Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments

FRITZ ALDER and JO VAN BULCK, imec-DistriNet, KU Leuven
JESSE SPIELMAN and DAVID OSWALD, University of Birmingham
FRANK PIESSENS, imec-DistriNet, KU Leuven

This article analyzes a previously overlooked attack surface that allows unprivileged adversaries to impact floating-point computations in enclaves through the Application Binary Interface (ABI). In a comprehensive study across 7 industry-standard and research enclave shielding runtimes for Intel Software Guard Extensions (SGX), we show that control and state registers of the x87 Floating-Point Unit (FPU) and Intel Streaming SIMD Extensions are not always properly sanitized on enclave entry. We furthermore show that this attack goes beyond the x86 architecture and can also affect RISC-V enclaves. Focusing on SGX, we abuse the adversary's control over precision and rounding modes as an ABI fault injection primitive to corrupt enclaved floating-point operations. Our analysis reveals that this is especially relevant for applications that use the older x87 FPU, which is still under certain conditions used by modern compilers. We exemplify the potential impact of ABI quality-degradation attacks for enclaved machine learning and for the SPEC benchmarks. We then explore the impact on confidentiality, showing that control over exception masks can be abused as a controlled channel to recover enclaved multiplication operands. Our findings, affecting 5 of 7 studied SGX runtimes and one RISC-V runtime, demonstrate the challenges of implementing high-assurance trusted execution across computing architectures.

CCS Concepts: • **Security and privacy** → **Systems security**; **Operating systems security**; **Side-channel analysis and countermeasures;**

Additional Key Words and Phrases: Trusted execution, Intel SGX, FPU, ABI, side channels

ACM Reference format:
Fritz Alder, Jo Van Bulck, Jesse Spielman, David Oswald, and Frank Piessens. 2022. Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments. *Digit. Threat.: Res. Pract.* 3, 2, Article 13 (February 2022), 26 pages.
https://doi.org/10.1145/3491264

1 INTRODUCTION

In recent years, several **Trusted Execution Environments** (**TEEs**) [36] have been developed as a new security paradigm to provide a hardware-backed approach of securing software. Their promise is that applications can

This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/S030867/1, EP/V000454/1, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM), by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by gifts from Intel Corporation. The results feed into DsbDtech. Fritz Alder and Jo Van Bulck are supported by a grant of the Research Foundation – Flanders (FWO). The authors would also like to thank Wei Sim, whose MSc project supported some of the work in Section 4.

Authors' addresses: F. Alder, J. Van Bulck, and F. Piessens, imec-DistriNet, KU Leuven, Celestijnenlaan 200A – bus 2402, B-3001, Belgium; emails: fritz.alder@acm.org, {jo.vanbulck, frank.piessens}@cs.kuleuven.be; J. Spielman and D. Oswald, University of Birmingham, Edgbaston; emails: JXS1366@student.bham.ac.uk, d.f.oswald@bham.ac.uk.

2576-5337/2022/02-ART13 $15.00
https://doi.org/10.1145/3491264

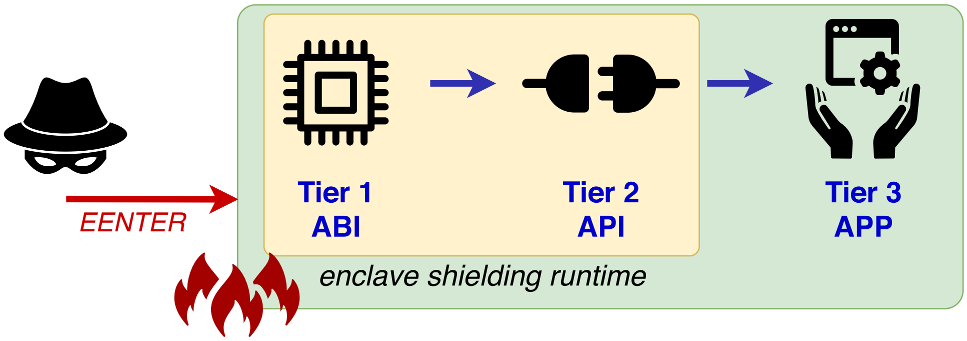

Fig. 1. Enclaved application binaries are transparently shielded by sanitizing untrusted ABI and API-level state.

be run in so called *enclaves* to be isolated and protected from the surrounding, potentially untrusted **Operating System** (**OS**). This allows for a radical reduction of the size of the **Trusted Computing Base** (**TCB**) to the point where only the enclave application itself and the underlying processor need to be trusted. TEEs hence offer the compelling potential of securely offloading sensitive computations to untrusted remote platforms [6, 24, 37]. However, the isolation guarantees provided by any TEE only hold in so far as the trusted in-enclave software properly scrutinizes the untrusted interface that is exposed to a potentially hostile environment. Especially in the context of Intel **Software Guard Extensions** (**SGX**) [14], a state-of-the-art TEE widely available on recent Intel processors, the last years have seen a considerable effort by academia and industry to develop *shielding runtimes* that aid secure enclave development by transparently protecting application binaries inside the TEE. Besides the canonical open-source SGX-SDK [13] reference implementation by Intel, several other mature enclave runtimes have been developed, including Microsoft's OpenEnclave [38], Fortanix's Rust-EDP [18], Graphene-SGX [48], and SGX-LKL [44]. Similarly, shielding runtimes have been developed for TEE architectures beside Intel's SGX, such as for the RISC-V-based Keystone enclave [29] or OP-TEE [42] for ARM TrustZone.

Attacks on enclave shielding runtimes. A recent systematic vulnerability assessment [53] of enclave runtimes has shown that shielding requirements are not sufficiently understood in today's TEE runtimes. Particularly, it was shown that popular SGX shielding systems suffered from a wide range of often subtle, yet crucial interface sanitization oversights. From this analysis, we conclude that the complex enclave shielding responsibility can be broken down into two successive tiers of interface sanitizations, as illustrated in Figure 1. In the first tier, immediately after entering the enclave protection domain, the trusted runtime should sanitize low-level machine state and establish a trustworthy ABI. This bootstrapping phase is typically implemented in a minimal assembly stub that sets up a trusted stack and initializes selected CPU registers before calling second-stage code written in a higher-level language. At this point, the trusted shielding runtime is responsible for providing a secure **Application Programming Interface** (**API**) abstraction by sanitizing untrusted arguments, such as pointers, before finally handing over control to the shielded application binary written by the enclave developer. Any sanitization oversight in either of the phases of the trusted runtime, or in the application tier itself, may nullify all of the enclave's pursued security objectives.

This is especially apparent for a long line of confused-deputy enclave attacks [9, 28, 43, 53] that abuse untrusted pointer passing in the shared address space to trick a victim enclave program into inadvertently dereferencing secure memory locations chosen by the attacker. Such API-level pointer sanitization vulnerabilities have been widely studied, both in the context of conventional user-to-kernel exploits [11] and more recently in TEE scenarios [9, 28, 35, 43, 53]. However, as these vulnerabilities fully manifest at the programmer-visible API level, principled solutions have been developed to thwart this category of pointer poisoning attacks, e.g., by means of developer annotations and automatic code generation as in Intel's `edger8r` [13], a secure type system as in

Fortanix's Rust-EDP [18], or by automatically scrutinizing the enclave API through symbolic execution [28], and even formal interface verification efforts [55, 57]. Furthermore, prior work exists to analyze enclave code via symbolic execution in order to reason about API-level attack surfaces [12]. Another example of insufficient API-level sanitization is the lack of scrubbing in uninitialized structure padding reported by Lee and Kim [31], causing leakage of confidental data from enclave memory.

ABI-level attacks. We argue that ABI-level vulnerabilities, on the other hand, are generally more subtle and harder to reason about as they do not manifest at the program level, but instead exploit implicit assumptions made by the compiler regarding the integrity of the low-level machine state, which may not always hold in the enclave's hostile environment. Due to their low-level nature, this class of ABI-level vulnerabilities hence falls explicitly out of the scope of established language-level security mechanisms like memory-safe type systems. Prior work [16, 53] has for instance exploited improper stack pointer initialization or insufficient sanitization of x86 flags to induce severe memory-safety issues in otherwise perfectly secure applications. It remains unclear, however, whether other ABI-level attack surfaces exist, to what extent they endanger the enclave protection model, and if they are limited to triggering evident memory-safety misbehavior or could also induce more indirect and stealthier errors in enclaved computations.

In this article, we analyze a subtle and previously overlooked ABI-level attack surface arising from enclave interactions with the processor's underlying FPU and SSE vector extensions. Specifically, we show that insufficient FPU and SSE control register initialization at the enclave boundary allows to adversely impact the integrity, and to a certain extent even the confidentiality, of enclaved floating-point operations executing under the protection of a TEE. Our analysis of this attack surface in popular Intel SGX shielding runtimes revealed re-occurring ABI-level sanitization oversights in five different runtimes, including widely deployed production-quality implementations such as Intel's SGX-SDK [13], Microsoft's OpenEnclave [38], and Fortanix's Rust-EDP [18]. Furthermore, an analysis of the ARM and RISC-V reduced instruction set architectures shows that this attack surface is not limited to the notoriously complex x86 instruction set architecture. Specifically, while the OP-TEE [42] runtime for ARM TrustZone properly sanitizes the FPU, we were able to reproduce the attack also in the Keystone runtime [29] on RISC-V. This lack of secure FPU initialization allows unprivileged adversaries to influence the rounding and possibly even the precision of enclaved floating-point operations, introduce indefinite values, and mask or unmask selected floating-point exception types. Interestingly, in contrast to prior research [16, 53] on ABI-level attacks, which induce direct memory corruptions in the victim program, uninitialized FPU and SSE configuration registers pose a significantly less straightforward threat and necessitate more insightful exploitation methodologies. Our work therefore contributes novel attack techniques that abuse the adversary's control over FPU state from two complementary angles.

First, we explore the use of rounding and precision control poisoning as an "ABI-level fault-injection" primitive to silently corrupt supposedly secure enclaved floating-point operations. In several case studies that mainly focus on the widely available Intel SGX-TEE, we show that such subtle floating-point corruptions can break the overall security objective of enclaved applications that operate as a service in an untrusted cloud environment, without ever breaking confidentiality. This threat is especially relevant for legacy applications that employ the x87 FPU, which can be maliciously downgraded from 64-bit double-extended precision to a mere 24-bit single precision mode. We illustrate that such attacks on the x87 FPU can lead to persistent misclassification in an exemplary enclaved image recognition neural network, as well as subtle, yet visible quality-degradation artifacts in 3D rendering algorithms. To the best of our knowledge, these case studies for the first time explore a new and stealthy class of *integrity-only* attacks that purposefully disturb the end result of outsourced enclave computations without ever breaching confidentiality, thus potentially defeating even severely reduced "transparent enclave execution" paradigms [47]. This perspective represents a notable change in direction compared to prior TEE attack research, which has so far only focused on abusing enclaved execution integrity flaws as a stepping stone to ultimately breach confidentiality, e.g., through memory-safety misbehavior [7, 30, 53], undervolting [41], or incorrect transient-execution paths [10, 51, 52]. By contrast, our work shows that, even when the processed data are not considered secret and the enclave binary is free from any application-level vulnerabilities, current widely used shielding systems cannot always safeguard the correctness of outsourced computation results.

Controlled-channel attacks. In a second and complementary angle, we explore the impact of ABI poisoning on the confidentiality of enclaved floating-point operations by showing that attacker-induced FPU or SSE exceptions can be abused as an innovative new type of controlled-channel attack [62]. Using this technique, we show that attackers can deterministically detect the occurrence of x87 instructions in secret-dependent code paths and may even partially reconstruct SSE operand values in straight-line code.

Specifically, in cases where an enclave multiplies a user-controlled input with a secret learned parameter, such as the weights in a neural network, attackers may partially reconstruct the secret multiplier by forcefully enabling floating-point exceptions before entering the victim enclave and abusing the mere occurrence or absence of a subsequent "denormal operand" exception for a carefully chosen input as an unconventional side channel. This technique is closely related to a powerful class of controlled-channel attacks that have previously abused side-channel leakage from x86 CPU exception events to spy on memory addresses accessed by a victim Intel SGX enclave through either page faults [62], segmentation faults [23], or alignment-check exceptions [53]. Our ABI-level attacks, on the other hand, directly reconstruct full data operand values for selected floating-point operations, and, hence, for the first time extend the threat of controlled-channel attacks beyond leaking address-related metadata for memory operations.

Our contributions. In summary, we make the following main contributions:

- A novel ABI-level fault-injection attack that allows unprivileged adversaries to influence the precision, rounding, and exception behavior of x87 or SSE floating-point operations in at least five popular Intel SGX enclave shielding runtimes and at least one RISC-V enclave shielding runtime.
- An innovative controlled channel that abuses floating-point exceptions to recover enclaved multiplication operands, including a proof-of-concept of weight extraction from enclaved neural networks.
- An exploration of a new class of quality-degradation attacks that stealthily compromise the integrity of supposedly secure outsourced enclave computation results.
- A demonstration of practical FPU attacks in an end-to-end machine learning case study enclave and a larger analysis of attacker-induced floating-point errors on the SPEC suite.

Finally, we formulate recommendations for principled ABI sanitization and we argue that this attack surface is non-trivial to patch. Specifically, our analysis revealed insufficient FPU sanitization patches in two production-quality runtimes [18, 38] that were explicitly aware of this attack surface. We show that, despite the initial patches for these runtimes, it was still possible for ABI-level unprivileged attackers to silently override the outcome of trusted in-enclave x87 computations with indefinite NaN outcomes.

Responsible disclosure. The main security vulnerabilities exploited in this work have been assigned CVE-2020-0561 by Intel, for the sanitization oversight in the Intel SGX-SDK, and CVE-2020-15107 by Microsoft, for the remaining attack surface after the initial mitigation attempt in OpenEnclave. While the initial mitigation attempt in OpenEnclave served as inspiration for our work, both the issue in the Intel SGX-SDK and the remediation of insufficient patches were then responsibly disclosed through the proper channels for the affected production runtimes. Intel, Microsoft, Fortanix, and Go-TEE acknowledged the issue and applied our recommended patches in the enclave entry code for the SGX-SDK v2.8, OpenEnclave v0.10.0, and the Rust compiler v1.46.0, respectively. We provide our case studies and proof-of-concept exploits as open-source artifact for other researchers to independently evaluate and build upon our findings.[1]

[1] https://github.com/fritzalder/faulty-point-unit.

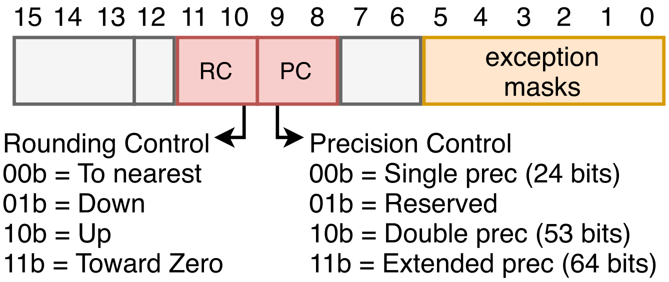

Fig. 2. Layout of the x87 FPU control word.

2 BACKGROUND

This section introduces the necessary background on SGX enclaves and Intel processor support for floating-point computations through the x87 FPU and SSE vector extensions, respectively. We also briefly introduce the necessary background for RISC-V and ARM-based TEEs.

2.1 Intel SGX

Intel SGX [14, 26], are a set of hardware instructions that allow to create trusted regions of code called *enclaves* that are shielded from the surrounding, potentially untrusted **Operating System** (**OS**). The SGX promise is that enclave applications can access almost all capabilities of the user-mode x86 instruction set, while at the same time being provided with strong hardware-backed memory isolation and the capability of attesting code to remote parties. SGX protects enclave memory from outside access and provides instructions to enter and exit enclave mode. When encountering exceptions or interrupts during enclaved execution, the CPU securely saves and scrubs the full extended register set inside the enclave, to be later restored when the enclave is resumed. However, on initial enclave entry into registered call gates, named `ecalls`, the cleansing and sanitization of registers is the responsibility of the software. Due to this challenge, multiple enclave shielding runtimes (cf. Figure 1) have emerged that take over this sanitization on enclave entry, bring the processor into a clean state, and then forward execution to the intended application binary inside the enclave. This not only lowers application developer effort to adopt enclaved execution but also streamlines the mitigation of vulnerabilities on ABI-level. While a 64-bit operation is the norm for SGX enclaves, a 32-bit compatibility mode is officially supported.

2.2 x87 FPU

The x87 FPU [26] provides an environment to perform floating-point and other math operations. For this, the x87 FPU has eight 80-bit data registers that are used internally as a register stack during computation of FPU instructions. The 80 bits in the registers are designed to ensure a high precision inside the FPU to minimize floating-point errors of data that is returned back from the data registers to memory. With 1 bit used for the sign and 14 bits used for the exponent, one 80-bit register utilizes 64 bits to store the significand of a floating-point variable which Intel calls *double-extended precision*. The internal data registers of the x87 FPU by default utilize the full 64 bits of the significand during computations. In addition, the x87 FPU also contains a control register that can be set with the *FPU Control Word* as shown in Figure 2. This control register allows to specify two additional precision formats, namely *double precision* with 53 bits used for the significand and *single precision* with only 24 bits for the significand. These additional precision modes enable compatibility with the IEEE Standard 754 and legacy programs or older programming languages.

Besides limited precision, another important aspect of floating-point operations is the rounding mode. Whenever a floating-point number can not be represented exactly with the given precision, the FPU needs to make a decision whether to choose the next higher or next lower possible representation. By default the x87 FPU will

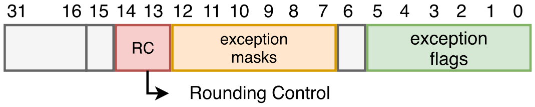

Fig. 3. Layout of the MXCSR control/status register.

round to the nearest value, but developers can choose to override this in the control word and enforce *rounding up*, *rounding down*, or *rounding toward zero*. Naturally, the impact of the rounding mode is greater for computations in single-precision mode than for computations in double-extended precision as rounding errors accumulate faster and the distance between two floating-point numbers that can be represented with the given precision is larger.

Figure 2 shows those fields of the FPU control word that control the behavior of FPU operations in red. These are the **Precision Control** (**PC**) bits 8 and 9, and the **Rounding Control** (**RC**) bits 10 and 11. Fields that control the masking of floating-point exceptions are shown in orange in the figure. Bits 0–5 can be used to mask any of the 6 floating-point exceptions that may be triggered by the x87 FPU. Notable examples of exceptions the FPU might encounter include underflow when a result becomes *subnormal*, also referred to as "denormal", and overflow when the result can no longer be represented in the respective floating-point type. Exceptions are *masked* by default, instructing the FPU to continue with some safe default values. However, in case programmers want to be notified about these events, individual exception types can be unmasked by clearing the respective bits in the FPU control word, e.g., through the C library function `feenableexcept()`. When encountering an unmasked exception, the FPU will stop operation and programmers can register a custom `SIGFPE` signal handler through the OS. Lastly, the remaining non-relevant bits in the FPU control word are marked gray. These are bits 6,7, and 13–15 which are reserved and bit 12, which exists for compatibility reasons and is not meaningful anymore for current versions of the x87 FPU.

Importantly, since the x87 FPU control word defines global program behavior, it is expected by the ABI to be initialized to a predefined sane state `0x37f` that should be preserved across function calls, except for procedures that have the explicit intention of globally changing the FPU configuration [17, 34]. Furthermore, on Intel processors supporting MMX technology [26], the eight x87 floating-point registers can also be utilized as general-purpose MMX vector registers. However, since the MMX registers are internally aliased to the x87 FPU register stack, care should be taken when mixing MMX and x87 instructions. Specifically, any MMX instruction marks the entire x87 stack as in-use and developers are required to issue a special `emms` instruction to clear the register stack before executing any subsequent x87 operation. Failure to do so may produce unexpected results, and compiler ABIs hence demand that "the CPU shall be in x87 mode upon entry to a function" [34].

2.3 Streaming SIMD Extensions (SSE)

In order to further speed up floating-point arithmetics, recent Intel processors include vector extensions that operate independently of the x87 FPU and allow for high performance of parallelized calculations. The line of **Streaming SIMD Extensions** (**SSE**) [26] supports parallel floating-point operations on 128-bit vector registers holding either four 32-bit single-precision or two 64-bit extended-precision floating-point numbers. In contrast to the x87 FPU, which calculates intermediate results with 80 bits of precision, SSE processes a vector of operands in parallel with a fixed (but lower) precision that cannot anymore be dynamically degraded by the developer.

Similar to the x87 control word, SSE offers a global `MXCSR` control register to configure the rounding mode and exception behavior, as shown in Figure 3. The SSE rounding control bits 13–14 (red) and floating-point exception mask bits 7–12 (orange) work identical to those described earlier for the x87 FPU. In addition, `MXCSR` provides status flags 0–5 (green) that indicate whether one of the six floating-point exceptions occurred and configuration bits to specify the behavior when encountering subnormal numbers and underflow conditions. Specifically, bit

15 is called the Flush-To-Zero bit and can be used to enter a mode that flushes the result to zero whenever an underflow is encountered which slightly reduces precision of the calculations for the benefit of increased performance. Bit 6 can be used to enter the Denormals-Are-Zeroes mode that treats all subnormal numbers as zeroes. Neither of these two modes is compatible with the IEEE Standard 754 and both of them are disabled by default [26]. Again similar to the x87 control word, the configuration bits in the global `MXCSR` register are expected by the ABI to be initialized to a pre-defined state `0x3f80` and preserved across function calls [17, 34].

The performance gain of parallelized SSE vector floating-point operations is leveraged by most modern compilers. For example `gcc`, the GNU Compiler Collection, defaults to the SSE when compiling for 64-bit targets [20]. Similarly, Microsoft Visual C++ defaults to the SSE for modern 64-bit applications [39]. For compatibility with 32-bit and legacy systems, both compilers also provide options to compile applications without the SSE and with all math operations purely executed by the x87 FPU. In `gcc`, this compiler option is called `-mfpmath=387`. At the same time, the x87 FPU remains fully supported also for modern 64-bit applications and default compilation options. One notable example is the `C` data type `long double` which is defined as "at least as large as the float type, and it may be larger" [20]. Some compilers as such aim to use the maximum available precision for this data type, which means utilizing the full 80-bit precision of the x87 FPU instead of the 64-bit precision provided by the SSE. For example, `gcc` will default to x87 instructions whenever a `long double` variable is involved and will regularly switch data between the FPU and SSE data register stacks if the SSE was utilized by a support library such as `libm`. Furthermore, `gcc` provides an experimental compilation option called `-mfpmath=both` to utilize a combination of SSE and x87 FPU for increased performance beyond just using it for `long double` variables [20]. Overall, the x87 FPU, while not being the default compilation target for all platforms anymore, is still relevant for calculations that require the high precision of `long double` variables or for legacy applications.

2.4 Other Processors Architectures

In addition to x86, we also briefly discuss the handling of floating-point state on two other mainstream architectures, namely, ARM and RISC-V. Together with x86 (and Intel SGX), these represent all widely used processor architectures for implementing TEEs—ARM provides the TrustZone extensions, while RISC-V is used in various research TEEs [5, 29, 59].

RISC-V. RISC-V defines 32 registers for floating-point data [58, Section 11.1] and a control and status register for floating-point operations, named the `fcsr` register [58, Section 11.2]. This register contains two main pieces: `frm` which globally controls the utilized rounding mode, and `fflags,` which indicates the accumulated floating-point exceptions since this register was last cleared.

Rounding modes in RISC-V are mainly controlled through three `rm` bits encoded into each floating-point instruction. This allows to set the rounding mode on a per-instruction basis if necessary. In accordance with the C99 standard however, RISC-V also provides the global `frm` rounding mode setting in the `fcsr` register—similar to the rounding mode settings in x87 and SSE. Instructions can either specify their own rounding mode or specify the `DYN` rounding mode that defaults to the global parameters in the `frm` setting. In general, this makes ABI-level attacks possible also for RISC-V architectures. We demonstrate the potential impact of this in Section 3.2.

Floating-point exceptions in the default RISC-V specification however do not result in an abortion of the current execution flow as they are not handled by an exception handler, also called trap handler in RISC-V [58, Section 11.2]. Instead, floating-point exceptions are marked in the `fflags` register and detecting that such an exception occurred is purely the responsibility of software. This means that it is the software responsibility to check the `fflags` register after utilization of the FPU and, importantly, also clear both the `fcsr` register and all data registers that contain floating-point data. If an enclave does not clear the `fflags` before returning control to the attacker, then the attacker can use the state of these flags as a side channel similar to the controlled channel case study we describe in Section 4 for Intel SGX. For example, with the default RISC-V specification and a default compilation with `gcc`, this side channel remains open to an attacker.

When an enclave developer uses the gcc flags -fp-trap=all, a check for floating-point exceptions is added after each floating-point calculation and a trap inside the enclave is executed. Even though the trap is not passed to the untrusted code, an attacker might still be able to determine whether an exception has occurred, e.g., from the timing behaviour or error messages. Note that the enclave shielding runtime also still needs to sanitize the fcsr and FPU registers on every context switch to prevent the attacker from gaining information on confidential computations inside the enclave.

ARM. Similar to the x86 architecture, ARM exposes status and control information through the ABI [3, Section A1.5]. In 32-bit mode (AArch32), the FPCSR combines status flags (e.g., zero, carry) and control flags (e.g., rounding mode) [4]. In 64-bit mode (AArch64), control and status information have been split into two registers, FPSR and FPCR. In both cases, the rounding mode can be configured in a similar way to x87 and SSE, and floating-point exceptions can be masked through certain bits in the control register. While we do not further consider ARM processors in the following, we note that the TEE runtime OP-TEE for TrustZone appropriately handles the floating-point state when switching between untrusted and trusted code.[2] Still, this fact highlights that ABI-level attacks are a concern beyond Intel architectures.

3 POISONING FPU STATE REGISTERS

This section first elaborates on the assumed attacker capabilities and system model. Thereafter, we analyze the different attack avenues that may arise in case of insufficient ABI-level sanitization, and we provide a toy example that illustrates their impact on the integrity of exemplary enclave computations. Finally, we conclude with a systematic vulnerability assessment of this attack surface across 7 widely used SGX shielding runtimes.

3.1 Attacker and System Model

We assume the standard Intel SGX threat model [14] where only the processor and the software executing inside the enclave are to be trusted. Notably, while Intel SGX explicitly excludes the OS from the TCB and aims at protecting even against adversaries who have gained root access to the target platform [54], we demonstrate our exploits with a considerably weaker attacker model. Particularly, we only assume user-space code execution in the untrusted host application so as to invoke the enclave with custom ABI-level register settings and to optionally install signal handlers via the OS interface. This falls within the capabilities of any unprivileged user who has access to the enclave binary.

Following widespread industry practice [6, 8, 18, 21, 25, 38, 44, 49], we assume the use of a shielding runtime that intervenes on enclave entry and exit to transparently protect the enclaved application binary from its untrusted environment. Specifically, we consider the explicit security objective of the shielding runtime to be to *(i)* make sure that an enclaved application behaves exactly like on a trusted OS, and *(ii)* prevent any avoidable information leakage beyond what is allowed through explicit interaction with the application. As an example of the first requirement, previous research has shown that the shielding runtime should clear the direction flag in the x86 status register on enclave entry to avoid unexpected memory corruption for string operations [53]. As an example of the second requirement, runtimes should scrub low-level CPU registers that do not form part of the calling convention before exiting the enclave to avoid leaking intermediary state [53].

We assume that the Intel SGX TEE is properly patched against microarchitectural vulnerabilities [10, 51, 52], such that the shielding system can provide enclaved computation results to remote parties as if they were executed on a trusted OS. In this respect, we consider it to be the objective of the shielding runtime to transparently protect *any* ABI-compliant x86 application binary. The latter can include legacy libraries and can be generated by an arbitrary compiler, as long as ABI-level calling conventions [17] are respected, that can hence make use of the full power of the x86 instruction set permitted inside SGX enclaves. In some of our case studies, only when

[2] https://github.com/OP-TEE/optee_os/blob/adb776/core/arch/arm/kernel/thread.c#L1312.

explicitly mentioned, we may emphasize this point by instrumenting the compiler to make increased use of the x87 FPU instead of more modern SSE features by means of the `-mfpmath=387` `gcc` compiler flag. It should be stressed; however, that the resulting application binaries remain fully legit ABI-compliant x86 code that may for instance also have been generated by older or more specialized compilers [20].

3.2 ABI Poisoning Attacks

While trusted code can be relied on to respect ABI calling conventions [17, 34], this does not hold anymore for `ecall` functions exposed to the untrusted world. The shielding runtime hence has the crucial responsibility to bridge this trust semantics gap by sanitizing the ABI on enclave entry. Before showing in Section 3.3 that this requirement is not sufficiently understood in today's widely used SGX shielding runtimes, we first elaborate below on what are the exact security implications of insufficient initialization of x87 and SSE registers, respectively.

Poisoning x87 FPU state. When the shielding system does not cleanse the x87 control word, attackers may execute the unprivileged `fldcw` instruction before entering the enclave to control all bits described in Section 2.2 and Figure 2. In fact, executing this instruction at any point before entering the enclave suffices to successfully implement the attack as long as the x87 control word does not get modified in-between. Since programs rarely modify the x87 control word as long as they are not performing floating-point operations, the attack may often be performed in advance instead of right before the actual ecall. In the following, we assume however that the attacker loads the desired x87 control word as the last instruction before switching into the enclave, which ensures that the x87 control register is in the desired state. The immediately obvious impactful fields the attacker can target are bits 8–9 to degrade the precision and bits 10–11 to alter the rounding mode for enclaved x87 floating-point operations. We will show in Sections 5 and 6 that the impact of a maliciously downgraded x87 precision can be especially devastating in larger applications. Additionally, by selectively unmasking floating-point exceptions and registering a signal handler, attackers may be informed of certain, possibly secret-dependent, FPU events that would otherwise pass unnoticed.

Furthermore, when the shielding runtime does not explicitly initialize the x87 register stack, it may be incorrectly left in MMX mode. For this, it suffices that the attacker executes any MMX operation that is not followed by an `emms` instruction before entering the enclave. Since an ABI-compliant enclave application expects the CPU to be in x87 mode with all registers available, any following attempt to load data into an x87 register will cause an unexpected FPU register stack overflow event, as the CPU still is incorrectly in MMX mode with all eight floating-point registers marked as in-use. The exact behavior in this case will depend on the corresponding exception mask bit in the FPU control word. In the default case where exceptions are masked, the processor will silently replace the intended x87 destination register with an indefinite value (`NaN`) and continue execution. We experimentally confirmed that such attacker-injected unintended `NaN` values are silently propagated further, which is a clear violation of computational integrity and may further cause unexpected or incorrect behavior depending on the victim application.

Alternatively, in the case where exception bits in the x87 control word are craftily unmasked before enclave entry, the attacker will be notified by means of an FPU exception signal whenever the enclave loads an x87 register. This technique is somewhat similar to prior controlled-channel attacks on Intel SGX, which have abused memory contention through page-fault exceptions [62] to spy on enclave-private page accesses. Essentially, by adversely filling the FPU register stack with MMX instructions before enclave entry, the attacker causes unexpected contention that can be used as side channel to learn subsequent use of the FPU by the enclave. We experimentally verified that this technique can be abused as an innovative controlled channel to deterministically recognize x87 instructions in a secret-dependent code path. We note that privileged attackers could further improve the temporal resolution of this novel FPU controlled channel by relying on the SGX-Step [54] enclave execution control framework to exactly pinpoint on which instruction the exception has been raised. SGX-Step leverages carefully scheduled timer device interrupts and has been shown to deterministically

Table 1. Proof-of-concept Attack Executed Inside an Enclave

FPU	Rounding	`arccos(-1)` $= \pi$	`2.1 * 3.4 = 7.14`
Single precision	To nearest	3.1415926535897932385128089	7.1399998664855957031250000
	Downward	3.1415926535897932382959685	7.1399998664855957031250000
	Upward	3.1415926535897932385128089	7.1400003433227539062500000
	To zero	3.1415926535897932382959685	7.1399998664855957031250000
Double precision	To nearest	3.1415926535897932385128089	7.1399999999999996802557689
	Downward	3.1415926535897932382959685	7.1399999999999996802557689
	Upward	3.1415926535897932385128089	7.1400000000000005684341886
	To zero	3.1415926535897932382959685	7.1399999999999996802557689
Extended precision	To nearest	3.1415926535897932385128089	7.1400000000000001156713613
	Downward	3.1415926535897932382959685	7.1400000000000001152376805
	Upward	3.1415926535897932385128089	7.1400000000000001156713613
	To zero	3.1415926535897932382959685	7.1400000000000001152376805
MMX	Any	-NaN	-NaN

advance production enclaves exactly one instruction at a time [40, 54]. FPU poisoning adversaries can, hence, precisely establish the relative instruction offset of enclaved x87 operations by merely counting the number of SGX-Step interrupts before detecting the FPU exception signal.

We finally note that the above x87 FPU poisoning attacks can even impact programs that were never explicitly compiled as x87 FPU programs. Section 2.3 indeed explained that some compilers, including gcc, still utilize the x87 FPU in certain scenarios such as for `long double` data types.

Poisoning SSE state. Compared to the x87 FPU, the more recent SSE floating-point extensions include less configuration bits and hence also expose a smaller ABI-level attack surface. However, we found that when the shielding system does not sanitize the control bits in the `MXCSR` register, attackers may execute the unprivileged `ldmxcsr` instruction before entering the enclave to control all bits described in Section 2.3 and Figure 3. Similar to the FPU attacks described above, this allows the attacker to maliciously alter the in-enclave rounding mode through bits 13–14 and to arbitrarily unmask floating-point exceptions through bits 7–12. Unlike the x87 FPU, the precision of SSE floating-point operations is fixed and can hence not be overridden by the attacker.

We demonstrate below that poisoning the SSE rounding mode may adversely impact the integrity (i.e., the expected outcome) of certain in-enclave floating-point computations. Section 4 furthermore introduces a case study which exploits the adversary's control over the denormal-operand SSE exception mask as an innovative controlled channel to reconstruct secret in-enclave multiplication operands.

A toy example. We exemplify the threat of ABI-level poisoning attacks on the integrity of enclaved floating-point computations by means of two types of math operations: one complex operation that relies on the standard math library included in the Intel SGX-SDK, and one example of a simple multiplication of two floating-point numbers. The complex example is an approximation of the number π by calculating `arccos(−1)` with the `acosl` function provided by `math.h` and the second example is a calculation of `2.1*3.4`. To achieve a maximum precision, the code utilizes variables of the `long double` type, which the compiler translates to predominantly x87 FPU instructions. For completeness, both the minimal C code and the resulting assembly instructions can be viewed in Appendix A. The enclave was compiled with a recent gcc v7.4.0 with standard compilation flags under Ubuntu 18.04.1 and with the Intel SGX-SDK v2.7.1. All evaluations were performed on an Intel i5-1035G1.

Table 1 shows the attack in practice by listing the results of an executed enclave with attacker-primed FPU registers before the `ecall` into the enclave. For all depicted values, the FPU, CW, and the `MXCSR` were set to the desired value via the `fldcw` and the `ldmxcsr` instruction, respectively, right before the enclave was entered.

Illustrated are four FPU groups of possible attack modes available to an ABI poisoning adversary, with the expected (unpoisoned) default mode highlighted. In the first three FPU groups, the attacker sets the x87 FPU control word to operate in either single-precision, double-precision, or extended-precision mode. These precision modes are then combined with each of the four available rounding modes set in both the FPU control word and the `MXCSR` register to affect the operation of the x87 FPU as well as SSE instructions. The last FPU group targets the MMX mode by marking all x87 registers as in-use, as described above, which always yields `NaN` independent of the rounding mode. For readability, all computation results are listed with a precision of 10^{-30} and cut off after the last digit.

As a first interesting observation, the results of the calculation of π listed in the middle column remain unaffected by the choice of the x87 precision mode. Up to the order of 10^{-19}, the calculated approximation is identical with the actual value of π across all possible x87 precision modes. Only the rounding mode can degrade the precision of this single math library calculation in the order of 10^{-19}. Specifically, the rounding modes to nearest and upward both achieve the baseline precision, while the rounding modes downward and toward zero have a degraded performance. This example shows that even when relying on standard math libraries, the attacker can partly degrade the quality of calculations. At the same time, it is evident that although the compiler relied on the x87 FPU to satisfy the precision requirements of the `long double` data type, the results remain unaffected by the modified precision mode. The reason for this is the fact that the `acosl` library function is internally implemented using SSE instructions, and hence the actual computation is not performed by the x87 FPU in this case. Listing 3 in Appendix A shows that the compiler-generated code transfers the x87 data into the SSE registers and similarly retrieves the data after `acosl` has returned. In summary, the attack surface is somewhat limited whenever the victim code utilizes library functions that are not compiled to x87 instructions.

The capabilities of an attacker that target victim code. which solely relies on x87 calculations; however, can be seen in the right column of Table 1. The right column of the table lists the results of the calculation $2.1 * 3.4$ which is performed without any external libraries and is, as such, by default compiled into pure x87 instructions due to its `long double` data type. Notice that this simple multiplication already experiences a floating-point representation error in the highlighted-base mode, which is an inherent consequence of limited-precision numerical representations. However, the table clearly shows that ABI attackers can significantly magnify the error with several orders of magnitude. While in the default extended-precision mode, the error for our exemplary multiplication lies in the order of 10^{-19}, this error increases to the order of 10^{-16} in double-precision mode and lastly to the order of 10^{-7} in single-precision mode. Observe that for each precision mode, rounding upward yields the next higher floating-point number that can be represented in the given precision, whereas the other three rounding modes yield identical results for this particular example. It is important to note that any successive calculation on the corrupted result in larger applications would be exposed to an ever increasing floating-point error. In this respect, our example also highlights a remarkable discrepancy: while attentive enclave developers would aim at utilizing the maximum available precision and minimize the effects of inherent floating-point imprecisions, the usage of the `long double` data type for this purpose also exposes the enclave to increased attack surface for x87 ABI attackers.

The last row finally shows the impact of the MMX attack that always silently replaces the expected outcome with an incorrect `-NaN` result. As discussed previously, this error results from the x87 FPU not being able to determine a usable floating-point register on the register stack and aborting the calculation.

Poisoning RISC-V FPU state. As mentioned in Section 2.4, similar to SSE/x87, the RISC-V FPU has a global `frm` control of the rounding mode for C99 compatibility. However, individual floating-point instructions can also specify the preferred rounding mode. We verified that the Keystone [29] RISC-V research TEE does not sanitize the state of `frm` on enclave entry. Furthermore, the respective compiler (gcc 10.2.0) emits instructions (e.g., `fmul.d`) that respect the global rounding mode for computations with `double` values. We developed a proof-of-concept (using Keystone's QEMU RISC-V emulation) that performs the multiplication $2.1 * 3.4$ inside

Table 2. Marked Runtimes Were Demonstrated to Not (★) or Only Partially (⯪) Sanitize FPU/SSE State, Whereas Empty Symbols (○) Indicate that the Runtime was Not Vulnerable at the Time of Our Initial Analysis (Nov 2019)

	SGX-SDK*	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Go-TEE	Enarx	Keystone
Exploit	★	⯪	○	★	★	★	○	★
Patch 1	`xrstor`	~~`ldmxcsr/cw`~~	`fxrstor`	–	~~`ldmxcsr/cw`~~	`xrstor`	`xrstor`	–**
Patch 2		`xrstor`			`xrstor`			

When applicable, applied and potentially remediated patches are provided

*Includes derived runtimes such as Apache Teaclave's Rust SGX SDK [45] (formerly Baidu Rust-SGX [57]) and Google's Asylo [22].

**as of April 2021.

a Keystone enclave. The untrusted host sets different rounding modes as for x87/SSE. With this, we reproduced the results of Table 1 (for "double precision" only).

Furthermore, Keystone also does not cleanse the `fflags` status bits that indicate whether floating-point exceptions have been raised: the untrusted host can clear the exception flags, run the enclave, and then test if any exceptions (e.g., underflow or overflow) were asserted due to the enclaved computations. Thus, if the compiler does not explicitly check for and trap on floating-point exceptions (the default for RISC-V gcc), this can be abused as controlled channel, cf. Section 4.

3.3 TEE Runtime Vulnerability Assessment

In order to methodologically assess the prevalence of ABI-level FPU poisoning attack surface in real-world SGX shielding runtimes, we performed a comprehensive vulnerability assessment of the seven open-source projects summarized in Table 2. Our selection was motivated by a recent extensive study [53] of popular Intel SGX shielding runtimes, which we extended with two newer runtimes [8, 21] that were not analyzed before. Particularly, we examined all predominant SGX shielding solutions in use by industry, namely, Intel's SGX-SDK [25], Microsoft's OpenEnclave [38], Fortanix's Rust-EDP [18], and RedHat's Enarx [8], as well as three relevant runtimes that were, at least initially, developed as research projects, namely, Graphene-SGX [48], SGX-LKL [44], and Go-TEE [21]. In addition, as a non-SGX example, we also considered the RISC-V TEE Keystone [29]. This wide selection highlights that our ABI-level vulnerabilities apply to both research and production code, emerging safe languages like Rust and Go as well as traditional unsafe languages like C or C++, and SDK-based secure function interfaces as well as library OS-based system call shielding systems. Furthermore, the discovered vulnerabilities are not unique to x86, but can also emerge in other CPU architectures like RISC-V, albeit to a smaller extent due to the reduced amount of ABI state.

A first conclusion from Table 2 is that prior to October 2019, i.e., before the initial Patch by Microsoft OpenEnclave, *all* 7 SGX runtimes were originally vulnerable to the ABI poisoning attacks described in this work. Indeed, our initial analysis was motivated by a partial ABI hardening patch in OpenEnclave in October 2019, which subsequently appears to have been picked up by Graphene-SGX developers as well. For the remaining runtimes, we then performed our initial analysis in November 2019 where we experimentally demonstrated that the SGX-SDK, Rust-EDP, SGX-LKL, and Go-TEE all similarly lacked any form of FPU or SSE register sanitization. We reported these issues and in the case of the SGX-SDK, this can be tracked via CVE-2020-0561/Intel-SA-00336, which also affects derived runtimes, such as Apache Teaclave's Rust SGX-SDK [45] (formerly Baidu Rust-SGX [57]) and Google's Asylo [22], that build on top of the SGX-SDK.

A second tendency in Table 2 relates to the mitigation strategies applied in the different runtimes. Particularly, following our recommendations for more principled ABI sanitization, Intel responded to our disclosure by patching the shielding runtime with an explicit xrstor instruction that fully initializes the entire processor-extended state on every enclave entry. This is also the mitigation applied by Enarx[3] and Go-TEE. Note that SGX-LKL is depicted in Table 2 as not to sanitize the FPU/SSE state because of their unmaintained assembly entry code into the shielding enclave. However, SGX-LKL has been in a migration process in order to utilize the code base of Microsoft OpenEnclave in favor of self-written assembly stubs. As such, once SGX-LKL is fully migrated to utilize OpenEnclave, it will inherit the mitigations implemented there.

In response to our disclosure, Rust-EDP adopted the original mitigation strategy of OpenEnclave, which merely sanitizes the SSE configuration register and the x87 control word through the ldmxcsr and fldcw instructions, respectively. While this approach appears sufficient at first sight, and avoiding a full xrstor may indeed be motivated from a performance perspective, we make the crucial observation that fldcw does not clear the x87 register stack and hence cannot protect the enclave against the MMX poisoning attack variants described above. Specifically, we experimentally demonstrated that on the initially patched Rust-EDP and OpenEnclave runtimes, we can still forcibly put the processor in MMX mode before entering the enclave and cause the outcome of trusted in-enclave x87 FPU operations to be incorrectly replaced with NaN values, which are further propagated silently and may cause application-specific misbehavior. Hence, while the initial patches in these runtimes do severely reduce the attack surface by cleansing MXCSR and the FPU control word, they fail to fully shield the enclave application binary from our attacks. To fully rule out MMX attack variants as well, the runtime should minimally execute an additional emms instruction to place the FPU in the expected x87 mode. The mitigation implemented by the Graphene developers who used an fxrstor instruction is sufficient to also rule out this followup MMX attack as it cleanses all state related to the FPU, MMX, XMM, and MXCSR registers. However, in light of our findings, we explicitly recommend that shielding runtimes adopt the more principled and future-proof strategy of cleansing the entire processor-extended state through xrstor on every enclave entry. Both OpenEnclave and Rust-EDP acknowledged the remaining attack surface of an insufficient ldmxcsr/cw mitigation, and our recommended full xrstor approach was integrated into their respective projects. Microsoft additionally assigned this followup issue CVE-2020-15107.

Finally, we found and reported[4] the issue in Keystone in April 2021. As Keystone is currently a research prototype and not used in production environments, we included the vulnerability in this article, even though a patch is not available yet. We note that this issue may not be specific to Keystone only, as any alternative enclave runtimes on RISC-V would have to properly sanitize the fcsr register as well. Hence, similar to the situation in the Intel SGX landscape, any additional (closed-source) RISC-V enclave runtimes [33] may be vulnerable to our attacks as well.

4 CASE STUDY: FLOATING-POINT EXCEPTIONS AS A SIDE CHANNEL

Background. Apart from compromising computations, an adversary can also use the FPU state registers to obtain side-channel information about floating-point computations inside SGX enclaves. Notably, this side channel also applies to floating-point operations carried out using the SSE extensions, i.e., with standard compiler settings and without the special requirement to use the x87 FPU. The base for this side channel are the exception mask bits that can be set in the MXCSR register right before entering the enclave and the fact that an attacker can register a custom signal handler for floating-point exceptions (SIGFPE). Crucially, for SGX enclaves, the signal handler is untrusted code. This is similar to other controlled-channel attacks, e.g., attacks based on page faults [62], segmentation faults [23], or alignment-check exceptions [53]. Note that in contrast to user-space code, the exact reason for the exception (e.g., underflow or overflow) is not passed on to the signal handler when

[3] Enarx is an ongoing project, still under active development, which is only included for completeness here. The specific runtime entry sanitization code was committed in March 2020, in completion of a longer-standing documented issue.

[4] https://github.com/keystone-enclave/keystone-sdk/issues/72.

```
void secret_mul(double input) {
    double internal =  secret * input;
    // further computations on internal value ...
}
```

Listing 1. Example enclave code vulnerable to secret extraction through a floating-point exception side channel.

triggered from within SGX. However, we show that this can be overcome by only unmasking one exception at a time and executing the enclave multiple times with the same input operands.

In this section, for the sake of simplicity, we focus on double operands, i.e., the 8-byte IEEE 754 double-precision binary floating-point format [60]. In this case, the smallest normal number is $n_{min} \approx 2.2250738585072014 \cdot 10^{-308}$ (hex 0x0010000000000000), while the largest subnormal is $d_{max} \approx 2.2250738585072009 \cdot 10^{-308}$ (hex 0x000FFFFFFFFFFFFF). Whenever the result of a computation is $\leq d_{max}$, an underflow exception will be triggered. A similar upper bound exists above $n_{max}(\approx 1.7976931348623157 \cdot 10^{308})$ where overflow exceptions will be thrown. As described in the following, forcing the calculation of a denormal number can be used as a side channel to infer one possibly secret operand of an enclaved floating-point computation (in this particular example a multiplication) if the other operand is attacker-controlled.

Attack scenario. For simplicity, we first focus on a single multiplication of two floats `secret * input`, but note that the method can be extended to multiple such multiplications by recovering the secret operand one-by-one. We subsequently also show how our technique can be used to partially recover the weights of an in-enclave neural network implementation.

For our initial proof-of-concept, we created an `ecall` on Intel SGX-SDK v2.7.1, which multiplies a secret value with an input. The `gcc` compiler by defaults generates the SSE instruction `mulsd` for the multiplication in Listing 1. Note that the enclave API does not expose the internal result value to the attacker and we merely focus on the side-channel signal whether an exception was raised or not.

Secret recovery. To recover `secret`, in the first step, we determine if its magnitude is ≤ 1. This can be achieved by passing n_{min} as input: if an underflow exception is raised, $|\text{secret}| < 1$, because the result of the multiplication is less than n_{min}. In the following, we describe an attack for the case that $|\text{secret}| < 1$, but we verified that a similar procedure can be used for the other case where $|\text{secret}| \geq 1$ by leveraging the overflow exception (cf. Algorithm 2 in Section B). Next, knowing that $|\text{secret}| < 1$, we use binary search to gradually approximate the secret. More precisely, the attack proceeds as in Algorithm 1: the input is set to 0.5, and if no underflow occurred, the search continues in the lower half $[0, 0.5]$ and otherwise in the upper half $[0.5, 1]$. This process is repeated until the difference between the upper and lower bound is below an attacker-chosen minimal value `epsilon`.

For our experiments, we set epsilon $= 0.00001 \cdot 10^{-308}$. For this bound, Algorithm 1 requires a fixed number of 1,040 invocations of the ecall to recover a secret operand. We ran this algorithm for 1,000 random, uniformly distributed secrets in the interval $[0, 1[$, and computed the difference between the actual and the recovered secret. The histogram of the error is shown in Figure 4. The maximum observed error was $3.667689888908754 \cdot 10^{-6}$, with the average error being $6.2648851729085662 \cdot 10^{-7}$.

Neural network weight extraction. Extending the previous example, we can leverage this controlled channel to recover multiple enclaved multiplication operands, for example, the weights of a simple neural network. Consider an implementation where the weights of the network are secrets stored securely inside an SGX enclave. The first layer of the network involves multiplying n attacker-controlled inputs x_i with secret floating-point weights w_i, where $f()$ is the activation function and b is the bias, to compute an output z of the layer:

$$z = f\left(b + \sum_{i=1}^{n} x_i * w_i\right).$$

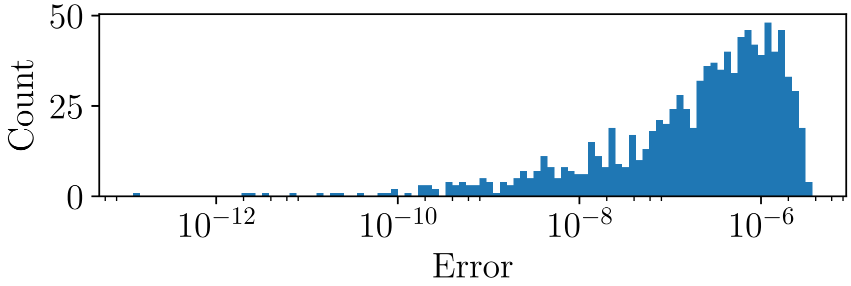

Fig. 4. Histogram over the error of the recovered secret for 1,000 samples (x-axis in log scale).

ALGORITHM 1: Binary search algorithm to recover a secret value based on underflow exceptions for operands <1

```
Result: recovered_secret
low = 0;
high = 1;
while abs(high - low) >= epsilon do
    mid = (low + high) / 2;
    secret_mul(mid);
    recovered_secret = $n_{min}$ / mid;
    if underflow exception raised then
        // continue search in upper half
        low = mid;
    else
        // continue search in lower half
        high = mid;
    end
end
```

We demonstrate the (partial) extraction of weights for two pre-trained feed-forward neural networks, which both use a version of the Genann Neural Networks Library [61] modified to run inside an SGX enclave. The enclave code includes two simple networks—a network that replicates a binary AND operation (cf. Figure 5) and a classifier based on the iris dataset [15]—with slightly different topologies. The AND network has two inputs, a single hidden layer with two nodes, and a single output node. The iris network has four inputs, a single hidden layer with four nodes, and three outputs corresponding to confidence in the three output classes.

Separately, we developed a userspace program that collects user input, instantiates the enclave, and (via the `ecall` interface) executes secure inference on the network using the provided arguments. This program also registers the floating-point exception handler and exits with a non-zero error code if a floating-point exception is raised within the enclaved code.

An attacker can go input-by-input for the network and execute the binary search procedure downwards (from an overflow state) and upwards (from an underflow state). By monitoring for raised exceptions and scanning in the appropriate direction, the threshold between "exception raised" and "valid calculation" again leaks the hidden operand, i.e., the secret weight. Due to the nature of the two exception sources (underflow and overflow), only the largest and smallest weights can be recovered using this method, as the program exits as soon as the first floating-point exception is raised on the largest or smallest weight, respectively.

Special care must be taken for weights that are less ≤ 1, because the underflow binary search only converges on the nearest order of magnitude and not the true value. The attack is able to adapt to this circumstance by re-running the scan recursively with a lower bound that grows by powers of 10. Using this method, the weights

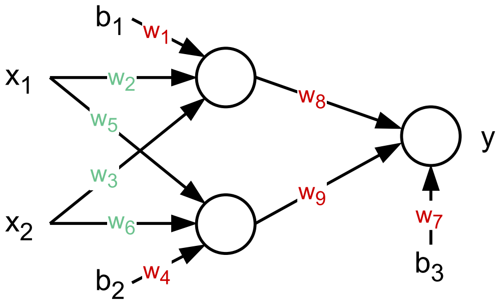

Fig. 5. Structure of the AND network: the green weights are recoverable via our attack, because they are connected to the inputs and there are only two weights per input. Red weights cannot be recovered.

with largest and smallest magnitude in the input layer can be reliably recovered. A proof-of-concept script that leverages the userspace program to perform this attack is included in the paper artifact.

In the case of the AND network, the recovery of the largest and smallest weight between each input and each of the two nodes in the first hidden layer is enough to recover all input weights, as depicted in Figure 5. However, in the case of the iris network, each input is connected to four nodes in the hidden layer, meaning that only eight of 16 input weights are recovered. It is worth noting that even though the precise value of any unrecovered weights remains unknown, it is known that they are all bounded by the largest and smallest weights.

Applicability to real-world models. While stealing an entire network would allow an adversary to perform unlimited inference, it might also allow them to craft adversarial examples, e.g., in the context of spam filtering, or, via a model inversion attack [19], recover training data such as private medical data. Though we demonstrated our attack on a small feed-forward neural network, any topology that directly (without normalization) connects input nodes to hidden layers (such as Recurrent (RNN), Residual (ResNet), or **Long Short Term Memory** (**LTSM**) networks) is vulnerable to this attack. Depending on the complexity of the architecture, the min/max of each input weight may not represent a sizeable percentage of the overall weights in the network though this information could still reduce the task of duplicating or "stealing" a model.

Tramèr et al. [46], propose a so-called "equation-solving attack" for learning the parameters (weights) of a neural network classifier from API outputs, which include a class labels and confidence scores. As our adversary has direct access to the network; however, there is no need to interact via an API and the full output of the model can be obtained directly. Using stochastic gradient decent and a number of queries equal to two times the number of unknown parameters, Tramèr et al. were able to produce a duplicate model which is over 99.8% accurate. By leaking weights using our method, the number of unknowns could be reduced, which would both reduce the time needed to resolve a model and help it converge on a network that is more similar to the original. Alternatively, assuming the architecture and hyperparameters of the model are known (or recoverable [56, 63]), it might be possible to train a new model on a similar set of input data with the recovered weights locked, in effect a shallow form of transfer learning.

Limitations. The attack has certain limitations. First, there is no way to recover the bias weights, because these are not connected to inputs and thus can not be intentionally over or underflowed by providing chosen inputs. Therefore, even if the activation function for a node can be ascertained, it is impossible to estimate that node's output without the bias' contribution, which makes propagating this attack deeper into the network difficult. Another issue is normalization: if the inputs to the network are normalized in any way, it may be impossible to choose the proper inputs to cause overflow and/or underflow exceptions. Finally, the sign of the weight is not recovered; only its magnitude is discovered.

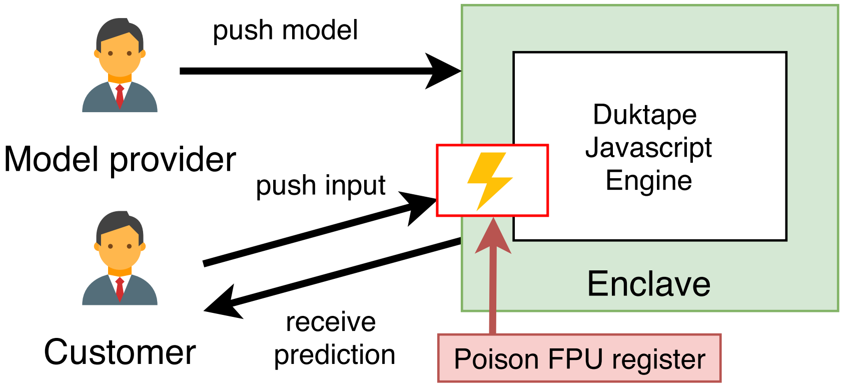

Fig. 6. MLaaS system model with enclaves.

In addition, the position of the recovered largest/smallest weight (in cases where there are more than two per input) remains unknown. However, note that for SGX, the enclave code can be single-stepped [54], which allows to exactly pinpoint on which instruction an exception has been raised. This allows us to determine at which position a recovered weight is located.

5 CASE STUDY: ATTACKING MACHINE LEARNING PREDICTIONS

Background and system model. The core attributes of TEEs are ideally suited for offloading sensitive computations into the cloud. With conventional systems, a sensitive workload needed to either be self-hosted or entrusted to an external cloud provider that is bound by contracts and confidentiality clauses. Both solutions require extensive (legal) planning and are attributed with an increased cost compared to the benefit of conventional cloud computing. When utilizing TEEs, on the other hand, a customer can place her sensitive computation inside an enclave that is executed on the cloud provider's premises. The TEE will guarantee the confidentiality and integrity of the performed workload, while the cloud provider will do his due diligence to achieve a high availability of the paid service to preserve his reputation. Additionally, customers that utilize the service can be ensured that the cloud provider will not learn the potentially confidential inputs or outputs.

Figure 6 illustrates such a TEE-based cloud computing service: A **Machine Learning as a Service (MLaaS)** example of a model provider who gives paid access to his model to customers. In this case study, we assume that the model provider has spent enough resources on the training of the model to make a direct access of customers to the model undesirable. The model provider is assumed to train the model in a trusted setting and then pushes the trained model directly into the enclave that provides the service to customers. Customers then communicate with the enclave and perform evaluations and predictions of their input without learning the machine learning model. Additionally, the enclave can guarantee privacy such that neither the model provider nor the cloud provider learn the customer's input.

We assume that the cloud provider can behave maliciously as long as his actions stay hidden from the model provider and the customer.

Experimental evaluation. We base our case study on earlier work from Alder et al. [1] who placed the Duktape Javascript engine [50] in an Intel SGX enclave and utilized it to provide Machine Learning with the ConvNetJS Javascript library [27]. This setup allows to provide machine- learning predictions from Javascript code executed inside an Intel SGX enclave. We adjust this system to prototype a simple service where a user requests evaluations of her input from a machine- learning model inside the enclave. As a platform for this service, we utilize a standard exemplary convolutional neural network from the ConvNetJS library that classifies

Table 3. MNIST Data Set Predictions with the x87 FPU and with SSE for Different Rounding Modes and Precisions

			Prediction class count (predicted digit)										Average error compared to baseline
	Rounding mode	**Accuracy**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**(SSE, rounding to nearest)**
x87 Single precision	Round to nearest	4%	0	12	14	2	10	32	0	30	0	0	0.176046466527088413256407761764
	Rounding down	8%	0	0	100	0	0	0	0	0	0	0	0.167963971736379585886211884826
	Rounding up	4%	0	12	14	2	10	32	0	30	0	0	0.176046434092910736302073360093
	Round to zero	8%	0	0	100	0	0	0	0	0	0	0	0.167963875521444400140680386357
x87 Extended precision	Round to nearest	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000005544406357383
	Rounding down	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000330733402271493
	Rounding up	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000314522247559579
	Round to zero	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000524157807065445
SSE	Round to nearest	100%	9	14	8	10	14	8	9	14	3	11	0.0
	Rounding down	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000330733402271493
	Rounding up	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000314522247559579
	Round to zero	100%	9	14	8	10	14	8	9	14	3	11	0.000000000000000524157807065445

images of handwritten digits from the MNIST dataset into their machine counterpart of 0– 9. We utilize the demo example to perform the training of a neural network on a trusted machine outside of the enclave and export the trained classifier to be used by our MLaaS enclave to classify future inputs. Such a training step is equivalent to a model provider training the neural network in a trusted environment, as it has not been subject to ABI-level fault injection by our attack yet. With the exported neural network and the ConvNetJS library, the enclave aims at evaluating customer inputs in a trusted environment. Finally, we simulate the customer with repeated requests with MNIST input digits to the enclave and measure the reported class and the reported confidence of the neural network associated with each class. Again, we perform the attack by modifying the FPU, CW, and the MXCSR directly before entering the enclave. To showcase the potential worst-case impacts of our attack, we consider two distinct scenarios with different victim enclave binaries created using Intel SGX-SDK v2.7.1: one binary was generated with default compilation flags and hence uses primarily SSE instructions, whereas the other binary was generated by additionally passing the -mfpmath=387 compilation flag to explicitly instruct gcc to use the x87 FPU for floating-point computations.

Table 3 shows the results of 100 input evaluations for all rounding modes when using the SSE, or the x87 FPU in extended or single-precision mode. Evaluations with the x87 double-precision mode are not shown as we found these results to be identical to runs with the x87 extended-precision mode. All depicted configurations were executed on the same set of inputs to ensure repeatability. For the highlighted baseline scenario, i.e., SSE and the default rounding mode of rounding to the nearest value, the trained model expectedly predicts 100% of the provided digits correctly. When adversely changing rounding modes through the untrusted ABI, small errors in the order of 10^{-16} are clearly introduced. Importantly, however, the results indicate that such small perturbations are insufficient to affect the predicted digit class and the model still holds the same overall accuracy. This observation also holds for the x87 victim enclave binary when utilizing the x87 FPU in extended-precision mode. However, when ABI-level attackers maliciously reduce the FPU to a single-precision mode, the x87 victim enclave binary can interestingly be coerced into one of two roles. When rounding to nearest or rounding up, the trained model will simply have a gravely decreased accuracy with only 4% of the given input classified with the correct digit. Alternatively, when forced to round down or toward zero, the trained model will predict *every* given input as the digit 2, regardless of the actual input. The average error in single-precision mode lies in the range of 10^{-1}, which easily scrambles and rearranges the prediction percentages of each input evaluation.

Discussion. While the overall effectiveness of this attack was shown to heavily depend on the way in which the enclave application was compiled, which may not always be under the control of the attacker, the case study clearly highlights the fallacy of the shielding runtime to protect an ABI-compliant enclaved application binary from its untrusted environment. The results especially underline the threat for larger legacy 32-bit [23] or specialized applications that heavily rely on the x87 FPU, or even just require high precision via the `long double` data type that might get compiled to utilize the x87 FPU. Our example MNIST attack illustrates that, for certain enclaved application binaries, an ABI-level adversary has the potential to inject faults that purposefully and stealthily disrupt the overall security objective of the outsourced application, without needing to break any confidentiality or availability guarantees. Furthermore, this attack can stealthily target specific customers to allow a malicious cloud provider to degrade the neural network performance for specific victims. Such a degradation in performance may for instance allow the adversary to shift the customer's favor greatly toward a competing product or drive away customers from the model provider while the adversary at the same time would have little to no risk of being detected.

6 CASE STUDY: SPEC BENCHMARKS

To evaluate the theoretical impact of our ABI-level fault-injection attacks on larger and more varied applications, we perform a larger-scale synthetic attack evaluation on the SPEC-CPU 2017 benchmark programs outside of Intel-SGX. While it is not straightforwardly possible to run the SPEC benchmark programs inside an SGX enclave, we argue that the induced faults into floating-point computations are independent of the surrounding execution environment and a common benchmark will help to better understand the possible impact of our attacks on an objective baseline computation.

Experimental evaluation. Our experimental setup runs outside Intel SGX and compiles the SPEC suite twice with `gcc` v6.2.0, one time with default settings and one time with an additional `-mfpmath=387` flag to enforce the usage of the x87 FPU for a maximum demonstration of the attack's impact. We then run the *reference* workload of the `fprate` class to generate meaningful evaluation results. The `fprate` class of benchmarks is explicitly designed around floating-point calculations and as such forms a relevant candidate to evaluate the impacts of our attack. It is important to note, that the SPEC benchmark evaluation scripts already account for floating-point errors by allowing a workload-specific error margin before a benchmark is marked as failed. Similar to the previous case studies, we perform the attack by executing `fldcw` and `ldmxcsr` instructions before executing the SPEC benchmarks. As such, the attacker performs the same steps as when attacking enclave code as the execution of the SPEC benchmark can be seen as equivalent to entering the enclave in this respect.

Table 4 shows the benchmarks in the `fprate` class and a marker indicating whether the benchmark succeeded or failed for both the default SSE binary, as well as for the x87 binary in single-precision mode. In the highlighted baseline mode of to-nearest rounding with the SSE, all SPEC benchmarks succeed. When maliciously changing the rounding mode before execution of the SPEC benchmark; however, multiple tests already fail due to a too high accumulation of floating-point errors. Furthermore, when considering a simulated maximum-impact attack on an x87 binary in single-precision mode, the attacker can, depending on the rounding mode, further degrade floating-point computations and cause even more benchmarks to fail. Under this attack, only four benchmarks in to-nearest rounding mode or one benchmark in to-zero rounding mode still succeed.

Discussion. To better understand the nature of the induced floating-point errors, we performed an additional manual analysis of two benchmarks: the *526.blender_r* image rendering benchmark and the *511.povray_r* ray-tracing benchmark.

526.blender_r image rendering. Blender[5] is an open-source content creation suite which includes the entire 3D production pipeline. The `blender` benchmark in Spec 2017 renders a single frame of a scene from a short film. While the `blender` benchmark is designed to be resilient against expected floating-point

[5] https://www.blender.org/.

Table 4. Benchmarks with SPEC-CPU 2017 Under Compilation with the x87 FPU and with the SSE, both Shown for Different Rounding Modes

	Rounding mode	bwaves	cactuBSSN	namd	parest	povray	lbm	wrf	blender	cam4	imagick	nab	fotonik3d	roms	specrand
Single precision	To nearest	✓	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗
	Downward	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
	Upward	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
	To zero	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
SSE	To nearest	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Downward	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗
	Upward	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓	✓	✗	✓	✗
	To zero	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗

Listed are all workloads in the `fprate` test class and their result in the given configuration.

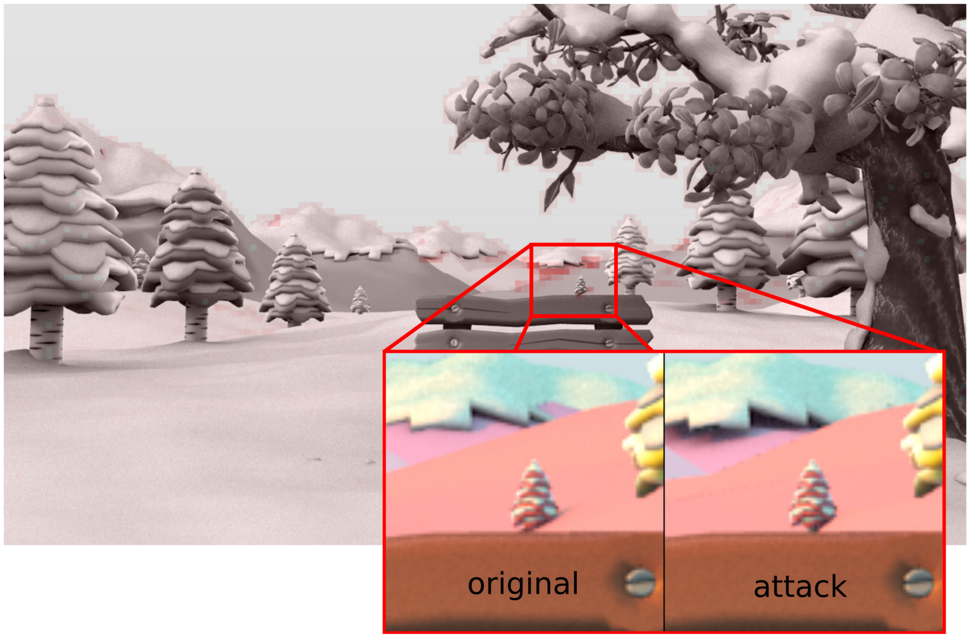

Fig. 7. Composite image of the Blender benchmark in Spec-CPU 2017 under attack by our FPU attacker in x87 single precision mode when rounding toward zero. Areas in red differ from the expected render image with the zoomed-in area showing differences visible to the human eye.

perturbations that do not exceed the internal error threshold, we found that the x87 binary in single-precision mode and with rounding toward zero can lead to subtle-yet-visible quality degradations in the rendered 3D images.

Figure 7 shows an example rendering with the difference between the expected original and an attacked scene marked in shades of red. While most of the scene is colored in a light shade of red that already stands for a small difference between the expected and calculated output, some parts of the screenshot are marked more clearly such as the framed mountain scenery or the hills to its left. In the zoomed in portion of the framed scenery, it can be seen that the expected baseline image (left) shows a tree shadow and snow cover on the mountains. With the attack (right), however, the shadow is missing and the contours of the mountains are lower, making the snow cover appear to float. It is evident that the visual perturbations between the baseline and attacked rendering are small, yet the fact that they are visible even for human observers clearly illustrates the potential impact of insufficient ABI shielding on the integrity of an outsourced enclave rendering service. If these perturbations are

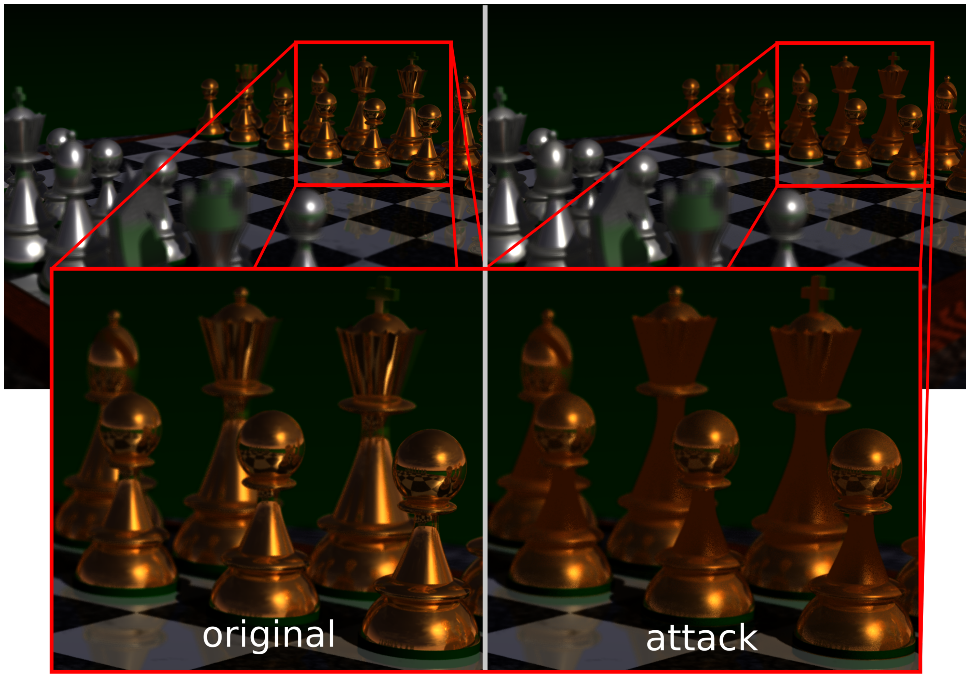

Fig. 8. Composite image of the Povray benchmark in Spec-CPU 2017. The image shows a comparison of the baseline result to the result under attack by our FPU attacker in x87 single precision mode when rounding up. The zoomed-in areas highlight a region where the quality of calculated reflections and raytracing has been visibly degraded.

inserted into a each frame of a sequence of images played back at multiple frames per second, the impact of the degradation are even more noticeable due to irregularities between frames, visible as a flickering effect. Such an attack may for instance be relevant when an untrusted cloud rendering provider has an economic incentive to stealthily degrade the quality of rendered images from a client or when an attacker aims at stealthily inserting quality degradation for monetary gain such as blackmail.

511.povray_r ray-tracing. POV-Ray[6] is an open-source ray-tracing application, which renders 3D images. The `povray` benchmark of SPEC 2017 renders a chess board with realistic reflections of other pieces and of the surrounding environment, including lights. Figure 8 shows a comparison of the original rendered image with default benchmark settings to the attacked scene under influence of a single precision attacker that rounds upwards.

Under the influence of the attack, especially in the zoomed-in portion of the benchmark, it is visible that multiple reflections and specular highlights are non-existent or severely degraded. This is most evident for the chess pieces of the king and queen, but is also visible for the middle part of the pawns. In all these instances, the reflection and highlights are almost completely degraded or lost, making this attack arguably more noticeable to the human eye than the perturbations in the previous benchmark. Similar to the previous blender benchmark, these perturbations will become more noticeable if they are part of a sequence of images played back at a constant frame rate.

From the SPEC analysis, we conclude that common applications may widely fail when unexpectedly interfaced with a malicious ABI and that attacker-induced floating-point errors in larger applications may propagate into

[6] http://www.povray.org/.

subtle corruptions of the expected result. The exact impact of such attacks will always be application-specific, however, and require careful analysis by the attacker depending on the x87 or SSE processor features used in the victim application.

7 CONCLUSIONS AND LESSONS LEARNED

With the wide availability of SGX in mainstream Intel processors, an emerging software ecosystem of enclave shielding runtimes has developed in recent years to ease the adoption process and enable developers to largely transparently enjoy SGX protection guarantees. But despite the considerable advances and developer efforts behind these runtimes, API and ABI-level issues continue to pose a threat to the promise of transparently shielding enclave applications [28, 53].

In this work, we presented novel ABI-level attacks on the largely overlooked x87 FPU and SSE state that allow an unprivileged adversary to impact the integrity of enclaved floating-point operations, in terms of the rounding mode, precision, and silently introduced `NaN` values. We furthermore explored an innovative controlled-channel attack variant that abuses attacker-induced floating-point exceptions to partially breach the confidentiality of otherwise private enclaved floating-point operations. In a comprehensive analysis of this vulnerability space in seven popular Intel SGX runtimes, developed by both academia and industry, we were able to provide a proof-of-concept attack for five of them. Moreover, our analysis revealed that two previously patched production runtimes remained vulnerable to `NaN` injection, further highlighting the intricacy of fully mitigating this ABI-level attack surface. While the eventual impact of our FPU poisoning attacks remains intrinsically application-dependent, we have presented several case studies that illustrate the potential exploitability in selected application binaries.

The fundamental issue can be mitigated by simply setting the x87 FPU control word as well as the SSE `MXCSR` register into known states when entering enclaved execution. Mitigating the followup MMX issue requires an additional `emms` instruction to place the FPU in the expected x87 mode. Regarding more principled mitigation strategies, however, we explicitly recommend that shielding runtimes perform a full `xrstor` to initialize the complete processor-extended state whenever the enclave is entered. Although this may come with a slightly increased cost in performance, we believe that our findings underscore the need for shielding runtimes to move away from selective register cleansing on an ad-hoc case-by-case basis, in order to more systematically prevent any orthogonal ABI-level issues that may arise in current or future processor extensions. Six of the seven investigated enclave shielding runtimes have now opted to perform such a full `xrstor` or in the case of Graphene perform an equivalent `fxrstor`, while SGX-LKL will inherit the `xrstor` mitigation from Microsoft OpenEnclave in the future.

In the wider perspective, we were also able to reproduce the attack for the Keystone TEE on RISC-V, despite its simpler architecture with a reduced instruction set. Our work highlights the challenges of implementing a high-assurance TEE on top of complex instruction set architectures like x86, with arguably too many neglected legacy features and strict backwards compatibility. Counterintuitively, however, our work also highlights that these challenges are not unique to complex instruction set architectures, but that they remain even when utilizing modern reduced instruction set architectures like RISC-V. In the context of floating-point operations, this can be attributed to the C99 convention to treat the FPU state as global and controlled by a number of functions—CPU designs that seek compatibility to C99 are likely to map this into FPU state and control registers.

We argue that, in an era where the research community is increasingly looking into subtle microarchitectural CPU vulnerabilities [10, 32, 51, 52], the strictly architectural attack surface of today's complex processor features remain not sufficiently understood—even if the underlying architectures are using a reduced instruction set. As such, an important avenue for future work is to further extend and apply specialized symbolic execution tools, such as TeeRex [12] or Guardian [2], to safeguard against ABI-level vulnerabilities in enclave runtimes.

APPENDICES

A PROOF-OF-CONCEPT ENCLAVE CODE

This appendix lists the C source code (Listing 2) and compiled assembly (Listing 3) for the benchmark toy example enclave discussed in Section 3.2 and Table 1. The assembly code in Listing 3 was output by gcc v7.4.0 under Ubuntu 18.04.1 and the Intel SGX-SDK v2.7.1 using the default compilation flags.

```
#include <stdint.h>
#include <math.h>

long double ecall_acosf(int a) {
return acosl(a);
}
long double ecall_mul(long double a, long double b) {
return a*b;
}
```

Listing 2. Code to perform basic floating-point operations inside the enclave.

```
<ecall_acosf>:
 push %rbp
 mov %rsp,%rbp
 sub $0x20,%rsp
 mov %edi,-0x4(%rbp)
 fildl -0x4(%rbp)
 lea -0x10(%rsp),%rsp
 fstpt (%rsp)
 callq 4450 <acosl>
 add $0x10,%rsp
 fstpt -0x20(%rbp)
 mov -0x20(%rbp),%rax
 mov -0x18(%rbp),%edx
 mov %rax,-0x20(%rbp)
 mov %edx,-0x18(%rbp)
 fldt -0x20(%rbp)
 leaveq
 retq

<ecall_mul>:
 push %rbp
 mov %rsp,%rbp
 fldt 0x10(%rbp)
 fldt 0x20(%rbp)
 fmulp %st,%st(1)
 pop %rbp
 retq
```

Listing 3. Compiled assembly of Listing 2.

B SEARCH ALGORITHM BASED ON OVERFLOW EXCEPTIONS

This appendix lists the additional Algorithm 2 to recover secrets for operands >1. It functions analogous to Algorithm 1 described in Section 4. We note that for brevity, both Algorithms 1 and 2 use standard floating-point variables for secret recovery. However, if desired, these algorithm could be likely re-written (although in a less clear manner) using the binary representation of the double operands instead.

ALGORITHM 2: Binary search algorithm to recover a secret value based on overflow exceptions for operands >1

```
Result: recovered_secret
// Maximum representable double
max_double = 1.7976931348623157e308;
low = 1;
high = max_double;
cnt = 0;
while cnt < 100 do
    mid = low / 2 + high / 2;
    secret_mul(mid);
    recovered_secret = max_double / mid;
    cnt++;
    if overflow exception raised then
        // continue search in lower half
        high = mid;
    else
        // continue search in upper half
        low = mid;
    end
end
```

REFERENCES

[1] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 185–199.

[2] Pedro Antonino, Wojciech Aleksander Wołoszyn, and AW Roscoe. 2021. Guardian: Symbolic validation of orderliness in SGX enclaves. In *Proceedings of the 2021 on Cloud Computing Security Workshop*. Association for Computing Machinery, 111–123. DOI: 10.1145/3474123.3486755

[3] ARM. 2021. *Arm Architecture Reference Manual Armv8*. ARM DDI 0487G.a. Retrieved November 15, 2021 from https://developer.arm.com/documentation/ddi0487/gb/ ARM DDI: 0487G.a.

[4] ARM. 2021. *FPSCR, the Floating-point Status and Control Register*. Retrieved from https://developer.arm.com/documentation/dui0068/b/Vector-Floating-point-Programming/VFP-system-registers/FPSCR--the-floating-point-status-and-control-register.

[5] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A security architecture with customizable and resilient enclaves. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 267–283.

[7] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *Proceedings of the 27th USENIX Security Symposium*. 1213–1227.

[8] Mike Bursell. 2019. Trust No One, Run Everywhere—Introducing Enarx. Retrieved November 15, 2021 from https://next.redhat.com/2019/08/16/trust-no-one-run-everywhere-introducing-enarx/.

[9] S. Checkoway and H. Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 253–264.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre attacks: Stealing intel secrets from SGX enclaves via speculative execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE.

[11] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 5:1–5:5.

[12] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *Proceedings of the 29th USENIX Security Symposium*. 841–858.

[13] Intel Corporation. 2017. *Intel Software Guard Extensions SDK for Linux OS: Developer Reference*. Retrieved November 15, 2021 from https://download.01.org/intel-sgx/sgx-linux/2.15/docs/.

[14] V. Costan and S. Devadas. 2016. Intel SGX explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[15] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. Retrieved November 15, 2021 from http://archive.ics.uci.edu/ml.

[16] J. Edge. 2008. CVE-2008-1367: Kernel doesn't clear DF for signal handlers. https://bugzilla.redhat.com/show_bug.cgi?id=437312 Retrieved from https://bugzilla.redhat.com/show_bug.cgi?id=437312.

[17] A. Fog. 2018. Calling conventions for different C++ compilers and operating systems. Retrieved November 15, 2021 from http://www.agner.org/optimize/calling_conventions.pdf http://www.agner.org/optimize/calling_conventions.pdf.

[18] Fortanix. 2019. Fortanix Enclave Development Platform — Rust EDP. Retrieved November 15, 2021 from https://edp.fortanix.com/.

[19] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1322–1333.

[20] Free Software Foundation. 2020. GCC, the GNU Compiler Collection. Retrieved November 15, 2021 from https://gcc.gnu.org/.

[21] Adrien Ghosn, James R Larus, and Edouard Bugnion. 2019. Secured routines: Language-based construction of trusted execution environments. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 571–586.

[22] Google. 2019. Asylo: An open and flexible framework for enclave applications. Retrieved November 15, 2021 from https://asylo.dev/.

[23] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*. Springer, 44–60.

[24] IBM. [n.d.]. Data-in-use protection on IBM cloud. Retrieved November 15, 2021 from https://www.ibm.com/blogs/bluemix/2017/12/data-use-protection-ibm-cloud-ibm-intel-fortanix-partner-keep-enterprises-secure-core/.

[25] Intel Corporation. 2019. Intel Software Guard Extensions – Get Started with the SDK. Retrieved November 15, 2021 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[26] Intel Corporation. 2020. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes*. Reference no. 325462-062US. Retrieved November 15, 2021 from https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/get-started.html.

[27] Andrej Karpathy. 2014. Convnetjs: Deep learning in your browser (2014). Retrieved November 15, 2021 from http://cs.stanford.edu/people/karpathy/convnetjs.

[28] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN attacks: On insecurity of enclave untrusted interfaces in SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.

[29] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems*. 1–16.

[30] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Security Symposium*. 523–539.

[31] S. Lee and T. Kim. 2017. Leaking uninitialized secure enclave memory via structure padding. *arXiv preprint* arXiv:1710.09061. Retrieved from https://arxiv.org/abs/1710.09061.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium*. 557–574.

[33] Jingbin Liu, Yu Qin, and Dengguo Feng. 2020. SeRoT: A secure runtime system on trusted execution environments. In *Proceedings of the 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 30–37.

[34] H. J. Lu, David L Kreitzer, Milind Girkar, and Zia Ansari. 2015. System v application binary interface. *Intel386 Architecture Processor Supplement, Version 1.1* (7 December 2015).

[35] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. Ryn Choe, C. Kruegel, and G. Vigna. 2017. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the NDSS 2017*.

[36] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2018. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers* 67, 3 (2018), 361–374. DOI: 10.1109/TC.2017.2647955

[37] Microsoft. 2017. Retrieved November 15, 2021 from https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[38] Microsoft. 2019. Open Enclave SDK. Retrieved November 15, 2021 from https://openenclave.io/sdk/.

[39] Microsoft Corporation. 2020. Microsoft Visual C++. Retrieved November 15, 2021 from https://docs.microsoft.com/en-us/cpp/.

[40] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled instruction-level attacks on enclaves. In *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 469–486.

[41] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41th IEEE Symposium on Security and Privacy*.

[42] OP-TEE c/o Linaro. 2019. Open Portable Trusted Execution Environment. Retrieved November 15, 2021 from https://www.op-tee.org/.

[43] S. Pinto and N. Santos. 2019. Demystifying ARM trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 130.

[44] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. arXiv:1908.11143. Retrieved from https://arxiv.org/abs/1908.11143.

[45] The Apache Software Foundation. 2020. Apache Teaclave (Incubating). Retrieved from https://teaclave.incubator.apache.org/.

[46] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction apis. In *Proceedings of the 25th {USENIX} Security Symposium ({USENIX} Security 16)*. 601–618.

[47] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE.

[48] Chia-Che Tsai, Donald Porter, et al. 2017. Graphene-SGX library OS — A library OS for Linux multi-process applications with Intel SGX support. Retrieved November 15, 2021 from https://github.com/oscarlab/graphene.

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

[50] Sami Vaarala. 2020. Duktape embeddable javascript engine. Retrieved from URL https://duktape.org/. (2020).

[51] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*.

[52] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

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

[54] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 4:1–4:6.

[55] N. van Ginkel, R. Strackx, and F. Piessens. 2017. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. 105–123.

[56] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–52.

[57] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards memory safe enclave programming with rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2333–2350.

[58] Andrew Waterman and Krste Asanovic. 2019. The RISC-V instruction set manual, volume I: Unprivileged ISA, document version 20191213. *RISC-V Foundation* (December 2019).

[59] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-Reza Sadeghi. 2019. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. In *NDSS 2019*.

[60] Wikipedia contributors. 2020. Double-precision floating-point format — Wikipedia, The Free Encyclopedia. Retrieved November 15, 2021 from https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=960696001 [Online; accessed 16-June-2020].

[61] Lewis Van Winkle. [n.d.]. Genann Neural Networks Library. Retrieved November 15, 2021 from https://github.com/codeplea/genann.

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

[63] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. 2020. Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures. In *Proceedings of the 29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2003–2020.

Received May 2021; accepted August 2021