Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks

Andreas Kogler and Daniel Gruss, *Graz University of Technology;*
Michael Schwarz, *CISPA Helmholtz Center for Information Security*

https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-minefield

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks

Andreas Kogler
Graz University of Technology

Daniel Gruss
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern CPUs adapt clock frequencies and voltage levels to workloads to reduce energy consumption and heat dissipation. This mechanism, dynamic voltage and frequency scaling (DVFS), is controlled from privileged software but affects all execution modes, including SGX. Prior work showed that manipulating voltage or frequency can fault instructions and thereby subvert SGX enclaves. Consequently, Intel disabled the overclocking mailbox (OCM) required for software undervolting, also preventing benign use for energy saving.

In this paper, we propose Minefield, the first software-level defense against DVFS attacks. The idea of Minefield is not to prevent DVFS faults but to deflect faults to trap instructions and handle them before they lead to harmful behavior. As groundwork for Minefield, we systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over at least 57 000 cycles. Minefield places highly fault-susceptible trap instructions in the victim code during compilation. Like redundancy countermeasures, Minefield is scalable and enables enclave developers to choose a security parameter between 0 % and almost 100 %, yielding a fine-grained security-performance trade-off. Our evaluation shows a density of 0.75, *i.e.*, one trap after every 1-2 instruction, mitigates all known DVFS attacks in 99 % on Intel SGX, incurring an overhead of 148.4 % on protected enclaves. However, Minefield has no performance effect on the remaining system. Thus, Minefield is a better solution than hardware- or microcode-based patches disabling the OCM interface.

1 Introduction

With a variety of use cases for modern computers, CPUs have an increasing number of features, also for security and performance. One feature available on Intel, AMD and ARM CPUs are trusted execution environments (TEEs). TEEs enable running code in a secure environment isolated from the rest of the system with the security goal of protecting code and data even from a compromised operating system or hypervisor.

To accommodate today's performance and efficiency goals, modern CPUs operate at various clock frequencies and voltage levels to adapt to the current workload. When the workload is low or energy must be saved due to thermal or battery constraints, the voltage level and the clock frequency are lowered. This mechanism, DVFS, is available on ARM, Intel, and AMD CPUs and can be controlled from privileged software. However, the modified voltage and frequency affect all security domains on the CPU.

Previous work [15, 32, 38, 40, 41, 48] has shown that an attacker can manipulate voltage and frequency using DVFS to inject faults into victim computations. The typical target of these attacks are TEEs since DVFS requires root privileges [32,38,40,41,48] or physical access [15], both of which are allowed in TEE threat models. The attacker uses DVFS to bring one or more CPU cores into a state where faults can occur with a very low probability. Thus, the CPU mostly still allows regular operation, *i.e.*, it does not cause a system crash. In this state, certain operations are more likely to experience a fault. Previous work has identified several of these operations, e.g., multiplication operations, pointer arithmetics, and AES-NI instructions. However, most instructions have not yet been analyzed for their fault probability.

In response to the DVFS attacks on Intel CPUs, Intel issued a CVE (CVE-2019-11157) and modified the SGX remote attestation process to verify that overclocking mailbox (OCM) and its model-specific registers (MSRs) allowing software-undervolting are disabled via a microcode update. The voltage regulators responsible for the core's voltage are connected to a bus receiving commands from CPU components, e.g., the OCM. In contrast, VoltPillager [15] directly sends these commands over the bus bypassing the CPU and the OCM. Therefore, disabling the OCM still leaves Intel CPUs without a fully-integrated voltage regulator (FIVR) design [11] vulnerable to VoltPillager [15] style attacks. Disabling the OCM led to complaints [17, 22] as the OCM is used to reduce overheating problems, thus increasing system performance and stability by undervolting the CPU. Some online guides even explicitly recommend reverting BIOS updates to get back the undervolting feature, breaking the ability to run SGX enclaves securely on these machines [17]. Disabling undervolting only if SGX is enabled impacts the entire system performance and stability if enclaves are used and is thus also not a desirable tradeoff. Selectively disabling undervolting while an enclave is active requires complex microcode changes, as the CPU has to ensure stable voltage levels on any reentry of the enclave. While it is unclear whether this is even possible with a microcode update, we also expect a high impact on the performance of enclaves.

In this paper, we propose the first software-level defense that probabilistically protects secure enclaves against all known DVFS attacks. As an empirical foundation for our defense, we systematically analyze DVFS attacks and categorize them based on the type of fault and its properties (e.g., spatial granularity, temporal granularity, and reproducibility). As part of our empirical analysis, we develop a framework to scan the x86 instruction set for DVFS fault susceptibility. Our analysis of the instructions with the highest fault probability confirms the implicit assumption from previous works that multiplications are most susceptible to faults. Hence, we rely on this instruction for our fault-deflection mechanism. We also analyze the temporal constraints and observe a timing gap of at least 57.8 µs between transitions from one voltage level to another. With the resulting weak control over the temporal fault location, state-of-the-art attacks have to repeat the victim operation millions of times [32, 38, 40, 41]. Currently, an attacker cannot precisely predict when and where during these operations the fault occurs.

Our defense, Minefield is a pure software-level defense implemented as a compiler extension. The basic idea of Minefield is not to prevent DVFS faults but to deflect them into trap instructions that are placed in the victim code during compilation, so that they cannot be weaponized anymore. The number of trap instructions scales as a security parameter from 0 % to almost amount for 100 % of the code base, yielding a fine-grained security-performance trade-off. Note that a security parameter of 100 % would refer to a program that consists only of trap instructions and no other instructions. Our evaluation shows that a trap density of 0.5, *i.e.*, one trap after every second instruction, mitigates the known DVFS attacks on Intel CPUs, namely Plundervolt [38], V0ltpwn [32], Voltjockey [41], and VoltPillager [15]. More specifically, in an attack on *mbedTLS* RSA-4096, a trap density of 2 mitigates more than 99 % of all attack attempts. Thus, Minefield is a viable defense against DVFS attacks on Intel SGX enclaves.

We carefully evaluate the performance impact on SGX enclaves with different Minefield security levels. Both runtime and memory overhead for the enclave scale up with the chosen security level. For a trap density of 0.75, which mitigates the known DVFS attacks in more than 99 % of the cases, Minefield incurs an overhead of 148.4 % on protected SGX enclaves on average. However, in some configurations Minefield even outperforms RSA redundancy protections, and the performance of normal-world applications remains entirely unaffected. Thus, Minefield is a better-suited mitigation against DVFS attacks on SGX enclaves than hardware- or microcode-based patches that disable the OCM entirely and also considers hardware-based undervolting attacks like VoltPillager [15].

While our evaluation focuses on Intel CPUs, we argue that the approach is applicable to ARM and AMD CPUs. Hence, Minefield can also be extended to prevent DVFS attacks on ARM TrustZone [48] and AMD SEV.

Contributions. The contributions of this work are:

1. We present a novel framework to systematically analyze the effects of DVFS faults on the entire x86 instruction set.
2. We propose a compiler extension Minefield, the first software-level defense against all known DVFS attacks.
3. We analyze different security levels and show that known DVFS attacks can be mitigated in 99 % of cases.
4. We evaluate the performance overheads of Minefield and show that the runtime overhead for SGX enclaves is below 150 % while mitigating 99 % of attacks.

Outline. Section 2 provides background, and Section 3 our threat model. Section 4 presents the high-level overview and poses research questions for Section 5. Section 6 details our implementation. Section 7 evaluates the security and performance. Section 8 discusses limitations. Section 9 concludes.

2 Background

In this section, we provide background on Intel SGX and DVFS, the mechanism behind the attacks we mitigate.

2.1 Intel SGX

Intel Software Guard Extension (SGX) [16, 27] is a trusted execution environment (TEE). To protect code and data on an untrusted system, an application is split into an untrusted and a trusted part, which is executed within a so-called SGX enclave. The enclave's execution state and memory cannot be accessed from other processes or the operating system. In the SGX threat model, only the CPU is trusted. Enclave memory is encrypted and integrity-protected in DRAM in a dedicated region called Enclave Page Cache (EPC), mitigating certain software-level and physical attacks. Thus, SGX even protects enclaves on systems compromised in software or hardware.

While these protections apply for the enclave's execution state (e.g., register values) and memory contents, scheduling, and page-table management are still performed by the untrusted operating system. Memory-safety violations [35], race conditions [50], or side channels [7, 45] can still lead to exploitation. Controlled-channel attacks [51], abuse, e.g., page-table entries or the APIC timer interrupt to precisely control the execution flow of a victim application [10, 43, 51]. Transient-execution attacks, e.g., Foreshadow [9], ZombieLoad [44], can precisely leak information from enclaves.

2.2 Power Management (DVFS)

Modern CPUs in smartphones, laptops, and servers, have different energy requirements. Especially mobile devices require constant energy balancing. Operating systems try to maximize the battery runtime while still providing sufficient computing power to handle the user's tasks. For dynamic adaption to the user's needs, modern CPUs implement Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows changing the voltage and frequency from privileged software via model-specific registers (MSR) [27]. However, the overclocking mailbox (OCM) interface allows to change the alignment between voltage and frequency, e.g., reduce the operating voltage at a specific frequency.

Undervolting and overclocking have become important to personal computer owners, especially for gaming computers (overclocking) and laptops (undervolting). While system stability has always been a concern in these communities, only recently researchers discovered that these interfaces can be abused for attacks. The first DVFS-based fault attack [48] overclocked an ARM CPU, leading to fault injection in the TrustZone trusted execution environment. More recently, several works have explored undervolting as a means to inject faults into the Intel SGX trusted execution environment [15, 32, 38, 41]. These works have in common that they modify the operating voltage during the execution of critical instructions leading to a computational error propagating into the result of these instructions. These faulty results lead to incorrect behavior inside a (correct and bug-free) program. These results then lead to exposure of secret data from enclaves, e.g., by faulting index calculations of array accesses. Faulty results can also occur within cryptographic primitives, e.g., enabling differential cryptoanalysis on AES-NI.

The main difference between previous works is the way the operating voltage is changed. VoltJockey [41], V0ltpwn [32], and Plundervolt [38] assume the SGX threat model and use privileged access to the OCM. These attacks can be mounted purely from software and only require access to the OCM MSR. VoltPillager [15], on the other hand, uses additional hardware to send messages directly to the voltage regulator unit on the mainboard. Hence, currently, there is no software mitigation against it, leaving SGX enclaves unprotected.

3 Attacker Model

In this section, we provide the attacker model for Minefield. We base our attacker model on the previously published attacks [15, 32, 38, 40, 41, 48] and our own experiments.

Attacker Privileges. Our mitigation, Minefield, works under the widely adopted SGX threat model and assumes a privileged attacker who controls the operating system and the BIOS. As for the hardware, the attacker has direct physical access to the CPU and the mainboard, enabling the attacker to mount DVFS attacks [15, 32, 38, 40, 41, 48]. This includes attacks like VoltPillager [15] intercepting and issuing bus commands to the onboard voltage regulators circumventing the OCM. We assume that the faulting behavior of VoltPillager does not differ from the software issued undervolt as both approaches influence the core voltage (see Section 8).

The enclave does not require the OCM to be disabled by a given local attestation. Hence, if the enclave is built using Minefield, the attestation does not have to verify whether the microcode disabling the undervolting functionality is active. The attacker does not exploit bugs inside the enclave's code, nor the software surrounding the enclave initialization, nor side-channel attacks to extract secret information from the enclave. As our defense focuses on fault attacks, we consider side-channel attacks [43] (e.g., cache attacks on SGX) an orthogonal problem. However, we discuss the implications of the mitigation on side-channel robustness in Section 8.

Fault-Injection Capabilities. The attacker can attack the enclave execution with DVFS attacks and induce faults inside the results of machine instructions. We assume that the attacker controls the environment with the same precision as in known DVFS attacks [15, 32, 38, 40, 41, 48]. Importantly, no previous DVFS attacks was able to:

1. precisely target an arbitrary bit inside an instruction result (but mounted attacks that work with random bit flips),
2. precisely control how many bits flip (but report various faults from a single bit to multi-byte flips [38]),
3. precisely control the timing (undervolting windows are multiple microseconds),
4. precisely control which instruction is faulted (*i.e.*, many instructions are at risk of fault due to the length of the undervolting window). Certain instructions are found to be more susceptible to DVFS-based fault injection.

We assume that the attacker has the capabilities from these previous works since there is currently no indication that the OCM enables even stronger and more precise attacks.

No single-stepping. In particular, no known attack can combine single-stepping, e.g., using controlled-channel attacks to target a specific instruction, with a DVFS-based fault attack. Given the significant amounts of code executed during context switches in controlled-channel attacks, there is reasonable doubt that such an attack can be mounted reliably. Furthermore, controlled-channel attacks can also be mitigated using T-SGX [46], entirely preventing single-stepping of SGX enclaves, or other interrupt-monitoring mechanisms [13, 21]. Hence, we assume that the attacker cannot target a single instruction this way. We discuss possible mitigations against a stronger attacker with single-stepping in Section 6.2.1.

4 High-Level Overview of Minefield

In this section, we provide a high-level overview of Minefield and the research questions we have to answer before designing it. The main goal is allowing the operating system to still control the undervolting of the CPU while ensuring that

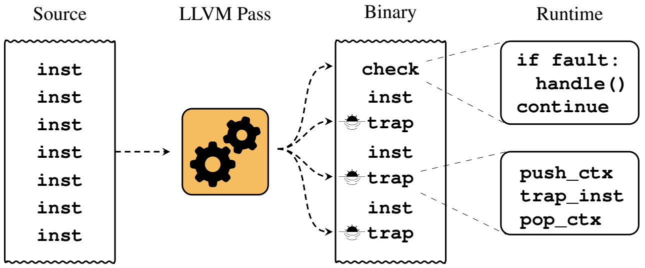


Figure 1: An overview of Minefield. The compiler part of Minefield interleaves the instruction stream with trap instructions and code to detect faults in these instructions. A library is linked to the enclave handling detected faults at runtime.

enclaves cannot be exploited. Minefield relies on an LLVM compiler extension to automatically place trap instructions in the code. Minefield has a security parameter to fine-tune the application-specific security-performance trade-off based on the required security guarantees. This makes Minefield an easily adaptable mitigation without changes in the protected software and with individual security levels per application.

Figure 1 shows an overview of Minefield and its components. Minefield consists of an LLVM compiler extension (Section 6.1) used to compile SGX enclaves, as well as a runtime library (Section 6.2) to check for faults and handle them. The compiler extension compiles unmodified source code and adds additional trap instructions to the binary. The trap instruction is an instruction highly susceptible to DVFS faults. Hence, the result of the trap instruction is used to detect faults that the enclave can then handle. Designing Minefield requires answering three research questions as follows.

4.1 Research Questions

While our approach may appear intuitive, a thorough analysis of DVFS-based fault attacks is necessary to ensure that Minefield is not built upon potentially wrong assumptions. Furthermore, even if the assumptions as outlined in the threat model hold, there remains a set of unanswered questions on the precise attacker capabilities. In the following, we ask three research questions we need to answer.

> *RQ1:* Is there an **instruction** highly susceptible to faults, and if so, how can we find this instruction?

Although all the published attacks [32, 38, 48] show that instructions can be faulted based on concrete instances of instructions, there is no comprehensive analysis on which instructions can be faulted. V0ltpwn [32], Plundervolt [38], and VoltJockey [41] indicate that multiplications are highly susceptible to faults on all evaluated systems. However, without a comprehensive analysis, this remains an assumption that must be further analyzed, as we do in Section 5.1.

> *RQ2:* What is the temporal and spatial **precision** of DVFS-based fault attacks?

Previous work induced single faults by repeating the target application until the fault hit the correct instruction [32,38,40, 41]. However, the precision for inducing faults is unknown. In Section 5.2, we analyze the capabilities of an attacker using DVFS to inject faults. We show that faults cannot be injected with arbitrary precision. Moreover, we show that there is a minimum time window between two undervolts.

> *RQ3:* How can an enclave react when **detecting** a fault?

Detecting a fault is not sufficient. An enclave has to react to the fault as well. Without replay protection in SGX [37], attacks could be repeated at a high frequency, even if a fault is detected. Thus, it is insufficient to simply terminate the enclave, especially when an attacker can arbitrarily retry inducing a fault. We discuss possible solutions in Section 5.3.

Based on our analysis in Section 5, we present the design and implementation of Minefield in Section 6.

5 Analysis of Research Questions

In this section, we analyze the capabilities of software-based fault-injection attacks to answer the research questions asked in Section 4.1. We exhaustively test the fault susceptibility of x86 instructions (RQ1) using an automated framework in Section 5.1. Moreover, we analyze the capabilities of an attacker to inject faults (RQ2), *i.e.*, the type of fault, as well as the spatial and temporal precision, in Section 5.2. Finally, we discuss the handling of detected faults (RQ3) in Section 5.3.

5.1 RQ1: Instruction Susceptibility to Faults

To determine the fault characteristics, we analyze instructions of the x86 instruction set during critical undervolting conditions and monitor the faults injected into the results. The goal is to find a suitable trap instruction with the highest fault susceptibility that is used by Minefield to detect faults. Therefore, we implement an analysis framework to determine instructions that are usable as trap instructions and to get a more in-depth insight into how undervolting affects instructions.

Design. Our framework is designed to exhaustively test all unprivileged x86 instructions for multiple levels of undervolting. The design of our framework is illustrated in Figure 2. The basic idea is to test an instruction multiple times. In each test, the instruction is run once in a stable environment and once in an undervolted environment. For both runs, the same randomly-generated inputs are chosen. The framework records the output values for both runs and compares them. If the output differs, the undervolting led to a fault, and the instruction, its parameters, and the undervolting level are reported. As outlined, this test is performed multiple times for

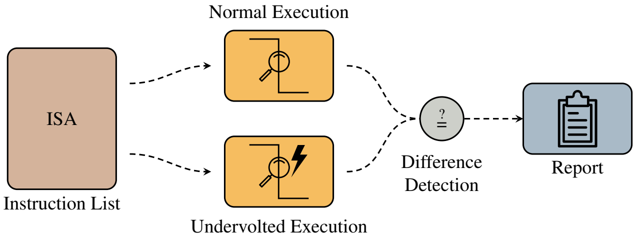


Figure 2: The framework to find trap instructions. Based on a machine-readable list of instructions, the framework executes all unprivileged instructions with random arguments, once normally and once undervolted. If the output differs, the instruction is reported as being susceptible to faults.

each instruction, namely with different input parameter values and undervolting levels and CPU frequencies.

Our test framework stores the bit difference of the expected result and the faulted result, *i.e.*, the bit location where the fault occurred. Furthermore, the framework analyzes the type of the fault, *i.e.*, whether it is a stuck-at-zero or stuck-at-one fault, or a bit flip. To measure the precision of the fault injection, we also record the temporal and spatial distance between two faulted operations. Consequently, our framework can determine the lowest observed temporal and spatial distance between two faults. This can later be used as a basis to determine appropriate security levels.

Implementation. To test the instructions defined in the x86 ISA, we use the list from Abel et al. [2], which contains all x86 instructions, including all ISA extensions, as well as the input, output, and side effects of the instruction. The framework automatically generates assembly code to parametrize these instructions. The generated assembly code is placed inside a loop to repeat the instruction multiple times. It is then compiled into a dynamic library for the test environment to load and evaluate. The framework allocates buffers for the instructions' state, e.g., registers and flags, and runs the instruction loop. The instruction in the loop then fills the buffers with all the changed state produced by the 1 000 000 iterations. Instructions that change the program's control flow are handled by setting the jump destination to an instruction after the jump that sets a flag to indicate that the jump was either taken or not.

The framework uses the same undervolting mechanism as Plundervolt [38], V0ltpwn [32], and VoltJockey [41], namely the OCM MSR 0x150. This MSR allows reducing the operating voltage for a short duration by modifying the voltage offset. When the undervolted execution is completed, the nominal voltage is restored, and the results are analyzed for bit errors. In this step, the framework compares the results of the undervolted instruction with the normal execution of the instruction. Each loop iteration is independent of the previous, so a fault inside one iteration is only visible in the iteration's outputs and does not influence other iterations.

To exhaustively test and analyze the instructions, we split the analysis into three distinct phases. First, we search for *faultable* instructions across all the tested CPUs at a fixed frequency and vary the undervolting offset until we see repeated system freezes or unrecoverable machine check errors. The framework monitors the response time and restarts over a remote power switch to recover from a system freeze. We store each reported faulted instruction into a global set of faultable instructions. Second, we use the set of faultable instructions to characterize the faulting behavior further. We execute each faultable instruction on each physical core of each CPU, with both varying frequency and undervolting offset. This analysis shows the minimum undervolt needed to observe a fault over the tested frequencies for each core. To evaluate the effect of other instructions on the faulting instruction, we tested each faultable instruction with all the other faultable instructions and evaluate if the faulting behavior is influenced.

Results. We analyzed 5 Intel CPUs with different microarchitectures, each running the same image with Ubuntu 21.04 with Kernel version 5.11. We list the exact CPUs in Table 2 (Appendix B). For each CPU, we analyze each physical core, resulting in a total of 26 analyzed cores. Our experiments did not observe different faulting behavior for the sibling threads, but we observed differences between the physical cores. In the instruction finding phase, we executed 1258 instructions and instruction variants, *i.e.*, same instruction but with different mnemonics, from the base, SSE, SSE2, FMA, AVX, AVX2, and AES instruction set, fixed the frequency to 3000 MHz. This analysis revealed 71 faultable instructions variants with 12 unique instructions. We analyzed the first faulting point for each of these unique instructions by varying the frequency from 2000 MHz to 4000 MHz (if available) in 500 MHz steps. Table 2 (Appendix B) shows the faulting point results.

From our experiment, we found that `imul` has the highest fault probability. Table 2 shows the analysis of `imul` in combination with different instructions. `imul` does not only fault well in isolation, but this behavior is also observable when combined with other instructions. The `imul` instruction faults in 92.1 % of all cases when other instructions also fault. For 1.5 % of the faulty instructions we need an additional `aesenc` instruction to detect the faults. On one CPU, we did not observe any faults with AES and hence used a `vorpd` instruction to detect the remaining 6.4 % faultable instructions.

Moreover, the `imul` instruction already suffers from faults at *smaller* undervolting offsets. This is in accordance with recent works [32,38] that focus on `imul` as well. Hence, `imul` is ideal as a trap instruction to monitor if the CPU is driven near the specification limits. In Section 7.1, we also show that this property holds when using `imul` in full programs.

5.2 RQ2: Fault-Injection Capabilities

The security level of Minefield is related to the fault-injection capabilities of the attacker. Prior work [32,38,40,41] did not

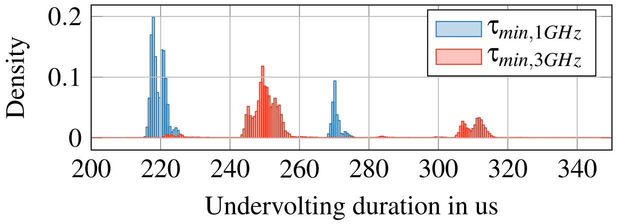


Figure 3: The minimal undervolting time window for two distinct CPU frequencies.

comprehensively analyze properties, e.g., the precision, of the faults but simply measured the probability of being able to fault the target instruction at any point when running it in a long loop. However, to provide strong security guarantees, we analyze DVFS-based fault injection in more detail, *i.e.*, the fault model [5]. The fault model includes types of errors, temporal and spatial precision, and the number of faults that can be injected in one execution [20]. Our fault model is based on prior work [32, 38, 40, 41] and our own experiments, and considers spatial and temporal *precision* and the *type* of faults.

Temporal and Spatial Precision. Previous work [32, 38, 40, 41] did not analyze where faults can be injected. Typically, faults can occur at a random location [6], or can be induced for instruction sequences or surgically for single instructions. So far, no paper has shown that it is possible to induce DVFS faults with such surgical precision. Our experiments also indicate that targeting a single instruction with DVFS faults is impractical. Figure 3 shows a histogram for the experimentally measured minimum undervolting time at two different CPU frequencies. The average undervolting duration at 1 GHz is around 220 µs, which are 220 000 cycles. The shortest undervolting duration we observe is 57.8 µs, *i.e.*, 57 800 cycles. Hence, to target a single instruction, an attacker would have to target a code sequence where the victim instruction is the only instruction susceptible to a fault within this window.

The minimal length of the undervolting window also influences the timing of faults. First, the minimal duration of the undervolting limits the frequency in which an attacker can induce faults. The CPU requires time to change the voltage. This is true both for reducing as well as increasing the voltage. As shown in Figure 3, the durations are not constant but subject to variations in the microsecond range, depending on the CPU frequency and also the CPU itself. We do not have an explanation for this effect. However, as a consequence, undervolting precisely the same instruction sequence multiple times is infeasible. When undervolting, there is always a non-negligible probability that several instructions before or after the targeted instruction are undervolted as well.

Fault Types. In addition to the precision, it is also important what types of faults can be injected. Typical fault models consider stuck-at-zero, stuck-at-one [18], random faults [23], or flips in one or more bits [19, 33]. There are more specialized fault models, e.g., bits with a bias [39].

We based our analysis of the fault types on our experiments and the results in prior work [32, 38, 41]. Table 1 (Appendix A) shows the detailed fault characteristics of each observed fault of our previous analysis. We confirm that a fault can influence one bit to multiple bytes. Further analysis revealed that we observe stuck-at-zero faults for instructions executing bitwise logical operations, *i.e.*, *VAND*, *VXOR*, and *VOR*. However, the faults of `imul` and further susceptible instructions behave randomly, *i.e.*, all observed bit positions can flip in both directions. Moreover, the affected bits differ between the physical CPU cores [32]. Hence, for the fault model, we assume that an attacker can flip between one and all bits of `imul`'s result to random values.

There is no difference if an ALU instruction is faulted or the address generation in a load or store instruction. In all cases, an attacker cannot choose the location of the bits, the number of bits, or the values of the bits.

5.3 RQ3: Handling Faults

While detecting a fault is a vital requirement for Minefield, it is not sufficient for protecting an enclave if the fault is not handled correctly. Hence, an important part of Minefield is the fault handler. We identified two different strategies for handling faults such that they cannot be exploited.

Cancel. The straightforward approach is to stop further execution as soon as a fault is detected. Aborting ensures that no further instructions are executed that potentially consume the faulted data. An abort handler does not require any change to the enclave code. To abort the enclave, the handler can either execute an illegal instruction, e.g., `ud2`, or simply stop execution by entering an endless loop. Note that Minefield does not suffer from false positives (cf. Section 7.1), *i.e.*, enclave execution is never wrongly aborted.

While abort handlers are straightforward, they may open new attack surface: An attacker could repeat the attack at a high frequency to increase the chance of bypassing our detection in one of the runs. Potentially, by knowing where the detection was triggered, the attacker might even improve the attack further. Without secure persistent storage and replay protection, the enclave developer must provide additional infrastructure to prevent the enclave from being started again. The SGX ecosystem already provides the EPID attestation method [31], allowing to identify a specific CPU, practically solving the replay protection problem if a remote trusted third party is available. We discuss different solutions in Section 8.

One possibility to reduce the frequency of restarts is to use monotonic counters [24]. These counters can only be read and incremented and are persistent across enclave restarts and also system restarts. Hence, by incrementing the counter on a fault, the enclave can track the number of total faults and decide not to start when a certain number of faults was detected. However, even with the counters, it is not possible to entirely prevent arbitrary execution of the enclave as the counters can be destroyed by re-installing the Intel PSW or by removing the BIOS battery [37]. Still, this at least slows down an attacker and might make an attack infeasible. We further discuss the availability of monotonic counters in Section 8.

Retry. A different approach is to try to "hide" the fault and prevent its weaponization by restoring the state before the fault and repeating the instruction. The retry handling is more complex, as instructions are not generally idempotent. Thus, the retry handler cannot simply re-execute the instruction before the fault or the current basic block. To use a retry handler, a developer has to define checkpoints in the enclave code to which a fault handler can safely jump back. Inside the retry handler, a developer can then choose to which checkpoint to return based on where the fault was detected. The implementation of such checkpoints could make use of the already existing `setjmp` and `longjmp` C functions.

The retry handler has the advantage that the enclave can continue execution in the presence of faults. Thus, this approach has similar advantages to multiple executions with a majority vote [5, 14], without the disadvantage of always executing code multiple times. The obvious disadvantage is that the developer has to take care of checkpoints at which execution can be retried. Moreover, the retry handler might provide an attacker with valuable information. As an attacker can monitor the execution time of the enclave, an attacker might learn that the fault was successfully injected. However, an attacker only learns that the fault definitely hit a trap instruction and not if the fault hit the target instruction. While this cannot be weaponized directly, it introduces a side channel (see Section 8).

5.4 Results

Based on the results from analyzing the research questions, we provide a solid fault model for software-based DVFS fault attacks. We show that there are indeed instructions that are more susceptible to faults than others. We confirm that `imul` instruction exploited in prior work [38] is indeed highly susceptible to faults, making it a perfect choice for Minefield's trap instruction. Furthermore, our analysis shows that an attacker cannot surgically induce faults. Both the temporal and spatial precision are limited by the minimal undervolting window of multiple microseconds. Hence, next to the instruction targeted by an attacker, there are always other instructions that are executed in the undervolted state as well. We can use this non-uniformity to place instructions with a higher chance to attract faults as trap instructions and enforce that the results of these trap instructions are not altered or faulted. Thus, these trap instructions enable us to protect the real instructions with a relatively simple mechanism against undervolting attacks. Depending on the number of inserted instructions, any induced fault is likely to also fault at least one of these inserted instructions.

6 Implementation of Minefield

In this section, we discuss the implementation details of Minefield. The implementation consists of two parts. The first part is a configurable LLVM compiler extension (Section 6.1) for adding additional trap instruction to enclave code at compile time. The second part is the runtime environment integrated into the enclave for detecting and handling induced faults (Section 6.2). Finally, Section 6.3 describes how the changes are integrated into the SGX toolchain.

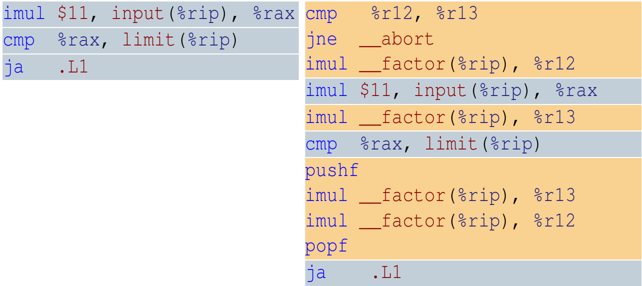
6.1 Compiler Extension

The compiler extension implements a *Machine Function Pass* inside the LLVM 11 [34] backend. The *Machine Function Pass* allows inspecting each program's function on an x86 machine instruction level. Implementing Minefield in the backend ensures that it is language agnostic, as long as there is an LLVM frontend for the desired language. The compiler extension is responsible for placing trap instructions and generating the code that checks whether a fault was induced.

Trap Instructions. Based on our fault susceptibility analysis and the fault model (cf. Section 5), we select `imul` as a default trap instruction, as it has the highest probability to fault when undervolted on our tested systems. In addition to this default trap instruction, a developer can also provide a different trap instruction to the compiler extension, e.g., a pair for AES and multiplication instruction. To keep track of the current state of our `imul` instruction, we use two distinct instances of the trap that only differ in the register operand. By placing both of these instances in alternating order, we ensure that the values in the two registers are at most one execution of the trap instruction apart. This placement is independent of the chosen trap instruction. The compiler also ensures that the basic block contains an even number of traps by adding an additional trap if necessary. This ensures that the registers must always have the same content at the start of a basic block, eliminating the need to store additional information about the current correct value as Figure 4b shows.

Placing Trap Instructions. Generally, trap instructions are placed between existing instructions, as illustrated in Figure 4. However, several practical obstacles have to be handled by the compiler. The trap instruction modifies the content of a register. Hence, the compiler has to know that the register used in the trap instruction is clobbered. Moreover, both the trap instruction and the code for detecting fault might modify the CPU flags, *i.e.*, the `rflags` register.

In addition to the problems of inserting a trap itself, we also have to decide *when* to insert a trap instruction. Inserting more traps leads to better security guarantees, while it impacts the performance negatively. We provide tuning parameters to find a trade-off between the performance and the provided security by the mitigation. We denote this parameter as the *placement density*. This parameter defines the ratio of trap instructions



```
imul $11, input(%rip), %rax
cmp  %rax, limit(%rip)
ja    .L1
```

(a) unmodified

```
cmp   %r12, %r13
jne  __abort
imul __factor(%rip), %r12
imul $11, input(%rip), %rax
imul __factor(%rip), %r13
cmp  %rax, limit(%rip)
pushf
imul __factor(%rip), %r13
imul __factor(%rip), %r12
popf
ja    .L1
```

(b) modified

Figure 4: Figure 4a shows the unprotected assembly instructions while Figure 4b shows the trap instruction sequence generated by Minefield with a placement density of 1.

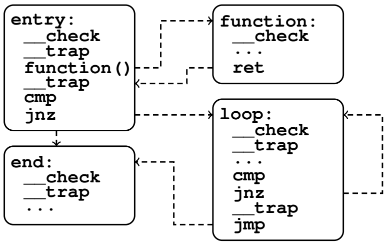


Figure 5: Traps are inserted between normal instructions. To ensure the correctness of comparisons, no trap instruction is inserted directly between comparison and conditional jump but only at the two jump destinations. Trap instructions are checked at the beginning of each basic block.

to existing instructions, e.g., a density of 0 means that no trap instruction is placed, a density of 0.5 places a trap after every second instruction. If this parameter is chosen higher than one, we place multiple trap instructions after the original instruction. We also ensure that at least two trap instructions are placed inside a basic block such that the mitigation also works if the placement density is low.

As the compiler extension is implemented in the backend, inserting the trap instructions is straightforward. The compiler simply iterates over each of the instructions inside the basic blocks and can directly insert new instructions.

Handling Register Clobbering. If the placement density is greater or equal to 1, Minefield places a trap instruction basically after every instruction. As loading and storing a value from and to memory after every executed instruction incurs a high overhead, the trap instead operates solely on values stored registers. Minefield dedicates two general-purpose registers to fault checking to minimize the performance impact. For the general-purpose registers, we use R12 and R13. These registers have no special use in the System V Application Binary Interface. Both registers are defined to be callee-saved. Thus, no other function can change the content of the registers. As a result, Minefield even supports calling functions that have not been compiled with the compiler extension, making it fully backward compatible with existing code. Reserving a general-purpose register inside the LLVM compiler infrastructure already excludes the register from the complete pipeline. Thus, no additional precautions are necessary.

In addition to the modification of the register, a trap can also modify the rflags register [26], thus changing the architectural state and, with that, potentially the semantics of the original program. To prevent saving and restoring the flags all the time, we rely on the liveness analysis of LLVM. The LLVM infrastructure records which registers are currently alive and in use and which instruction consumes a given register (variable liveness analysis). Therefore, we can monitor when the flags register is in use. Typically, the content of the flag register is only relevant between a comparison and a conditional operation, e.g., a conditional jump. Minefield can simply omit the placement of trap instructions between an instruction that modifies and an instruction that acts on the flags register. This approach increases the performance without significantly reducing the security guarantees, as faults are checked in any case at the beginning of a basic block. We evaluate the correctness of this approach in Section 7.3.

Without relying on the liveness, the rflags would have to be saved before and restored after executing the trap. However, saving and restoring the state is expensive, as it involves pushing the flags to the stack (pushf) and restoring it from the stack afterward (popf). While this ensures the correctness of the generated code, it adds a non-negligible performance overhead to the fault checks. For testing purposes, we provide an additional compiler option to fall back to this slower approach and not use the liveness analysis of LLVM.

Fault-detection Code. To detect faults in the trap instruction, the compiler extension creates code for the fault detection. Minefield supports two ways of checking whether a fault occurred in a trap instruction. This check can either be *immediate*, *i.e.*, the detection code is inserted after every trap instruction. Alternatively, the check can be *lazy*, *i.e.*, the check is only performed at the start of a basic block. Nevertheless, despite the used method, a check is always performed at the basic block's beginning by simply comparing R12 and R13.

Both approaches have their advantages and disadvantages. *Immediate* checking results in a larger binary size and also a larger performance overhead. However, with *immediate* checking, the time between a fault and the detection of the fault is minimized. When using *lazy* checking, the trap instruction is verified at the beginning of each basic block. Hence, with lazy checking, the number of checks is reduced, increasing the performance but potentially increasing the time window in which a fault could be exploited.

Immediate checking seems intuitive. However, for instructions that only operate on registers and do not perform a memory access, *immediate* checking does not provide additional security since the faulted value is not visible outside of the registers of the CPU. With this observation, we can extend the *immediate* checking method to only check the trap instructions right before either a load or a store is executed.

This extension ensures that neither the load's address nor the store's address or data was previously faulted.

Basic Blocks and Control Flow Changes. There are two main reasons we chose to verify the trap instructions at the beginning of basic blocks, as shown in Figure 5. First, as per definition, control flow changes can only target the beginning of a basic block and never target instructions inside the basic block. We can ensure that checks placed at the beginning of basic blocks are always executed, regardless of the control flow conditions [1, 47]. Second, a basic block has one entry point but can have multiple exit points. A basic block can be exited by either calling a different function, by returning, or by jumping to a different basic block. Therefore, checking all the possible exit paths requires more checks that impact the performance without providing any benefits. Nevertheless, we still have to perform checks before return instructions since in LLVM, call instructions can be placed inside basic blocks.

6.2 Runtime Fault Handling

The second part of Minefield is the runtime library, statically linked into the enclave, which handles the detected fault. By default, an abort-handler callback is called when a fault is detected. The implementation of the actual fault handler is the responsibility of the enclave developer. This allows maximum flexibility for the developer, as depending on the threat model, there are different reactions to a detected fault.

Minefield also provides two default fault handlers that can be used in many scenarios. These fault handlers can either retry or cancel the execution of the enclave when a fault is detected, as outlined in Section 5.3.

6.2.1 Monitoring Controlled-Channel Attacks

As already described in the attacker model (cf. Section 3), there is no combined controlled-channel DVFS fault attack and due to the significant amount of code, it is unlikely that such an attack could be implemented reliably. Controlled-channel attack frameworks, such as *sgx-step* [10] enable attackers to essentially single-step (and zero-step) an enclave. Thus, the attacker can step an enclave precisely to a single target instruction inside an enclave. If this technique could be combined with a DVFS attack, it could bypass our defense. There is a strong indication that such a combination cannot be mounted reliably, *i.e.*, the system freezes instead because of the substantial amount of micro-code executed during enclave entry. We also empirically validated this in our own experiments. When undervolting during enclave entry, the system easily freezes while the CPU restores the enclave state. We also emphasize that appropriate countermeasures against controlled-channel attacks (including single-stepping of SGX enclaves) already exist, e.g., T-SGX by Shih et al. [46].

Integrated mitigation. As T-SGX would incur additional overhead, we also propose a more integrated solution to prevent single-stepping-assisted DVFS fault attacks. Similar to previous work [13, 21], we can utilize the Save State Area (SSA) of the enclave to monitor any interruptions. When an enclave gets interrupted and the control flow is passed to the interrupt handler, the state of the enclave is stored inside the SSA of the enclave, including all the registers and additional enclave state. We can write a magic value to the position of the enclave `RIP` field inside the SSA and later check if this magic is still present or if it was overwritten by the CPU when exiting the enclave asynchronously. Frequent interruptions can be handled as described in previous works [13, 21].

6.3 Toolchain Integration

At the time of writing, Intel does not officially support LLVM to build SGX enclaves with their SGX SDK. Compiling the SDK with clang instead of gcc fails due to gcc-specific features used in the SDK. Hence, to use the SDK for evaluation with Minefield, we had to apply small changes to the SDK version 2.10 to make it compatible to LLVM.[1]

We only compile the trusted part of the SGX SDK with Minefield. This is sufficient, as only the trusted part is an attack target. The untrusted part is under the control of the attacker. Thus, there is no benefit in protecting this part. The protected part is responsible for the enclave initialization, the enclave entry calls, and the enclave exits. As a small part of the enclave entry and exit code is written in assembly, it needs to be manually patched, as the current prototype of Minefield does not support assembly files. In addition to the manual patching, we adopted the enclave entry function to set up the registers required for Minefield. There is no need for additional modifications inside the source code.

7 Evaluation

In this section, we evaluate the security (Section 7.1), performance (Section 7.2), and correctness (Section 7.3).

7.1 Security

We evaluate the security of Minefield by evaluating the probability to detect induced faults for two different applications, a victim highly susceptible to fault attacks as used by Murdock et al. [38], as well as a practical application based on *mbedTLS*. In both scenarios, we evaluate different undervolting levels and placement densities between 0 and 2 and show that Minefield can successfully protect these applications.

Setup. We use SGX enclaves built with Minefield. All experiments are conducted on an Intel Core i5-8265U running Ubuntu 20.04.1 LTS. We focus on two different enclaves, one

[1] Our changes to the SDK, the source of Minefield, and the test enclaves are provided at: `https://github.com/iaik/minefield`.

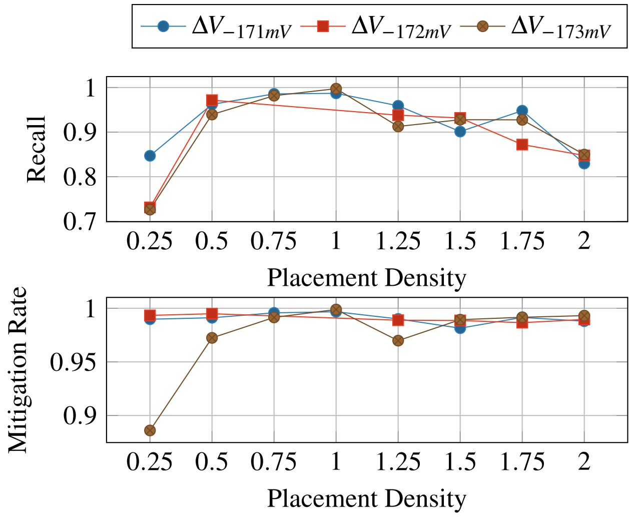


Figure 6: The *recall* over the placement density for several undervolting offsets for the `imul` experiment.

enclave containing the Plundervolt multiplication proof-of-concept [38], and another enclave running *mbedTLS* [3]. As an abort handler, we use a function that reports faults without terminating the applications.

As a detection metric, we use the *recall* of Minefield. In our setup, the recall is calculated by dividing the number of experiments where the target was successfully faulted and the mitigation detected the fault ($\mathcal{F}\&\mathcal{D}$) by the number of the experiments where the target was faulted ($\mathcal{F}$). The *recall* is bounded between zero and one, where zero means that no fault was detected and one that all faults were detected. We use the *recall* instead of the *F-score* as a security metric since the *precision* of Minefield is consistently one. This is because the detection cannot observe *false negatives*. The check is entirely deterministic. Thus, it is not possible to detect a fault although there was no fault. Moreover, if we observe a fault inside a trap instruction, the system is already driven near the specification limits. Hence, even if the fault is only in a trap instruction, the execution of the enclave is no longer safe and should be terminated. As a consequence, there is no case where the trap instruction triggers without the system being compromised. For the two enclaves, we evaluate the *recall* for different voltage offsets and vary the placement density between 0 and 2 (cf. Section 6.1).

7.1.1 Highly-susceptible Toy Victim

In the first scenario, we use a toy application inspired by the Plundervolt proof of concept [38]. In this application, we target four `imul` instructions executed 30 720 times inside a tight loop. The multiplications use the result of the previous iteration as input. Thus, any fault induced in a multiplication propagates to the final iteration's result. To detect a fault, it is sufficient to compare the final result to the ground truth. The fault propagation has the advantage that no additional fault-checking code has to be inserted. This toy application has a high probability that a fault can be induced. Moreover, every fault is effective, leading to a change in the final result. Hence, from a defender's perspective, this application is close to the worst case, as nearly every instruction has to be protected.

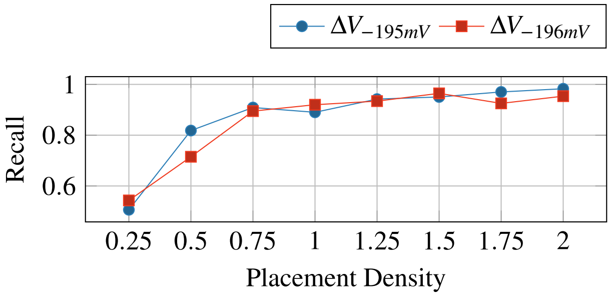


Figure 7: The *recall* over the placement density for several undervolting offsets for the *mbedTLS* experiment.

Figure 6 shows the recall when compiling this code with Minefield and inducing faults. We test different undervolting levels for which the rest of the system is still stable. We observe that the undervolting level itself does not significantly impact the probability of inducing a fault. As expected, the recall increases with the placement density of Minefield.

With a placement density of 0.5, we already recognize 80 % of the faults. If we further increase the placement density to 1, we can detect nearly all the faults for the different voltage offsets. We also observed that on one of our cores, the recall started to decline when increasing the placement density above 1.25. However, this does not directly correlate with the security. In that case, the overall probability of inducing a fault decreased drastically. Hence, we were rarely able to induce a fault at all. To better visualize this effect, we show in Figure 6 the mitigation rate, *i.e.*, the inverse of the probability that an attacker faults the target instruction without the mitigation detecting it. For all placement densities above 0.75, the mitigation rate never dropped below 95 %.

This toy application shows that even if most instructions are susceptible to faults, and any fault is exploitable, Minefield can protect an application. Especially with a high placement density, there is a nearly arbitrarily adjustable trade-off between performance impact and security guarantees.

7.1.2 Real-world Victim

The second scenario uses a more realistic application. We protect *mbedTLS* [3] version 2.13 with Minefield. Due to its small codebase and simplicity, it can be easily used inside SGX enclaves [52]. As a constant target for side-channel attacks, *mbedTLS* also provides side-channel resilient implementations of the provided cryptographic algorithms [4].

For our evaluation, we focus on the RSA signature algorithm of *mbedTLS*. As shown in previous work [38], a fault during the signing can be sufficient to recover the private key. Hence, for the evaluation, we focus on the underlying *binary modulo exponentiation* function `mbedtls_mpi_exp_mod`, which is directly used inside the library's RSA algorithm. We choose the input parameters for the function to represent a 4096-bit key. After the execution of the algorithm, we check whether the result of the function was faulted and also determine whether Minefield detected the fault. For each placement density, we perform 2000 encryptions. The voltage is reduced for each of these 2000 encryptions before entering the function and restored after the return from the encryption.

Figure 7 shows the recall for the *mbedTLS* experiment. We observe a relatively high detection rate of 90 % with a low placement density of 0.75 across two voltage offsets of −195 mV, *i.e.*, the first offset where we observe faults and −196 mV, *i.e.*, the last offset where the system did not freeze. Compared to the toy application, the undervolting offset of the *mbedTLS* example is lower since the executed codebase is more extensive.

7.1.3 Results

For both our victims, Minefield reliably detects induced faults. Especially for higher placement densities, the probability of faulting a target instruction without triggering Minefield is very low. While the level of undervolting does not have a huge impact, we observe a trend that the mitigation performs slightly better if the CPU is driven more into critical conditions, *i.e.*, if we undervolt the CPU more.

For the analysis, we do not consider faults detected by Minefield that had no effect on the victim computations. In a real scenario, it is also desirable that these faults trigger the enclave's abort or retry mechanic, as a stable execution cannot be guaranteed. During our experiments, we observed on average 10 times more faults inside the trap instructions compared to the target `imul` instruction. This result also supports our choice for using `imul` as the default trap instruction.

7.2 Performance of Minefield

For the performance evaluation of Minefield, we evaluate three different metrics: the runtime overhead (Section 7.2.1), the increase in code size (Section 7.2.2), as well as the one-time compile-time overhead (Section 7.2.3).

7.2.1 Runtime Evaluation

To evaluate the performance, we use the well-known *SGX nbench* benchmark suite for SGX. Additionally, we also evaluate the performance impact on *mbedTLS*, as we used this library for the security benchmark as well (see Section 7.1). For *mbedTLS*, we also compare the performance overhead of Minefield to the integrated fault-mitigation technique.

SGX nbench. *SGX nbench* [49] is an adoption of the traditional nbench benchmark suite for SGX enclaves. The benchmarks focus on classical benchmarks executed inside the enclave environment. We use this benchmark to evaluate the performance impact on actual performance code mitigated with Minefield including the SGX SDK. Each of the benchmarks is executed 25 times over a total duration of 2 h and 51 min. Figure 8 shows an average overhead for a placement density 1 of 191.51 %. The overhead linearly increases to 400.12 % for a placement density of 2. In all cases, the standard deviation was below 1 %.

mbedTLS. *mbedTLS* [3] already hardens the software implementation of its RSA algorithm against fault attacks. The `mbedtls_rsa_private` function used for encrypting data with the RSA key decrypts the complete ciphertext after encryption and compares if the decrypted message matches the provided function's input message. This is only possible since *mbedTLS* stores also the public key inside the private context.

With the built-in fault check, the RSA implementation takes on average 13.9 ms (n = 1000, $\sigma_{\bar{x}}$ = 0.064) for one encryption with a 2048-bit key, where the public key is large, *i.e.*, it only has 6 leading zeros. When disabling the internal fault check, the same encryption takes on average 6.9 ms (n = 1000, $\sigma_{\bar{x}}$ = 0.045). Hence, the runtime overhead of the internal check of *mbedTLS* is 100.99 %. When using the same parameters with a small public key, *i.e.*, the key has 2031 leading zeros, the overhead decreases to 1.13 %. For comparison with Minefield, we compile the version without the internal check with Minefield. Figure 9 shows the performance comparison over different placement densities. For large public keys (6 leading zeros), Minefield always performs better, regardless of the placement density. With a placement density of 0.75, we increase the performance by 71.42 % for full-length public keys compared to the internal verification of *mbedTLS*. We show in Section 7.1 that with a placement density of 0.75 we already achieve a recall of 90 %. For small public keys (2031 leading zeros) and at the same placement density, we only decrease the performance by 17.23 % compared to the internal verification of *mbedTLS*.

7.2.2 Code-Size Evaluation

As Minefield inserts additional code into an application, we compare the size of binaries created with Minefield and with the same compiler without any placed trap instructions. For evaluating the code size, we use the benchmarks used to evaluate the runtime overhead in Section 7.2.1. The code size is especially relevant for SGX, as the amount of physical memory usable by SGX is limited for all enclaves running on the system. We further discuss the memory impact in Section 8.

The code size is increased by the constant size of the runtime library linked to the enclave code (cf. Section 6.2). Additionally, there is a variable increase based on the number of instructions in the enclave and the placement density. Fig-

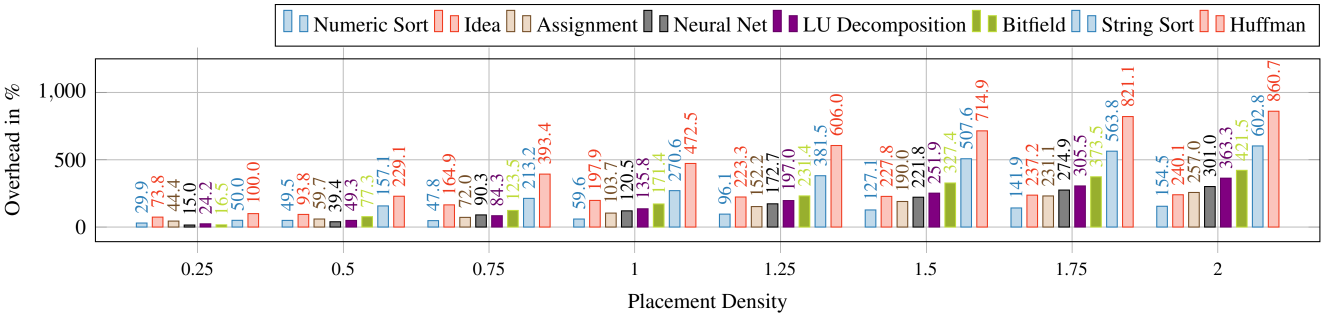


Figure 8: The performance overhead of Minefield for the *sgx-nbench* benchmark over multiple placement densities. We observe a linear overhead with increasing placement density.

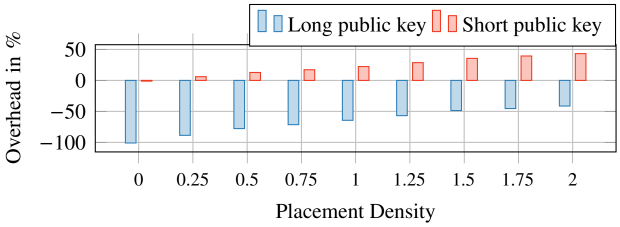


Figure 9: The performance comparison between the *mbedTLS* RSA verification and Minefield-protected version.

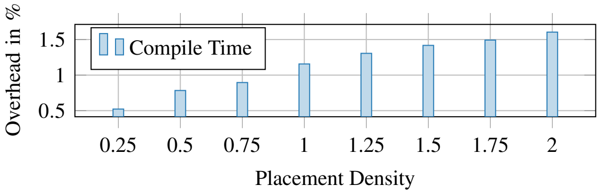


Figure 11: The compile-time increase for the various benchmarks over the placement density parameter of Minefield.

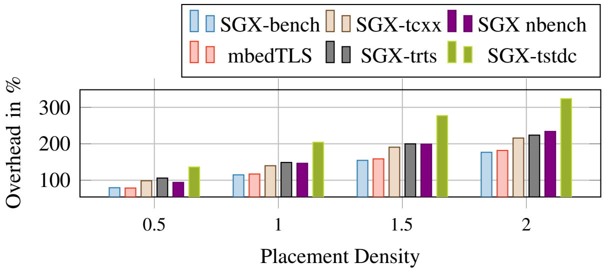


Figure 10: The increase of the code size for the various benchmarks over the placement density parameter of Minefield.

ure 10 shows the code size increasing over the placement density parameter for the *SGX nbench*, *mbedTLS* and *SGX-bench* benchmarks. In addition, we also show the increase in code size for trusted SGX SDK components such as the runtime system, the C library, and the C++ library.

As expected, we observe a nearly linear increase of the code size when using Minefield. However, even for large applications such as *SGX nbench*, protected with a placement density of 1, the absolute increase is only 274.5 kB.

7.2.3 Compile-Time Evaluation

We analyze the impact on the compile time that Minefield has on enclaves. As a baseline, we compile the benchmarks without any mitigations enabled. We compare the compile time for different placement densities to this baseline.

Figure 11 shows the average increase in compile time for the benchmarks. The placement density does not have a significant impact on the compile time. For a placement density of 0.5, the overhead is negligible, with on average around 0.78 %. Even for a placement density of 2, the overhead is only around 1.6 %, which amounts to less than 0.5 s for *mbedTLS*.

We conclude that the overhead on the compilation time is negligible, especially as this is a one-time overhead for the developer. This small overhead also makes it feasible to use Minefield in the development process, and not only for the final compilation of an SGX enclave.

7.3 Correctness of Minefield

In addition to the performance and security evaluation, we also verify that our Minefield prototype does not introduce any correctness problems.

Compiler Correctness. We explicitly tested the compiler by running a C compiler test suite [12] to ensure that we did not introduce any bugs. The test suite confirmed that the compiler changes did not have any adverse effect on the correctness. We also confirmed that *mbedTLS* works correctly with Minefield by running it without undervolting both in SGX and as a native application. For SGX, the SGX-nbench benchmark verifies the correctness of the computed results in addition to the performance. We did not encounter any errors.

Integration Correctness. In addition to the correctness of the compiler itself, we also evaluated the correctness of our integration with the SGX SDK. We relied on SGX-bench [42] to test the enclave interactions. SGX-bench is a small test suite for SGX enclaves to determine the performance of, e.g., enclave entries, enclave initialization, and enclave ocalls. We did not run into any bugs or crashes when running the test cases, showing that Minefield successfully works with the SGX SDK. Moreover, running our test enclaves with *mbedTLS* without undervolting also showed that the integration works.

8 Discussion and Limitations

Current Mitigations. The currently active countermeasure against undervolting attacks on SGX enclaves prohibits the user from applying voltage offsets to the CPU. This removes a feature to gain additional performance or increase thermal thresholds against throttling. In addition, software-based fault mitigations either handwritten in cryptographic software or per compiler extension often focus on calculating results multiple times to check the correctness of the results against each other. While these mitigations provide a high level of security against fault attacks, they induce additional software complexity and performance overhead based on the type of instructions protected.

Hardware Undervolting. Starting from the 11th generation, Intel CPUs reuse fully integrated voltage regulator (FIVR) designs, previously abandoned after the 4th generation [11]. CPUs without a FIVR design expose the voltage regulators, allowing an attacker within our threat model (cf. Section 3) to mount VoltPillager [15] attacks. The voltage regulators are connected to a bus that receives commands from the CPU. VoltPillager directly sends these commands over the bus bypassing the CPU and the OCM. Although we performed the instruction analysis via the OCM, we argue that the observed fault behavior is independent of how the undervolt is issued. Therefore, we assume that Minefield is also applicable against hardware-based undervolting like VoltPillager, where the current mitigation to disable the OCM is ineffective.

Persistent Failing. Li et al. [36] show that AMD SEV's "security-by-crash" is exploitable, similarly using Minefield without hindering an attacker from arbitrarily often restarting an enclave might result in an undetected fault. Intel SGX does not support any local replay-protected persistent state that is also protected against an attacker with physical access. Hence, an enclave cannot securely store any data that could be used to detect how often the enclave has already been started, without using a trusted remote server. Thus, an attacker can always restart an enclave arbitrarily often. Even when relying on the monotonic counters [24] for counting restarts, an attacker can, e.g., remove the BIOS battery to destroy the counter, effectively resetting it [37]. The support for monotonic counters was discontinued in the Linux SGX-SDK [29]. However, the latest available documentation of the Windows SGX-SDK [28] (March 2020) still lists these functions.

If a trusted remote server is available, we can either implement the replay protection with Intel's EPID scheme or other rollback preventions. Intel's EPID group signature remote attestation scheme [31] can verify that enclaves are part of a certain CPU group. Moreover, EPID supports the *named-base* mode that allows linking two signatures, *i.e.*, the verifier can determine if two signatures originate from the same signer. Therefore, when using the named-base mode, the verifier can deny the data exchange with enclaves that repeatedly restart, observe faults, or do not terminate. Matetic et al. [37] present a rollback prevention for persistent state based on a distributed system. Hence, by relying on such a technique, Minefield could also implement persistent failing without requiring any hardware change. This would restrict an attacker to only a developer-defined number of induced faults per physical CPU.

Performance and Memory Overhead. The performance and memory overhead of Minefield is adjustable by the placement density (cf. Section 6.1) and affects only occasionally running SGX workloads, allowing the remaining system to benefit from undervolting and the resulting performance and energy gains. Unfortunately, adjusting the placement density also affects the security guarantees. We propose the following extensions for future work to reduce Minefield's overhead without affecting security.

First, Table 2 shows a margin between `imul` faults and faults of different instructions. Minefield can utilize this margin by protecting regular `imul` instructions with additional redundancy or replacing them with functional equivalents. Due to this margin, the trap `imul` instructions observes substantially more faults at these lower voltages than the other susceptible instructions. Thus, increasing the detection capabilities retaining the same security guarantees with lower placement densities, improving performance.

Second, we can reduce the impact on branch prediction by replacing the check's `cmp` and `jne` instructions with instructions generating a GP-fault if Minefield detects a fault. We propose using `xor` to calculate the registers' difference followed by `popcount` giving the number of bit errors. Adding this number to the higher 16 bit of a 64 bit address makes the address non-canonical if a fault was detected. When accessing a non-canonical address, the CPU raises a GP-fault causing an asynchronous enclave exit [27]. The enclave can only be resumed at the internal signal handler, stopping further faulty code execution [24].

Finally, the SGX driver ensures that enclaves that exceed the available EPC memory (usually 128 MB) can execute without limitations by swapping EPC pages [16]. Accessing a non-present EPC page introduces a latency of 13 103 cycles (n = 1 000 000, $\sigma_{\bar{x}} = 0.925$) to swap it back into the EPC on our Intel i5-8265U. We analyzed Intel and Synaptics production enclaves and found that their enclave sizes are below 3 MB. Furthermore, Intel's Ice Lake CPUs increase the available EPC memory up to 1 TB [30]. Therefore, we find Minefield's memory overhead for these enclaves tolerable.

Side Channels. Minefield does not protect against classical side-channel attacks on enclaves. Side channels are orthogonal to fault attacks and are thus out of scope for Minefield. Intel sees it as the developers responsibility to ensure that their code is free of side channels [25]. Importantly, Minefield does not introduce any new or additional side channels, as we decouple the instructions responsible for fault detection from the actual data processed by the enclave. However, if the enclave is already susceptible to side-channel attacks, Minefield might amplify the side-channel leakage. In the worst case, this can enable the exploitation of side channels that were previously considered not exploitable. For example, the inserted trap instructions might change a secret-dependent control flow within a cache line to a secret-dependent control flow on a cache line or even cache set granularity. Hence, for complete side-channel protection, developers have to ensure that all algorithms handling secrets are data oblivious [25].

Other Architectures. The idea of Minefield is not restricted to any given architecture and has two requirements. First, the architecture needs an instruction that is more susceptible to undervolting faults than others. Second, all targets that can be faulted must be compilable with Minefield. If the architecture meets these requirements, Minefield can probabilistically protect code running on the system. The performance depends on all the susceptible instructions of that architecture.

Minefield is also applicable to other TEE alternatives such as ARM TrustZone and AMD SEV. Qiu et al. [40] target ARM TrustZone with software undervolting faults and exploit faults in AES and RSA computations. Minefield could protect the target AES and RSA code if we port the compiler extension and the runtime library. Due to AMD's x86 instruction set, Minefield is directly applicable to AMD SEV workloads. As of writing this paper, there are no known software undervolting attacks against AMD. However, Buhren et al. [8] exploit AMD SEV by inducing hardware undervolting faults to compromise the secure coprocessor responsible for transparent encryption. In this case, the attack compromises AMD SEV by exploiting code not protected by Minefield, breaking the second requirement and rendering the defense ineffective.

9 Conclusion

In this paper, we presented Minefield, the first software-level defense against DVFS attacks. We systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over a sequence of at least 57 thousand cycles. The trap instructions Minefield places in the victim code during compilation are highly susceptible to faults. Our evaluation showed that a density of 0.75 traps per instruction, *i.e.*, 1-2 traps after every second instruction reliably mitigates the currently known DVFS attacks on Intel CPUs, namely Plundervolt, V0ltpwn, VoltJockey, and VoltPillager. Minefield allows fine-grained selection of the performance-security tradeoff. For this strong security level, we observe overheads of 94.4 % on average on protected SGX enclaves. The performance of the remainder of the system is entirely unaffected. Thus, we conclude that Minefield is an important alternative to a solution in hardware or microcode that comes with the prohibitive effect of disabling the OCM entirely.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Dave Tian, for their guidance, comments and suggestions. Additional funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005. doi:10.1145/1102120.1102165.

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 673–686. ACM, 2019. doi:10.1145/3297858.3304062.

[3] ARM. mbed TLS, 2020. URL: https:///tls.mbed.org.

[4] ARM. Security Advisories - Tech Updates - Mbed TLS, 2020. URL: https://tls.mbed.org/tech-updates/security-advisories.

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*, page 7. ACM, 2010. doi:10.1145/1873548.1873555.

[6] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. doi:10.1007/3-540-69053-0_4.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[8] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, 2021.

[9] 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 William Enck and Adrienne Porter Felt, editors,

27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, pages 991–1008. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[10] 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, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017. doi:10.1145/3152701.3152706.

[11] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. Fivr—fully integrated voltage regulators on 4th generation intel® core™ socs. In *2014 IEEE Applied Power Electronics Conference and Exposition-APEC 2014*, pages 432–439. IEEE, 2014.

[12] Andrew Chambers. c-testsuite, 2020. URL: https://github.com/c-testsuite/c-testsuite.

[13] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating speculative-execution attacks on SGX with hyperrace. In *2019 IEEE Conference on Dependable and Secure Computing, DSC 2019, Hangzhou, China, November 18-20, 2019*, pages 1–8. IEEE, 2019. doi:10.1109/DSC47296.2019.8937682.

[14] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced simdization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 57–64. IEEE Computer Society, 2017. doi:10.1109/FDTC.2017.10.

[15] 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 Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 699–716. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: http://eprint.iacr.org/2016/086.

[17] Douglas Black. Intel & OEMs are disabling undervolting. Here's how to re-enable it, 2020. URL: https://www.ultrabookreview.com/37095-dells-disabling-undervolting-on-their-laptops-heres-how-to-re-enable-it/.

[18] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 108–118. IEEE Computer Society, 2013. doi:10.1109/FDTC.2013.18.

[19] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential fault intensity analysis. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 49–58. IEEE Computer Society, 2014. doi:10.1109/FDTC.2014.15.

[20] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France*, volume 153 of *IFIP*, pages 159–176. Kluwer/Springer, 2004. doi:10.1007/1-4020-8147-2_11.

[21] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 217–233. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss.

[22] Hacker News. Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX, 2019. URL: https://news.ycombinator.com/item?id=21759683.

[23] Ludger Hemme. A differential fault attack against early rounds of (triple-)des. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 254–267. Springer, 2004. doi:10.1007/978-3-540-28632-5_19.

[24] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[25] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations, 2019. URL: https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations.

[26] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[27] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[28] Intel. Intel SGX SDK Developer Reference for Windows* , 2020. URL: https://software.intel.com/content/www/us/en/develop/download/sgx-sdk-developer-reference-windows.html.

[29] Intel. Unable to find alternatives to monotonic counter application programming interfaces (apis) in intel software guard extensions (intel sgx) for linux to prevent sealing rollback attacks, 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html.

[30] Intel. What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel Xeon Scalable Processor Platforms?, 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html.

[31] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services, 2016.

[32] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1445–1461. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[33] Yoongu Kim, Ross Daly, Jeremie S. 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 *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014. doi:10.1109/ISCA.2014.6853210.

[34] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

[35] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 523–539. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[36] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: rollback protection for trusted execution. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1289–1306. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[38] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1466–1482. IEEE, 2020. doi:10.1109/SP40000.2020.00057.

[39] Sikhar Patranabis, Abhishek Chakraborty, Phuong Ha Nguyen, and Debdeep Mukhopadhyay. A biased fault attack on the time redundancy countermeasure for AES. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2015. doi:10.1007/978-3-319-21476-4_13.

[40] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 195–209. ACM, 2019. doi:10.1145/3319535.3354201.

[41] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2019, Xi'an, China, December 16-17, 2019*, pages 1–6. IEEE, 2019. doi:10.1109/AsianHOST47458.2019.9006701.

[42] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves, 2018. URL: https://github.com/sgxbench/sgxbench.

[43] Michael Schwarz and Daniel Gruss. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Secur. Priv.*, 18(5):18–27, 2020. doi:10.1109/MSEC.2020.2993896.

[44] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 753–768. ACM, 2019. doi:10.1145/3319535.3354252.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2017. doi:10.1007/978-3-319-60876-1_1.

[46] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/.

[47] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62. IEEE Computer Society, 2013. doi:10.1109/SP.2013.13.

[48] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1057–1074. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[49] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX, 2017. URL: https://github.com/utds3lab/sgx-nbench.

[50] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2016. doi:10.1007/978-3-319-45744-4_22.

[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. doi:10.1109/SP.2015.45.

[52] Fan Zhang. mbedtls-SGX: a TLS stack in SGX, 2019. URL: https://github.com/bl4ck5un/mbedtls-SGX.

A Faulting Bits

Table 1 shows the analysis of which bit flips we observed for the instructions. We recorded each bitflip, analyzed the direction to which the bit flipped, and reported the overall flip tendency of the faulty bits. We observe that logical vector instructions have a high probability for stuck-at-zero bitflips. Furthermore, we also found that some vector operations introduce interesting faulting behavior, e.g., the vector comparison instruction shows bitflip within a given element.

B Instruction Analysis

Table 2 shows the detailed first faulting points for all the faulted instructions we found. We tested the faulted instruction across all our available CPUs with SGX support. We compiled the faultable instruction with the trap instruction to verify that the generated faults are indeed detectable with a given trap instruction. We executed it with the reported undervolting offset.

Table 1: The distinct faulted bit positions for each of our tested CPUs in mask form and the corresponding probability for each bit to fault towards one. Note that all the outputs of the instructions are present in the mask, *i.e.*, `imul` also contains the `rflags` register. We see that parallel vector operations show only stuck-at-zero flips, where other instructions like multiplications show a more random distribution.

	Core i9-9900K		Core i7-8700k		Xeon E3-1505M		Core i7-6700K (1)	
Instruction	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$
IMUL	`0x00000000000000000000783fff000000`	46.8 %	`0x00000000000000000000001fff800000`	48.3 %	`0x00000000000000000ff000fffc000000`	48.7 %	`0x000000000000000000000007f8000000`	40.8 %
AESENC	`0x00000100000000000010000004188e04`	50.7 %	`0x00000000000000000000000004198e04`	49.4 %				
VANDN*	`0x000000a0022a0080000000a208a80000`	0.0 %			`0x10000050405455400000000000105c00`	0.0 %		
VAND*	`0x000000a002020080000000a000000000`	0.0 %			`0x00010100014554000000000500144400`	0.0 %		
VOR*	`0x000200a0022200820000002a02ac0020`	0.0 %			`0x00000010001414420000000000140e00`	0.0 %		
VPCLMULQDQ	`0x00000000000000000000000200000000`	0.0 %	`0x3fd7ff7ffffffffffffffffffffffbff`	51.1 %				
VPCMP*	`0x00a2aa82002000000000000000000000`	0.0 %	`0xffffffff00000000ffffffffffffffff`	33.2 %				
VPSRAD	`0x0aa2aaa0000200000000000000000000`	0.0 %						
VSQRTPD	`0x0000000001f4ffff0000000000c3ffff`	52.6 %	`0x000c000000001dff0000000000001fff`	47.0 %				
VXOR*	`0x000000a0222a0080000000a20a800000`	0.0 %			`0x10000050405455420000001500145e10`	0.0 %		
VPMAX*			`0xffffffffffffffffffffffffffffffff`	50.2 %				
VPADDQ			`0x00000101010001000000000000000000`	73.3 %				

Table 2: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
Core i9-9900K	2000	0	⬠ 244											
		2	⬠ 255	⬠ 255			⬠ 255							
		4					⬠ 255							
		5	⬠ 250											
		6	⬠ 250											
	2500	0	⬠ 187	⬠ 202	⬠ 201		⬠ 197	⬠ 197						
		1	⬠ 195	⬠ 197										
		2	⬠ 196	⬠ 197	⬠ 210									
		5	⬠ 197											
		6	⬠ 197				⬠ 210							
		7	⬠ 201											
	3000	0	⬠ 154	⬠ 165	⬠ 165	⬠ 170	⬠ 165	⬠ 163		⬠ 172				
		1	⬠ 160	⬠ 159	⬠ 172	⬠ 200	⬠ 165	⬠ 166					⬠ 167	
		2	⬠ 160	⬠ 165	⬠ 180	⬠ 174	⬠ 169	⬠ 172		⬠ 185				
		3	⬠ 170		⬠ 172	⬠ 177	⬠ 172	⬠ 171	⬠ 180					
		4	⬠ 163	⬠ 173	⬠ 177	⬠ 202	⬠ 171	⬠ 175					⬠ 179	
		5	⬠ 164	⬠ 165	⬠ 175		⬠ 199	⬠ 176						
		6	⬠ 158	⬠ 176	⬠ 210		⬠ 177							
		7	⬠ 165	⬠ 172	⬠ 182		⬠ 209							
	3500	0	⬠ 156	⬠ 160	⬠ 167	⬠ 166	⬠ 162	⬠ 159		⬠ 167		⬠ 175	⬠ 166	
		1	⬠ 161	⬠ 152	⬠ 166	⬠ 166	☆ 156	⬠ 167					⬠ 161	
		2	⬠ 159	☆ 156	⬠ 200	⬠ 178	⬠ 165	⬠ 174		⬠ 180		⬠ 180	⬠ 172	
		3	⬠ 162	⬠ 169	⬠ 172		⬠ 170	⬠ 173	⬠ 167	⬠ 174			⬠ 175	
		4	⬠ 155	⬠ 165	⬠ 177	⬠ 180	⬠ 172	⬠ 204				⬠ 185	⬠ 167	
		5	⬠ 162	⬠ 165	⬠ 180	⬠ 207	⬠ 172	⬠ 204					⬠ 174	
		6	⬠ 154	⬠ 164			⬠ 177	⬠ 215	⬠ 175	⬠ 180			⬠ 176	
		7	⬠ 161	⬠ 166			⬠ 180			⬠ 176			⬠ 179	
	4000	0	⬠ 161	⬠ 161	⬠ 180	⬠ 170	⬠ 170	⬠ 168		⬠ 175			⬠ 166	
		1	⬠ 167	☆ 155		⬠ 200	⬠ 169	⬠ 195					☆ 155	
		2	⬠ 163	☆ 163			⬠ 175		⬠ 195				⬠ 162	
		3	⬠ 166	⬠ 170					⬠ 190				⬠ 166	
		4	⬠ 160	⬠ 163									⬠ 161	
		5	⬠ 175	⬠ 166									⬠ 169	
		6	⬠ 162	⬠ 166					⬠ 165				⬠ 168	
		7	⬠ 164	⬠ 194						⬠ 175				
Core i7-8700k	1500	0	⬠ 264	⬠ 264										
		1	⬠ 262											
	2000	0	⬠ 230	⬠ 230		⬠ 245	⬠ 240	⬠ 241	⬠ 242					
		1	⬠ 227				⬠ 242		⬠ 250					
		2	⬠ 242											

Table 2: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of -1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		IMUL	AESENC	VANDN*	VAND*	VOR*	VXOR*	VPCLMULQDQ	VPSRAD	VPMAX*	VPCMP*	VSQRTPD	VPADDQ
		5	⬠ 250											
	2500	0	⬠ 204	⬠ 207		⬠ 235	⬠ 230	⬠ 237	⬠ 214		⬠ 230	⬠ 225	⬠ 223	⬠ 227
		1	⬠ 200		⬠ 232		⬠ 229	⬠ 234	⬠ 215					
		2	⬠ 214			⬠ 241	⬠ 240							
		5	⬠ 221											
	3000	0	⬠ 194	⬠ 189	⬠ 227	⬠ 230	⬠ 222	⬠ 223	⬠ 195					
		1	⬠ 193	⬠ 222	⬠ 220	⬠ 220	⬠ 215		⬠ 195					
		2	⬠ 214	⬠ 220		⬠ 235	⬠ 224	⬠ 230						
		3	⬠ 223	⬠ 225			⬠ 235		⬠ 224					
		5	⬠ 214											
	3500	0	⬠ 209	⬠ 187	⬠ 230		⬠ 225	⬠ 228	⬠ 208				⬠ 229	
		1	⬠ 190	⬠ 221			⬠ 222	⬠ 223	⬠ 202				⬠ 230	
		2	⬠ 223	⬠ 220									⬠ 240	
		3	⬠ 245										⬠ 245	
		5	⬠ 244											
	3700	0	⬠ 202	⬠ 192			⬠ 225	⬠ 229				⬠ 215	⬠ 226	
		1	⬠ 185	⬠ 220			⬠ 218		⬠ 198		⬠ 235		⬠ 220	
		2	⬠ 217	⬠ 230										
		3	⬠ 214								⬠ 240			
		5	⬠ 244											
Xeon E3-1505M	800	0	⬠ 277											
		1				⬠ 282								
	1500	0	⬠ 202		⬠ 222	⬠ 221	⬠ 219	⬠ 220						
		1	⬠ 212		⬠ 210	⬠ 213	⬠ 210	□ 211						
		2	⬠ 215		□ 217	□ 217	⬠ 209	□ 209						
		3	⬠ 208		□ 212	⬠ 215	□ 209	□ 209						
	2000	0	⬠ 190		⬠ 209	⬠ 211	⬠ 208	⬠ 207						
		1	⬠ 200		⬠ 200	⬠ 200	□ 195	⬠ 195						
		2	⬠ 205		⬠ 208	⬠ 207	□ 203	⬠ 205						
		3	⬠ 197		⬠ 200	⬠ 201	□ 195	□ 195						
	2500	0	⬠ 160				⬠ 175	⬠ 175						
		1					□ 159	□ 160						
		2	⬠ 165					⬠ 170						
		3	⬠ 160		⬠ 169		□ 150	□ 160						
	3000	0	⬠ 172											
		1					□ 154	□ 155						
		2	⬠ 165											
		3				⬠ 173	□ 160	⬠ 159						
	3290	3					□ 158							
	3300	0	⬠ 175											
		1					□ 148	⬠ 154						
		2	⬠ 160											
		3	⬠ 170		⬠ 165		□ 156	⬠ 160						
Core i7-6700K (1)	2000	2	⬠ 249											
		3	⬠ 241											
	2500	3	⬠ 242											
Core i7-6700K (2)			No Faults Found											