334*. URL: https://cdrdv2.intel.com/v1/dl/getContent/619320. [Accessed: 5 June 2020].

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

[97] Intel. *L1 Terminal Fault SA-00161*. Aug. 2018. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. [Accessed: 28 October 2022].

[98] Intel. *Voltage Regulator-Down 11.1: Processor Power Delivery Design Guide*. 2009. URL: https://www.intel.com/content/www/us/en/power-management/voltage-regulator-down-11-1-processor-power-delivery-guidelines.html. [Accessed: 29th July 2019].

[99] *Introducing Rainbow: Donjon's side-channel analysis simulation tool*. 2019. URL: https://github.com/Ledger-Donjon/rainbow. [Accessed: 20th February 2022].

[100] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *MASCAT: Stopping Microarchitectural Attacks Before Execution*. ePrint 2016/1196. 2016.

[101] Thomas Jakobs and Gudula Rünger. “On the Energy Consumption of Load/Store AVX Instructions”. In: *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2018, pp. 319–327.

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

[103] Simon Johnson. *An update on 3rd Party Attestation*. Dec. 2018. URL: https://software.intel.com/en-us/blogs/2018/12/09/an-update-on-3rd-party-attestation. [Accessed: 30 July 2020].

[104] M. Joye, P. Manet, and J. B. Rigaud. “Strengthening hardware AES implementations against fault attacks”. In: *IET Information Security* 1.3 (2007), pp. 106–110. DOI: 10.1049/iet-ifs:20060163.

[105] Marc Joye and Michael Tunstall, eds. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012. DOI: 10.1007/978-3-642-29656-7.

[106] David Kaplan, Jeremy Powell, and Tom Woller. “AMD memory encryption”. In: *White paper* (2016).

[107] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. “Hardware Designer’s Guide to Fault Attacks”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2295–2306.

[108] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. “MAGIC: Malicious aging in circuits/cores”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.1 (2015).

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

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

[111] 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.

[112] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. DOI: 10.1145/360248.360252.

[113] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*. ACM, 2009, pp. 207–220.

[114] 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: *2019 IEEE Symposium on Security and Privacy (S&P)*. 2019.

[115] Andreas Kogler, Daniel Gruss, and Michael Schwarz. “Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks”. In: *USENIX Security Symposium*. 2022.

[116] Kokke. *Tiny AES in c*. 2017. URL: https://github.com/kokke/tiny-AES-c. [Accessed: 2022-02-24].

[117] Oliver Kömmerling and Markus G. Kuhn. “Design Principles for Tamper-Resistant Smartcard Processors”. In: *Smartcard ’99*. 1999.

[118] 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)*. IEEE. 2016, pp. 431–442.

[119] Maha Kooli and Giorgio Di Natale. “A survey on simulation-based fault injection tools for complex systems”. In: *Proceedings - 2014 9th IEEE International Conference on Design and Technology of Integrated Systems in Nanoscale Era, DTIS 2014*. 2014.

[120] Thomas Korak and Michael Hoefler. “On the effects of clock and power supply tampering on two microcontroller platforms”. In: *Proceedings - 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014* (2014), pp. 8–17. DOI: 10.1109/FDTC.2014.11.

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

[122] 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.

[123] Butler W Lampson. “A note on the confinement problem”. In: *Communications of the ACM* 16.10 (1973), pp. 613–615.

[124] Michael Le, Andrew Gallagher, and Yuval Tamir. “Challenges and Opportunities with Fault Injection in Virtualized Systems”. In: *First International Workshop on Virtualization Performance: Analysis, Characterization, and Tools* (2008).

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

[126] 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.

[127] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. “PLATYPUS: Software-based Power Side-Channel Attacks on x86”. In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2021.

[128] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)* (2018). eprint: 1801.01207.

[129] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. *Frequency Throttling Side-Channel Attack*. 2022. DOI: 10.48550/ARXIV.2206.07012.

[130] Chen Liu, Monodeep Kar, Xueyang Wang, Nikhil Chawla, Neer Roggel, Bilgiday Yuce, and Jason M. Fung. “Methodology of Assessing Information Leakage through Software-Accessible Telemetries”. In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2021, pp. 259–269. DOI: 10.1109/HOST49136.2021.9702271.

[131] Yifan Lu. “Attacking hardware AES with DFA”. In: *arXiv preprint arXiv:1902.08693* (2019).

[132] Yifan Lu. “Injecting Software Vulnerabilities with Voltage Glitching”. In: *CoRR* abs/1903.08102 (2019). DOI: 10.48550/ARXIV.1903.08102.

[133] Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter, and Koen De Bosschere. “Mitigating Smart Card Fault Injection with Link-Time Code Rewriting: A Feasibility Study”. In: *Financial Cryptography*. 2013.

[134] Manjaro. *Undervolt intel CPU.* URL: https://wiki.manjaro.org/index.php?title=Undervolt_intel_CPU. [Accessed: 30th September 2019].

[135] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. “Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals”. In: *NDSS ’19.* San Diego, CA, USA, 2019.

[136] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, Srdjan Čapkun, and E T H Zürich. “Thermal Covert Channels on Multi-core Platforms This paper is included in the Proceedings of the”. In: *24th USENIX Security Symposium (USENIX Security 15)* (2015).

[137] Norm Matloff. “Below C Level: An Introduction to Computer Systems”. In: (May 2012). URL: https://www.cs.ucdavis.edu/~matloff/matloff/public_html/50/PLN/CompSystsBookS2011.pdf. [Accessed: 20th September 2021].

[138] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. “Evaluation of CPU frequency transition latency”. In: *Computer Science - Research and Development* 29.3-4 (2014), pp. 187–195. DOI: 10.1007/s00450-013-0240-x.

[139] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. “Flicker: An Execution Infrastructure for Tcb Minimization”. In: *SIGOPS Oper. Syst. Rev.* 42.4 (Apr. 2008), pp. 315–328. DOI: 10.1145/1357010.1352625.

[140] 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.* 2015.

[141] Milosch Meriac. *Heart of Darkness - exploring the uncharted backwaters of HID iCLASS security.* Tech. rep. Bitmanufaktur GmbH, 2010.

[142] Microsoft. *Azure confidential computing.* Retrieved from archive.org. URL: https://web.archive.org/web/20191206233429/https://azure.microsoft.com/en-gb/solutions/confidential-compute/. [Accessed: 6th December 2019].

[143] Microsoft. *Open Enclave SDK.* URL: https://github.com/openenclave/openenclave. [Accessed: 23 September 2020].

[144] Miha Eleršič. *Guide to Linux undervolting for Haswell and never Intel CPUs.* 2019. URL: https://github.com/mihic/linux-intel-undervolt. [Accessed: 29 July 2019].

[145] Charlie Miller. "Battery Firmware Hacking". In: *Black Hat USA* (2011), pp. 3–4.

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

[147] Kit Murdock. "Plundervolt: Pillaging and plundering SGX with Software-based Fault Injection Attacks". In: Redhat Research Day. URL: https://www.youtube.com/watch?v=Czfs8i2EA_0. Presented on: 24rd January 2020.

[148] Kit Murdock. "Plundervolt: Software-Based Fault Injection Attacks against Intel SGX". In: S&P. URL: https://www.youtube.com/watch?v=5Mr1FCZ7VBQ. Presented online on: 19th March 2020.

[149] 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)*. 2020.

[150] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Frank Piessens, and Daniel Gruss. "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble". In: *IEEE Security and Privacy* 18.5 (Sept. 2020), pp. 28–37. ISSN: 1540-7993. DOI: 10.1109/MSEC.2020.2990495.

[151] Onur Mutlu. “The RowHammer problem and other issues we may face as memory becomes denser”. In: *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*. 2017. DOI: 10.23919/DATE.2017.7927156. arXiv: 1703.00626.

[152] Anh Quynh Nguyen. *Capstone Disassembler*. URL: www.capstone-engine.org. [Accessed: 20 April 2022].

[153] Anh Quynh Nguyen and Hoang Dang. *Unicorn Engine*. URL: https://github.com/unicorn-engine/unicorn. [Accessed: 24 February 2022].

[154] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A Survey of Published Attacks on Intel SGX”. In: *arXiv preprint arXiv:2006.13598* (2020). DOI: 10.48550/ARXIV.2006.13598.

[155] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. “Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 479–498.

[156] NotebookReview. *The ThrottleStop Guide*. 2019. URL: http://forum.notebookreview.com/threads/the-throttlestop-guide. [Accessed: 29th July 2019].

[157] Colin O’Flynn and Zhizhang (David) Chen. “ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research”. In: *COSADE 2014*. 2014, pp. 243–260. DOI: 10.48550/ARXIV.2006.13598.

[158] O’Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. *Spectre attack against SGX enclave*. Jan. 2018.

[159] Johannes Obermaier and Stefan Tatschner. “Shedding too much Light on a Microcontroller’s Firmware Protection”. In: *Usenix WOOT ’17*. 2017.

[160] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. “Error detection by duplicated instructions in super-scalar processors”. In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75.

[161] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin, Kapil Vaswani, Manuel Costa, and Aastha Mehta. “SGX-Enabled Oblivious Machine Learning”. In: *Usec* (2016).

[162] Thales Paiva, Javier Navaridas, and Routo Terada. “Robust Covert Channels Based on DRAM Power Consumption”. In: Sept. 2019, pp. 319–338. DOI: 10.1007/978-3-030-30215-3_16.

[163] Matthias Payer. “HexPADS: a platform to detect “stealth” attacks”. In: *ESSoS*. 2016.

[164] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.

[165] Stjepan Picek, Lejla Batina, Domagoj Jakobovic, and Rafael Boix Carpi. “Evolving Genetic Algorithms for Fault Injection Attacks”. In: *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*. 2014, pp. 1106–1111. DOI: 10.1109/MIPRO.2014.6859734.

[166] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. “Fault attacks, injection techniques and tools for simulation”. In: *Hardware Security and Trust: Design and Deployment of Integrated Circuits in a Threatened Environment*. 2017, pp. 27–47. DOI: 10.1007/978-3-319-44318-8_2.

[167] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. “Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections”. In: *ICST ’14*. 2014, pp. 213–222. DOI: 10.1109/ICST.2014.34.

[168] 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”. In: *arXiv preprint arXiv:1908.11143* (2019).

[169] Rui Qiao and Mark Seaborn. “A new approach for rowhammer attacks”. In: *Proceedings of the 2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016*. 2016. DOI: 10.1109/HST.2016.7495576.

[170] 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. London, United Kingdom: ACM, 2019, pp. 195–209. DOI: 10.1145/3319535.3354201.

[171] André Quadt. “Smart Watts”. In: *E-Energy*. Ed. by Arnold Picot and Karl-Heinz Neumann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 85–93.

[172] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. “Flip Feng Shui: Hammering a Needle in the Software Stack”. In: *USENIX Security ’16*. Austin: USENIX Association, Aug. 2016, pp. 1–18.

[173] 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*. IEEE Computer Society. 2005, pp. 243–254.

[174] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. “FIVER – Robust Verification of Countermeasures against Fault Injections”. In: *TCHES* 2021.4 (Aug. 2021), pp. 447–473. DOI: 10.46586/tches.v2021.i4.447-473.

[175] RightMark Gathering. *Throttlestop, RMClock Utility*. URL: http://cpu.rightmark.org/products/rmclock.shtml. [Accessed: 29 July 2019].

[176] *Riscure*. URL: https://www.riscure.com/. [Accessed: 01 November 2022].

[177] Riscure. *FiSim*. URL: https://github.com/Riscure/FiSim. [Accessed: 24 February 2022].

[178] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. "Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells". In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 89–98. DOI: 10.1109/FDTC.2013.17.

[179] J. M. Rushby. "Design and Verification of Secure Systems". In: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. SOSP '81. Pacific Grove, California, USA: Association for Computing Machinery, 1981, pp. 12–21. DOI: 10.1145/800216.806586.

[180] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. "A Diagonal Fault Attack on the Advanced Encryption Standard". In: *IACR Cryptology ePrint Archive* 2009 (Jan. 2009), p. 581.

[181] 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: *2019 IEEE Symposium on Security and Privacy (S&P)*. 2019, pp. 88–105.

[182] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. "Intrinsic Rowhammer PUFs: Leveraging the Rowhammer effect for improved security". In: *Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust, (HOST)*. 2017.

[183] Horst Benjamin Schirmeier. “Efficient fault-injection-based assessment of software-implemented hardware fault tolerance”. PhD thesis. Dortmund University of Technology, 2016.

[184] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *CCS*. 2019.

[185] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *CoRR* abs/1702.08719 (2017).

[186] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: *Black Hat Briefings*. Mar. 2015.

[187] 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.

[188] Kristoffer Myrseth Severinsen. “Secure Programming with Intel SGX and Novel Applications”. MA thesis. 2017. URL: https://www.duo.uio.no/handle/10852/60352.

[189] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. New York, NY, USA: ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[190] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2016, pp. 138–157.

[191] Daniel P Siewiorek and Robert S Swarz. *The theory and practice of reliable system design*. Digital press, 1982.

[192] Sergei Skorobogatov. “Fault attacks on secure chips”. In: *Design and Security of Cryptographic Algorithms and Devices* (2011).

[193] Sergei P. Skorobogatov. *Copy Protection in Modern Microcontrollers*. URL: https://www.cl.cam.ac.uk/~sps32/mcu_lock.html. [Accessed: 29 July 2019].

[194] Sergei P. Skorobogatov and Ross J. Anderson. “Optical Fault Induction Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. 2002, pp. 2–12.

[195] statista. *Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025*. URL: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide. [Accessed: 24th February 2022].

[196] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. “Microcontrollers as (In)Security Devices for Pervasive Computing Applications”. In: *Proceedings of the IEEE* 102.8 (2014), pp. 1157–1173. DOI: 10.1109/JPROC.2014.2325397.

[197] Adrian Tang. “Security Engineering of Hardware-Software Interface”. PhD thesis. Columbia University, 2018.

[198] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *USENIX Security ’17*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1057–1074.

[199] “The art of fault injection”. In: *Control Engineering and Applied Informatics* 13.4 (2011), pp. 9–18.

[200] tibersam. *ARCHIE AES Example*. 2021. URL: https://github.com/tibersam/archie-aes-example. [Accessed: 24 February 2022].

[201] 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)*. IEEE. 2017, pp. 19–34.

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

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

[204] 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.

[205] 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)*. 2020.

[206] 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, Nov. 2019.

[207] 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, Oct. 2018.

[208] 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*. ACM. 2017, p. 4. URL: https://github.com/jovanbulck/sgx-step.

[209] 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, Aug. 2017.

[210] Jan Van den Herrewegen, David Oswald, Flavio D. Garcia, and Qais Temeiza. “Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (Dec. 2020), pp. 56–81.

[211] 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.

[212] 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)*. 2016, pp. 167–179.

[213] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. “Examining ACE analysis reliability estimates using fault-injection”. In: *Proceedings - International Symposium on Computer Architecture*. 2007.

[214] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: *Proceedings of the USENIX Security Symposium (USENIX)*. 2022.

[215] Zhenghong Wang and Ruby B. Lee. "Covert and Side Channels Due to Processor Architecture". In: *22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006, pp. 473–482. DOI: 10.1109/ACSAC.2006.20.

[216] Zhenyu Wu, Mengjun Xie, and Haining Wang. "Energy Attack on Server Systems". In: *USENIX Workshop on Offensive Technologies (WOOT)* (2011).

[217] 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.

[218] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *ACM CCS '17*. ACM, 2017, pp. 2313–2328. DOI: 10.1145/3133956.3134046.

[219] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation". In: *Hardware and Systems Security* 2.2 (2018), pp. 111–130. DOI: 10.1007/s41635-018-0038-1.

[220] Fan Zhang, Shize Guo, Xinjie Zhao, Tao Wang, Jian Yang, Francois Xavier Standaert, and Dawu Gu. "A Framework for the Analysis and Evaluation of Algebraic Fault Attacks on Lightweight Block Ciphers". In: *IEEE Transactions on Information Forensics and Security* (2016). DOI: 10.1109/TIFS.2016.2516905.

[221] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds". In: *RAID*. 2016.

[222] Siqi Zhao and Xuhua Ding. "FIMCE: A Fully Isolated Micro-Computing Environment for Multicore Systems". In: *ACM Trans. Priv. Secur.* 21.3 (May 2018). DOI: 10.1145/3195181.

[223] Igor Zhirkov. *Low-level programming: C, assembly, and program execution on Intel® 64 architecture*. 2017. DOI: 10.1007/978-1-4842-2403-8.

[224] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. "A survey on fault injection techniques". In: *The International Arab Journal of Information Technology* 1.2 (2004), pp. 171–186.

[225] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. "Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 110–115.