KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Interface Sanitization and Real-Time Scheduling for Enclaved Execution

Fritz Alder

Supervisors:
Prof. dr. ir. F. Piessens
Prof. dr. J.T. Mühlberg
Dr. ir. J. Van Bulck

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

October 2023

Interface Sanitization and Real-Time Scheduling for Enclaved Execution

Fritz ALDER

Examination committee:
Prof. dr. ir. H. Neuckermans, chair
Prof. dr. ir. F. Piessens, supervisor
Prof. dr. J.T. Mühlberg, supervisor
Dr. ir. J. Van Bulck, supervisor
Prof. dr. ir. W. Joosen
Prof. dr. N. Smart
Prof. dr. D. Oswald
(University of Birmingham)
Prof. dr. S. Shinde
(ETH Zurich)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

October 2023

Preface

Writing this preface is both a daunting and exciting conclusion of my four years at KU Leuven. Exciting, because I have been looking forward to doing research and a Ph.D. since starting my master's and since I delved ever deeper into the fascinating rabbit hole of cyber security. A global pandemic roughly six months after I moved to Belgium did not make it easy for me and many others but it at least never deterred me from wanting to do more research. At the same time, writing this preface is daunting because throughout my career I have been surrounded by great, compassionate, and truly intelligent people. Trying to name all of you who have impacted me directly or indirectly, or have helped me along the way is nearly impossible. But I shall try.

First and foremost, I wish to thank my supervisor Frank Piessens, for being simply the best supervisor I could have wished for. It is rare to see a supervisor who not only gives his students full freedom with no questions asked but also manages to always be available and responsive when needed. And somehow you manage to do it while always giving meaningful insights. You keep saying advice is worth what you pay for it and I feel that my debt to you must be immense by now. Sincerely thank you.

I next want to thank Jan Tobias Mühlberg. Not only did you immediately take me in after I sheepishly asked whether there were open positions, but you also gave me the freedom to go fully astray from what we originally planned for this dissertation.

The last person on my supervisory committee is Jo Van Bulck, who partly was the reason for my diversions of the original research trajectory. Thank you for too many things to count here. I really enjoyed working with you for these years and loved our little hack sessions, extended design discussions, disagreements, and late-night conference discussions.

I wish to thank the external as well as the local members of my Ph.D. jury: Herman Neuckermans, Wouter Joosen, Nigel Smart, David Oswald, and Shweta

Shinde. I feel honored to have had all of you on my jury and truly enjoyed our interactions throughout the process. I especially wish to thank David for the years of fruitful and fun collaboration.

Outside of my jury, I first wish to thank my direct collaborators Gianluca Scopelliti, Lesly-Ann Daniel, Jan Pennekamp, Roman Matzutt, Christoph Baumann, and Sepideh Pouyanrad. I thoroughly enjoyed working with you. I am especially happy that Gianluca decided to still work with me after doing his master's thesis with me.

Next, I want to thank my colleagues and friends in and around DistriNet: Job Noorman, Hans Winderix, Márton Bognár, Vera and Snezhok Rimmer, Merve Turhan, Gilang Hamidy, Antoon Purnal, Gerald Budigiri, Tobias Reinhard, Annick Vandijck, An Makowski, Katrien Janssens, and all the other friendly faces that made the office a nice place to be at.

Outside of KU Leuven, I want to thank the people in and around the Secure Systems Group in Finland. Thank you, Asokan and Andrew Paverd, for still taking me into your group after I initially (stupidly) refused your offer, and thank you for the best start I could have wished for in my research career. Thank you also to Ilhan Gurel, Thomas Nyman, Sebastian Szyller, and Koen Tange. Thank you, Lukas Prediger, for the (almost) weekly chats, and thank you to the other members of Group One for persistently being a great exchange family. I also wish to thank Stefan and Christian Teutrine, as well as Anika Werner. Also thank you, Finn, for bringing everyone some joy.

Coming to a close, I wish to thank my family and most importantly, my parents. Love needs constant care and reassessment. And you two have so much of it, you keep radiating that love to everyone you meet. That I keep moving around in the world and you by now know time zones and flight tracker websites by heart is a result of your success as parents, not your fault – I promise! Being so far away so often (and at the worst times) meant that I saw my older brother Florian only a fraction of what I would have wished for these past years. Still, I am immensely grateful to have grown up with you and for the bond we share. Ich bin stolz auf euch drei.

Last, I thank my partner Sonja for so many things. I am continuously amazed and happy to have found you. Thank you for always being there for me and suffering through hundreds of iterations of my slide graphics and endless excited descriptions of weird security topics. I look forward to wherever the road leads us.

— Fritz Alder
October 2023, Leuven

This work was supported by a Ph.D. Fellowship from the Research Foundation — Flanders (FWO) and by the Research Fund KU Leuven.

Abstract

Modern computing is increasingly characterized by an abundance of connectivity between networked devices and a sharing of resources on local devices. While this development has created a range of positive opportunities in terms of productivity and technical capabilities, it also opens up modern systems to security issues that were not as critical in the previously insulated systems. Protecting confidentiality and integrity has thus become an integral concern and isolation mechanisms already enabled a type of computing where programs share their resources with other, entirely untrusted, programs.

One approach to ensure the security on such systems are hardware isolation approaches, such as trusted execution environments (TEEs). TEEs aim to isolate programs and shield them from accesses by any other part of the system that is not within the trusted computing base. Specifically, hardware-based TEEs achieve this by employing modifications to the underlying computing architecture that limit access to specific interactions and deny any other access. One type of TEE protects code in so-called *enclaves* that draw the protection boundary at the program level and usually require coordinated interactions between an untrusted and a trusted program within the same address space.

This dissertation advances the state of the art for this type of TEEs in two directions. First, we investigate availability guarantees on lightweight architectures and equip TEEs for real-time applications. We do this with a hardware-software co-design that places a real-time scheduler inside of an enclave, in order to provide other enclaves with strong availability guarantees. This allows us to combine the approach of openly sharing resources between mutually distrusting parties with the realm of safety-critical devices that must meet real-time deadlines. Our solution can be seen as a first step to apply modern TEE capabilities to the slow-moving but critical area of real-time and mixed-criticality systems on lightweight computing architectures.

Second, this dissertation investigates software responsibilities of Intel Software Guard Extensions (SGX) enclave shielding runtimes. This growing and diverse ecosystem is not sufficiently understood and we make contributions in two parts. First, we manually find and report issues at the low-level transition between enclave and untrusted domain. Our work shows that every extended architectural feature in a processor that the enclave may rely on must be adequately sanitized and initialized to a secure state before being used inside the isolated area. The results show that low-level configuration registers for floating-point accelerators are widely overlooked, and the impacts of this misconfiguration are more dangerous than may be intuitive. Since modern processor architectures are increasingly complex and legacy features are rarely removed, we then develop a tool that helps in automatically finding interface vulnerabilities. This tool, named Pandora, targets the crucial area of enclave shielding runtimes that provide the basis for most projects running in enclave-based TEEs like Intel SGX. Pandora saves the enclave memory at creation time and then uses symbolic execution to simulate execution of this truthful view of the enclave. We use Pandora to automatically detect multiple vulnerabilities across various enclave shielding runtimes, and use it to help vendors in validating their applied mitigations. Our work on Pandora is the first analysis of arbitrary Intel SGX enclaves that is able to automatically find vulnerabilities such as the vulnerability class of improper pointer alignment.

In summary, this dissertation extends the range of applicability of TEEs and secures TEEs by uncovering new vulnerabilities and automatically finding known vulnerabilities in enclave software. Our work thus serves as a fundament for future work to strengthen the capabilities of future TEEs and helps projects to secure their software on existing TEEs against known vulnerabilities.

Beknopte Samenvatting

Het gebruik van moderne informatietechnologie leidt tot meer productiviteit en technische mogelijkheden, zowel door de toenemende connectiviteit in computernetwerken, als door het delen van computersystemen door meerdere, onderling onvertrouwde applicaties. Deze evoluties stellen hedendaagse computersystemen echter ook in toenemende mate bloot aan beveiligingsproblemen en onderstrepen het belang van het beschermen van vertrouwelijke informatie en integriteit van berekeningen.

Een belangrijke bouwsteen om de veiligheid van computersystemen te garanderen, is hardwarematige isolatie, waar recente ontwikkelingen op het gebied van vertrouwde uitvoeringsomgevingen bijzonder veelbelovend lijken. Zulke vertrouwde uitvoeringsomgevingen stellen aanpassingen voor aan de onderliggende computerarchitectuur om beschermde programma's strikt af te schermen in zogenaamde *enclaves*, die geïsoleerd zijn van alle andere, onvertrouwde software op het doelapparaat, inclusief zelfs het besturingssysteem.

Dit proefschrift draagt bij aan het verbeteren van hedendaagse vertrouwde uitvoeringsomgevingen op twee manieren. Ten eerste richten we ons op kleine, ingebedde computerchips en onderzoeken we hoe vertrouwde uitvoeringsomgevingen kunnen ingezet worden in real-time omgevingen. Ons resulterende hardware-software co-design zondert een real-time planningsalgoritme af in een bevoorrechte enclave, om zo de beschikbaarheid van afzonderlijke applicatie-enclaves te garanderen. Deze oplossing vormt een eerste stap om vertrouwde uitvoeringsomgevingen uit te breiden met sterke beschikbaarheidsgaranties, zodat dat ze kunnen worden ingezet voor het beveiligen van kritische, ingebedde apparaten die strikte real-time deadlines moeten halen.

Ten tweede onderzoeken we de resterende softwareverantwoordelijkheden voor enclaves die gebruik maken van de wijdverspreide SGX-extensies in recente Intel-processoren. Dit proefschrift bevordert de veiligheid van het groeiende en diverse SGX-ecosysteem op twee manieren. Als eerste bijdrage ontdekken en rapporteren

we subtiele gebreken in de cruciale initialisatielogica die verantwoordelijk is om de overgang naar de enclave in goede banen te leiden. Meer bepaald laat onze studie van populaire SGX-ontwikkelingsomgevingen zien dat het veilig initialiseren van processorconfiguratieregisters voor zwevendekommabewerkingen op grote schaal over het hoofd werd gezien. In verschillende praktische aanvalsscenario's tonen we bovendien aan dat de gevolgen van deze verkeerde configuratie gevaarlijker zijn dan intuïtief kan lijken. Dit vormt de motivatie voor onze tweede bijdrage, waar we een automatische methode onderzoeken om interfacekwetsbaarheden in SGX-ontwikkelingsomgevingen te vinden. Daarbij ontwikkelen we een praktische toepassing, Pandora genaamd, die het geheugen van een willekeurige SGX-enclave waarheidsgetrouw kan reconstrueren om vervolgens een symbolische uitvoering van de enclave nauwgezet te simuleren. Pandora is in staat om automatisch meerdere onbekende kwetsbaarheden te detecteren in verschillende SGX-ontwikkelingsomgevingen en kan bovendien helpen om de toegepaste mitigaties te valideren.

Samengevat breidt dit proefschrift het toepassingsgebied van vertrouwde uitvoeringsomgevingen uit naar real-time omgevingen, en dragen we bij aan het beter begrijpen en valideren van resterende softwareverantwoordelijkheden. Ons werk kan dus als basis dienen voor toekomstig onderzoek om de mogelijkheden en veiligheid van opkomende vertrouwde uitvoeringsomgevingen te versterken, zowel op hardware als op software vlak.

Contents

Abstract v

Beknopte Samenvatting vii

Contents ix

List of Figures xiii

List of Tables xv

List of Abbreviations xviii

1 Introduction 1
- 1.1 Confidential Computing . . . 2
  - 1.1.1 Enclave Architectures . . . 5
  - 1.1.2 Sancus . . . 6
  - 1.1.3 Intel SGX . . . 7
- 1.2 Dissertation Motivation and Contributions . . . 10
  - 1.2.1 Availability Guarantees for Enclaves . . . 10
  - 1.2.2 Interface Sanitization in Intel SGX . . . 11
  - 1.2.3 Open Science and Ethical Considerations . . . 13
- 1.3 Other Contributions . . . 14
- 1.4 Dissertation Outline . . . 17

2 Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves 19
- 2.1 Introduction . . . 21
- 2.2 Problem and Assumptions . . . 23
  - 2.2.1 Generalized Base Platform . . . 23
  - 2.2.2 A Running Example . . . 25
  - 2.2.3 Security & Availability Guarantees . . . 26

2.3 Design . . . 28
2.3.1 TEE Architecture . . . 29
2.3.2 Exception Engine . . . 30
2.3.3 Atomicity Monitor . . . 33
2.3.4 Enclaved Scheduler . . . 35
2.4 Prototype Implementation . . . 36
2.4.1 Background: Sancus and RIOT . . . 36
2.4.2 Modifications to Sancus . . . 38
2.4.3 Modifications to RIOT . . . 40
2.5 Experimental Evaluation . . . 41
2.5.1 Case Study . . . 41
2.5.2 Performance Evaluation . . . 43
2.6 Discussion and Security Analysis . . . 46
2.7 Conclusion . . . 49

3 Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments 51
3.1 Introduction . . . 54
3.2 Background . . . 59
3.2.1 Intel SGX . . . 59
3.2.2 x87 FPU . . . 59
3.2.3 Streaming SIMD Extensions (SSE) . . . 61
3.2.4 Other Processor Architectures . . . 63
3.3 Poisoning FPU State Registers . . . 64
3.3.1 Attacker and System Model . . . 64
3.3.2 ABI Poisoning Attacks . . . 65
3.3.3 TEE Runtime Vulnerability Assessment . . . 70
3.4 Case Study: Floating-point Exceptions as a Side Channel . . . 73
3.5 Case Study: Attacking Machine Learning Predictions . . . 78
3.6 Case Study: Spec Benchmarks . . . 82
3.7 Conclusions and Lessons Learned . . . 86

4 Pandora: Principled Symbolic Execution of Intel SGX Enclaves 89
4.1 Introduction . . . 91
4.2 Background and Related Work . . . 93
4.3 Problem Statement and Overview . . . 96
4.3.1 Research Gap . . . 96
4.3.2 Solution Overview . . . 98
4.4 Enclave-Aware Symbolic Execution (*G1*) . . . 100
4.4.1 Modeling x86 Instruction Semantics . . . 100
4.4.2 Taint Tracking of Attacker Inputs . . . 100
4.4.3 Enclave-Aware Memory Model . . . 101
4.4.4 Enclave Entry and Reentry . . . 103

4.4.5 Path Exploration and State Reduction . . . 103
4.5 Runtime-Agnostic Enclave Loading (*G2*) . . . 104
4.6 Pluggable Vulnerability Detection (*G3*) . . . 106
4.6.1 ABI-Level CPU Register Sanitization . . . 107
4.6.2 Untrusted Pointer Value Sanitization . . . 109
4.6.3 Untrusted Pointer Alignment Sanitization . . . 110
4.6.4 Control-Flow Hijacking Validation . . . 111
4.7 Evaluation . . . 112
4.7.1 Selftest Validation Framework . . . 112
4.7.2 SGX Runtime Ecosystem Analysis . . . 114
4.8 Discussion . . . 119
4.9 Conclusion . . . 120

5 Conclusion **123**
5.1 Summary of Contributions . . . 124
5.2 Future Work . . . 125
5.3 Concluding Remarks . . . 129

A Additional Resources for Aion **131**
A.1 Atomicity State Machine . . . 131
A.2 Case Study Source Code in C . . . 132

B Additional Resources for Faulty Point Unit Attacks **137**
B.1 Proof-of-concept Enclave Code . . . 137
B.2 Search Algorithm Based on Overflow Exceptions . . . 138

C Additional Resources for Pandora **141**
C.1 Pandora CLI and Report Generation (*G4*) . . . 141
C.2 Static Analysis of Enclave Runtimes . . . 142
C.3 Pandora Breakpoints . . . 143
C.4 Vulnerability Details . . . 144

Bibliography **147**

Curriculum Vitae **163**

List of Publications **165**

List of Figures

1.1 Overview of confidential computing architecture paradigms . . 4
1.2 Enclave architecture call hierarchy 6
1.3 Intel SGX shielding runtime ecosystem 9

2.1 Aion use case example 25
2.2 Aion system overview 28
2.3 Exception engine overview of Aion 31
2.4 `clix` behaviour in Aion 34
2.5 Aion modifications to the status register 39

3.1 Intel SGX enclave model overview 55
3.2 Layout of the x87 FPU control word 60
3.3 Layout of the `MXCSR` control/status register 61
3.4 Histogram of error for secret recovery 75
3.5 Structure of the AND network 76
3.6 MLaaS system model with enclaves 79
3.7 Blender benchmark in SPEC CPU 2017 under attack 84
3.8 Povray benchmark in SPEC CPU 2017 under attack 85

4.1 Enclave shielding runtime overview 94
4.2 Overview of the Pandora architecture 99

A.1 Aion atomicity state machine 132

C.1 Example of an HTML report generated by Pandora 145
C.2 Command line interface of Pandora 146

List of Tables

2.1 Aion scheduler overhead 44
2.2 Aion activation latencies 45

3.1 Proof-of-concept attack executed inside an enclave 68
3.2 Affected runtimes by the FPU attacks 71
3.3 MNIST predictions under attack 81
3.4 SPEC CPU 2017 benchmarks 83

4.1 Comparison of symbolic-execution tools for SGX 97
4.2 Overview of Pandora results 113

C.1 List of breakpoints added by Pandora 143
C.2 Detailed overview of Pandora results 144

List of Abbreviations

ABI	application binary interface
API	application programming interface
AVX	advanced vector extensions
CCA	confidential compute architecture
CI	continuous integration
CLI	command line interface
CPU	central processing unit
CVE	common vulnerabilities and exposure
DRPW	device register partial write
EDL	enclave definition language
EDP	enclave development platform
ELF	executable and linkable format
EPC	enclave page cache
FPU	floating-point unit
GOT	global offset table
GPU	graphics processing unit
ISA	instruction set architecture
libOS	library operating system

MCDT MXCSR configuration-dependent timing

MMIO memory-mapped I/O

OS operating system

SBDR shared buffers data read

SDK software development kit

SECS SGX enclave control structure

SEV Secure Encrypted Virtualization

SGX Software Guard Extensions

SIMD single instruction multiple data

SSA state save area

SSE Streaming SIMD Extensions

TCB trusted computing base

TCS thread control structure

TDX Trust Domain Extensions

TEE trusted execution environment

TOCTOU time-of-check time-of-use

TPM trusted platform module

VM virtual machine

Chapter 1

Introduction

With an ever-decreasing cost of computation resources comes an ever-increasing number of opportunities these resources present. Where devices like the oft-cited and highly specialized Apollo guidance computer were unique at their time, devices of similar capabilities have nowadays become commonplace. On the other end of the computing spectrum, highly capable server hardware has become ubiquitous and enables extensive computations on a large scale. Across this spectrum, modern computing architectures are defined by two fundamental characteristics: connectivity to a network of other devices and the increased sharing of devices between potentially heterogeneous stakeholders. Both aspects significantly impact the security of underlying systems. Networking and potentially even global access to a system make any vulnerability an immediate and possibly global threat to a large number of affected systems. Similarly, sharing a platform between potentially unaffiliated stakeholders threatens the confidentiality and integrity of computations if the users are not adequately isolated from each other. In embedded computing, such as safety-critical applications, availability is often an additional concern, and global networking or sharing resources with unknown parties is thus rarely implemented to limit the system's exposure to possible attacks.

Many approaches exist to shield secrets from the impacts of arbitrary network attackers or harmful stakeholders on the same platform. Cryptography allows secret communication between software components over an untrusted network. This is also known as protecting data in transit, and cryptography has established itself as the de-facto standard in modern Internet communication. Encryption is furthermore a powerful paradigm to address the issue of protecting against stakeholders that may access archived data, also called protection of data at

rest. However, to protect data that needs to be accessed, also called protection of data in use, established encryption algorithms are insufficient as they require decryption prior to computation on encrypted data. To address this, multi-party computation [133] and homomorphic encryption [1] are two active areas of research. An orthogonal approach to pure cryptography is to modify the underlying hardware to isolate to-be-protected code from the potentially harmful environment. While these approaches often still utilize cryptographic primitives to protect communication and data at rest, their benefit is that data in use must not be encrypted since the computing party has exclusive access to it. Theoretically, this requires little to no computational overhead to operate on data in use as the overhead is spent in terms of additional hardware to access the data and not during runtime.

In the trajectory of this Ph.D., we focused on the area that is nowadays known as confidential computing, or trusted execution environments (TEEs). TEEs are a type of hardware primitive that isolates software components from the surrounding system [38, 122]. Other hardware approaches with similar or overlapping goals include trusted platform modules (TPMs) [148], virtualization isolation, capability architectures [172], and CPU privilege levels.

This dissertation makes contributions on two fronts. First, we address the gap of applying TEE architectures in safety-critical systems. Our work Aion, for the first time, enables availability on the class of embedded enclave architectures, prototyped on the research architecture Sancus [106]. Second, we address the growing complexity of server TEE architectures such as the x86-based Intel Software Guard Extensions (SGX) [11, 98] and the need for proper sanitization when entering and computing in those complex isolation environments. In this part, we first show that floating-point accelerators can be prone to fault-injection attacks via their status and control registers if not properly sanitized on enclave entry. As a follow-up, our contribution Pandora combines insights from this earlier work and related work. With Pandora, we can automatically analyze enclaves from the first instruction to detect a range of vulnerability classes.

1.1 Confidential Computing

TEEs are hardware extensions to processor architectures that allow software to shield a region in memory from unwanted access [38]. In doing so, hardware-based TEEs that we consider in this dissertation place their root of trust in exactly these instruction set architecture (ISA) changes. The trusted computing base (TCB) of the resulting shielded environment is thus only the hardware and its associated microcode, as well as the shielded environment itself [122].

This is a drastic reduction to the TCB in the absence of a TEE where the complete software stack that an application is running on has to be trusted. Figure 1.1 illustrates the different levels of encapsulation that modern TEEs provide. Without trusted hardware on the left of the figure, an application receives only protection guarantees resulting from virtual memory management and can, for example, be accessed by the underlying guest operating system (OS) which in turn can itself be accessed by the underlying hypervisor. The only protection with virtual memory comes from the underlying layer creating virtual address spaces that prevent parallel entities on the same level from accessing an entity's data. For a guest OS, this means that the hypervisor is trusted to set up this protection properly, and for an application, the guest OS is trusted to shield an application from other applications running on the same system. In this case, the TCB is quite large, as any module in the aforementioned stack has to be trusted not to leak secrets or impact the application's integrity. The promise of trusted execution is to disrupt this stacked layering of trust. Confidential computing and specifically TEEs present two alternative models of layering: enclave shielding and virtual machine (VM) shielding [38]. In enclave shielding, the application itself is isolated from the surrounding environment, such as the guest OS and other applications. To keep the TCB of an enclaved application minimal, a legacy application can be split up into an untrusted and an enclaved application, and communication between the two can be set up to provide only security-critical services from inside the enclave. Essentially, however, the TCB of the small enclave application only consists of itself and the associated code inside the trusted environment, and the second, untrusted, part of the application must be seen as potentially compromised. To address the complexity of splitting applications into a trusted and an untrusted part, VM shielding draws a larger isolation boundary and also includes the guest OS itself and co-located applications from the underlying hypervisor. This widens the TCB but has the benefit of easing the applicability of this approach since developers can seamlessly adopt many legacy applications into such a VM-based TEE environment.

The choice between both approaches lies in the size of the TCB versus the usability and direct applicability of established and legacy code in the isolation environment. Enclave architectures may suffer in their usability from the small TCB as dynamic libraries may not be usable as the guest OS is untrusted. Similarly, enclaves cannot execute system calls directly from within the enclave. They must either execute system calls via the untrusted application or substitute them for secure versions from within the enclave. VM architectures may accommodate this limitation at the cost of a larger TCB and associated higher computational overhead per isolated application.

While the specific guarantees by a TEE are not clearly defined across the

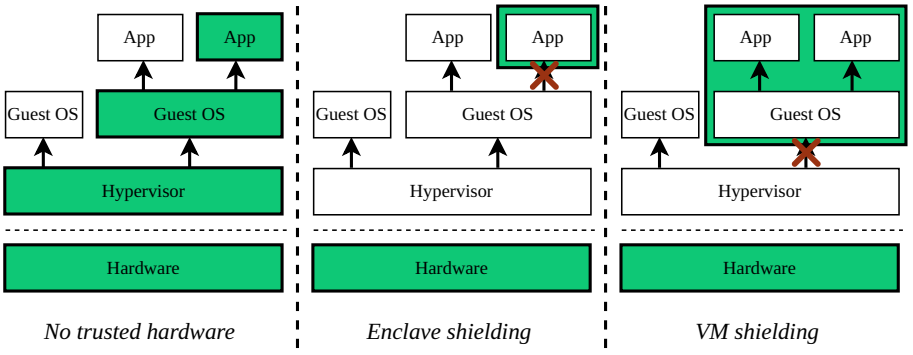


Figure 1.1: Overview of confidential computing architecture paradigms. Green boxes (bold frame) denote the trusted computing base of an application. Trusted hardware prevents access from the guest OS to the application or from the hypervisor to the guest OS, respectively.

ecosystem, the following properties are relevant in the context of this dissertation and can also be seen as the core parts of confidential computing [37]:

Data confidentiality and integrity: Software running inside the isolated environment can protect the confidentiality and integrity of data and ensure that access to the protected data is restricted to the TCB of the isolated software and that untrusted sources cannot tamper this data.

Code integrity: The untrusted environment cannot modify the code executing inside the isolated environment. While the untrusted environment may be able to trigger interrupts that disrupt the execution, all switches into the isolated environment are restricted to predefined entry points.

Attestability: The state of the software loaded in the isolated environment can be proven and reported to an outside stakeholder who can use this proof to establish trust into and create a secure communication channel with the software running in the TEE.

Additional orthogonal properties may be common across multiple TEEs and be useful for certain applications. Such additional properties include confidentiality of isolated code, recoverability from flaws in the underlying hardware, authentication of isolation environments prior to their launch, or direct access to memory-mapped I/O (MMIO) devices from within the isolated environment [37]. We will introduce additional concepts where necessary and when other properties are relevant to the research described in this dissertation.

This dissertation exclusively focuses on the group of enclave architectures. Where relevant, we draw parallels to and reflect insights onto other VM-based architectures such as ARM confidential compute architecture (CCA) [15], Intel Trust Domain Extensions (TDX) [71], AMD Secure Encrypted Virtualization (SEV) [9], and Keystone [86]. In this dissertation, we investigate state-of-the-art, off-the-shelf enclave architectures such as Intel SGX and explore new hardware-software co-designs on research prototype architectures. In the following, we first give an overview of the intricacies of enclave architectures and then introduce the two TEEs that are relevant in this dissertation: Sancus [106] and Intel SGX [40]. Sancus is used as the underlying architecture for the first contribution presented in Chapter 2, and Intel SGX is the underlying architecture for the second and third contribution presented in Chapter 3 and 4.

1.1.1 Enclave Architectures

The work presented in this dissertation primarily centers around enclave architectures. These architectures have two core benefits over the VM-based approach: they require a simpler architecture and no virtualization support and thus also apply to light-weight embedded platforms, and they allow to keep the TCB minimal versus VM-based TEEs where the isolation spans a complete guest OS. However, a downside of enclave architectures usually stems from the same lack of a trusted OS in the sense that all code has to statically be included in the loaded enclave instead of dynamically accessing libraries from a shared page set up by the OS. This complicates the compilation process, bloats up enclaves as multiple enclaves cannot share the same virtual memory pages with shared libraries, and complicates the development as developers have to ensure that all functionality is included within the single binary. Similarly, system calls within enclaves require a potentially costly switch from the enclave to the untrusted OS. Any such switch endangers an enclave's security properties and must be handled carefully to neither leak secrets nor rely on potentially harmful information when returning. Both issues may increase development complexity for enclave applications as legacy applications have to be either rewritten for this new paradigm or encapsulated into an enclave runtime that can transparently provide secure versions of system calls and library accesses.

Enclave Call Hierarchy To reduce the exposure of an enclave to attacks, all TEEs discussed in the following only support access by jumping to a single pre-defined entry point in the enclave boundary. From this single entry point, the enclave then multiplexes calls to its enclave functions while allowing the developers to carefully design and potentially verify the security of the entry point code. Figure 1.2 illustrates the common call hierarchy from the point of

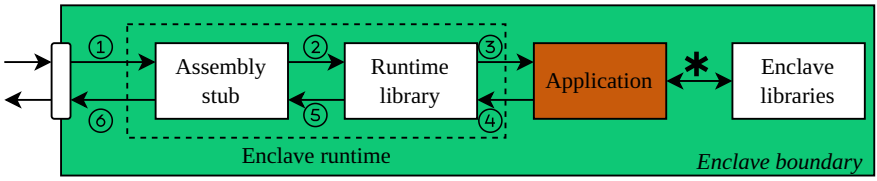


Figure 1.2: Call hierarchy in enclave architectures. On enclave entry, (1) an assembly stub sanitizes inputs and (2) passes execution to the runtime, which (3) calls the application. Application executions may perform (∗) multiple calls to enclave libraries and (4) – (6) either complete an `ecall` by returning the call via the runtime and the assembly stub or perform an `ocall` to the untrusted world.

entering into the single entry point of the enclave to the return to the untrusted world, usually in the form of the untrusted part of an enclaved application. As a common first step, a small section of code, usually programmed in assembly, sanitizes the potentially malicious register state. This section of the program is also known as the application binary interface (ABI) sanitization and also includes a setup of an in-enclave stack and preparing the register state for the switch to code written in higher-level programming languages. The expectation of compilers for higher-level programming languages is usually that the ABI adheres to specific guidelines, such as the System-V ABI [93]. One important task of the ABI assembly stub in enclaves is thus to prepare the ABI and set up the configuration and interface for the compiler-expected state. Depending on whether an enclave runtime is used or not, a dedicated runtime library may serve as a middle point between the ABI sanitization and the enclave application. This runtime library may take over parts of the sanitization of the application programming interface (API), e.g., sanitization of passed pointers. Finally, the runtime library switches to the enclave application, which may make arbitrarily many calls to statically linked libraries within the enclave boundary. Whenever this application enclave has finished its computation, the call is usually returned via the runtime library and the assembly stub to sanitize the ABI and API state, this time to scrub it from secrets that may leak to the untrusted OS.

1.1.2 Sancus

On the lower end of the computing spectrum, lightweight processor architectures such as the Texas Instruments MSP430 have a 16-bit architecture, run at clock speeds of 1-20 MHz and lack many extended processor features that have become

commonplace in modern, more complex processor designs. The core benefits of these processor families are the very low costs per chip and the very low power consumption, combined with the fact that not every application requires extensive computational capabilities. Sancus [106] is an open-source open-hardware research prototype TEE that is built on top of such a lightweight architecture, specifically the openMSP430 core by Olivier Girard.[1] The main contribution of Sancus is that it brings an enclave architecture to the low-level openMSP430 processor that before supported neither hardware isolation mechanisms nor even a cryptographic coprocessor. Data confidentiality and integrity are achieved with a program counter-based access control that only grants read or write access to a protected region if the current execution is within this region. Code integrity is achieved by limiting switches to the enclave to the first address in the enclave address range and preventing any jumps into the middle of a protected region. The above guarantees allow multiple mutually distrusting enclaves to coexist on the same device without compromising their confidentiality or integrity. To enable secure communication between enclaves, the Sancus design allows enclaves to directly call other enclaves and retrieve the identifier of a caller enclave. Remote attestation with external stakeholders is secured with symmetric keys that are deterministically derived based on device identity and the initial hash of the protected region. Lastly, Sancus, as a hardware-software co-design, also provides compiler modifications to LLVM [92] that allow developers to initialize, program, and interact with enclaves conveniently.

While the Sancus architecture is in many ways simpler than higher-level architectures like Intel SGX, its simplicity allows for a range of features that do not exist on these more complex architectures: secure access to MMIO peripherals from within an enclave, enclaves as interrupt handlers, and driver enclaves that can expose peripherals to other enclaves or untrusted tasks.

Sancus has served as the basis for a range of applications and extensions, such as partial exploratory support for interruptible enclaves [156, 158], side-channel resistant compilers [25, 170], automotive applications [155, 162, 169], and smart home applications [103].

1.1.3 Intel SGX

Intel SGX is an enclave architecture that, as of 2023, has been available on processors of the Core-i series from 2015 (generation “Skylake”) to 2020 (generation “Comet Lake”) as well as on server processors since 2015 [72].

[1] `https://opencores.org/projects/openmsp430`

In contrast to MSP430, x86 uses a more complex memory architecture based on virtual address spaces. Instead of the program counter-based access protection used in Sancus, Intel SGX uses an enclave page cache (EPC) map that tracks the assignment of virtual pages to enclaves. Since the underlying operating system is seen as fully untrusted but still manages the memory allocation, the assigned EPC pages are protected on enclave creation when the enclave is loaded and the pages are added to the enclave address range (called the `ELRANGE`) [40]. Upon finalization of the enclave's initialization, an enclave-specific value called the MRENCLAVE value is calculated from all initial enclave pages. In cooperation with some architectural enclaves, this MRENCLAVE value can be used in remote attestation to authenticate the enclave to remote parties and establish a secure channel with remote stakeholders or other enclaves. Importantly, however, this MRENCLAVE value also requires that all initializations of the same enclave result in the same hash, independently of where the enclave has been loaded in the virtual address space [11]. This requires additional complexity within the enclave as common software relocation mechanisms and position-independent code must be initialized within the enclave instead of untrusted software being able to reposition code after loading it into memory.

Once an enclave has been initialized, it can only be entered via special instructions that jump to a predefined enclave entry [98]. This enclave entry is usually written in assembly and follows the same principles as described above in Section 1.1.1. Aside from these `ecall`s into the enclave, Intel SGX also defines `ocall`s as the reverse operation where an enclave calls a function outside the enclave that then returns to the enclave. Additional complexity is added by special in-enclave structures such as the SGX enclave control structure (SECS) and thread control structure (TCS) pages that contain important architectural enclave information and local enclave thread information, respectively.

Enclave Shielding Runtimes

Several shielding runtimes have been developed for Intel SGX and have become the de-facto standard for developing enclave applications. These shielding runtimes provide multiple benefits to an application developer: they govern the switches between untrusted and trusted world for `ecalls` and `ocalls`; they administer the relocation of position-independent code inside the enclave as part of an initialization step; they provide crucial functions that ease the programming of applications inside the enclave, like functions to check whether a pointer lies inside or outside the enclave; and they often provide an interface to marshall and securely copy data structures into and out of the enclave. Figure 1.3 illustrates the landscape of Intel SGX applications and a part of the ecosystem of these shielding runtimes that has developed as of 2023. While

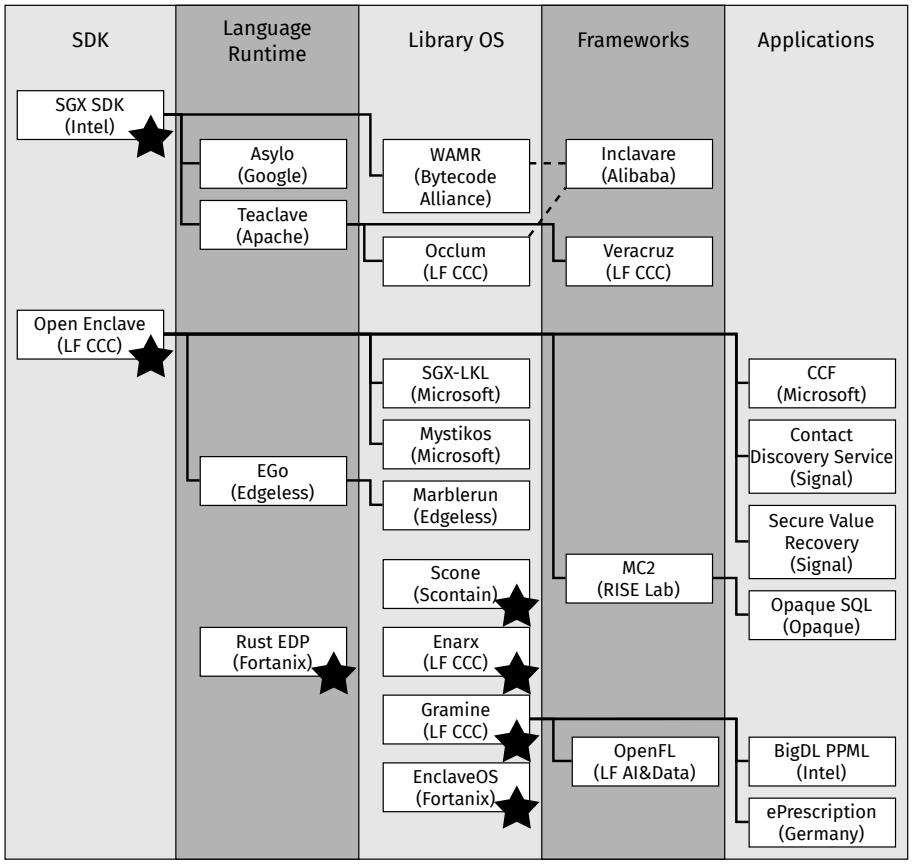


Figure 1.3: Intel SGX shielding runtime ecosystem of production-ready projects. Most projects depend on only a few root projects (★) that provide core enclave functionality.

many applications are based on the Intel SGX software development kit (SDK), a growing number of projects are based on other shielding runtimes such as Open Enclave [101] or Gramine [144]. Notably, only a few root-level projects exist that serve as the basis TCB for many other projects. All highlighted projects in the figure provide entry-level assembly code that sanitizes the potentially malicious input from the untrusted environment and also provides developers with the functionality to either transparently or actively interact with the untrusted environment securely.

1.2 Dissertation Motivation and Contributions

In light of this background, we present the three core contributions of this dissertation: Chapter 2 extends the protection guarantees of existing embedded TEEs and Chapter 3 and 4 uncover and automatically detect interface sanitization issues in Intel SGX.

1.2.1 Availability Guarantees for Enclaves

As a first contribution, Chapter 2 presents Aion, a hardware-software co-design that can ensure the availability of multiple mutually distrusting parties at the same time. While Sancus can provide strong confidentiality and integrity guarantees to mutually distrusting enclaves, similar to many other TEE architectures, it does not guarantee the availability of any of its protected regions. Specifically, any enclave protected on a Sancus system can be ensured that its confidentiality is protected at all times and that when it executes, its code integrity is protected as well. However, other enclaves or even untrusted code can impact the availability of enclaves by either never scheduling the enclave or triggering interrupts to occur when the enclave is executing. This inevitably results in attackers being able to disrupt deadlines that enclaves may need to meet in a mixed-criticality or even safety-critical system.

Enclave architectures already provide strong security guarantees and are ideal for protecting multiple stakeholders at once that execute their code on a mixed-criticality platform. To additionally guarantee availability, we designed Aion as an extension to enclave architectures. At its core, Aion moves the real-time scheduler of the system into an enclave. This ensures that the scheduler is protected from outside influences and has the additional benefit that other enclaves do not need to trust the scheduler enclave for confidentiality and integrity since the security protections by the TEE also apply across enclaves. If these other enclaves are willing to trust the scheduler enclave for availability, however, they can request to experience periodic scheduling or similar scheduling guarantees as needed. The scheduler enclave then serves as a small TCB for availability only and must ensure that it only approves scheduling guarantees when they are possible even in a worst-case scenario. To provide hard guarantees on the upper bound of this worst-case scenario, we additionally equip Aion with a concept of bounded atomicity. Only the scheduler enclave is in full control of interruptability and can disable interrupts. All other enclaves and untrusted software on the system only have access to a special `clix` instruction that disables interrupts for a bounded number of cycles. A specially designed availability monitor and an exception engine for enclaves ensure the correct

handling of all edge cases in hardware. With this hardware-software co-design, Aion can provide hard real-time guarantees to multiple stakeholders at once, allowing system designers and developers to specifically design their real-time applications with strong confidentiality, integrity, and availability protections.

Our work, for the first time, combines lightweight embedded TEEs with mixed-criticality systems requiring strong hard real-time guarantees. While architectures like ARM TrustZone have existed for multiple years and are starting to be applied in embedded applications that require availability guarantees, Aion demonstrates that availability protections are also possible without placing the scheduler into the TCB for confidentiality and integrity. Since ARM TrustZone is similar to the VM-based model explained above, this nuanced trust relationship is not possible, as the scheduler and the complete guest OS are fully trusted by the protected applications. Aion furthermore demonstrates that even the architecture class below classical TrustZone devices, *i.e.*, the architecture class of 16-bit processors, can benefit from TEE capabilities with strong availability guarantees.

> Publication data:
> F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg. "Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves". In: *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS'21)*. ACM, 2021, pp. 1357–1372

1.2.2 Interface Sanitization in Intel SGX

The second part of this dissertation is concerned with the interface between the untrusted world and Intel SGX enclaves.

As a first contribution in this part, we show in Chapter 3 that former register sanitization in Intel SGX shielding runtimes was insufficient. Specifically, most shielding runtimes omitted to set configuration and control registers of the x87 floating-point unit (FPU) and of the Streaming SIMD Extensions (SSE) to a safe value upon entering the enclave. At the time of investigation, all seven investigated shielding runtimes were vulnerable to at least some form of FPU register attack. Our work shows that even seemingly minor control registers can have potentially grave impacts due to the strong integrity protection promises of Intel SGX. We highlight this with three case studies and illustrate that unexpected floating point precision and rounding modes can majorly impact the resulting quality of generated output. These attacks are non-trivial to mitigate and show that exception masks can even leak enclave secrets in some cases.

This contribution does not stand alone in showing that the interface between untrusted world and Intel SGX enclaves must be properly sanitized. Van Bulck et al., for example, showed in 2019 that all enclave interaction on both ABI and API can be problematic if not handled carefully [157]. At the same time, several of the investigated shielding runtimes are of high quality and investment by the respective vendors, and the functions and assembly stubs that perform sanitizations can be seen as having received extensive care from their developers. For example, our study of shielding runtimes from 2022 showed that it is common for these shielding runtimes to have changed as many lines of code in the assembly stubs of up to two times their actual size since the project began [153]. Yet, issues on both the ABI and API levels keep reoccurring, as happened in early 2023, for example, when Intel provided new recommendations of SSE configuration values that refine the earlier recommendations that were also applied as a response to our attacks [73].

As a second contribution in this part, and this dissertation's third and final contribution, we thus present a framework in Chapter 4 that allows symbolic execution of Intel SGX enclaves and finding known classes of interface vulnerabilities, such as the ones reported in our earlier contribution. This framework, called Pandora, brings several innovations to, for the first time, allow *principled* and *truthful* symbolic execution of enclave binaries. Pandora uses a novel initial phase to dump the full enclave memory at enclave creation time and uses this dump to truthfully simulate enclave execution beginning at the first assembly instruction. Prior work has already attempted and partially succeeded in symbolically executing Intel SGX enclaves but has focussed on the applications instead of the crucial but vulnerable shielding runtimes. Our work, for the first time, takes care to fully encompass all functionality provided by these enclave shielding runtimes without skipping or simulating essential functionality. In addition, Pandora is written in a modular approach and can utilize the analyses provided by the underlying symbolic execution engine angr and pass information on to a new system of powerful plugins that can flexibly be enabled on demand. These plugins extend the capabilities of angr and perform the actual vulnerability detection and reporting. We use Pandora to find seventeen new vulnerabilities across eight partially closed-source shielding runtimes.

Publication data:
F. Alder, J. Van Bulck, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI poisoning attacks on Intel SGX". in: *Annual Computer Security Applications Conference (ACSAC)*. 2020, pp. 415–427

F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck. "Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes". In: *In submission.* 2023

1.2.3 Open Science and Ethical Considerations

All software vulnerabilities discussed in this work have been reported to the respective vendors, and where possible, we have supported the vendors in validating the applied mitigations. Overall, the work in this Ph.D. trajectory has directly or indirectly led to the finding of 211 vulnerability instances, the assignment of twelve common vulnerabilities and exposures (CVEs), and software patches across nine projects. In several instances, our active engagement with the open-source maintainers has led to more refined patches, thus improving end-user security in the emerging Intel SGX software ecosystem. Furthermore, all research performed during this dissertation has been open-sourced. For the three core contributions of this dissertation, the details are:

Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves The Sancus hardware and compiler infrastructure changes have been included in the Sancus core architecture on GitHub.[2] All software modifications to the Riot OS [17] were published as open source in their own repository under the same Sancus organization[3], and a Docker container with the Aion architecture is automatically built and provided on GitHub.

Faulty Point Unit: ABI Poisoning Attacks on Intel SGX During the course of this contribution, we found and responsibly disclosed six vulnerabilities across six runtimes, one of which is a runtime on the RISC-V architecture, leading to the assignment of two CVEs. The artifact and corresponding Docker container of this contribution are fully available on GitHub[4] and continuous integration regularly verifies its reproducibility up to this date. As part of the ACSAC conference, the artifact received the highest ACM rating of "Artifacts Evaluated - Reusable v1.1", and the paper subsequently received a "distinguished paper with artifacts" award.

[2] `https://github.com/sancus-tee`
[3] `https://github.com/sancus-tee/sancus-riot`
[4] `https://github.com/fritzalder/faulty-point-unit`

Pandora: Principled Symbolic Execution of Intel SGX Enclaves At this time, Pandora is in submission. During working on this project, we found 174 new vulnerability instances across 10 runtimes, leading to the assignment of 7 CVEs. Once this submission concludes, Pandora will be released as open-source, together with extensive documentation to support its use during the development processes of enclave software[5].

1.3 Other Contributions

In parallel to the three major contributions outlined in this dissertation, I authored or contributed to several additional publications that are summarized as follows.

Efficient and Timely Revocation of V2X Credential In vehicle-to-everything communication, vehicles are expected to communicate with an abundance of other vehicles and communication parties alongside the road. To ensure the security and safety of all involved participants, malicious or misbehaving parties must be punishable in a timely manner, e.g., via revocation of their credentials or pseudonyms. This paper presents a mechanism for self-revocation of vehicle pseudonymous credentials based on a TEE inside each vehicle. The presented mechanism is formally verified with the Tamarin prover and can provide a predictable upper bound on revocation time based on configurable parameters. In addition to the formal verification, the paper presents a statistical model of the expected size of the revocation list of each network participant, and a simulation to showcase its feasibility even for larger networks.

Gianluca Scopelliti is the main author of this work and developed the core design together with Christoph Baumann. I contributed with discussions on the design and the statistical model of the certificate revocation list as well as writing of the concerned chapters.

> G. Scopelliti, C. Baumann, F. Alder, E. Truyen, and J. T. Mühlberg. "Efficient and Timely Revocation of V2X Credentials". In: *31st Annual Network and Distributed System Security Symposium (NDSS'24)*. The Internet Society, 2024

[5] `https://github.com/pandora-tee`

About Time: On the Challenges of Temporal Guarantees in Untrusted Environments This paper investigates how time can be accessed by enclaves from within a TEE and classifies five levels of this access: no access to trusted time; checking the monotonicity of untrusted time; access to an external trusted time source; access to the trusted time source with a known delay; and lastly atomic access to the trusted time source. We found that Intel SGX can only provide access to an external trusted time source when using Intel platform services that are not always available. Furthermore, in the future, Intel SGX enclaves may access trusted time atomically if never interrupted. Other TEEs like Intel TDX or Arm CCA have the potential to provide access with a known delay to a trusted time source, albeit most TEEs cannot access time atomically. Our work concludes by enumerating multiple applications of trusted time in TEEs and rating them according to their required trusted time level.

I was the main author of this paper and collaborated with Gianluca Scopelliti and Jo Van Bulck under the supervision of Jan Tobias Mühlberg.

> F. Alder, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg. "About Time: On the Challenges of Temporal Guarantees in Untrusted Environments". In: *6th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2023

End-to-End Security for Distributed Event-Driven Enclave Applications on Heterogeneous TEEs Hardware isolation primitives provide strong local assurances that can be attested to outside parties. This paper introduces a framework that links local peripheral input on one device to a decision made on another. With this so-called *authentic* execution, we can guarantee that a decision made on a device has only been made as a response to peripheral input received by a remotely attested device. Conversely, the output would not have been created without adequate input. Our open-source framework supports Sancus, Intel SGX, and ARM TrustZone to allow for heterogeneity, and we showcase it with a case study of a smart home application.

Gianluca Scopelliti is the first author of this paper. I contributed with the mentoring of Gianluca Scopelliti during his master's thesis, contributed to the current designs of the authentic execution framework, and aided in writing the final text.

> G. Scopelliti, S. Pouyanrad, J. Noorman, F. Alder, C. Baumann, F. Piessens, and J. T. Mühlberg. "End-to-End Security for Distributed Event-Driven

> Enclave Applications on Heterogeneous TEEs". In: *ACM Transactions on Privacy and Security (TOPS)* (Apr. 2023)

A Case for Unified ABI Shielding in Intel SGX Runtime As a result of our investigation of FPU sanitization in Intel SGX and the earlier work on Intel SGX interface sanitization [157], this paper makes a case against the heterogeneous landscape of enclave assembly code. Since most enclave shielding runtimes need to perform the same register sanitization and need to set up their own secure environment, we argue that unification of the small code base that performs this task is necessary. Some intricacies would be necessary to allow for a diverse set of environments and programming languages that follow the execution of the unified stub, but a clear benefit would be the starkly reduced patch timelines to mitigate known ABI-level issues.

Jo Van Bulck was the main author of this paper, and we shared work equally under the supervision of Frank Piessens.

> J. Van Bulck, F. Alder, and F. Piessens. "A Case for Unified ABI Shielding in Intel SGX Runtimes". In: *5th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2022

Secure End-to-End Sensing in Supply Chains This paper serves as a case study of how to protect complex supply chains by combining trusted hardware and blockchain-backed ledgers. In supply chains, shipments are passed from the producer over multiple shipment providers down to the final customer. A common issue in this environment is trust, as in modern supply chains, stakeholders rarely trust all other stakeholders that preceded it in the order of events. If a shipment arrives with faults, a stakeholder may not always be able to prove that the defect existed when it received the shipment. Similarly, even the absence of faults must be proven, as a disrupted cold chain cannot always be immediately detected. To tackle these issues, our work designs an architecture that relies on embedded TEEs like Sancus to draw a secure, integrity-protected path from the sensor to a blockchain-backed ledger in a cloud environment. With this architecture, stakeholders no longer need to place trust in prior stakeholders but only need to trust the underlying hardware and its integration into the shipping unit.

Jan Pennekamp is the first author of this paper. He, I, Roman Matzutt, and Jan Tobias Mühlberg had equal parts in writing under the supervision of

Frank Piessens and Klaus Wehrle. This paper resulted in a bachelor's thesis at RWTH-Aachen University, which I co-supervised as an external advisor.

> J. Pennekamp, F. Alder, R. Matzutt, J. T. Mühlberg, F. Piessens, and K. Wehrle. "Secure End-to-End Sensing in Supply Chains". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–6

1.4 Dissertation Outline

The remainder of this dissertation is structured as follows. In the first part of our contributions, Chapter 2 presents our work on Aion, which brings strong availability guarantees to lightweight enclave architectures. Chapter 3 then begins the second part of our contributions with our attacks on Intel SGX via the FPU configuration registers. To aid developers in this fast-paced landscape of interface vulnerabilities, Chapter 4 continues this second part with our third contribution and presents Pandora that can automatically detect common vulnerabilities in enclave binaries. Chapter 5 then concludes this dissertation and presents opportunities for future work.

Chapter 2 to 4 are based on the previously peer-reviewed publications as marked in the respective chapters and only contain minor modifications to fit the style of this text. A preamble precedes each chapter to place the work in the context of this Ph.D. trajectory and to present related work in the area since publication.

Chapter 2

AION: Enabling Open Systems through Strong Availability Guarantees for Enclaves

This chapter was previously published as:

> **F. Alder**, J. Van Bulck, F. Piessens, and J. T. Mühlberg. "Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves". In: *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS'21)*. ACM, 2021, pp. 1357–1372

Preamble

This first contribution has been the final result of long-going discussions in our research group on interruptability and availability guarantees on hardware-based trusted execution environments (TEEs). Many of these discussions started before this Ph.D. trajectory began in 2019, as Jo Van Bulck already discussed some weak points of architectures like Sancus when it comes to availability in his master's thesis [152]. A follow-up publication then outlined necessary steps to

take to make Sancus enclave a candidate for real-time applications [156]. Many of the ideas and contributions of that prior work have impacted the design of Aion.

The core issue with TEE architectures like Sancus at the time of publication was that lightweight embedded security architectures prioritized the confidentiality and integrity of a system at the cost of availability. For example, both the original Sancus work [107] as well as the follow-up Sancus 2.0 version [106] were configured to reset the processor when a violation occurs, and this was common in similar architectures like SMART [50] or VRASED [109]. This, however, is a stop-gap approach for any system that is supposed to provide availability, as it gives attackers a convenient opportunity to reset the processor and impact real-time deadlines. Similarly, neither the original Sancus 2.0 work nor any related work at that time like SMART intended enclaves to be interruptible. As a first work, TrustLite [83] explored a hardware-level interrupt engine for enclaves that uses an exception engine to securely store the register state to a secure location before sanitizing it for the untrusted exception handler. Van Bulck et al. [156, 158] implemented an interrupt mechanism to Sancus enclaves that is similar to this earlier work by TrustLite. Busi et al. [29] formally verified a Sancus interrupt handler that was hardened against side channels. An important drawback of these designs is, however, that interruptability requires the cooperation of the enclave, as enclaves are able to completely disable interrupts and stall the system. Orthogonally, Masti et al. [97] presented a means of bounded atomicity in embedded system for scheduling decisions. With Aion we refine and combine these multiple directions of research and present a solution to the long-standing development of enclave interruptability and availability guarantees. Aion ensures that enclaves can be interrupted without impacting their security but cannot disable interrupts as a means of unboundedly disrupting the availability of the whole system.

After the publication of Aion, several works have approached the same issue of availability on the ARM TrustZone architecture. RT-TEE [166] provides the secure world with an event-driven hierarchical scheduler and device drivers for I/O. This allows RT-TEE to guarantee availability even in the case that the normal world is compromised. The authors showcase their prototype on two types of processors and present several case studies showing their approach's feasibility. Furthermore, Van Eyck et al. built MrTEE as a practical framework to build real-time compliant applications with commodity software and hardware without requiring extensive software modifications, as is the case with RT-TEE [161]. As a result of their underlying TrustZone architecture, both essentially differ from the guarantees provided by Aion. Where Aion only requires enclaves to trust the scheduler for availability only, as confidentiality and integrity are protected by the underlying hardware mechanisms, TrustZone only

provides the security guarantees for the whole secure world. Thus, the scheduler that is executed in the secure world must be fully trusted by the protected applications, both for availability as well as for confidentiality and integrity. The TrustZone approaches are thus different in their isolation granularity from the approaches discussed and presented in this chapter.

With the conference publication, AION contributed an open-source artifact that was integrated into the Sancus project[1]. This integration was not performed as a duplication of the code base of Sancus but was instead embedded into the existing code base. A Sancus core can be configured via pre-compiler directives that define whether a Sancus core is generated with availability capabilities or not. Together with this embedding into the existing core, we also integrated the availability engine into the existing continuous integration pipeline of Sancus and enabled regular testing of the AION hardware functionality, as well as the regular building of Docker containers that can be used as a basis for development with either AION or Sancus. Additionally, as part of this artifact, AION uses a new, faster hardware simulator based on Verilator[2] that can generate cycle-accurate simulations of AION within seconds instead of the prior simulator that required several minutes for the same task. Lastly, AION was the basis for a master's thesis that investigated remaining issues concerning the schedulability of systems built on AION [65].

2.1 Introduction

With the increased connectivity of devices all across the computing spectrum comes an increasing demand for systems that are not locked down but are more dynamic and open to changes after they are deployed in the real world. An *open* system runs software components (tasks, processes, ...) from several stakeholders that do not necessarily trust each other. The resources of such system, including memory, devices, and the CPU, must be shared among these software components without introducing security vulnerabilities that would allow a malicious component to violate the security expectations of another component. Traditionally, Operating System (OS) kernels have the responsibility of enforcing appropriate isolation between components, and, hence, the OS kernel has been part of the Trusted Computing Base (TCB).

However, experience has shown that operating system kernels can have vulnerabilities too, and several approaches have been explored to reduce the amount of trust in the OS kernel:

[1] `https://github.com/sancus-tee`
[2] `https://verilator.org/`

First, there is a long line of work in reducing the *size* of kernels (e.g., move to microkernels), or relying on simpler hypervisors or security monitors for enforcing isolation [23, 94, 129]. The key idea is that the trusted layer of software gets smaller, but all software components still need to fully trust the system software for any of their security properties.

Second, formal verification of system software has been proposed as a mechanism to reduce the likelihood of vulnerabilities, and, hence, to better justify the level of trust in system software[66, 82].

Third, work in the trusted computing research area has developed the idea of Trusted Execution Environments (TEEs) or enclaves [8, 19, 28, 83, 86, 98, 106]. These approaches make it possible to remove most (if not all) system software from the TCB, but they cannot guarantee all desired security properties. More specifically, while integrity and confidentiality of enclaves can be guaranteed with a TCB consisting of just the enclave software itself and the hardware, no availability guarantees can be provided. More generally, these systems can provide strong guarantees for resources (like memory) that are *spatially* shared, but not for resources (like CPU time) that are *temporally* shared. In the best case (for instance, in Intel SGX), the operating system kernel can *preempt* temporally shared resources from misbehaving enclaves, at the cost of having to trust the kernel for availability properties. In other cases, there are no availability guarantees in the presence of malicious enclaves.

The objective of this paper is to improve the state-of-the-art in this third approach. We propose a hardware/software co-design that supports classic enclave-like isolation of software components in an open system, and that improves on that classic isolation by also providing availability guarantees. Our system supports the secure *temporal* sharing of resources (including CPU and I/O devices) among mutually distrusting software components with a small TCB. More specifically, a given enclave software component needs to trust: *(i)* its own code and the hardware for confidentiality and integrity properties, and *(ii)* its own code, the hardware, the drivers of the shared devices it requires access to, and a small, trusted scheduler enclave for availability properties. Crucially, since the scheduler is only trusted for availability, our design protects the confidentiality and integrity of vital enclave applications even against a misbehaving scheduler. Furthermore, when the scheduler is well-behaved, our design can provide strong availability guarantees (including real-time guarantees) to software components in the presence of arbitrary malicious software on the platform outside the TCB (including malicious enclaves, malicious drivers for devices not used by this specific component, and system software besides the trusted scheduler).

Our design targets small embedded systems (specifically, our prototype is based

on a TI MSP430 16-bit processor running the RIOT OS), both because these can benefit most from availability and real-time guarantees, and because this allows us to focus on the essence of our design: building on preemption combined with a safe bounded atomicity primitive. Extensions to larger systems, such as for instance Intel SGX-scale processors, are not in the scope of this paper, and are left for future work.

In summary, the contributions of this paper are:

- a novel hardware-software co-design of a security architecture for open systems that extends the strong security properties of modern hardware TEEs with strong guarantees on enclave availability, even in the presence of powerful software adversaries on the same platform.
- a prototype implementation built by extending an existing TI MSP430-based TEE and by extending the existing RIOT IoT operating system.
- a case-study driven evaluation of the security and availability provisions and the costs of the design.

2.2 Problem and Assumptions

To illustrate the problem and our platform requirements, we first discuss the base platform that we use as a starting point for our work. We then describe a simple application scenario with specific security and availability needs that cannot be realized with classic TEE implementations. Finally we generalize this to derive platform requirements and discuss these in the context of related work.

In general, we aim to support *open* systems, which are systems that allow multiple distrusting stakeholders to dynamically load arbitrary applications at runtime. While it is obviously possible to combine an open system with priority-based scheduling, the interesting and most difficult case is dealing with mutually distrusting stakeholders executing code with the same priority. Only in this case resources have to be divided fairly.

2.2.1 Generalized Base Platform

The base platform we start from is an embedded TEE that provides an enclave-like isolation mechanism. This base platform supports the creation of enclaves that offer the following security guarantees. First, the software in an enclave is

isolated from all other software on the same platform, including system software such as the operating system. Second, enclaves support (local and remote) attestation: they can provide cryptographic evidence about their identity (characterized by a cryptographic hash of the binary code of the enclave). These security guarantees rely on a small trusted computing base, sometimes even only the hardware.

More specifically, in terms of isolation, the base platform guarantees that: *(i)* the data section of an enclave is only accessible while executing code from the code section of that same enclave, and *(ii)* the code section can only be entered through one or more designated entry points. These isolation guarantees are simple, but they have been shown to be strong enough and useful to enforce confidentiality and integrity properties of enclaved applications or modules. For instance, Patrignani et al. [113] show how encapsulation mechanisms from Java-like object-oriented languages can be securely compiled to a platform that supports enclaves. This implies that confidentiality and integrity properties of the enclave can be guaranteed in an *open* system: an enclave developer only needs to trust (or verify) the code of their own enclave (and possibly other enclaves that the enclave depends on, such as device driver enclaves). As a consequence, mutually distrusting enclaves can co-exist on the platform, and neither one needs to trust the other to maintain its own security, which is limited to confidentiality and integrity. The construction and the benefits of such a base platform is well understood, and Maene et al. [96] provide a survey of existing platforms.

However, these platforms lack any kind of *availability* guarantee. On some platforms [50, 106, 109] enclaves can protect themselves from being interrupted (and, hence, get atomicity guarantees) for security purposes, but as a consequence a misbehaving enclave can abuse such atomicity guarantees to disrupt the system and make it unavailable to other enclaves. On systems [28, 83, 86, 98] where enclaves are interruptible, on the other hand, enclaves do not get any guarantees on progress. For instance, enclaves might never get scheduled, or when they are scheduled they can get interrupted again without having made any progress. Also, enclaves may need to acquire resources other than memory or CPU, e.g., access to I/O devices like sensors or communication channels, and no guarantees can be provided that the enclave can acquire these within a bounded time span. Note that some Memory-Mapped I/O (MMIO) devices may only use a specific memory region to interact with the applications. This means that this memory region needs to be temporally shared between applications as a spatial sharing may not be possible for certain control or status bits. Finally, some platforms handle security violations in such a way that a security violation from one enclave can impede the progress of another one. For instance, a security violation might lead to a platform reset [50, 106, 109].

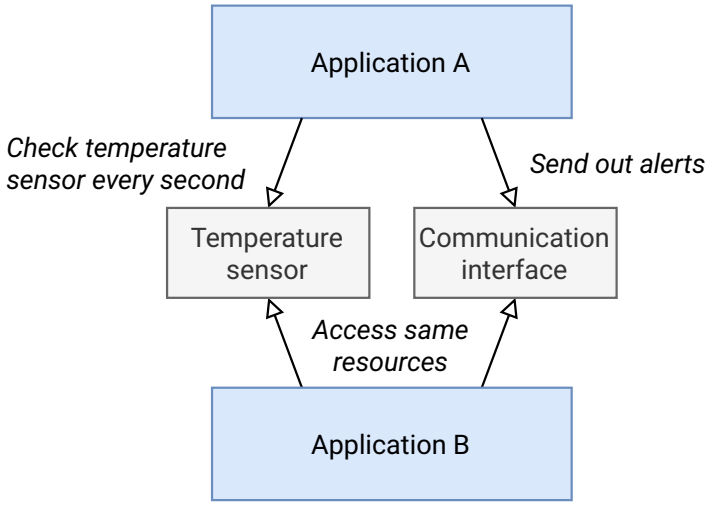


Figure 2.1: Simple example of two applications periodically accessing the same shared resources.

This set of shortcomings leads us to the problem we set out to solve in this paper: how can an enclave platform provide availability guarantees, while maintaining the desired strong confidentiality and integrity guarantees, *i.e.*, in particular that only the hardware plus the enclave itself and any dependent enclaves need to be trusted or verified. By doing so, the platform we design is the first enclave platform to provide a strong notion of availability for mutually distrusting enclaves, where neither one needs to trust the other to maintain its own security, which includes confidentiality, integrity, *and availability* properties.

2.2.2 A Running Example

Figure 2.1 depicts a scenario with two applications $\mathcal{A}$ and $\mathcal{B}$ that execute periodically, monitoring the same temperature sensor. Each application will trigger an alert if the temperature exceeds a programmed threshold. These alerts are communicated over the same, shared communication interface. We assume an open system where all system resources, including the CPU, the sensor and the communication interface, may be used by multiple applications. The deploying stakeholders of $\mathcal{A}$ and $\mathcal{B}$ are neither aware of each other's applications, nor would they trust each other's applications to behave collaboratively. However, both stakeholders consider their applications to be critical as harm may be caused if the alarms are not triggered within strict time bounds. The stakeholders

do trust the execution platform to uphold a notion of spatial and temporal isolation for their respective applications, and they may rely on primitives such as remote attestation to be ensured of their application execution on the intended platform. In regards to input and output from the temperature sensor and to the communication interface, the applications trust the utilized peripherals and an attacker controlling one of the peripherals themselves or a failed sensor are out of scope of their attacker model. This means that the platform aims to provide guarantees only up to the device boundaries and tamper-resistant sensors or resilience against network denial-of-service attackers are left to orthogonal research. At the same time, peripheral drivers on the system and their communication with attached devices are in scope of the guarantees as long as the attached peripheral is responsive.

While the spatial isolation properties required by our running example are generally well understood in existing TEE platforms, these platforms do not provide the required *availability* guarantees. This includes the temporal sharing of MMIO devices. Specifically, the requirements of $\mathcal{A}$ and $\mathcal{B}$ to run periodically, make progress, and get a guaranteed opportunity to send out the alert cannot be realized with existing TEE platforms. Especially not considering that in our example, no application trusts any other application on the device, for example considering them as compromised by a strong software adversary.

2.2.3 Security & Availability Guarantees

We follow the established attacker model in TEE research (cf. Maene et al. [96]), where all software that is not explicitly part of an application's TCB is considered to be under the control of the attacker. We consider hardware-level attacks to be out of scope for our prototype. In particular, an attacker cannot physically disconnect components or control peripherals.

Under this model, the platform should provide the same guarantees as the described generalized base platform above, *i.e.*, confidentiality and integrity of mutually distrusting applications combined with the possibility to attest applications to remote parties. As a generalization of the availability requirements of the running example, the platform has to provide the following *additional* availability guarantees for a bounded number of protected applications:

- *Bounded activation latency:* the platform guarantees a specific finite bound on the maximal time that can elapse between an event (in the example case, a timer interrupt) and the execution of the first instruction of an enclave that wants to act on the event.

- *Guaranteed progress:* the platform guarantees that within a specific time interval T (e.g. a second), at least x percent of the CPU cycles goes to the monitoring application (where T and x can be configurable, but an application can securely attest these values to a remote stakeholder).
- *Guaranteed device access:* device drivers can be programmed to provide assurance to an application that it can acquire access to all devices it needs within a specific finite time T. Obviously, the temperature monitoring application needs to trust (or verify) the code of the sensor driver and communication channel driver, and use it appropriately to get these guarantees. But an important point is that *no other applications competing for the same resources need to be trusted.*
- *Safety independence:* faults in the executions of other applications do not impact the availability of the temperature monitoring application. Only the application itself (including dependent code) must be trusted (or verified) not to have faults (including security faults) to preserve availability.
- *No trust hierarchy:* the same guarantees can be given to multiple mutually distrusting applications. Two independent applications can perform monitoring tasks and compete for the communication channel to send out alerts, and both of them will get the availability guarantees we discussed, without either having to trust the other. It is in this sense that our platform is truly an *open* system: progress and real-time guarantees can be offered to a number of protected applications that run at the *same* priority.

Considering the last guarantee, we note that equivalent guarantees can only be given to competing applications up to an upper limit depending on the nature of the resource. Intuitively, no realistic guarantee can be given if the requirements exceed the available schedulability of the resource. This restriction spans across all shared resources such as time (managed and guaranteed by the scheduler), and attached peripherals (such as the temperature sensor and communication interface drivers). We see it as a software responsibility of each (trusted) resource driver to only provide a guarantee if this guarantee can realistically be given.

In summary, these guarantees make it possible to ensure for our example applications $\mathcal{A}$ and $\mathcal{B}$ that temperature alerts will be sent out within a hard real-time bound in the presence of buggy or malicious code on the platform. More specifically, the protected $\mathcal{A}$ is capable of achieving its goals even if $\mathcal{B}$ is malicious and attempts to monopolize resources, and vice versa. In Section 2.5 we will show how this simple application can be realized on our platform with

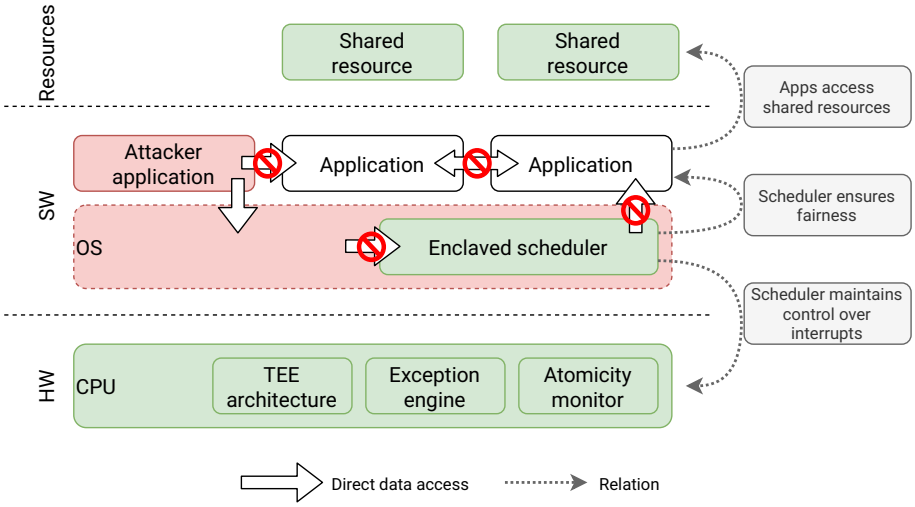


Figure 2.2: System overview with trusted components highlighted in green. The scheduler has exclusive control over interrupts and can enforce a periodic scheduling, but cannot access protected application enclaves directly.

the above availability guarantees. To the best of our knowledge, no other TEE is capable of providing these combined security and availability guarantees.

2.3 Design

In the following we present the design of AION that, based on conventional light-weight embedded TEE architectures, can bring strong temporal isolation guarantees to multiple, mutually distrusting applications. We base our prototype implementation on Sancus, but stress that the general design of AION is independent of the underlying platform. Figure 2.2 shows an overview of the AION system and its core components.

The first core component is the underlying hardware-based **TEE architecture** that provides the core guarantees of confidentiality and integrity. In the following, we focus only on TEE characteristics that are necessary in addition to the established protection mechanisms, e.g., how interrupts or violations of the TEE's security policy are handled. We are confident that these additions could be implemented on top of all discussed light-weight embedded TEE architectures. The second component of our design is a hardware-based **exception engine**

that is triggered whenever an interrupt or violation occurs. This exception engine cannot only interrupt unprotected but also protected, *i.e.*, enclaved, applications. Furthermore, the exception engine is triggered on any violation of a platform policy such as reading from protected memory or jumping into the middle of a protected code region. The third and fourth elements of Aion are an **atomicity monitor** and an **enclaved scheduler**. The hardware-based atomicity monitor ensures that the enclaved scheduler is the only entity that has full control over handling any system events, e.g. interrupts or violations. For this, the atomicity monitor implements a notion of *bounded atomicity* and carefully controls interrupt behavior during context switches, e.g., when entering an enclave. The sofware-level enclaved scheduler is the handler of all events on the system, and orchestrates the execution flow of the system when events occur. All four components play together to enable the scheduler to issue fair scheduling decisions. We will now detail these four core pillars of our design.

2.3.1 TEE Architecture

We build Aion around TEEs that provide memory isolation for dynamic enclaves. From the investigated TEEs, TyTAN [28] and Sancus [106], support these requirements natively.

In addition, Aion requires two additional features that need to exist to design our security architecture. First, violations of the TEE security policy should not result in a reset or in blocking the system. A system reset is a common solution to violations since illegal writes or reads from protected memory regions may only be detected after the offending instructions has completed. If an architecture detects a security violation after it occurred, a system reset prevents any malicious code to use the result or side effect of this access. In Aion, however, the platform must not be impacted by any offending instruction but instead proceed with an exception and hand control over to a handler of this violation. It is crucial that offending instructions do not complete but are instead either stopped or their effects rolled back before control is handed over to a violation handler in constant time. As such, the handler of the violation must not necessarily be privileged or trusted by any party.

Second, TEE-internal hardware operations must be interruptible. While we discuss preempting enclaves in Section 2.3.2, some operations of the TEE architecture may need a large amount of cycles to complete. Common examples of such operations are cryptographic operations or the enabling or disabling of enclaves. Adversaries in Aion are capable of arbitrary code execution and may attempt to stall the system by issuing long-running cryptographic operations. To prevent this, the TEE architecture must support the preemption of these

operations. A successful or unsuccessful completion must be notified by the hardware to the issuer of the operation when control is resumed so that benign applications can restart the operation in case of an interrupt; the policy for this must be part of the hardware-software contract to enable developers to design enclaves that can make progress. Additionally, the hardware must ensure that any cryptographic state is cleared and removed from memory before interrupts are handled to prevent information leakage. We implement our prototype of Aion on Sancus which builds on MSP430 and has no cache or advanced microarchitecture. Therefore, execution time is fully deterministic and only depends on the instruction type and memory accesses. This simplifies our approach but does not limit generality: Aion can be implemented on any TEE-platform for which a WCET-analysis is possible. Determining upper bounds for the execution of scheduler operations is the only strict requirement for Aion.

Also note that while remote attestation may on first glance not seem essential to Aion, attestation in Aion provides remote stakeholders with the guarantee that *(i)* the right code is loaded untampered in a protected application enclave; *(ii)* the scheduler enclave was loaded correctly, ensuring a fair availability policy; and *(iii)* expected implementations of shared drivers are used. We thus see attestation as an integral part of how Aion would be used in practice.

2.3.2 Exception Engine

Whenever an interrupt or a policy violation occurs, the exception engine in Aion is responsible for switching from the current job to the enclaved scheduler. This ensures that the scheduler can always fairly schedule the next application and ensure that all applications maintain a fair share of the resource CPU time. In its operation, the exception engine distinguishes between two types of exceptions: *interrupts* due to periodic or aperiodic events and *violations* of platform policies. Figure 2.3 shows a high-level flow of the exception engine. Note that violations are always handled immediately after the offending instruction completes but the handling of interrupts is delayed by the platform-specific global interrupt-enable flag. An immediate handling of violations ensures that even in atomic sections, dangerous violations are immediately handled and the offending job can be punished.

Handling interrupts On interrupts, the exception engine has to store the current state of the running job in a way that execution can be resumed at a later point. For this, the exception engine needs to distinguish whether the current execution is of an unprotected application or whether an enclave is being executed. For unprotected applications, the behavior of the exception engine

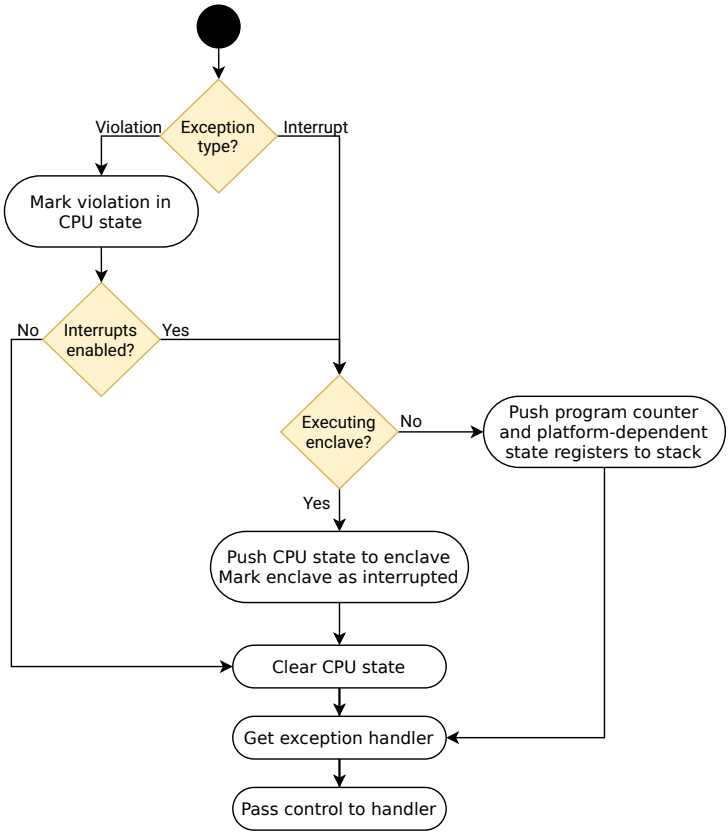


Figure 2.3: High-level flow of the exception engine. Two main paths are distinguished: interrupts and violations. On interrupts, context state is saved in the enclave. On violations, a marker is first set in the CPU state.

is the same as for regular platforms where usually only the current program counter and potential state registers need to be saved on the program's stack. Since the running program is unprotected, the process of storing the program state in the application's memory region can be a responsibility of the scheduler and be done in software.

For protected applications, however, *i.e.*, enclaves, the exception engine needs to store all context information of the running job in the enclave's protected memory. Depending on the implementation platform, the context information usually entails all CPU registers. This process is done in hardware because the enclaved scheduler should not have access to the protected memory of

the interrupted enclave and can thus not perform this process in software. After storing the context information in the enclave, all context information is cleared before handing execution over to the enclaved scheduler. Since enclaves can only ever be entered through predefined call gates, the enclave's entry routines must on their next execution, furthermore, also be able to detect whether the enclave was interrupted previously. Thus, the exception engine also leaves a marker for the enclave that it should restore its execution context instead of accepting potential execution parameters that could overwrite a currently running execution flow. The specifics of this marker can be left to the implementation of AION, e.g., storing a single bit at a known location is sufficient.

Handling violations In contrast to interrupts, violations do not occur during the normal behavior of a platform but are usually the result of an unauthorized attempt by an adversarial job. We consider two types of violations that are both handled by the exception engine: security and availability violations. Security violations are defined by the TEE architecture and revolve around the hardware protections of the TEE such as protecting memory regions or preventing illegal jumps into the middle of protected regions. Availability violations on the other hand are introduced by the atomicity monitor and occur whenever a program attempts to enter too long atomic periods or attempts to illegally prolong the current atomic period. We explain the atomicity monitor and how it enforces an upper bound on all atomic periods below.

For both types of violation, we can assume that they are not usually triggered by a benign job and it can be assumed that if a job experiences one, it is either controlled by the adversary, a victim of the adversary, or being tricked by the adversary, e.g., to access another protected memory region through an unchecked pointer [157]. Since the last example can be ruled out by proper input vetting of enclave code, we design AION around the assumption that any violation is the result of an adversary. To alleviate the impact this may have on applications that do suffer a policy violation during benign behavior, we additionally introduce a violation marker that is set on enclave violations in the CPU context to inform the enclave that it recently suffered a violation. The exact implementation is left to architecture specifics, but any available bit in a status register suffices as long as it cannot be set by software.

Figure 2.3 shows the behavior of storing violations on the left side. After setting the violation marker, the whole CPU state is stored as it would be for an interrupt. On its next entry, the enclave can check that its last operation was aborted due to a violation. However, if interrupts were not enabled at the time of the violation, the exception engine does not perform this context save to ensure

that it not accidentally overwrites an old interrupt context. This is needed since attackers could otherwise call into enclaves and create an availability violation at the cost of the called application.

If a violation occurs during the process of storing the CPU context, this process is aborted and the exception engine jumps ahead to clearing the CPU state. This ensures that the hardware cannot be tricked into performing memory writes to areas that the current enclave is not privileged to access.

2.3.3 Atomicity Monitor

To prevent attackers from impacting the availability of the system, it is necessary to block all attempts that completely disable interrupts. At the same time, the enclaved scheduler in Aion is the main driver of the resource CPU time and requires special privileges in regards to this resource. As such, the scheduler in Aion is the only entity that has the capabilities to disable interrupts on the platform. Since the scheduler is crafted carefully, this privilege does not change the availability guarantees of the system.

While denying any program aside from the scheduler the ability to disable interrupts is beneficial to the availability guarantees of the system, it is certainly not desirable to also prevent all benign usages of atomic sections. In addition to functionality issues that may arise for shared resources if they are interrupted in a critical state, there are also additional concerns in the context of enclaved programs. During entry of an enclave, atomic sections are crucial to allow the enclave to restore its interrupt context from memory before another interrupt context can be written over the current one. To overcome this limitation, we introduce a special `clix` instruction similar to the design of Masti et al. [97] which starts a bounded atomic period. Figure 2.4 shows the use and several edge cases of this `clix` instruction. When issuing a `clix`, the hardware disables interrupts for exactly x cycles after which it automatically enables interrupts again (Figure 2.4.a). Programs can choose x individually up to an upper bound that is set by the platform designer depending on the deployed shared resources. Any `clix` instruction that requests a number of cycles larger than the upper bound and any attempt to nest `clix` periods trigger an atomicity violation (Figure 2.4.b). We design atomicity violations to be handled by the exception engine as described previously and assume that the atomicity monitor clears all current state when experiencing a violation such as the current count of remaining cycles left in the `clix` period. Issuing atomicity violations ensures that attackers cannot perform `clix` instructions that are out of the bounds of the system's designers chosen acceptable worst-case latency between two interrupts. It, furthermore, ensures that attackers can never prolong their granted atomic

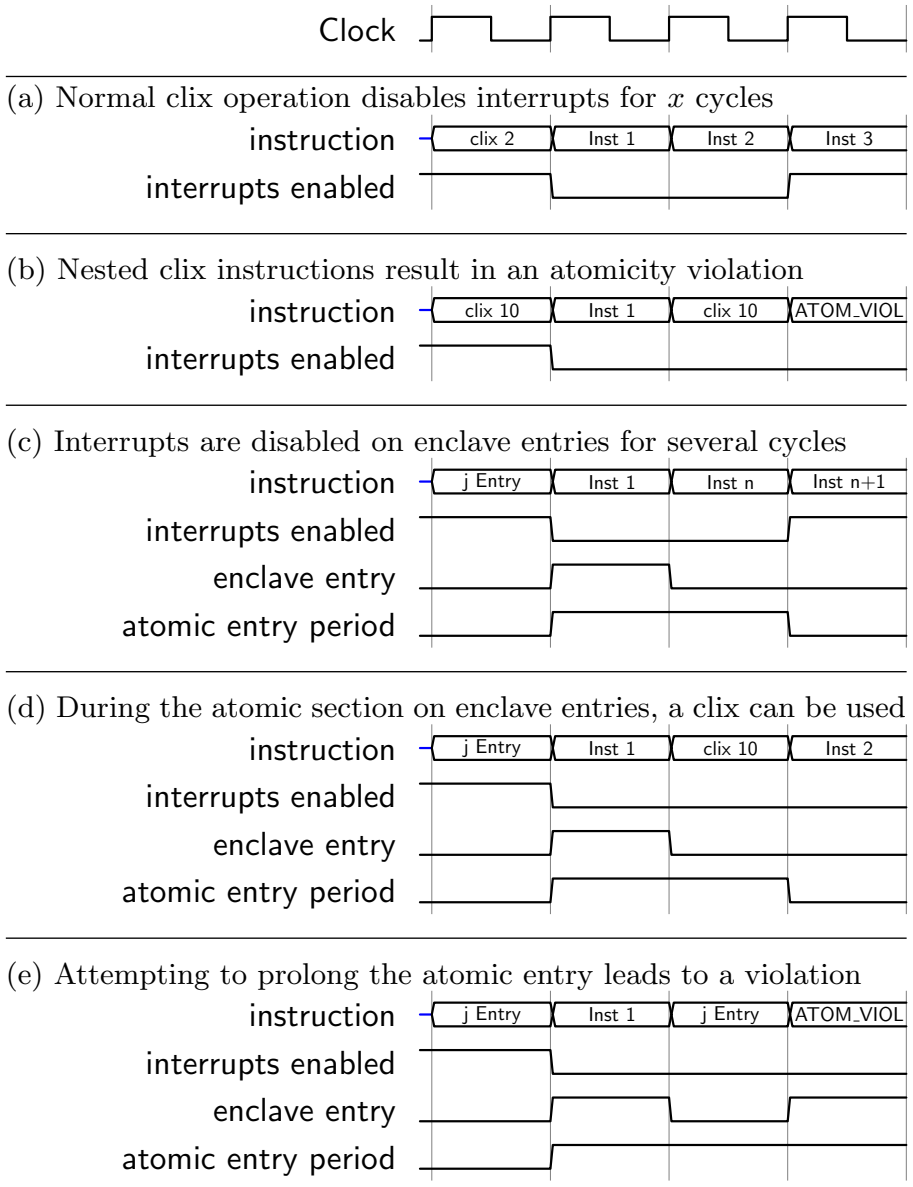


Figure 2.4: Representation of the desired behavior of bounded atomic sections. `clix` instructions temporarily disable interrupts but cannot be nested. On enclave entry, a short atomic period is started which can be prolonged with a `clix`.

period without at least experiencing one cycle of enabled interrupts in which an incoming event can be processed.

While the `clix` instruction technically allows to perform the critical part of an enclave entry in an atomic section, adversaries could still issue an interrupt right at the moment when an enclave is entered. This may lead to issues as an existing interrupt context in the enclave could be overwritten by the adversary's interrupt with a new context that points to the start of the enclave entry. Such a data loss and integrity violation is not acceptable. To prevent this, the atomicity monitor additionally ensures that on each entry of an enclave, *i.e.*, on each context switch into a new protected region, interrupts are disabled for a very limited amount of cycles as shown in Figure 2.4.c. This gives the enclave entry code enough time to issue a `clix` instruction of the length it needs to restore its interrupt context. Since the exact cycle duration that each application needs to be interruptible again may vary, we allow applications to define this cycle length via the `clix` instruction rather than automatically issuing a long atomic period at each enclave entry. Furthermore, this dynamic `clix` length at enclave entry allows each enclave to decide whether it wants to utilize several cycles of hardware-guaranteed progress before the scheduler could preempt this application again. For some applications, such a guaranteed immediate progress may be more valuable than other progress longer after the deadline. As can be seen in Figure 2.4.d, issuing a `clix` during the few cycles of an atomic entry period terminates the atomic entry and seamlessly proceeds into a `clix` period. However, any attempt to prolong this atomic entry is prevented with atomicity violations (Figure 2.4.e).

Our atomicity design serves two main purposes: First, Aion allows the use of atomic sections while at the same time maintaining hard limits on the activation latency of an arriving interrupt. Second, the length of issued atomic sections are purely in the responsibility of software under the restriction enforced by the hardware. This helps in the potential attestation of code that uses atomic sections and increases the performance of benign applications that do not always have to enter a long atomic period if this is not necessary.

A complete overview of the atomicity state machine can be seen in Figure A.1 in Appendix A.1.

2.3.4 Enclaved Scheduler

The previous core elements of Aion have laid the foundation for a trusted scheduler that is in full control of the shared resource CPU time. The exception engine ensures that all state is cleared and control is handed over to the scheduler on all interrupts and violations. The atomicity monitor limits the atomic periods

of any job besides the scheduler itself. To enable a scheduler to utilize this foundation and provide trusted scheduling, however, the scheduler must itself also be protected by the TEE architecture and, hence, run inside an enclave. This is crucial as the scheduler can only provide consistent and fair scheduling decisions if it is unaffected by any attempts of the adversary and if control is always deterministically returned to the same scheduler entry code. With the combination of these properties, the enclaved scheduler can provide a fair real-time scheduling of dynamic applications on an open system.

Practical implementations of AION benefit of a timer peripheral that is solely controlled by the scheduler. This allows the scheduler to ensure fair scheduling for configurable time periods and can also be used as a basis for a trusted time service for applications.

2.4 Prototype Implementation

We implemented AION on top of the Sancus TEE and the RIOT operating system, specifically Sancus 2.0 as presented by Noorman et al. [106] and RIOT in major version 2019.10 which bases on the original work of Baccelli et al. [17, 18]. We chose this combination as Sancus is an open-source architecture based on the 16-bit TI MSP430, running at 8 MHz, and RIOT is equally available as open-source and has support for MSP430 processors. Sancus already provides the desired confidentiality and integrity guarantees. However, certain modifications were still necessary, especially surrounding the additional requirements AION makes on the TEE architecture (cf. Section 2.3.1). Furthermore, because RIOT is designed to be a highly modular priority-based operating system, certain adjustments were required to the scheduler and the way threads are handled to implement an open system with this OS.

In the following we briefly describe Sancus and RIOT, and then discuss how we adapt these systems to implement our solution. The full source code of AION and the modified toolchains of Sancus and RIOT are available as open-source[3].

2.4.1 Background: Sancus and RIOT

The Sancus TEE Sancus [106, 107] is an open-source embedded TEE [96] with a hardware-only TCB that extends the memory access logic and instruction set of a low-cost, low-power openMSP430 [61] microcontroller. Sancus supports multiple mutually distrusting software components that each consist of two

[3] `https://github.com/sancus-tee/sancus-riot`

contiguous memory sections in a shared single-address-space. A hardware-level program counter-based access control mechanism [135] enforces that an enclave's private *data section* can only be accessed by its corresponding *code section*, which can only be entered through a single *entry point*. Sancus's generic memory isolation primitive can, furthermore, be used to provide secure driver enclaves with exclusive ownership over MMIO peripheral devices that are accessed through the address space. Since Sancus modules only feature a single contiguous private data section, however, secure I/O on Sancus platforms requires these small driver modules to be entirely written in assembly code, using only registers for data storage [108].

Sancus also provides hardware-level authenticated encryption, key derivation, and key storage functionality by extending the CPU with a cryptographic core. This cryptographic core can be used to implement secure communication as well as both local and remote attestation by employing a key hierarchy between the infrastructure provider, the application developer, and individual enclaves. Finally, Sancus comes with a dedicated C compiler that automates the process of enclave creation and hides low-level concerns such as secure linking, private call stack switching, and multiplexing user-defined entry functions through the single physical entry point.

RIOT OS RIOT is an open-source operating system for the IoT, which puts special emphasis on supporting real-time applications on resource-constrained devices [17, 18]. In contrast to other embedded OS kernels, RIOT provides the full set of features expected from an OS, ranging from hardware abstraction, kernel capabilities, system libraries, to tooling.

RIOT is designed to be tickless, which means that the scheduler is not executed at specific intervals but instead only when necessary. The standard RIOT model is a cooperative scheduling model where it is assumed that applications actively yield whenever they wish to pass control over to the next application. However, to also support periodic events, RIOT allows jobs to set timers to sleep for a period of time. For this, RIOT accesses the timer peripheral, tracks the passed time of the system, and maintains a list of active timers and the thread they are connected to. This setup is ideal for mixed-criticality systems as the highest priority job will always be scheduled next and can run as long as necessary until it either cooperatively yields to pass control over to the next job or until an interrupt arrives and stops the job. For applications of the same priority, however, RIOT assumes a fair and cooperative scheduling through yields which places all other applications of the same priority within an application's TCB.

The RIOT scheduler can provide scheduling decisions in constant time, *i.e.*, in O(1) due to its reliance on a bitmask that depends on the amount of configured

priority levels. Sleeping is implemented in O(n) due to an unlimited amount of possible timers.

2.4.2 Modifications to Sancus

We made multiple changes to the Sancus hardware to implement Aion. All of these changes are made under the assumption that a scheduler has a fixed enclave hardware ID of 1, *i.e.*, the scheduler is the first enclave that is loaded. Specifically, we *(i)* modified the exception engine to handle interrupts and violations according to Section 2.3.2, *(ii)* implemented an atomicity monitor component according to Section 2.3.3, *(iii)* placed restrictions on parts of the status register to only be modified by the scheduler, and *(iv)* made cryptographic operations interruptible (in an abandon-restart fashion).

All changes to the Sancus architecture are backwards-compatible with Sancus 2.0 [106] and the MSP430 specification. This was validated with the default tests provided by the openMSP430 project that Sancus is based on and with new tests for cases where we added functionality. To provide full backwards-compatibility with the specification, our availability restrictions do not come into effect before the first enclave, *i.e.*, the scheduler, is initialized.

In the following, we focus only on the essential aspects of our implementation that are not immediately derived from the design of Aion as presented in Section 2.3. Specifically, this applies to the exception engine, the status register, and the cryptographic core.

Sancus exception engine Sancus 2.0 originally only supports the preemption of non-enclave code. Thus, we extend the exception engine to perform the tasks as outlined in Section 2.3.2. In our implementation, we utilize a configurable location in the enclave data region to store the CPU context plus a violation marker when receiving an interrupt or a violation respectively. The context-saving hardware logic is subject to the same access-control checks as the interrupted enclave, and any violations during the processing of an interrupt or another violation lead to the abort of the context saving as shown in Figure 2.3.

In MSP430, an Interrupt Vector Table (IVT) at a fixed location in the memory layout is used to determine the handler of an interrupt. In our implementation, we assume that the scheduler registers itself for all interrupts and violations, and then locks the IVT by wrapping it in the protected section of a small driver enclave, thus preventing any further access to the IVT.

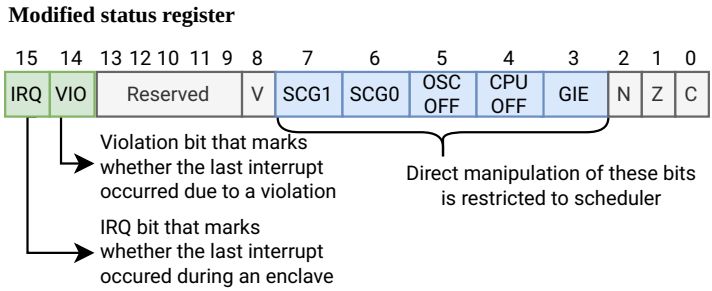


Figure 2.5: Overview of the status register and our changes. Bits highlighted in blue (bits 3-7) are restricted to the scheduler. Bit 15 marks whether the last interrupt occurred during an enclave. Bit 14 marks whether a violation occurred.

Status register modifications The MSP430 status register contains multiple flags e.g., for arithmetic operations and is stored on interrupts and restored together with the program counter on a `reti` instruction. However, in addition to these flags, the MSP430 status register also contains flags that are considered sensitive in Aion. Figure 2.5 shows an overview of the status register and our modifications. Most obvious, we restrict the disabling of the Global Interrupt Enable (GIE) bit to the scheduler. However, we allow the setting of this bit at all times which allows applications to terminate their own `clix` or atomic entry period ahead of schedule. Additionally, we also restrict bits 4 to 7 to the scheduler which could be used to completely switch off the platform, such as the CPUOFF flag, or which switch off the internal oscillator that is used as a timer. Furthermore, we add two flags to the reserved portion of the status register that are set by hardware and cannot be written from software: the IRQ flag (in bit 15) and the violation flag (in bit 14). The violation flag in bit 14 is set by the exception engine when it processes a violation and is the implementation of the violation marker as described in Section 2.3.2.

The IRQ flag in contrast exists for purely functional reasons and helps the scheduler to restore jobs as either unprotected code or as enclaves. Since by default the scheduler has no reliable method of deducing whether the hardware interrupted an enclave or unprotected code, the exception engine sets the IRQ flag after clearing the CPU state and before handing control to the scheduler.

Cryptographic core Finally, we changed the behavior of Sancus's cryptographic instructions to update the Zero flag (bit 1) in the status register to indicate whether the operation completed or was aborted due to an interrupt arrival.

The resulting abandon-restart semantics is similar to how Intel SGX handles long-latency cryptographic operations, such as `einit` [77, §40.3]. Specifically, whenever an interrupt request arrives during a cryptographic operation, the CPU resets the cryptographic core (without committing or leaking any internal state), sets the zero flag, and updates the program counter before state saving proceeds as usual via the exception engine. This behavior ensures that interrupt response times cannot be delayed by long-standing cryptographic operations (cf. requirements in Section 2.3.1). Interrupted cryptographic instructions can be simply restarted later when they are followed by a conditional jump that tests the zero flag.

2.4.3 Modifications to RIOT

In Aion, we need to protect the scheduler and its associated data structures from outside interferences. At the same time, it is desirable to provide a similar functionality as the unmodified RIOT. Thus, we map the scheduler enclave over the RIOT scheduler and incorporate core features of the RIOT timer. Since the scheduler is executed on every interrupt already, we also grant it exclusive access to the timer peripheral which we map into the protected memory region of the scheduler. This allows scheduling decisions not only based on expiring timers such as sleeping jobs, but it also allows other applications to use the scheduler as a source for trusted system timings. It, furthermore, enables the scheduler to be the only instance that monopolizes the shared resource of CPU time. In our prototype, the scheduler disables interrupts during its execution and will never interrupt itself. This increases the interrupt latency of our prototype and is not strictly necessary to uphold the defined guarantees. With more engineering effort, the scheduler could also be implemented to allow interrupts at carefully selected parts of its execution paths.

As discussed above, a fair scheduling can only exist if the default state of the system is schedulable. Any platform owner that accepts new application to be deployed to the open system must check that the requirements of the new application do not exceed the capabilities of the available shared resources. If the shared resources are schedulable, however, the Aion scheduler can enforce a fair share for each deployed application. For the prototype, we limit the number of maximum running or sleeping applications but allow the attacker to register additional applications up to this limit.

2.5 Experimental Evaluation

We evaluate Aion in two steps. First, we present a case study implementing the running example from Section 2.2.2. Then, we provide a cycle-accurate performance evaluation for all operations impacting the real-time performance of the hardware and the scheduler.

2.5.1 Case Study

We demonstrate the security and availability features of Aion by implementing the running example from Fig. 2.1. Our case study features three enclaved RIOT jobs that all run with the highest priority. These jobs implement the application enclaves $\mathcal{A}$ and $\mathcal{B}$, and an I/O enclave $\mathcal{I}$. The latter provides an interface to synchronously read the sensor and to asynchronously dispatch messages to a serial line. The enclaves make use of Sancus's TEE features [106], including isolation guarantees and secure linking between $\mathcal{A}$ and $\mathcal{I}$, and $\mathcal{B}$ and $\mathcal{I}$; they can further be remotely attested. All three enclaves schedule timer interrupts to be woken up at regular intervals.

In Listings 2.1 and 2.2 we illustrate interesting aspects of our implementation. Aion's development toolchain is based on that of Sancus and currently supports programming in C and assembly. We decided to present only the enclave entry functions (as opposed to internal functions that can only be called from within the same enclave) in Python-like pseudo code to reduce the complexity and focus on important security and software engineering aspects that are enabled by Aion. The C implementation of our case study is given in Appendix A.2 and as part of the open-source artifact.

I/O job and API Following Listing 2.1, $\mathcal{I}$ provides three entry points: `sync_input` returns a sensor reading; the code to operate the MMIO resource – a few assembly instructions – reside in the internal function `read_sensor`. The function first executes `clix` to ensure atomic execution of this operation. Following our semantics of `clix`, it is up to the developer to guarantee that `sync_input` completes with the end of the requested `clix` period. The execution of the `clix` itself is protected by the atomic entry period. Similar to `sync_input`, `async_output` is also an atomic function. But instead of performing the I/O operation immediately, the `payload` is buffered. The function may throw an exception if no free buffer is available for the specific calling context and we anticipate that $\mathcal{I}$ would provide guaranteed buffers for a number of protected jobs such as $\mathcal{A}$ and $\mathcal{B}$, while other jobs would have to share buffers. In our example, this decision is based on the Sancus `get_caller_id` primitive, which

```
def sync_input:
  CLIX <cycles to complete>
  return read_sensor()

def async_output(payload):
  CLIX <cycles to complete>
  try:
    i = buf_free(get_caller_id())
  if i != 0:
    output_buf[i] = payload
  except: fail

def async_io_task:
  while True:
    for i in output_buf
      output_buffer(i)
```

Listing 2.1: Pseudo-code of the I/O enclave $\mathcal{I}$ of our case study.

```
def worker:
  while True:
    t = sync_input()
    if (t > threshold):
      async_output("WARNING")
    sleep(1s)
```

Listing 2.2: Pseudo-code of the application enclaves $\mathcal{A}$ and $\mathcal{B}$

allows $\mathcal{I}$ to identify the calling enclave. We have hard-coded this for reasons of simplicity and discuss a more general implementation in Section 2.6. Finally, `async_io_task` is an interruptible function to output buffered payloads from `async_output`. The implementation of `output_buffer` would again be atomic to ensure non-interference during the I/O operation. Indeed, $\mathcal{I}$ is free to implement a wide range of policies for accepting and executing I/O operations. $\mathcal{A}$ and $\mathcal{B}$ can attest $\mathcal{I}$ to be ensured that they use an I/O implementation suitable to implement their requirements.

Application jobs The application enclaves $\mathcal{A}$ and $\mathcal{B}$ can be implemented as illustrated in Listing 2.2. A single function `worker` will use the functionality provided by $\mathcal{I}$ to acquire sensor readings, evaluate these readings, and, if necessary, queue a warning message with $\mathcal{I}$. We assume that $\mathcal{I}$ is programmed such that our $\mathcal{A}$ and $\mathcal{B}$ are guaranteed a free I/O buffer once per second, thus we do not handle the exception. Other applications, in particular code that is not enclaved, may not enjoy these guarantees and therefore need to handle the exception. The application then schedules a sleep of 1 s and is guaranteed

to be woken up by the scheduler when this time period is elapsed, plus the scheduling margins summarized in Table 2.2. Note that our application does not make use of `clix` and is therefore interruptible. Making the execution of $\mathcal{A}$ and $\mathcal{B}$ entirely atomic is neither feasible (nested `clix` with $\mathcal{I}$ are not allowed) nor intended, as this would reduce the responsiveness of the overall system. However, even if $\mathcal{B}$ would deviate from the behavior in listing Listing 2.2 by performing a `clix` or causing a violation, this would not impact the security or availability of $\mathcal{A}$, which we discuss more comprehensively in Section 2.6.

Our case study shows that applications and drivers can be implemented such that, even in the presence of an uncooperative or malicious application that monopolizes system resources and maximizes delays, well-behaving protected applications make progress with deterministic latencies.

2.5.2 Performance Evaluation

One core performance metric of Aion implementations is the activation latency of applications. This activation latency is the time from when an application should be scheduled up to the time when control is actually passed over to it and it can start executing. In the following we consider the best and worst-case activation latencies for our prototype. An important characteristic of our prototype implementation is that any operation that the scheduler performs itself is atomic, *i.e.*, interrupts are disabled during scheduler execution so that the scheduler will not interrupt itself. In addition to regular scheduling, the scheduler also offers multiple operations to applications that return back to the caller or switch to another application. This means that activation latencies of application may be delayed by currently running scheduler operations. We first evaluate the performance of each scheduler operation in the best and worst-cases and use the results from this evaluation to perform an in-depth analysis of the activation latency of pending applications.

All timing overheads below are measured in CPU cycles and were retrieved through repeated measurements with the prototype implementation in a cycle-accurate simulation of Aion with Verilator [134]. Note that all performance numbers depend on the implementation of the trusted scheduler and show observed cycles only. Our prototype can only be seen as a baseline for real-world performance, that could be improved substantially with additional development effort.

Table 2.1 shows an overview over the timing overhead of all operations that applications can request from the scheduler. All scheduler operations are carefully designed to have a constant worst-case execution time. The remaining differences between best and worst-case execution time mostly depend on the

Table 2.1: Detailed overhead in observed cycles for the operations provided by the scheduler.

Scheduler operation	Best case (cycles)	worst-case (cycles)
Create job	688	860
Exit job	512	736
Sleep	1124	1320
Yield	424	628
Get time	212	

amount of already scheduled or pending applications in the system. Since the prototype implementation places a sensible upper bound on the number of maximum running or sleeping applications, the worst-case execution times are strictly bounded and cannot be extended by adversaries. The longest operation that an adversary can attempt is to sleep while the maximum amount of other applications are already sleeping, which means that the scheduler needs to insert a new timer into a list of the maximum length. We observe a deterministic overhead of 1320 cycles for this operation.

Building on these first evaluation numbers, we craft an attacker that *(i)* enters an adversary-controlled enclave right before the victim deadline, *(ii)* executes a `clix` of the maximum length, and finally *(iii)* enters the scheduler with the worst-case sleep operation before the `clix` expires. At the end of the triggered scheduler operation, the scheduler will then detect the pending interrupt and process that interrupt instead of returning back to the adversary or another application. This represents the longest chain of events that an attacker can craft before a periodic enclave is executed. Table 2.2 shows the best and worst-case latencies that are possible for such an application deadline. In the absence of an attacker (*i.e.*, in the best case), interrupts are already enabled (*i.e.*, GIE=1) when the application is to be woken up, and the exception engine can process the interrupt immediately. In the presence of an attacker, however, the attacker would perform the sequence of steps as described above in order to delay the handling of the deadline. Since in our implementation, interrupts are disabled during scheduler operations, this prolongs the time until an interrupt is triggered by the time of the running operation. This bounds the worst-case latency between an issued interrupt and its actual processing in the scheduler by a maximum of 2330 cycles (10 cycles of atomic entry, 1000 of `clix` operation, followed by the 1320 cycles of the worst-case sleep operation). Note that the adversary does not benefit from creating a violation during the last cycles of the `clix` instruction as a violation is also handled by the scheduler which can check whether other interrupts are currently pending before resuming execution of a job.

Table 2.2: Detailed overhead for an event that preempts a running job. Shown are measurements with default Aion parameters and the overheads in the best and worst-case. Values in parentheses show the worst-case in the absence of an attacker and are zero for the crafted attacker scenario.

Task/Stage	Best case (cycles)	worst-case (cycles)
1. Interrupt arrival	0	10 + `clix` + 1320
2. Interrupt processing	7	(35)
3. Scheduler entry	157	(115)
4.1 Timer	1356	4075
4.2 Scheduler run	443	443
5 Scheduler resume	72	72
Activation latency	2035	5920 + `clix`

Processing the interrupt in hardware takes 7 cycles if an unprotected job is being interrupted, while interrupting enclaved jobs takes 35 cycles. The overhead stems from the additional work to store the CPU context in the enclave versus only storing the program counter and status register on the unprotected job's stack. This overhead is reversed on entering the scheduler for unprotected code (157 cycles) versus entering the scheduler after interrupting an enclave (115 cycles). In the crafted attacker scenario, the scheduler can detect the pending interrupt at the end of the running operation and before it would resume execution to the next application. Thus, in the worst-case, steps 2 and 3 are skipped by the scheduler as it can start processing the interrupt without needing to reenter itself.

In our prototype, processing a timer tick requires the processing of all software timers to evaluate whether a software timer is ready to be fired. This means that in the best case, no timer has to be processed, leading to a latency of 1356 cycles while in the worst-case, all 15 jobs currently have set a timer which leads to a latency of 4075 cycles. Identifying the next job to schedule takes a static duration of 443 cycles as periodic enclaves are always scheduled with the highest priority on the system. Resuming from the scheduler then takes 72 cycles.

Overall, our prototype can *guarantee* an activation latency of 2035 cycles in the best and 6920 cycles in the worst-case. This means that in the presence of an active adversary that controls all 14 other threads besides the victim thread and performs the sequence of steps as described above, our best-effort Aion prototype can guarantee that the first guaranteed application to be scheduled is served at the latest 6920 cycles after its trigger occurred. We discuss below what activation latency can be given to any application other than the first to be scheduled if multiple applications received guarantees simultaneously.

2.6 Discussion and Security Analysis

Confidentiality and integrity Firstly, our reliance on TEEs and enclaved execution protects $\mathcal{A}$ and dependent code from a range of attacks on confidentiality and integrity. TEEs and their limitations are well understood in general [96] and for Sancus in particular [106]. For example, it is clear that enclaved applications must be developed such that they are free of vulnerabilities that allow an attacker to hijack the enclave's control flow or to extract secrets. The TCB reduction provided by TEEs helps to implement secure enclaves, relying on extensive code reviews, testing, and formal verification, which are orthogonal lines of research.

An important consideration to nuance the architectural confidentiality guarantees offered by TEEs is information leakage through software-exploitable side channels [57]. Fortunately, the class of light-weight embedded systems considered by AION have a significantly reduced microarchitectural attack surface in comparison to notoriously complex x86 processors. In particular, known side-channel attacks on MSP430-Sancus platforms are mostly reduced to classic start-to-end timing [63], as well as more fine-grained interrupt latency timing attacks [158]. Side channels can generally be ruled out entirely by manually rewriting the application code to adhere to established constant-time programming best practices [57]. Alternatively, in the case of deterministic execution platforms such as MSP430, static code balancing solutions can provide an automated solution, either by transparently generating compensation code in the compiler backend [170] or statically analyzing execution path timings at the level of the generated assembly code [44, 117]. Finally, for the particularly relevant problem of interrupt latency timing side channels [158], recent work has proposed a provably secure, hardware-level padding defense for a simplified version of Sancus [29]. We leave integration of such architectural changes to further strengthen AION against side channels as future work.

Guaranteed availability Importantly, the activation latencies from Section 2.5.2 apply to $\mathcal{A}$ and $\mathcal{B}$ in our case study, even in the presence of strong software-level attackers who are capable of manipulating all software that is outside of the TCB. Specifically, we consider attackers that might attempt to *(i)* block the CPU by performing extensive uninterruptible computations, *(ii)* influence the scheduler to disrupt the execution of (other) jobs, *(iii)* block I/O resources through continuous use, or *(iv)* cause illegal memory access or atomicity violations.

First, considering attack *(i)*, a misbehaving or malicious enclave can try and prevent progress by using the `clix` instruction and potentially invoke a scheduler operation. This is limited to a fixed number of cycles after which the scheduler

will serve pending interrupts and schedule other jobs. In Aion, `clix` and scheduler operations are the only means by which an application can prevent interruption. Importantly, `clix` periods cannot overlap to form continuous uninterruptible sections.

Alternatively, in attack *(ii)*, the attacker could try and schedule many short sleeps to maximize scheduling effort. We consider this attack in our evaluation and show that it has a substantial but still deterministic impact on the available CPU cycles for applications (cf. Section 2.5.2), and that the attack does not impact the baseline guarantees. The attack can be prevented by a scheduling policy where sleep requests below a certain threshold are not accepted, or where a misbehaving job is terminated.

In attack scenario *(iii)*, attackers try to continuously use an I/O resource. This can be ruled out by implementing `clix`-based atomic interactions with I/O drivers, which are followed by a scheduler interaction. As we have illustrated in our case study, it is feasible to program enclaved device drivers that either synchronously or asynchronously serve application enclaves, where the entry functions for applications have a bounded execution time and return within a single `clix`. This prevents the attacker from continuously blocking a resource and guarantees deterministic worst-case latencies for the next scheduling decision. Sancus's secure I/O functionality [106] can be used to guarantee that no code other than the driver enclave has access to the memory addresses used to control the peripheral, excluding non-driver code to interfere with the peripheral.

Finally, considering attack *(iv)*, Aion's exception engine guarantees that all interrupts, including violations of platform policies, are handled by the scheduler and do not trigger a platform reset. The specifics under which jobs are scheduled and how violations are handled are subject to the protected scheduler implementation.

Aion provides real-time guarantees based on a deterministic worst-case latency that is followed by M cycles of progress. By means of specific scheduler implementations, more elaborate policies can be provided, including the "at least x % of the CPU cycles per interval T"-guarantee from Section 2.2.3. For this, the scheduler must *(a)* allow at most N jobs with availability guarantees, *(b)* implement round-robin scheduling among these N jobs, and *(c)* run on a platform where `clix` provides atomicity for M cycles. Then each of these N tasks is guaranteed to execute at least M CPU cycles (within a `clix`) of every $T = (5920 + M) + (1845 + M) * (N - 1)$ cycles. This is under the assumptions that $N - 1$ jobs are under attacker control, all attacker jobs are placed before the victim job in the round robin scheduling, and the attacker jobs all schedule a timer to be woken up together with the victim. Furthermore, each attacker job executes a maximum `clix` for M cycles, which ends in a scheduler invocation

where the job schedules a timer for the next period. Thus, the first scheduled job experiences the above calculated worst case delay while the scheduler will only need to perform steps 1, 4.2, and 5 from Table 2.2 for the remaining jobs. For our prototype implementation with 15 allowed jobs and a `clix` length of 1000, the absolute worst-case activation latency for the last-scheduled victim job is $6920 + 2845 * 14 = 46750$ cycles. This represents the absolute worst-case where the platform developer decided to provide the same guarantees to 14 attacker jobs other than the victim job and it shows that our system can give deterministic guarantees based on highly flexible platform configurations.

Using attestation Applications include dependent code in their TCB, e.g. device drivers or the scheduler, and trust these for availability. The trustworthiness of this code is to be established by validation techniques beyond the scope of this paper. Remote attestation of the application enclave, together with Sancus's secure linking mechanism [106], give the deployer the guarantee that the application is indeed executing on a platform with the intended properties. For this, the scheduler and I/O drivers must be provided as enclaves and implement scheduling and access policies in code, the identity of which is then part of the attestation procedure. Enclaves can make use of mutual attestation and rely on enclave IDs to identify each other and provide specific guarantees, such as the availability of output buffers for $\mathcal{A}$ and $\mathcal{B}$ in the case study.

Our case study illustrates these features in a rather static scenario and based on fixed enclave IDs. To enable the open system that we describe in this paper, where protected applications can be deployed at any time and without a platform reset, applications and driver enclaves need to provide APIs that allow an application to, e.g., request a guaranteed I/O buffer, and to communicate success or failure to the deploying stakeholder after the initial attestation. This allows the deployer to ascertain schedulability of a deployment.

The latter approach also enables the use of I/O devices that require more complex access policies and that cannot complete an I/O operation atomically. For instance, a sensor might need to be calibrated for a specific use and multiple applications may require different calibrations. We believe that the AION design is flexible enough to integrate adequate access logic for such scenarios into driver enclaves, yet our MSP430-Sancus platform, being a very light-weight 16-bit processor, has limitations regarding the implementation size of application and driver logic.

Summary, limitations, and future work As a result of our joint spatial and temporal isolation, an application's security is no longer impacted by faults

in other applications. Specifically, vulnerabilities in $\mathcal{B}$ may lead to $\mathcal{B}$ being compromised, and scheduling faults caused by $\mathcal{B}$ may lead to the termination of $\mathcal{B}$. But, these events do not affect the security and availability of $\mathcal{A}$, and vice versa. Importantly, Aion does not impose a hierarchy of trust or criticality on applications. We enable multiple mutually distrusting and non-collaborative applications that operate at the same "priority" to execute under equally strong security and availability guarantees.

We consider a hardware attacker with the ability to arbitrarily trigger external interrupts to be out of scope for Aion. However, a platform where the scheduler is capable of temporarily masking these interrupts or disabling interrupts completely, would be able to resist these attacks. We note however, that this could compromise the trusted time guarantees of the scheduler if a timer tick is missed.

A specific challenge of Aion comes with the use of cryptographic operations for attestation or secure communication, which may take many CPU cycles to complete. Sancus [106] implements these operations in hardware for reasons of security and performance. While Aion makes cryptographic operations interruptible, the state of the cryptographic engine is lost upon interruption and the operation needs to be restarted entirely. Therefore, these operation complicate timing analysis and may prevent applications from making progress if they cannot complete a cryptographic operation within a `clix`. There are several ways to address these issues, e.g. by making the crypto engine resumable, tuning the semantics of `clix` to specific progress requirements, or relying on cryptographic operations in software, which we will investigate in future work.

2.7 Conclusion

We presented Aion, a configurable security architecture that can preserve real-time availability guarantees for embedded systems even in the presence of a strong software attacker. This set of guarantees is especially of interest for open systems that execute arbitrary dynamically deployed code from multiple, mutually distrusting stakeholders which all request their same fair access to resources. Aion is the first design for modern TEE architectures that provides a strong notion of trusted scheduling, derived from preemption, bounded atomicity, and an enclaved scheduler. We implemented and evaluated a prototype of Aion on a light-weight TEE and conclude that our system can deterministically guarantee a worst-case latency of 6920 cycles until a protected job is scheduled. This allows platform developers to derive more complex scheduling policies that can enable a future class of truly open IoT systems.

Chapter 3

Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments

This chapter was previously published as:

> **F. Alder**, J. Van Bulck, J. Spielman, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments". In: *Digital Threats* 3.2 (Feb. 2022)

Which was a result of the conference publication:

> **F. Alder**, J. Van Bulck, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI poisoning attacks on Intel SGX". in: *Annual Computer Security Applications Conference (ACSAC)*. 2020, pp. 415–427

Preamble

This contribution presents a previously overlooked part of the x86 register set regarding application binary interface (ABI) sanitization in Intel Software Guard

Extensions (SGX) enclaves. In prior work, Van Bulck et al. [157] presented similar vulnerabilities in enclave shielding runtimes. Some of the issues in their work were concerned with the x86 flags register, and they demonstrated that these registers could even be used to extract enclave secrets. This work sheds light on a different part of the x86 architecture: the complex landscape of floating point calculations.

Initially, the x86 central processing unit (CPU) architecture was extended with an x87 floating-point unit (FPU) that can accelerate integer and floating point calculations. Later, single instruction multiple data (SIMD) instructions were added that allow to perform multiple such calculations simultaneously with a single instruction. For this, Intel added another processor extension to the legacy x87 FPU, called the Streaming SIMD Extensions (SSE). Nowadays, extensions like advanced vector extensions (AVX) build out this extended feature set even more. Crucially, however, the old FPU features remain in the processor feature list due to legacy reasons. And even in modern software such as the GNU Compiler Collection [56], the legacy x87 FPU is still used to perform extended precision math operations.

This complexity led us to investigate the status and control registers of these extended feature sets and how they are sanitized on enclave entry. As a result, our work leads to software fault injection attacks that purely require setting FPU precision and rounding modes as well as exception masks to malicious values before entering an enclave. Interestingly, Open Enclave, Enarx, and Gramine (formerly Graphene) had initial mitigations against incorrect FPU configurations before work in this contribution began. While their mitigations prevented an attack of setting the precision or rounding mode to a malicious value, specifically the mitigation applied by Open Enclave did not protect the enclave from maliciously blocking floating point registers. This blocking leads the FPU to silently return a `NaN` result, *i.e.*, to notify that the calculation failed and the result is not a number. In the case of enclaves, such a silent error can be highly problematic, and we worked with Microsoft for Open Enclave as well as with the Rust enclave development platform (EDP) developers to perform a restoration of the full extended CPU feature set on enclave entry. Such a full restore is more expensive in terms of computation time but is the only stop-gap approach to ensure that no partial mitigations can be found to be incomplete in the future. Overall, our contribution led to patches in four different projects, namely the Intel SGX software development kit (SDK), the Microsoft-maintained Open Enclave, the Rust compiler, and the research runtime Go-TEE.

Together with the conference publication, our attacks were published as an open-source artifact that was peer-reviewed and received the highest ACM rating of “Artifacts Evaluated - Reusable v1.1”. Our artifact makes use of a

Docker container that can reproduce the presented issues also on non-SGX hardware by means of using the Intel SGX simulation mode, considerably lowering the required effort to reproduce and for future research to extend our artifact. Internally, this artifact has been used by a master's student and has proven to be useful for the issues described in Chapter 4. This paper's and artifact's overall contribution subsequently received the "distinguished paper with artifacts" award at the ACSAC 2020 conference. The award led to the journal publication that is presented in this dissertation as part of a special issue in the journal Digital Threats: Research and Practice. The extended version of the paper that is also part of this dissertation below presents the same FPU attack on the RISC-V-based Keystone architecture, which shows that the underlying issues behind this attack vector are not constrained to a single architecture.

After publication, this contribution has led to several follow-up works and a master's thesis. To further investigate the structure and issues arising from the ABI layer in shielding runtimes, the master's thesis analyzed the Intel SGX SDK and discussed and proposed a unified ABI code [119]. We furthermore analyzed the enclave shielding runtime landscape and how vulnerability mitigations are applied across runtimes [153]. Fundamentally, this work also laid the foundation for Pandora which we present in Chapter 4. As part of a wider community outreach, we disseminated this work to the wider industry and developer community by presenting it at the Remote Chaos Experience (RC3) which was the online event for the Chaos Communication Congress 2020, and at the 2nd Intel SGX Community Workshop hosted by Intel.

Lastly, an interesting development after the publication of this work has been an advisory published by Intel in early 2022 [73]. This advisory, called the MXCSR configuration dependent timing, refines our earlier established recommendation from this work to set the MXCSR register to the value `0x1f80`. The new recommended value by this advisory is a value of `0x1fbf` to prevent timing side channels that result from setting exception bits. This progression from multiple partial patches to a thus-far final advisory of a specific value highlights the moving target nature of the seemingly straightforward issue of ABI sanitization. The new advisory further underlines the complexity involved in sanitizing extended processor feature sets when dealing with hardware-based isolation mechanisms.

3.1 Introduction

In recent years, several Trusted Execution Environments (TEEs) [96] have been developed as a new security paradigm to provide a hardware-backed approach of securing software. Their promise is that applications can 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 [21, 70, 99]. 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 SGX [40], 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 [78] reference implementation by Intel, several other mature enclave runtimes have been developed, including the Microsoft-maintained Open Enclave [101], Fortanix's Rust-EDP [53], Graphene-SGX [144], and SGX-LKL [118]. Similarly, shielding runtimes have been developed for TEE architectures beside Intel's SGX, such as for the RISC-V based Keystone enclave [86] or OP-TEE [111] for ARM TrustZone.

Attacks on enclave shielding runtimes. A recent systematic vulnerability assessment [157] 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 3.1. 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

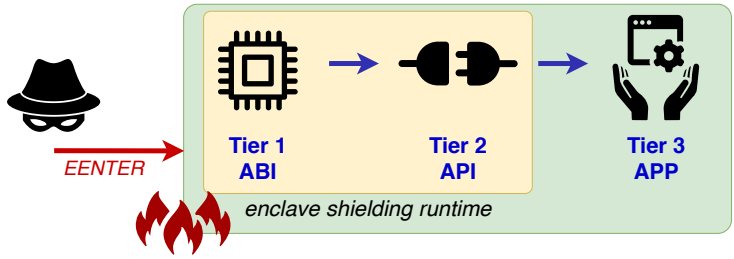


Figure 3.1: Enclaved application binaries are transparently shielded by sanitizing untrusted ABI and API-level state.

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 [32, 80, 116, 157] 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 [34] and more recently in TEE scenarios [32, 80, 95, 116, 157]. 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` [78], a secure type system as in Fortanix's Rust-EDP [53], or by automatically scrutinizing the enclave API through symbolic execution [80] and even formal interface verification efforts [60, 165]. Furthermore, prior work exists to analyze enclave code via symbolic execution in order to reason about API-level attack surfaces [35]. Another example of insufficient API-level sanitization is the lack of scrubbing in uninitialized structure padding reported by Lee and Kim [88], 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 [48, 157] 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 paper, 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 5 different runtimes, including widely deployed production-quality implementations such as Intel's SGX-SDK [78], the Microsoft-maintained Open Enclave [101], and Fortanix's Rust-EDP [53].

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 [111] runtime for ARM TrustZone properly sanitizes the FPU, we were able to reproduce the attack also in the Keystone runtime [86] 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 [48, 157] 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 [147]. 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 [22, 87, 157], undervolting [104], or incorrect transient-execution paths [33, 151, 154]. By contrast, our work shows that, even when the processed data is 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 [173]. 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 [173], segmentation faults [68], or alignment-check exceptions [157]. 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 [53, 101] 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 Open Enclave. While the initial mitigation attempt in Open Enclave 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, Open Enclave 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`

3.2 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.

3.2.1 Intel SGX

Intel Software Guard Extensions (SGX) [40, 77], 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 3.1) 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.

3.2.2 x87 FPU

The x87 FPU [77] 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

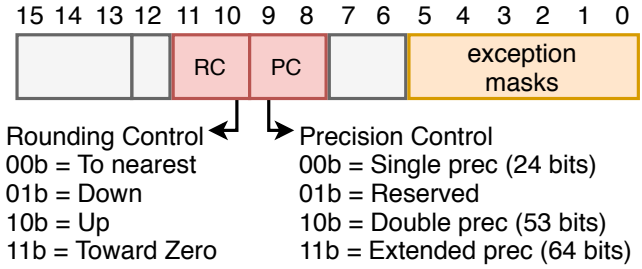


Figure 3.2: Layout of the x87 FPU control word.

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 3.2. 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 cannot 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 *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 3.2 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 to 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

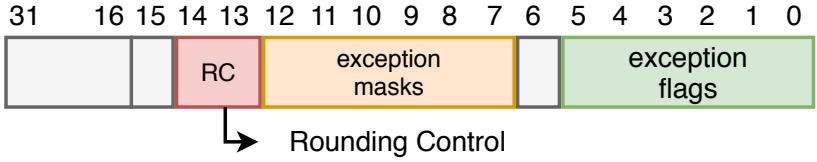


Figure 3.3: Layout of the `MXCSR` control/status register.

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 [52, 93]. Furthermore, on Intel processors supporting MMX technology [77], 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" [93].

3.2.3 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) [77] 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 3.3. 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 [77]. 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 predefined state `0x3f80` and preserved across function calls [52, 93].

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 [56]. Similarly, Microsoft Visual C++ defaults to the SSE for modern 64-bit applications [100]. 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” [56]. 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 [56]. 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.

3.2.4 Other Processor 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 [19, 86, 168].

RISC-V RISC-V defines 32 registers for floating-point data [167, §11.1] and a control and status register for floating-point operations, named the `fcsr` register [167, §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 3.3.2.

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 [167, §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 3.4 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 [13, §A1.5]. In 32-bit mode (AArch32), the `FPCSR` combines status flags (e.g., zero, carry, etc.) and control flags (e.g., rounding mode) [14]. 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.

3.3 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.

3.3.1 Attacker and System Model

We assume the standard Intel SGX threat model [40] 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 trusted computing base and aims to protect even against adversaries who have gained root access to the target platform [159], 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.

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

Following widespread industry practice [21, 51, 53, 58, 78, 101, 118, 149], 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 [157]. 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 [157].

We assume that the Intel SGX TEE is properly patched against microarchitectural vulnerabilities [33, 151, 154], 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 [52] 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 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 [56].

3.3.2 ABI Poisoning Attacks

While trusted code can be relied on to respect ABI calling conventions [52, 93], 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 3.3.3 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 3.2.2 and Figure 3.2. 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 3.5 and 3.6 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 [173] 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 [159] 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 advance production enclaves exactly one instruction at a time [102, 159]. 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 3.2.3 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 3.2.3 and Figure 3.3. 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 3.4 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

Table 3.1: Proof-of-concept attack executed inside an enclave.

FPU	Rounding	`arccos(-1)` $= \pi$	`2.1` $*$ `3.4` $=$ `7.14`
Single precision	To nearest	3.14159265358979323851280895	7.13999986648559570312500000
	Downward	3.14159265358979323829596852	7.13999986648559570312500000
	Upward	3.14159265358979323851280895	7.14000034332275390625000000
	To zero	3.14159265358979323829596852	7.13999986648559570312500000
Double precision	To nearest	3.14159265358979323851280895	7.13999999999999968025576890
	Downward	3.14159265358979323829596852	7.13999999999999968025576890
	Upward	3.14159265358979323851280895	7.14000000000000056843418860
	To zero	3.14159265358979323829596852	7.13999999999999968025576890
Extended precision	To nearest	3.14159265358979323851280895	7.14000000000000011567136130
	Downward	3.14159265358979323829596852	7.14000000000000011523768050
	Upward	3.14159265358979323851280895	7.14000000000000011567136130
	To zero	3.14159265358979323829596852	7.14000000000000011523768050
MMX	Any	`-NaN`	`-NaN`

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 B.1. 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 3.1 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 towards 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 B.2 in Appendix B.1 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 targets victim code which solely relies on x87 calculations, however, can be seen in the right column of Table 3.1. 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 to utilize 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 3.2.4, 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 [86] 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 a Keystone enclave. The untrusted host sets different rounding modes as for x87/SSE. With this, we reproduced the results of Table 3.1 (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 3.4.

3.3.3 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 7 open-source projects summarized in Table 3.2. Our selection was motivated by a recent extensive study [157] of popular Intel SGX shielding runtimes, which we extended with two newer runtimes [51, 58] that were not analyzed before. Particularly, we examined all predominant SGX shielding solutions in use by industry, namely Intel's SGX-SDK [78], Open Enclave [101], Fortanix's Rust-EDP [53], and RedHat's Enarx [51], as well as three relevant runtimes that were, at least initially, developed as research projects, namely Graphene-SGX [144], SGX-LKL [118], and Go-TEE [58]. In addition, as a non-SGX example, we also considered the RISC-V TEE Keystone [86]. 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

Table 3.2: 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). When applicable, applied and potentially remediated Patches are provided.

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

* Includes derived runtimes such as Apache Teaclave's Rust SGX SDK [143] (formerly Baidu Rust-SGX [165]) and Google's Asylo [64].
** as of April 2021

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 3.2 is that prior to October 2019, *i.e.*, before the initial Patch by Microsoft in Open Enclave, *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 Open Enclave 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 [143] (formerly Baidu Rust-SGX [165]) and Google's Asylo [64], that build on top of the SGX-SDK.

A second tendency in Table 3.2 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 3.2 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

[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.

a migration process in order to utilize the code base of Open Enclave in favor of self-written assembly stubs. As such, once SGX-LKL is fully migrated to utilize Open Enclave, it will inherit the mitigations implemented there.

In response to our disclosure, Rust-EDP adopted the original mitigation strategy of Open Enclave, 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 Open Enclave 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 Open Enclave 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 paper, 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 [91] may be vulnerable to our attacks as well.

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

3.4 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 [173], segmentation faults [68], or alignment-check exceptions [157]. 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 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 [10]. 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 3.1. Note that the enclave API does not expose the internal result value to the attacker

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

Listing 3.1: Example enclave code vulnerable to secret extraction through a floating-point exception side channel.

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 Appendix B.2). 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`.

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
```

For our experiments, we set epsilon $= 0.00001 \cdot 10^{-308}$. For this bound, Algorithm 1 requires a fixed number of 1040 invocations of the ecall to recover a secret operand. We ran this algorithm for 1000 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 3.4. The maximum observed error was $3.667689888908754 \cdot 10^{-6}$, with the average error being $6.2648851729085662 \cdot 10^{-7}$.

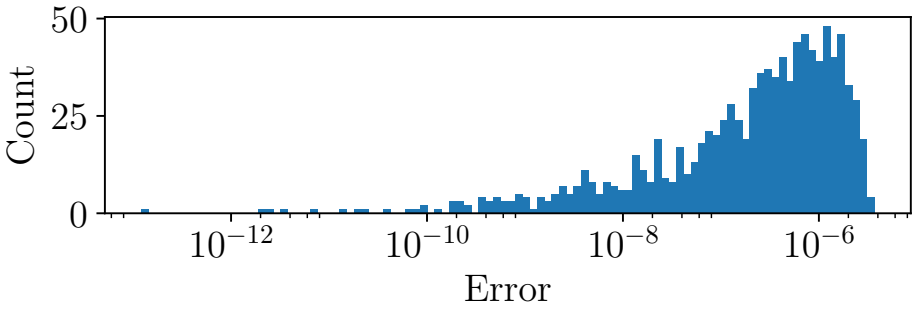


Figure 3.4: Histogram over the error of the recovered secret for 1000 samples (x-axis in log scale).

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:

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

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 [171] modified to run inside an SGX enclave. The enclave code includes two simple networks—a network that replicates a binary AND operation (cf. Figure 3.5) and a classifier based on the iris dataset [46]—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.

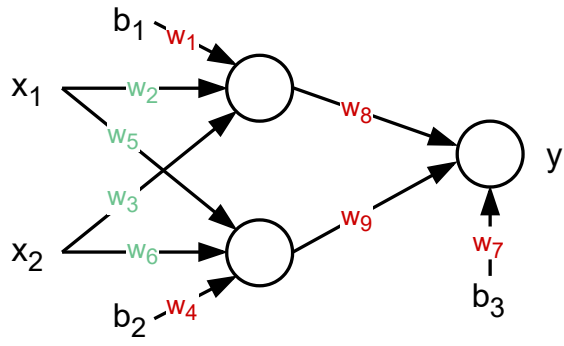


Figure 3.5: Structure of the AND network: the green weights are recoverable via our attack, because they are connected to the inputs and there are only two weights per input. Red weights cannot be recovered.

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 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 3.5. 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 [55], 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, Zhang, Juels, Reiter, and Ristenpart [146], 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, Zhang, Juels, Reiter, and Ristenpart 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 [163, 174]), 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 cannot 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.

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 [159] 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.

3.5 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 3.6 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.

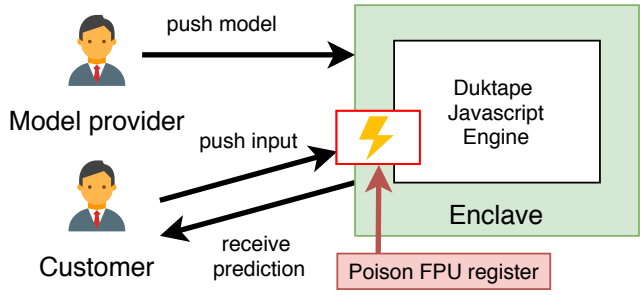


Figure 3.6: MLaaS system model with enclaves.

Experimental evaluation We base our case study on earlier work from Alder et al. [2] who placed the Duktape Javascript engine [150] in an Intel SGX enclave and utilized it to provide Machine Learning with the ConvNetJS Javascript library [79]. 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 images of handwritten digits from the MNIST dataset into their machine counterpart of 0 to 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 to evaluate 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 3.3 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 towards 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 [68] 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 towards 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.

Table 3.3: MNIST data set predictions with the x87 FPU and with SSE for different rounding modes and precisions.

	Rounding mode	Accuracy	Prediction class count (predicted digit) 0	1	2	3	4	5	6	7	8	9	Average error compared to baseline (SSE, rounding to nearest)
x87 Single precision	Round to nearest	4%	0	12	14	2	10	32	0	30	0	0	0.17604646652708841325 ↪ 6407761764
	Rounding down	8%	0	0	100	0	0	0	0	0	0	0	0.16796397173637958588 ↪ 6211884826
	Rounding up	4%	0	12	14	2	10	32	0	30	0	0	0.17604643409291073630 ↪ 2073360093
	Round to zero	8%	0	0	100	0	0	0	0	0	0	0	0.16796387552144440014 ↪ 0680386357
x87 Extended precision	Round to nearest	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000000055 ↪ 4406357383
	Rounding down	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000033073 ↪ 3402271493
	Rounding up	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000031452 ↪ 2247559579
	Round to zero	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000052415 ↪ 7807065445
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	0.00000000000000033073 ↪ 3402271493
	Rounding up	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000031452 ↪ 2247559579
	Round to zero	100%	9	14	8	10	14	8	9	14	3	11	0.00000000000000052415 ↪ 7807065445

3.6 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 3.4 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 4 benchmarks in to-nearest rounding mode or one benchmark in to-zero rounding mode still succeed.

Table 3.4: Benchmarks with SPEC CPU 2017 under compilation with the x87 FPU and with the SSE, both shown for different rounding modes. Listed are all workloads in the `fprate` test class and their result in the given configuration.

	Rounding mode	**bwaves**	**cactu BSSN**	**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	✔	✖	✖	✔	✔	✔	✖	✖	✔	✔	✔	✖	✔	✖

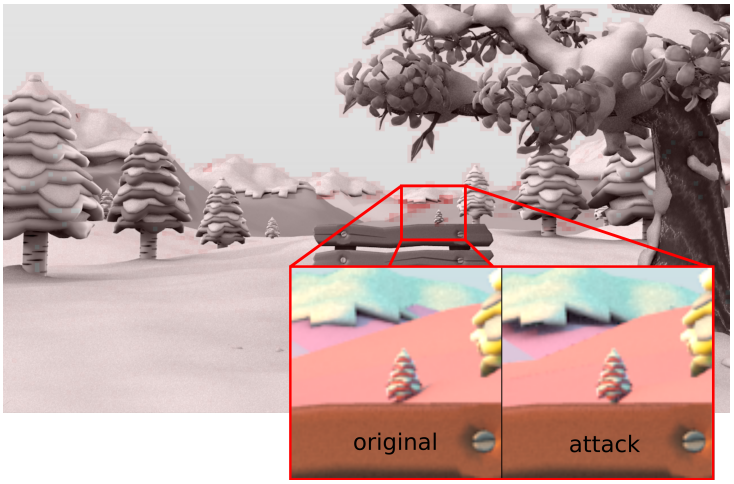


Figure 3.7: Composite image of the Blender benchmark in SPEC CPU 2017 under attack by our FPU attacker in x87 single precision mode when rounding towards zero. Areas in red differ from the expected render image with the zoomed-in area showing differences visible to the human eye.

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

Figure 3.7 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.

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

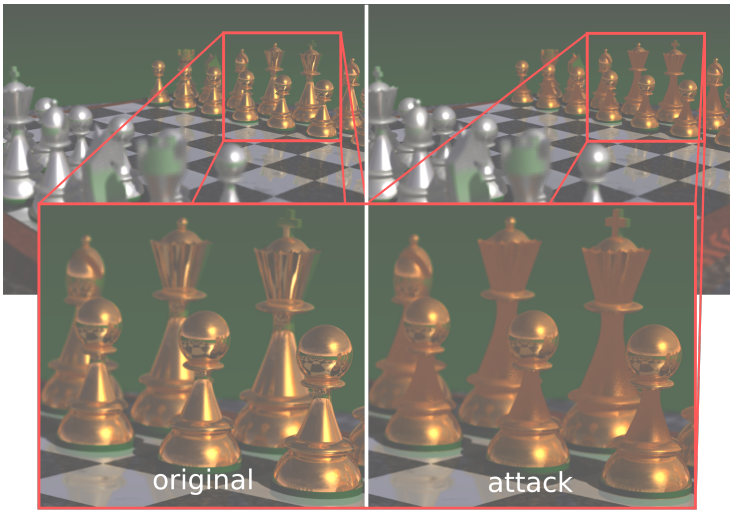


Figure 3.8: Composite image of the Povray benchmark in SPEC CPU 2017. The image shows a comparison of the baseline result to the result under attack by our FPU attacker in x87 single precision mode when rounding up. The zoomed-in areas highlight a region where the quality of calculated reflections and raytracing has been visibly degraded. Image brightened for print version.

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 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 to stealthily insert 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

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

chess board with realistic reflections of other pieces and of the surrounding environment, including lights. Figure 3.8 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 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.

3.7 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 [80, 157].

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 7 popular Intel SGX runtimes, developed by both academia and industry, we were able to provide a proof-of-concept attack for 5 of them. Moreover, our analysis revealed that 2 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 Open Enclave 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 [33, 89, 151, 154], 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 [35] or Guardian [12], to safeguard against ABI-level vulnerabilities in enclave runtimes.

Chapter 4

Pandora: Principled Symbolic Execution of Intel SGX Enclaves

This chapter is currently in submission as:

> **F. Alder**, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck. "Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes". In: *In submission.* 2023

Preamble

Earlier work by Van Bulck et al. and our contribution presented in Chapter 3 are two examples that illustrate the complexity involved when protecting the enclave interface with the untrusted world. Both works investigated major runtimes that can benefit from large development teams and are often developed or maintained by international companies. Still, as our earlier work on application binary interface (ABI) shielding runtimes showed, the effort to keep only the small assembly entry stub secure from all known vulnerabilities is immense. For the investigated production runtimes in that work, this means that relative to the size of the assembly stub, between over half (for the Rust enclave development platform (EDP)) to over quadruple (for Gramine) lines of code of the assembly

stub have been changed since the project started [153]. While unit tests can be written when a vulnerability is mitigated to ensure that it is not reintroduced, not all runtimes do this. Additionally, as the case of Open Enclave shows that is discussed in this contribution, even if unit tests are used, there is no guarantee that the tests cover all parts of the mitigated issues to ensure that a vulnerability cannot be reintroduced.

Multiple approaches exist to validate that a piece of code contains no vulnerabilities. Fuzzing is a popular dynamic analysis method that multiple related works apply, also in the realm of Intel Software Guard Extensions (SGX) [36, 45, 47, 62]. While full formal verification of a piece of code is another area of research, in this contribution, we investigate the static analysis approach of using symbolic execution.

Two prior works already worked on symbolic execution for Intel SGX: TEErex [35] and Guardian [12]. While both approaches were important in their contribution, neither is designed to find vulnerabilities within the enclave shielding runtimes. Instead, both prior works specifically targeted applications built on top of the Intel SGX software development kit (SDK) and abstracted away many functions provided by this SDK.

In this contribution, we aim to fill this previously overlooked gap and seek to automatically find vulnerabilities in the shielding runtimes themselves. Our tool Pandora is built on the same basis that the earlier two works used, which is the popular symbolic execution tool angr.[1] However, in contrast to the earlier tools, Pandora can analyze binaries compiled with any enclave shielding runtime and is fully SDK-agnostic. Our goal with Pandora is to provide a research platform for future research that can build on and extend Pandora with more vulnerability detection methods and expand its execution model. To aid and ease this future development, we put considerable effort into the usability and design of Pandora by not only providing a customizable and user-friendly command-line user interface but also by providing clear user reports generated in HTML and by following best practices in software development.

During working on this contribution, we have used Pandora to find 174 vulnerability instances across 10 runtimes, most of which are production runtimes. At this time, several of the vulnerabilities reported in this dissertation are still being mitigated and we are actively working with the respective vendors to help validate ongoing patch efforts. As a result, some industry vendors have already explicitly found the reports generated by Pandora helpful in pinpointing and mitigating the found issues. Some vendors also indicated an interest in integrating Pandora into their continuous integration and testing pipelines and we plan on working with the vendors to realize this upon publication of Pandora.

[1] `https://angr.io/`

4.1 Introduction

Recent years have seen the rise of trusted execution environments (TEEs) that provide strong, hardware-rooted protection of small software components, called *enclaves*, against hostile, possibly attacker-controlled system software. With the release of the SGX [40, 98], included in selected Intel processors from 2015 onwards, TEE protection is readily available in today's mainstream computing platforms, and even more recent technology, like the Trust Domain Extensions (TDX) [71] for upcoming Intel server processors, continues to rely critically on SGX enclaves. Thus, the widespread availability of SGX has boosted ongoing interest in enclave applications and limitations from both industry and academia.

While SGX hardware enforces that enclave memory cannot be accessed from the outside, enclave software remains ultimately responsible to be bug-free and should properly sanitize registers and pointer arguments in the shared address space. This non-trivial requirement has given rise to a sizable ecosystem of SGX *shielding runtimes* that support diverse enclave applications. Modern SGX development paradigms nowadays include *(i)* custom `C/C++` SDKs [78, 101] that directly expose a secure function call abstraction; *(ii)* numerous SGX-tailored library operating systems (libOSs) [16, 21, 118, 130, 149] to support lift-and-shift protection of existing legacy applications; and *(iii)* enclaved memory-safe language runtimes [49, 51, 53, 58].

The popularity of Intel SGX has, furthermore, triggered a long and ongoing line of attacks exploring limitations of this technology [105]. In this respect, a clear trend has been that, while some of the earlier SGX attacks [33, 104, 121, 123, 151] could still be mitigated fully transparently at the hardware level by means of CPU microcode patches, progressively more stringent demands have been placed on enclave software behavior to mitigate evermore specific vulnerabilities [5, 26, 33, 41, 73, 74, 76, 157] when interacting with the untrusted environment. This has increasingly made secure enclave software development, and especially the sanitization responsibilities for the numerous SGX shielding runtimes, a moving target (cf. Section 4.2).

While software mitigations for transient-execution and side-channel attacks have been widely studied for Intel SGX, and presently various compiler-based solutions [27, 59, 69, 76, 84, 131, 154] exist, the crucial aspect of validating the security of the enclave interface has received much less attention. Researchers have only recently started to explore more systematic analyses through fuzzing [36, 42, 112] or symbolic execution [12, 35, 80]. However, existing approaches fall short in that they focus on validating enclave application logic only, without considering vulnerabilities in the crucial shielding runtime, or even being compatible with diverse runtimes beyond Intel's SGX SDK. Furthermore,

existing approaches focus mainly on detecting memory-safety issues, without considering more subtle types of shielding responsibilities, such as untrusted pointer alignments [26, 74] and CPU register sanitizations [5, 73, 157]. These approaches, hence, are not fitted for the diverse and fast-changing SGX software ecosystem, where a subtle sanitization oversight in a shielding runtime may be the equivalent of a zero-day rootkit vulnerability in a commodity OS kernel.

To address these challenges, the main objective of our work is the development of a principled, tool-supported approach to validate the security of enclave software binaries using symbolic execution. We propose Pandora, an extensible, enclave-aware symbolic execution tool that is built upon the popular angr framework and extends it with several novel technical contributions. Particularly, we accurately implement missing, SGX-specific x86 semantics, conceive a proficient, enclave-aware symbolic memory model, and develop a generic enclave memory extractor. Thus, Pandora for the first time enables *truthful* and runtime-agnostic symbolic exploration of full enclave binaries, identical to the attested initial memory layout and including the crucial shielding runtime itself. Furthermore, to deal with the moving-target nature of secure enclave software development, we propose *pluggable* vulnerability detectors, extending the notion of angr breakpoints with SGX-specific memory-access and control-flow events that allow rapid scripting of powerful Pandora plugins.

Our extensive experimental evaluation on 10 different shielding runtimes from research and industry, with 4 plugins validating diverse sanitizations, highlights the delicacy and complexity of present SGX software responsibilities. We demonstrate the power of Pandora's truthful symbolic execution semantics by identifying several subtle vulnerabilities in commonly overlooked low-level enclave initialization and relocation code that cannot be analyzed with state-of-the-art enclave symbolic-execution tools. We, furthermore, are the first to construct an automated tool for wide-scale validation of intricate untrusted pointer-alignment software mitigations [74, 75] recently deployed throughout the SGX ecosystem in response to ÆPIC [26] attacks.

In the wider research landscape, we envision our open-source tool as a solid foundation to enable future science on validating the security of enclaved software, including low-level and fast-changing SGX software shielding runtimes.

Contributions In summary, our contributions are:

- We propose Pandora, an extensible, enclave-aware symbolic execution framework for truthful and principled validation of SGX binaries.

- Responding to the heterogeneity of the emerging SGX software landscape, we propose a universal enclave memory extractor and corresponding angr loader.
- Responding to the volatile and elusive SGX software responsibilities, we propose pluggable detectors for diverse vulnerabilities, from validating CPU register cleansing over untrusted pointer sanitization and alignment constraints to control-flow transitions.
- In an extensive experimental evaluation on 10 different SGX runtimes, Pandora autonomously confirmed 69 known and 174 new vulnerable code locations.

Disclosure and Artifacts We responsibly disclosed all findings to the respective vendors (tracked via 7 CVEs), providing them with comprehensive reports from our tool. We, furthermore, included recommendations for software mitigations and assisted in validating the applied fixes, which has uncovered remaining issues in at least one runtime.

In the spirit of open science, we provide a comprehensive open-source artifact[2] (to be released upon conclusion of submission) with self-contained HTML reports of all vulnerabilities from Table 4.2, multiple runtimes to test out Pandora, and documentation of how to reproduce our results. We will also include the binaries of all analyzed shielding runtime versions (where allowed by licensing) to provide a representative public data set of vulnerable enclaves that can serve as a baseline for future research.

4.2 Background and Related Work

Enclave Shielding Due to its strong attacker model, enclave software faces several additional security challenges compared to traditional user-space software. In current practice, these additional challenges are primarily handled by a shielding runtime that transparently intervenes on interactions with the untrusted environment, as shown in Fig. 4.1.

Intel SGX enclaves are embedded as a contiguous virtual address region within an untrusted, surrounding host application. As in-enclave software is allowed to freely dereference outside memory locations, the host application can efficiently communicate through the enclave's application programming interface (API) by passing pointers to arguments and return values in the shared virtual

[2] https://github.com/pandora-tee

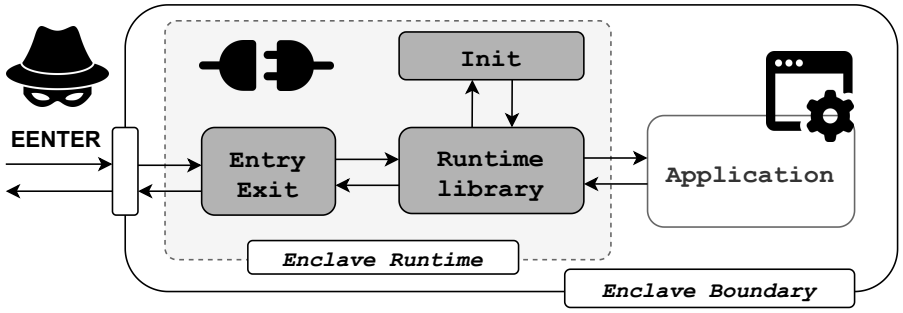


Figure 4.1: A shielding runtime transparently protects enclave applications by (1) cleansing CPU registers upon entry or exit events; (2) finalizing the initial memory layout, including any in-enclave relocations, upon first entry; and (3) sanitizing pointer arguments before handing control over to the application, which can call back via trusted standard library functions.

address space. However, this also opens the door to an especially powerful class of confused-deputy attacks, necessitating that the enclave shielding runtime adequately sanitizes any attacker-provided API pointers prior to dereference. Despite this requirement being well-known and the availability of automated methods—such as the `edger8r` tool to automatically generate interface sanitization code from developer annotations in the Intel SGX-SDK [78] and Open Enclave [101], or the Rust type system leveraged in EDP [53]—a continuous stream of vulnerabilities [12, 35, 36, 80, 157] has proven SGX pointer sanitization vulnerabilities to be particularly elusive and widespread in practice. As an example, Listing 4.1 illustrates how adequately sanitizing an elementary pointer-to-pointer argument can be non-trivial in practice.

Moreover, in response to the recently disclosed ÆPIC [26] and related memory-mapped I/O (MMIO) [74] stale data vulnerabilities in Intel processors, enclave software requirements for sanitizing untrusted pointer arguments have been considerably complicated. That is, not only does enclave software nowadays need to ensure that attacker-provided pointers properly fall entirely outside the protected enclave range, but any subsequent pointer dereferences also need to proceed at a certain alignment and size or need to be preceded and followed by fragile x86 instruction sequences to cleanse microarchitectural buffers and stall the CPU pipeline. These successive refinements of software responsibilities hence necessitated extensive and ongoing changes throughout the heterogeneous SGX software ecosystem.

A parallel moving-target evolution can be observed at the level of the ABI. An initial comprehensive study [157] has shown that secure initialization was

```
1 void encl_get_from_addr(struct user_arg *op) {
2     assert(is_outside_enclave(op, sizeof(*op)));
3     // Copy op->addr to avoid TOCTOU attacks
4     volatile void* ptr = (void*) op->addr;
5     assert(is_outside_enclave((void*) ptr, 8));
6     use(ptr); // Dereference op->addr
7 }
```

Listing 4.1: Example of API sanitization: lines 2 to 5 enforce that all attacker-controlled pointers lie outside the enclave prior to dereference.

widely overlooked for certain crucial CPU configuration flags, such as the x86 direction flag that may introduce memory-safety violations in otherwise secure code. Similar issues have since been shown for stack-pointer initialization in SGX enclave exception handlers [41] and for x87 and SSE floating-point configuration registers [5]. The latter was most recently refined once again in an Intel advisory [73] with additional SSE sanitizations to protect against certain operand-dependent floating-point instruction timing channels in otherwise constant-time code. A recent overview study [153] has documented how these ABI vulnerability disclosures necessitated several rounds of widespread patches throughout popular SGX shielding runtimes.

Symbolic Execution Symbolic execution [81] statically interprets a program using *symbolic* inputs (*i.e.*, mathematical terms) and collects *constraints* (*i.e.*, mathematical formulas over these terms) encoding programs paths. These constraints can be solved with an SMT solver to generate concrete inputs exercising the path or check security assertions. Its ability to systematically explore program paths and generate concrete inputs has made symbolic execution a tool of choice for intensive testing [62] and vulnerability analysis [31]. More recently, researchers have also started to apply symbolic execution to the specific context of Intel SGX enclaves [12, 35, 80]. We provide an extensive comparison of Pandora to these existing tools in Section 4.3.1. Some works [33, 175] have, furthermore, focused on detecting microarchitectural side-channel vulnerabilities in enclave applications using symbolic execution, but their goal is orthogonal to our scope of validating shielding responsibilities.

Fuzzing A well-known, complementary approach to static analysis via symbolic execution is dynamic concrete execution via fuzz testing. An orthogonal and concurrent line of work [36, 42, 47] has started to explore such fuzzing for Intel SGX enclave applications. Compared to symbolic execution, fuzzing can more easily scale to complex code bases by quickly generating test cases and may find bugs with fewer false positives. However, unlike symbolic execution,

fuzzing cannot provide any guarantees about the absence of bugs and requires carefully crafted test cases to trigger non-superficial application logic. Hence, in line with existing surveys [20, 132], we regard fuzzing-based approaches as complementary to symbolic validation.

4.3 Problem Statement and Overview

The combination of a varied and evolving Intel SGX runtime ecosystem with the frequent discovery of new attack techniques that necessitate additional software sanitizations makes the problem of principled enclave software validation particularly challenging and, indeed, largely unexplored for the fundamental shielding runtimes themselves. Therefore, we set the following goals:

- *G1 Truthful symbolic exploration.* Enclave-aware symbolic execution should closely mimic the real SGX hardware. Particularly, to not miss vulnerabilities in the runtime itself, the symbolic exploration should *(a)* start from the very first entry instruction without skipping initialization procedures or stubbing runtime library functions; and *(b)* operate on the *exact* initial memory contents, as remotely attested via MRENCLAVE [11], while accurately detecting and symbolizing any subsequent accesses to untrusted or unmeasured memory.
- *G2 Runtime-agnostic.* Validation should not be limited to enclaves developed with any specific single shielding runtime. The heterogeneous SGX ecosystem with ill-documented and varying enclave binary formats calls for a lightweight conversion approach to a unified format capturing the exact enclave memory layout.
- *G3 Extensible validation policies.* The system should support prompt reactions to evolving sanitization responsibilities by adding new or modified vulnerability detection plugins. This calls for an approach that decouples validation *policies* from enclave-aware symbolic execution *mechanisms*, such that plugins can solely focus on elegantly expressing the required software security invariants to be validated for explored paths.
- *G4 Accessibility.* The tool should be open-source and easy to use, including on closed-source binary targets. Reports should be easily interpretable by human analysts.

4.3.1 Research Gap

Initially, SGX software vulnerability research was mainly guided through manual code review [5, 41, 153, 157], whereas automated enclave analysis through

Table 4.1: Comparison of symbolic-execution tools for SGX.

	Runtime			Binary	Dump	Reentry	Plugins				
Tool	App	SDK	Entry	Init				Ptr	ABI	ÆPIC	Jmp	Open
TeeRex [35]	●	Intel	○	○	●	○	◐	◐	○	○	◐	○
Guardian [12]	●	Intel	●	○	◐	○	○	◐	◐	○	◐	●
COIN [80]	●	Intel	○	○	○	○	○	◐	○	○	○	●
Pandora	●	*any*	●	●	●	●	●	●	●	●	●	●

Features can be fully (●), partially (◐), or not (○) supported. Columns 4–7 denote whether the tool executes the runtime *entry* and *init*ialization phases; can handle *binaries* without additional specification; and uses the exact memory layout (*dump*).

symbolic execution has only more recently started to be explored [12, 35, 80]. Table 4.1 compares Pandora to these existing tools. In summary, existing approaches are mostly focused on application bug detection instead of principled validation of the absence of shielding runtime vulnerabilities. This means that they are inherently insufficient for truthful symbolic exploration (*G1*), as the focus is on analyzing enclave application logic only, while (largely) skipping the underlying shielding runtime and operating on inaccurate initial memory contents. Moreover, existing tools are ill-fitted for the diverse SGX ecosystem (*G2*), as they all make runtime-specific assumptions that strictly limit them to enclaves developed with Intel's SGX SDK only. Finally, existing tools focus mainly on a narrow set of classical memory-safety issues for pointers without principally supporting more intricate shielding responsibilities (*G3*), such as recently rolled out pointer-alignment ÆPIC mitigations [26, 74].

TEERex TeeRex [35] is a closed-source prototype to detect memory corruption vulnerabilities in enclave applications developed with the Intel SGX SDK. Similarly to our work, TeeRex is based on `angr` [132], a popular symbolic execution tool for binary code, and performs taint tracking of untrusted attacker arguments and memory accesses outside the enclave using unconstrained symbolic values.

In contrast to Pandora, however, TeeRex does not support truthful symbolic exploration (*G1a*), as it entirely skips analysis of the whole trusted runtime and directly performs symbolic execution of enclave application entry points, called `ecalls`. Moreover, TeeRex is inherently runtime-specific (vs. *G2*), as it relies on Intel SGX SDK-specifics to identify addresses of `ecall` functions, to hook specific pointer validation functions, and to set up an approximate, non-truthful initial memory layout (vs. *G1b*). With regard to vulnerability detection (*G3*), TeeRex only reports unconstrained and NULL-pointer dereferences and cannot detect more subtle pointer issues, or ABI and ÆPIC issues. Particularly,

by hooking the crucial validation functions (e.g., `is_outside_enclave` in Listing 4.1), TeeRex may miss logical partial validation errors [157] that will be caught by Pandora's precise enclave-aware memory model (cf. Section 4.7). Lastly, TeeRex is not openly available for independent usage, study, and reproduction (vs. *G4*).

Guardian Guardian [12] is similarly based on angr and can partially check API and ABI shielding policies. Regarding truthful exploration (*G1a*), Guardian is the only prior work that starts at the enclave entry point within the trusted runtime, but it nevertheless skips the complex enclave initialization phase, which may still contain critical vulnerabilities (cf. Section 4.7). Furthermore, similar to TeeRex, Guardian is constrained to binaries developed with specific versions of the Intel SGX SDK (vs. *G2*) and only constructs an approximate, non-truthful initial memory layout (vs. *G1b*). As to vulnerability detection (*G3*), Guardian validates a principled, yet fundamentally incomplete *orderliness* policy, where the developer is required to manually annotate execution phases (vs. *G4*). Guardian validates that, after the entry phase, an (incomplete) blocklist of ABI configuration registers has been cleared, and that untrusted memory outside the enclave is only accessible during execution of the shielding runtime, but not during the application phase. This simplified permission state-machine model may be overly conservative for applications and, more problematically, remains inherently insufficient to detect critical vulnerabilities (e.g., CVE-2018-3626 [157]) in the shielding runtime itself, as the latter is allowed unrestricted access to the full address space.

COIN COIN [80] uses concolic execution to find memory-safety vulnerabilities in enclave applications. COIN specifically targets applications developed on top of the Intel SGX SDK only (vs. *G2*) and requires the enclave source code (vs. *G4*) for extracting the parameters of `ecalls` in order to set up an approximate, non-truthful initial state (vs. *G1b*). Regarding vulnerability detection (*G3*), COIN is largely orthogonal to our work by focusing on traditional memory-safety application vulnerabilities instead of nuanced, enclave-specific shielding issues and skipping analysis of the runtime itself (vs. *G1a*).

4.3.2 Solution Overview

Figure 4.2 depicts a high-level overview of the Pandora software architecture, which we implemented in 5,934 lines of extensible Python code (as measured by `sloccount`). At Pandora's core, the engine component in the middle-right interacts with the underlying symbolic execution library angr [164]. This engine

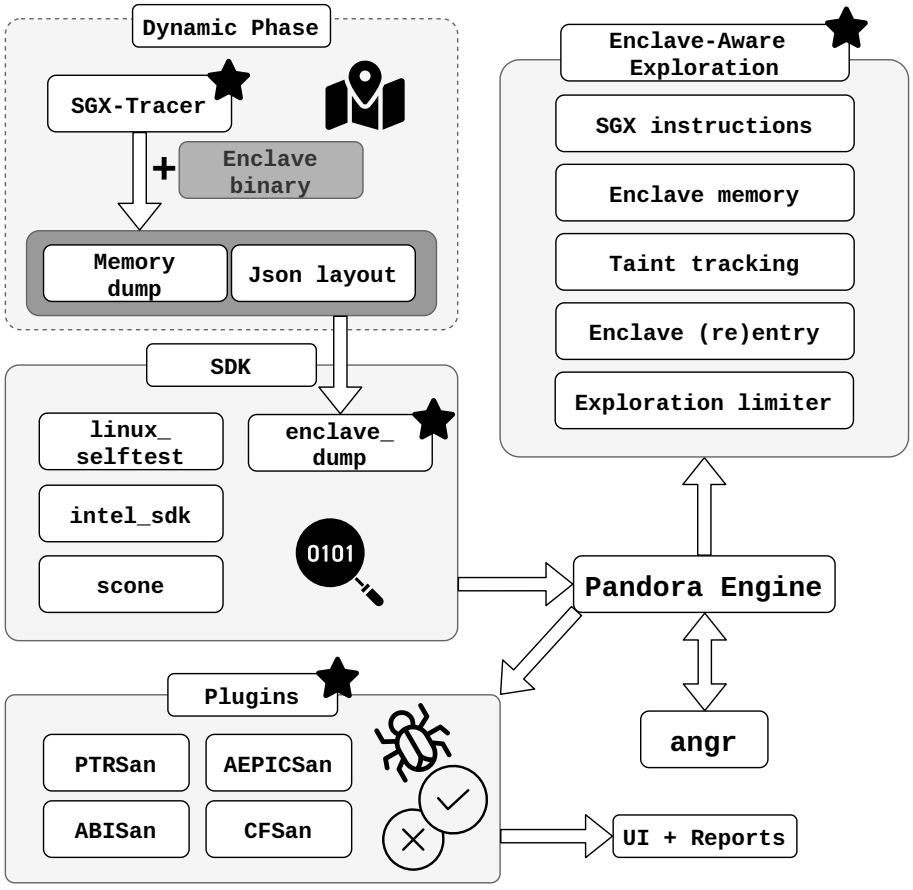


Figure 4.2: Overview of the Pandora architecture.

augments `angr` with accurate SGX semantics and drives the enclave-aware truthful symbolic exploration (*G1*), shown on the top right and described in Section 4.4. The engine is primed with the exact initial enclave image via a novel, runtime-agnostic dynamic memory extraction phase (*G2*) depicted on the left of the figure and detailed in Section 4.5. As such, Pandora is the first symbolic-execution tool that can find vulnerabilities before the application is even executed, *i.e.*, by not only executing the runtime entry procedures, but also the complete low-level enclave initialization phase.

While symbolically executing a binary, the Pandora engine may trigger vulnerability-detection plugins (*G3*), shown on the bottom left and described in

Section 4.6, based on subscribable events exposed by the SGX-aware exploration. After a completed run, Pandora bundles the findings of each plugin and formats them into convenient and interactive HTML reports (*G4*), shown in Appendix C.1, including severity levels, descriptions, disassembly, register dumps, and full basic-block backtraces to enable human analysts to readily investigate the reported issues.

4.4 Enclave-Aware Symbolic Execution (*G1*)

4.4.1 Modeling x86 Instruction Semantics

The underlying VEX representation used by angr does not have a symbolic model for many x86 instructions that commonly occur in enclave binaries. Most prominently, the `ENCLU` user leaf instructions [77] are used inside the enclave to perform architectural tasks, such as creating a local attestation report (`EREPORT`), generating cryptographic keys (`EGETKEY`), or exiting the enclave (`EEXIT`). While prior work faced similar angr limitations and either did not execute [35] or merely hooked and skipped [12] over these instructions, Pandora truthfully emulates used enclave instructions as closely as possible. For example, in `EREPORT`, we copy the relevant SGX enclave control structure (SECS) fields provided by the enclave loader, including the processor extended features request mask, into the generated report structure. When specific fields are not available and no sane defaults can be provided, values are symbolized to ensure that all possible paths are explored.

Furthermore, in response to advanced ABI attacks [5, 73], instructions like `XSAVE` and `XRSTOR` or their variants are commonly used to save and restore extended x86 register on enclave context switches. In contrast to prior work [12, 35], Pandora does not skip over these instructions, but carefully emulates their behavior as closely as possible. Where necessary, we add dedicated shadow registers to keep track of special x86 registers, such as MXCSR, which are not normally part of angr's execution model. As shown in Section 4.7, this precise register view enables Pandora plugins to accurately uncover subtle oversights, e.g., attacker-controlled registers when switching to enclave functions or insecure MXCSR configuration values.

4.4.2 Taint Tracking of Attacker Inputs

In order to accurately deal with attacker-controlled inputs, Pandora comes with a capable symbolic taint-tracking mechanism. Specifically, initial register

contents on enclave entry, as well as memory reads from outside the enclave or from uninitialized unmeasured pages inside the enclave (cf. Section 4.4.3), are transparently replaced with unconstrained symbolic values. Thus, the symbolic execution initially makes no assumptions about attacker-provided inputs, until specific constraints are added by any subsequent sanitizations performed by the enclave code. Pandora, furthermore, uses angr's annotation system to mark attacker-controlled symbolic values with an *attacker-taint*, which is conservatively propagated during symbolic execution and can be conveniently queried by plugins. For instance, plugins can check that values are properly sanitized (e.g., Section 4.6.1) or react differently based on whether a value is attacker-tainted or not (e.g., Section 4.6.2).

Note that Pandora's taint tracking mechanism only tracks direct data flows. That is, any indirect flows resulting from attacker-controlled control flow are ignored, as they are arguably less critical and may severely over-approximate taints, which would lead to abundant false positives.

4.4.3 Enclave-Aware Memory Model

Pandora features a fully enclave-aware memory model that truthfully simulates the enclave address space in a more accurate and expressive way than prior work, while also including reasonable performance optimizations. Particularly, we are the first to realize a precise, runtime-agnostic enclave memory model that properly recognizes attacker-controlled symbolic addresses and sizes and that takes into account novel attack surface from unmeasured SGX enclave pages.

Address-Space Partitioning

At its core, we implemented our enclave-aware memory model as an angr `MemoryMixin` extension that performs rigorous checks on every memory access. Particularly, we use angr's constraint solver to unambiguously decide for every accessed buffer with a possibly symbolic address and size whether it is restricted to *(i)* lie fully inside the enclave; *(ii)* lie fully outside the enclave; or *(iii)* partially touch the protected enclave range. Accesses to memory inside or outside the enclave will be handled differently, as outline below. Pandora plugins can, furthermore, subscribe to these respective events to check and report specific vulnerabilities (cf. Section 4.6).

Note that the above accurate classification is non-trivial to implement, and prior work side-stepped these intricacies by either hooking runtime-specific

pointer-validation functions [35] or ignoring the (possibly symbolic and attacker-controlled) size of memory reads [12]. Our fully symbolic memory model, on the other hand, allows to meticulously detect subtle oversights or logical errors in the crucial validation functions themselves. For instance, Section 4.7 discusses a particularly intricate finding where overflow protection logic was silently optimized away by the compiler.

Untrusted Memory Accesses

For accesses falling outside the protected enclave range, we model the strongest type of adversary that utilizes tools such as SGX-Step [159] to perform instruction-granular time-of-check time-of-use (TOCTOU) attacks. For example, an enclave checking an external pointer that resides in untrusted memory, before accessing this pointer again at a later time (as in Listing 4.1) may realistically receive two different values. Pandora truthfully simulates this by ignoring untrusted memory writes and fully symbolizing all untrusted memory reads with a fresh attacker-tainted symbolic value on *every access*.

Enclave Memory Accesses

In close accordance with the SGX specification [77], we distinguish two types of memory inside the enclave: measured and unmeasured pages. Measured enclave pages are attested as part of the MRENCLAVE enclave identity and are, hence, always demonstrably initialized to the exact value provided by the enclave loader. Unmeasured enclave pages, on the other hand, are protected from enclave creation time onwards, but their initial content is *not* attested as part of the MRENCLAVE enclave identity. These unmeasured enclave pages have many uses in enclaves, for example to reserve heap memory or to load additional code or data during execution that did not exist at enclave creation time yet. As the initial value of these pages is not part of the enclave identity, and thus under attacker control, enclave software must always securely overwrite these pages before first use. However, to the best of our knowledge, to date no sanitizer exists to validate this critical security property. To enable this with Pandora, we ensure that any read from unmeasured enclave memory initially returns an attacker-tainted symbolic value. Only when unmeasured bytes are securely initialized, we create an angr memory backing and the newly written secure values will be taken into account for future reads.

Pandora, furthermore, implements two types of safe performance optimizations. First, we remove measured and initialized unmeasured enclave memory that consists of all-zero bytes from the angr backend. Any reads from such regions

will statically return zero bytes until they are overwritten with non-zero data. Second, only for source and destination buffers that are constrained to fall entirely inside the enclave, we optionally hook common memory-management functions (`memcpy` and `memset`) and x86 `rep` string operations with custom `SimProcedures` that eliminate loop overhead, while still taking care to trigger any relevant `angr` mixins and breakpoints.

4.4.4 Enclave Entry and Reentry

During enclave lifetime, `EEXIT` and `EENTER` instructions can switch execution to and from the untrusted environment. Prior work [12, 35, 80] relied on parsing runtime-specific and fragile data structures to find out the supported `ecalls` in order to skip the crucial runtime entry and/or initialization phases entirely and immediately start executing at the respective application `ecall` function.

Enclave Entry

To truthfully execute entry into the enclave, we parse the actual thread control structure (TCS) from enclave memory to retrieve the entry point location and fill registers with the exact same values that they would receive from the architecture, such as the TCS address and `FS` and `GS` base addresses. All other registers are filled with unconstrained, attacker-tainted symbolic values to initiate Pandora's taint-tracking mechanism (cf. Section 4.4.2).

Enclave Exit

Pandora allows to truthfully build up enclave state by emulating a new `EENTER` with the same accumulated memory view after a symbolic path reached the `EEXIT` instruction. Hence, the enclave entry code in the runtime itself will perform any necessary checks and autonomously decide whether the entry request is an `ecall` or an `ocall` return and dispatch this request accordingly. The strength of this approach is that subtle attack vectors, like dereferencing a function pointer before in-enclave relocation (cf. Section 4.7) or returning from an `ocall` where no prior `ocall` was executed [157], can in principle be detected.

4.4.5 Path Exploration and State Reduction

Pandora's unique focus on truthful symbolic exploration of the *entire* enclave binary, including low-level shielding runtime code, comes with the potential cost

of state explosion. To reduce memory consumption for individual explorations, Pandora optionally supports depth-first exploration in addition to breadth-first exploration.

With regards to reentry, every path that reached `EEXIT` would have to be reentered in a naive approach, because the enclave may have accumulated relevant global state. However, we observed that many paths result in a clean failure that is reported to the untrusted world with the request to restart the enclave with correct parameters. To avoid exploring all of these semantically equivalent traces in parallel, we implement a novel *state uniqueness reduction* before reentering enclave exploration. That is, two symbolic `EEXIT` states are different from each other only if they have made different changes to the internal memory of the enclave. For example, two enclave traces that both result in no changes to the enclave except setting a specific bit indicating that the enclave failed, are equivalent and reentering both would be redundant. With this uniqueness criterion, we thus remove all non-unique enclave traces before preparing them for reentry. Note that this approach is a safe over-approximation, e.g., states may still be semantically equivalent even though they differ in some de-allocated stack variables. However, we found that our state uniqueness reduction is sufficient to greatly reduce the state space without risking that unique states may be lost.

4.5 Runtime-Agnostic Enclave Loading (*G2*)

Truthful symbolic execution naturally starts with an accurate representation of the initial enclave memory layout (*G1b*). Unfortunately, however, in contrast to well-established standards like the executable and linkable format (ELF) for Linux binaries, there exists no standardized format to distribute SGX binaries. Hence, over the last years, all SGX shielding runtimes have adopted their own custom formats to describe the additional information needed to correctly load the enclave, e.g., often by encoding opaque blobs into additional ELF metadata sections [78, 101]. This is especially problematic as Intel SGX requires a particularly involved, multi-stage loading process [40, 77].

First, the untrusted system software constructs the initial enclave memory layout, containing regions for code and data, and also including several unique enclave-specific data structures. The two most prominent data structures are the SECS structure describing, among others, the enclave load address and size, as well as the TCSs, describing the enclave entry point and thread-local data storage. Furthermore, as SGX enclaves are commonly compiled as position-independent code and loaded as dynamic libraries, the MRENCLAVE identity

must be independent of the load address. Hence, the enclave cannot rely on the untrusted loader to perform any remaining ELF relocations (e.g., for dynamic function-pointer tables). Thus, as a second loading step, enclave shielding runtimes generally include in-enclave code to perform any necessary ELF relocations upon the first enclave entry, *i.e.*, *after* the enclave has already been created and loaded into memory.

Static Analysis Notably, all prior works [12, 35, 80] on SGX-aware symbolic execution entirely side-step the aforementioned intricacies by restricting themselves to one particular runtime, specifically the Intel SGX SDK, and by loading the enclave largely as a normal ELF file. Particularly, existing approaches only take care to create approximate space for stack and heap and either skip to the application directly [35], or they manually patch fragile and version-specific global data structures to falsely mark the symbolic enclave as initialized and skip over the costly, low-level runtime initialization and relocation phases [12]. Thus, prior works simulate an inaccurate enclave memory layout (vs. *G1b*) and are, moreover, only compatible with one specific version of one specific runtime (vs. *G2*).

We argue that, with ample code review or reverse-engineering efforts, it is in principle possible to devise an approach that accurately mimics the runtime-specific loading process to construct a truthful initial memory layout, satisfying *G1b*. Indeed, Appendix C.2 describes such optional support we added to Pandora to load enclave binaries from selected runtimes based on static analysis of a given enclave binary. We found, however, that such a purely static-analysis approach is highly labor-intensive and inherently fragile, requiring to implement a custom loader for every studied enclave runtime, possibly even with changes across runtime versions. This would evidently limit the scope and not satisfy our vision of runtime-agnostic analysis for the sprawling SGX ecosystem that has become heterogeneous both in runtime capabilities as well as in programming languages available to the enclave developer.

Dynamic Enclave Memory Extraction To overcome the labor-intensity and inherent fragility of the above pure static analysis approach with runtime-specific loaders, Pandora supports a more powerful approach that requires a short-lived dynamic execution phase to load the binary-under-test once. Specifically, we developed a minimal standalone program, called SGX-TRACER, to passively observe the loading process of an enclave binary on actual Intel SGX hardware.[3] SGX-TRACER consists of about 400 lines of C code and uses the `ptrace` Linux

[3]Real SGX hardware may not even be a strict requirement, as SGX-TRACER could, in principle, also spoof the existence of the SGX driver.

system call to attach to the untrusted enclave host process and intercept all calls to the (in-kernel or out-of-tree) Intel SGX driver. SGX-TRACER can thus fully transparently *(i)* detect enclave creation via `ECREATE` and record crucial enclave SECS metadata, including load address and size; *(ii)* record the exact memory contents of all pages that are subsequently added via `EADD`; and *(iii)* track additional metadata and permissions for these pages, as well as locate special pages like TCSs, before the enclave identity is finalized via `EINIT`. This allows SGX-TRACER to accurately extract the *exact* initial enclave memory (*G1b*), as attested by MRENCLAVE, for *any* SGX process (*G2*).

The output by SGX-TRACER is stored as a binary dump and accompanying JSON file and can subsequently be used on non-SGX hardware by Pandora. Particularly, we developed a minimal angr loader to reconstruct a truthful symbolic memory view, including permissions of each page and whether the page is measured or unmeasured (cf. Section 4.4.3). This inherently runtime-agnostic loader makes Pandora compatible with *any* enclave dump extracted via SGX-TRACER, regardless of runtime-specific loading details.

One downside of utilizing an enclave memory dump for symbolic execution is that this process loses all debug symbols, including function names. Pandora can run without any of these symbols, but upon finding a potential vulnerability, the generated reports may be less understandable for human analysts (vs. *G4*). Hence, we implemented a custom symbol handler that can augment a plain memory dump extracted by SGX-TRACER with symbol information from the original ELF file, if optionally provided via a Pandora command-line option (together with a static offset).

4.6 Pluggable Vulnerability Detection (*G3*)

During symbolic exploration, angr triggers a set of breakpoints that can be hooked to investigate the symbolic state. Exemplary angr breakpoints are memory or register accesses and function calls. Pandora extends the legacy angr events with a set of eight new *enclave-specific breakpoints* (cf. Appendix C.3). Specifically, Pandora exposes breakpoints before and after enclave entry and exit, as well as breakpoints before and after symbolic memory reads and writes that are restricted to resolve fully inside, fully outside, or partially overlapping with the enclave memory range.

Pandora's enclave-aware breakpoints form the basis for our notion of pluggable vulnerability detection (*G3*). Specifically, specialized plugins can subscribe to relevant enclave events, as well as legacy angr breakpoints, to accurately validate certain software invariants during symbolic exploration. We created 4

plugins for a diverse set of enclave shielding runtime responsibilities at the levels of ABI register cleansing, API-level pointer arguments, ÆPIC-style pointer alignment considerations, and attacker-controlled control flows. Plugins can, furthermore, make use of Pandora's built-in reporting interface (*G4*) to conveniently summarize any findings in human-readable HTML reports that are automatically annotated with all relevant information, e.g., a severity score and description of the issue and how to reach the vulnerable state (cf. Appendix C.1).

4.6.1 ABI-Level CPU Register Sanitization

Enclaves share the CPU register set with their untrusted surrounding host process. An important responsibility of the shielding runtime is, therefore, to securely initialize any low-level configurations registers on enclave entry. Due to the intricacies of these low-level register manipulations, those sanitizations have to be carefully implemented in a fragile, hand-written assembly stub before a jump into higher-level languages can be securely made, compliant with ABI expectations [52, 93] by the compiler.

While the general concept of ABI-level sanitization is relatively well-understood across SGX shielding runtimes, an ongoing line of manually discovered vulnerabilities [5, 41, 73, 153, 157] has underlined the intricacies and challenges for secure register initialization in the complex x86 instruction set. Prior work on automated enclave software vulnerability detection has either fully ignored CPU register sanitization by focusing on API validation only [35, 36, 80], or resorted to a simplistic and incomplete blocklist approach that merely checks whether selected CPU registers have certain concrete safe values [12]. On the other hand, Pandora's `ABISan` plugin proposes a more principled approach based on taint tracking, which can autonomously discover insufficient register initialization or cleansing.

Attacker-Tainted Configuration Registers

The `ABISan` plugin hooks all `angr` register read events and relies on Pandora's taint-tracking mechanism (cf. Section 4.4.2) to report critical violations when attacker-tainted, uninitialized CPU configuration registers are read. To avoid evident false positives, `ABISan` only requires a concise allowlist for the x86 data registers, *i.e.*, the 16 general-purpose registers, 16 vector registers, and floating-point unit (FPU) register stack, which do *not* contain control or status bits and, hence, are allowed to be tainted with attacker inputs. Any other

attacker-tainted register reads will be automatically reported as critical policy violations.

Our systematic taint-tracking approach has two main strengths compared to simply checking that an incomplete subset of registers has been initialized to certain values [12]. First, `ABISan` can *autonomously* track all relevant occurrences where the attacker has influence over the result of a computation through control registers.[4] This may, in principle, even include yet unknown ABI attack avenues. For instance, we experimentally validated that `ABISan` can fully autonomously discover attacker-tainted reads from individual bits in the RFLAGS [157] register, e.g., the crucial direction flag for x86 `REP` string instructions, as well as a particularly subtle oversight for floating-point operations that required several rounds of patches in Rust-EDP and OpenEnclave to make sure that not only the x87 FPU control word is initialized, but also the internal x87 tag word [5]. Second, `ABISan` also enables tracking advanced attack vectors where the enclave would inadvertently restore tainted control registers prior to using them in a computation.

Enclave Entry Sanitization

Our `ABISan` plugin inspects the complete register state when reaching the first `CALL` instruction inside the enclave. Indeed, the first function call inside the enclave revealed to be a surprisingly effective heuristic for the switch from assembly sanitization code to the higher-level, compiler-generated API entry point: across the 10 investigated runtimes, only a single runtime performed a `CALL` from inside assembly code before jumping to `C` code, which we accommodated in our heuristic. Upon reaching the API entry point, `ABISan` warns for every control and data register that has not been entirely cleared of attacker-tainted data.

Thanks to Pandora's powerful taint-tracking mechanism and enclave-aware execution model, we were able to express the entire `ABISan` policy in only 142 lines of Python code. It is important to note that the flexible nature of our plugins allows for quickly reacting to the ever-changing landscape of recommendations to ABI sanitization responsibilities for Intel SGX. For example, initial research [5] first investigated issues with incomplete sanitization of floating-point control registers and recommended setting the MXCSR register to the ABI-specified value of `0x1F80` on enclave entry. More recently, however, Intel [73] further nuanced secure MXCSR initialization by recommending the value `0x1FBF`,

[4]The only limitation here is that we are restricted to the subset of x86 behavior that is emulated by angr. For instance, angr does not consider the alignment-check flag in RFLAGS, and, while floating-point precision and rounding modes are kept track of, we found that they are largely ignored in the underlying VEX symbolic-execution engine.

which additionally sets all floating-point exception status flags, to protect against subtle, one-cycle timing differences dependent on (possibly secret) floating-point operand values. We were able to swiftly incorporate this latest recommendation into `ABISan`'s validation policy. This demonstrates that our plugin system can react flexibly and promptly to such updated recommendations, which, as we will show in Section 4.7, require changes that propagate slowly throughout the Intel SGX software ecosystem.

4.6.2 Untrusted Pointer Value Sanitization

We implemented a capable `PTRSan` plugin in 120 lines of Python code that proposes three expressive security invariants to catch the pervasive issues of confused-deputy attacks via untrusted pointer arguments in the shared address space. Note that, in contrast to prior work [12, 35, 80], `PTRSan` is entirely independent of the runtime-specific sanitization function, solely relying on Pandora's built-in taint tracking and enclave-aware memory model. Hence, as demonstrated in Section 4.7, `PTRSan` for the first time allows to find subtle logical errors in the sanitization logic itself.

Address Inside or Outside Enclave

Any symbolic memory access that may resolve both outside or inside the enclave memory range is a clear violation of the distinction between trusted and untrusted memory regions. These cases are, hence, always reported as a critical issue of a pointer that has not been sufficiently constrained by the enclave software. Note that this case also covers accesses that may (partially) cross the enclave boundary.

Tainted In-Enclave Address

Attacker-tainted accesses that are constrained to resolve entirely in untrusted memory are clearly benign behavior of the enclave. On the other hand, attacker-tainted accesses that are constrained to always lie *entirely* in trusted enclave memory may still be benign behavior, e.g., an attacker-controlled, yet constrained index into an in-enclave array data structure. Hence, we only report a warning in these cases and mark them as potential issues that may warrant manual and application-specific further inspection. To simplify such further analysis, `PTRSan` reports the size and maximum address range of the tainted memory access. This criterion to warn for tainted in-enclave memory accesses thus ensures that no clear violation of secure memory accesses can

occur, at the potential burden of occasional false-positive warnings. These false positive are non-straightforward to eliminate generically, but we discuss possible enhancements and heuristics in Section 4.8.

Untainted Outside-Enclave Address

Untainted accesses that are constrained to always resolve entirely in enclave memory are clearly benign behavior of the enclave. However, if untrusted memory is ever accessed with an address that is *not* tainted by the attacker, `PTRSan` sees this as a critical issue hinting at unexpected behavior, e.g., an uninitialized or NULL pointer dereference.

4.6.3 Untrusted Pointer Alignment Sanitization

The recently disclosed ÆPIC [26] and MMIO stale data leakage [74] attacks on Intel SGX platforms have shown that enclave secrets may propagate from microarchitectural fill buffers into architectural, software-visible registers when dereferencing unaligned pointers to MMIO devices. While CPU microcode updates have since been released to transparently cleanse fill buffers upon enclave exits on affected processors, additional software mitigations are still necessary to prevent confused-deputy exploitation of these issues during enclave execution [74, 75]. That is, even when the enclave shielding runtime has properly checked that untrusted, attacker-tainted pointer arguments fall entirely outside the enclave memory range, as can be validated by `PTRSan`, SGX enclaves have no way of knowing whether these untrusted memory locations refer to vulnerable MMIO regions. Indeed, privileged adversaries can trivially map untrusted memory pages to arbitrary MMIO devices, including the x86 APIC configuration registers [159]. As such, dereferencing untrusted pointers during enclave execution may unintentionally expose secret stale data, and Intel explicitly advises that SGX shielding runtimes should additionally constrain untrusted pointer dereferences to certain safe combinations of alignments and lengths [74, 75]. Note that this holds both for outside-enclave reads and writes, through the shared buffers data read (SBDR) and device register partial write (DRPW) processor vulnerabilities, respectively.

In response to these dynamic challenges, we developed a specialized `ÆPICSan` plugin, which investigates the alignment of each symbolic memory access that may resolve outside the enclave. Specifically, in accordance with Intel's intricate software security guidance [74, 75], we validate that every untrusted read or write access resolving outside the enclave is minimally 8-byte aligned, *i.e.*, has the lower three address bits cleared. We, furthermore, ensure that

untrusted read accesses have a size that is always *maximally* eight bytes at a time, whereas untrusted writes should be in chunks of *multiples* of eight bytes at a time [74]. Finally, when detecting unaligned untrusted writes, `ÆPICSan` parses the disassembly of the current basic block to filter out safe cases where the vulnerable write is preceded by the `VERW` instruction to cleanse leaky microarchitectural buffers and directly followed by an `LFENCE; MFENCE` instruction pair to avoid inadvertent transient refills, as per Intel's software security guidance [74].

Our complete `ÆPICSan` validator requires only 103 lines of Python code, where the majority of code concerns parsing the disassembly. This clearly shows the strength of exposing Pandora's enclave-aware memory model (cf. Section 4.4.3) to individual plugins that may have partially overlapping functionality, e.g., `PTRSan` vs. `ÆPICSan`.

The recent SBDR/DRPW disclosures required extensive manual software mitigations, frequently encompassing several rounds of commits and pull requests, throughout the SGX runtime ecosystem. We are the first to provide any form of toolchain support for automatically detecting and validating SGX pointer-alignment considerations, and we are the first to perform a wide-scale investigation of such issues remaining in real-world enclaves (cf. Section 4.7).

4.6.4 Control-Flow Hijacking Validation

Lastly, Pandora includes a `CFSan` plugin, implemented in 110 lines of Python code, that validates enclave control-flow events. This plugin reports insecure jump targets according to the location of the target and whether the target is attacker-tainted.

First, similar to prior work [12, 35], we report a critical security issue when the attacker can arbitrarily control a jump target inside the enclave. Furthermore, similar to the false-positive heuristic for `CFSan`, we only report a warning when attacker-tainted jump targets are constrained to always fall entirely inside the enclave.

In addition to this first criterion, partially covered by prior work, `CFSan` also includes novel rules to detect any enclave jumps to attacker-controlled memory contents. Specifically, we found that several shielding runtimes feature unmeasured and executable memory pages, so as to dynamically load (encrypted) code at runtime. As explained in Section 4.4.3, this type of enclave memory is not part of the attested MRENCLAVE measurement and is, as such, initially attacker-controlled until it is first initialized by enclave software. Thus, any enclave jumps to unmeasured memory that has not yet been initialized are

reported as a critical security issue. While, apart from validation on our own test enclaves, we have not encountered such instances in our evaluation on real-world enclave binaries, we are the first to formulate and write a sanitizer for this nuanced class of novel unmeasured enclave vulnerabilities.

Finally, note that, in line with our goal of truthful symbolic execution, the Pandora base engine already intercepts any jumps to outside the enclave memory range or to non-executable pages inside the enclave, regardless of `CFSan`. We simply abort the symbolic execution paths for these cases, as both of these events would result in a runtime exception on real SGX hardware and would, hence, not be an exploitable vulnerability besides denial-of-service.

4.7 Evaluation

We evaluated the efficacy of Pandora and its vulnerability detection plugins in two distinct ways. First, we developed a concise unit-test validation framework, loosely based on the existing Linux selftest enclave [145], to precisely diagnose (known) vulnerabilities in small benchmark enclaves compiled with increasing levels of mitigations. Second, we performed a comprehensive ecosystem analysis on 10 relevant, real-world SGX shielding runtimes, uncovering over 174 newly found vulnerable code locations, tracked via 7 (anonymized) common vulnerabilities and exposure (CVE) identifiers. Additionally, further demonstrating the versatility of Pandora, we made our symbolic-execution tool autonomously reproduce over 69 previously known vulnerable code locations from the literature in older versions of the investigated runtimes. Table 4.2 provides an overview of all reported and reproduced issues, whereas a more detailed breakdown is included in Appendix C.4.

4.7.1 Selftest Validation Framework

The Linux kernel natively includes drivers for Intel SGX since the 5.11 release [145]. As part of this effort, Linux also contains a bare-metal selftest enclave that provides a minimal example to test the loading and execution of an enclave binary without relying on any particular SGX shielding runtime. This Linux selftest enclave consists of hand-crafted assembly routines for entry and exit, plus an `ecall` dispatcher that calls `C` functions. While this selftest enclave is not intended to be a production runtime, Linux developers have noted that its code may be copied and provides a "great starting point if you want to do things from scratch" [120]. Indeed, we found that at least one real-world SGX project directly built on the Linux selftest enclave to date: Alibaba Inclavare

Table 4.2: Evidence of Pandora finding and reproducing vulnerabilities both in production and research runtimes.

Runtime	Version	Prod	Src	Plugin	Instances	CVE
Newly found vulnerabilities in shielding runtimes (total 174 instances)						
EnclaveOS	3.28	✓	✗†	ABISan	1	
EnclaveOS	3.28	✓	✗†	PTRSan	15	CVE-2023-38022
EnclaveOS	3.28	✓	✗†	ÆPICSan	33	CVE-2023-38021
EnclaveOS	3.28	✓	✗†	CFSan	2	
GoTEE	b35f	✗	✓	PTRSan	31	
GoTEE	b35f	✗	✓	ÆPICSan	18	
GoTEE	b35f	✗	✓	CFSan	1	
Gramine	1.4	✓	✓	ABISan	1	
Intel SDK	2.15.1	✓	✓	PTRSan	2	CVE-2022-26509
Intel SDK	2.19	✓	✓	ÆPICSan	22	
↪ Occlum	0.29.4	✓	✓	ÆPICSan	11	
Linux selftest	5.18	✗	✓	ABISan	1	
↪ Inclavare	0.6.2	✗	✓	ABISan	1	
Linux selftest	5.18	✗	✓	PTRSan	5	
↪ Inclavare	0.6.2	✗	✓	PTRSan	2	
Linux selftest	5.18	✗	✓	CFSan	1	
↪ Inclavare	0.6.2	✗	✓	CFSan	1	
Open Enclave	0.19.0	✓	✓	ABISan	2	CVE-2023-37479
Rust EDP	1.71	✓	✓	ABISan	1	
SCONE	5.7	✓	✗	ABISan	2	CVE-2022-46487
SCONE	5.7	✓	✗	PTRSan	10	CVE-2022-46486
SCONE	5.7	✓	✗	ÆPICSan	11	CVE-2023-38023
Reproduced vulnerabilities in older versions (total 69 instances)						
GoTEE	b35f	✗	✓	ABISan	1	
Gramine	1.2	✓	✓	ÆPICSan	10	
Intel SDK	2.1.1	✓	✓	ABISan	1	CVE-2019-14565
Intel SDK	2.13.3	✓	✓	ÆPICSan	28	
Open Enclave	0.4.1	✓	✓	ABISan	1	CVE-2019-1370
Open Enclave	0.4.1	✓	✓	PTRSan	13	CVE-2019-0876
Open Enclave	0.4.1	✓	✓	ÆPICSan	13	
Rust EDP	1.63	✓	✓	ÆPICSan	2	

Legend: † Source code was made privately available; ↪ Based on above runtime.

Containers [7] uses it as a skeleton example of best-practice enclave runtime integration. We report Pandora's findings on these bare-metal enclaves in the next section.

We developed a unit-test framework based on the Linux selftest enclave. This test suite contains individually crafted enclave binaries featuring multiple levels of ABI register cleansing and input pointer(-to-pointer) sanitizations. These enclaves, thus, provide a controlled test environment to craft arbitrarily complex and challenging scenarios to validate the efficacy of our plugins and Pandora's enclave-aware symbolic memory model. Furthermore, they allow to prototype conceivable vulnerabilities that have not (yet) been encountered "in the wild", e.g., jumps to unmeasured and uninitialized pages (cf. Section 4.6.4).

4.7.2 SGX Runtime Ecosystem Analysis

Runtime Selection To explore the vulnerability landscape for real-world enclave software, we evaluated Pandora on a diverse set of 7 production-quality and 3 research-grade Intel SGX shielding runtimes. Note that, as discussed in Section 4.3, we opted to focus on validating the vital enclave shielding runtime itself, including indispensable, low-level initialization and entry code, rather than the more accessible challenge of validating higher-level application logic as explored in complementary prior work [12, 35, 36]. While the latter typically only affects a single (research) application that makes incorrect use of shielding abstractions, e.g., unchecked `user_check` pointers [78, 101], production-quality shielding runtimes are supposed to be thoroughly vetted and any vulnerabilities found would affect universally all applications developed on top.

Our runtime selection includes diverse enclave programming paradigms, including 2 SDKs (Intel SGX SDK [78] and Microsoft Open Enclave [101]), 4 libOSs (EnclaveOS [54], SCONE [124], Occlum [130], and Gramine [144]), 2 secured language runtimes (Rust-EDP [53] and Go-TEE [58]), and 2 bare-metal enclaves (Linux selftest [145], and Inclavare [7]). We included the bare-metal enclaves, as well as the academic Go-TEE research prototype runtime, to complement the insights from the more mature production ecosystem. Furthermore, while the majority of SGX shielding runtimes are developed as open-source software, our selection also includes two proprietary runtimes: EnclaveOS, with source code privately provided by the vendor, and SCONE, with only binaries available.

Due to the intricacies involved in building old runtime versions with often complex dependencies, we opted to limit our choice of known vulnerabilities to a representative sample across major runtimes. We see a systematic overview of the vulnerability landscape of past runtimes as an interesting and feasible

direction for future work and believe that Pandora could aid in such a survey. In the following, after describing our experimental setup, we highlight the most interesting findings of each plugin.

Experimental Setup We extracted exact enclave dumps via SGX-TRACER and ran Pandora on all runtimes with a time budget of 12 hours and a memory budget of 256 GB, whichever occurred first. Cloud instances with such memory budget are commercially available beginning at 4 $ per hour, making this limit feasible for occasional extensive validation with Pandora, e.g., as part of continuous integration (CI) for releases (as at least one vendor privately expressed interest in). Each runtime was explored twice: once with a default breadth-first exploration strategy and once with a depth-first strategy that eagerly followed the longest paths.

We note that in our experiments, the 256 GB memory limit was only hit twice, namely for the Intel SGX SDK 2.19 when using breadth-first search after approximately 8 hours, and for GoTEE as the enclave memory dump is exceedingly large at 64 GB. In all other cases, the memory consumption varied between 24.6 GB and 196.7 GB for breadth-first search and from 4.9 GB to 154.7 GB for depth-first search.

In some rare cases, our Pandora prototype crashed before reaching these limits due to remaining unsupported x86 instructions or due to crashes in the underlying `angr` and `z3` solver. For EnclaveOS specifically, we manually guided Pandora to skip two functions that either contain still unsupported AES-NI instructions, or execute a waiting loop that expects a second thread to fill data before continuing.

ABI Sanitization Issues

Following a recent overview study [153], Pandora promptly confirmed known ABI issues in older Intel SGX SDK and Open Enclave binaries, which have since been evidently mitigated (cf. Table 4.2). Nonetheless, Pandora found that the proprietary SCONE runtime still lacked any sanitization code for x87 and SSE floating-point configuration registers. We experimentally demonstrated that this lack of ABI sanitization, can be exploited in practice via a proof-of-concept exploit that successfully introduces rounding errors in an elementary "sconified" floating-point application. Following our responsible disclosure, tracked under CVE-2022-46487, these issues have been patched in the latest SCONE release 5.8.0.

Additionally, `ABISan` found that the academic GoTEE runtime, as well as the Linux selftest enclave and Inclavare, universally lack ABI entry sanitizations for RFLAGS and floating-point configuration registers. Interestingly, Inclavare took care to cleanse extended processor state on enclave exit, but not on entry. Highlighting the strength of `ABISan`'s taint policy, the plugin autonomously discovered that GoTEE even lacks secure stack pointer initialization, which could be exploited to obtain full code execution in this runtime (cf. as also reported by both `CFSan` and `PTRSan`).

Our systematic analysis, furthermore, identified an interesting case of *regression* in Open Enclave, which was assigned CVE-2023-37479 by Microsoft and mitigated in release 0.19.3. Particularly, in response to prior research [157], commit `efe7504` in Open Enclave included a patch to properly sanitize the x86 alignment-check flag. However, `ABISan` discovered that in current versions of Open Enclave, the alignment-check flag was no longer properly sanitized after the initial enclave sanitization routines have completed. Upon further investigation, we were able to conclude that Open Enclave accidentally reintroduced the once-fixed vulnerability with commit `16efbd6` in 2021, in a patch set to mitigate another attack [41] that places more stringent demands on stack-pointer initialization for exception handlers. This instance of unintended regression thus provides a clear illustration of the complexity of shielding responsibilities and the potential value of including an automated tool like Pandora in CI pipelines to test against known vulnerabilities before releasing new software versions.

A final and particularly widespread line of ABI sanitization issues follows from Intel's recent MXCSR configuration-dependent timing (MCDT) software guidance [73]. Particularly, Intel recommends that shielding runtimes set all floating-point exception status flags in the MXCSR register for the lifetime of the enclave to avoid subtle, operand-dependent timing differences in otherwise constant-time code on affected processors. Notably, this refined guidance did not result from an academic publication or security advisory and may have been easily missed by runtime developers. Indeed, `ABISan` detected that only the Intel SGX SDK and the dependent Occlum runtime properly set MXCSR according to the new recommendation, and *all* other runtimes did not. Following our disclosure, this has since been patched in Open Enclave, Rust-EDP, and EnclaveOS.

Pointer Sanitization Issues

The strength of the `PTRSan` plugin is to rigorously investigate issues with pointer dereferences across many enclave runtimes.

In the SCONE production runtime, `PTRSan` uncovered 10 unique critical issues: 8 entirely unconstrained, attacker-tainted pointer dereferences and 2 untainted outside-enclave reads. Although the source code was not available, Pandora was able to generate precise basic-block backtraces annotated with ELF symbols, aiding in our investigation and even the development of proof-of-concept exploits. We reported each issue, tracked as a bundle under CVE-2022-46486, to the SCONE developers who confirmed our findings and included patches in the latest release 5.8.0.

In EnclaveOS, `PTRSan` was able to detect a particularly subtle instance of an untrusted pointer dereference as part of a string length calculation, which is logically correct but can be abused as a capable side-channel oracle to precisely locate all null bytes in enclave memory [157]. Fortanix gave a *high* severity rating for this finding, tracked under CVE-2023-38022, and mitigated it promptly in version 3.29. As a second notable finding in EnclaveOS, Pandora autonomously detected that overflow protections were missing in the untrusted pointer validation logic of the enclave binary. Upon closer examination, we found that existing source-level overflow checks were silently optimized away by the compiler. Specifically, the source code utilized `void*` pointer arithmetic, which, unfortunately, is undefined behavior in `C`, leading to the compiler removing this check completely. Pandora correctly reported that, with this check missing, the attacker can cause untrusted pointers to wrap the address space via an unsigned integer overflow. This issue, thus, highlights the strength of Pandora's binary-level validation and accurate symbolic constraint solving of not only untrusted pointer values, but also their sizes.

Furthermore, as part of this research, `PTRSan` additionally confirmed an untrusted pointer dereference in the protected code loader of the Intel SGX SDK version 2.15.1, tracked via CVE-2022-26509 and patched in later versions. This issue underlines the importance of validating low-level runtime initialization code, as this pointer check was missing before any in-enclave relocations, including global variables containing the enclave base address and size needed in the validation function itself, had been performed.

In the GoTEE research runtime, `PTRSan` discovered numerous (31) unconstrained pointer dereferences, highlighting that even safe languages are not immune to oversights in pointer validation for SGX's unique attacker model. Furthermore, all bare-metal enclaves were found especially vulnerable without any pointer sanitization measures (as reported both by `PTRSan` and `ÆPICSan`). Likewise, the Inclavare enclave contains several vulnerable invocations of `memcpy` with unconstrained source and destination parameters, and the Linux selftest enclave contains 5 entirely unconstrained, attacker-tainted pointer dereference locations that can be trivially exploited to leak or corrupt arbitrary in-enclave memory locations.

Finally, for the known-vulnerable version 0.4.1 of Microsoft Open Enclave, Pandora correctly identified CVE-2019-0876 [157], which highlights the power of multiple reentries, as the vulnerability can only be triggered after the enclave has been initialized. In addition, `PTRSan` also reported a (presumably unknown) issue in this old runtime version, indicating a lack of pointer sanitization in the `oe_initialize_cpuid()` function.

ÆPIC Sanitization Issues

Pandora is the first tool to support automated analysis and validation of ÆPIC-style untrusted pointer alignment vulnerabilities in SGX enclaves. We, thus, employed our novel `ÆPICSan` plugin to perform a large-scale, automated analysis to assess the completeness of Intel's particularly complex and error-prone software mitigation guidelines [74, 75] in real-world enclave shielding runtimes. As result of this systematic analysis, Pandora found that SBDR and DRPW mitigations were missing entirely in GoTEE (18 unique instances), SCONE (11 instances; tracked via CVE-2022-46487 and mitigated in version 5.8.0), and EnclaveOS (33 instances; tracked via CVE-2023-38021 and mitigated in release 3.32). Existing mitigations in Gramine, Rust-EDP, and Open Enclave were found sufficient, but `ÆPICSan` autonomously discovered a missing SBDR sanitization in the enclave initialization phase of the latest version of the Intel SGX SDK (also inherited by the derived Occlum runtime), highlighting that adequately restricting untrusted pointer alignments is challenging even for mature runtime developers.

As expected, we additionally confirmed that `ÆPICSan` can automatically reproduce ample SBDR and DRPW issues in older versions of Gramine, Rust-EDP, Open Enclave, and the Intel SGX SDK without mitigations.

Control Flow Issues

The `CFSan` plugin found a delicate issue in EnclaveOS where the global offset table (GOT) is incorrectly accessed before relocation of the enclave has completed. The GOT is used to jump to functions in position-independent code and has to be securely initialized, *i.e.*, relocated, before it can be used inside enclaves. The issue found by Pandora, and confirmed and fixed by Fortanix, concerns an unusual trace where an error occurs during initialization, which results in the code calling a debug logging function.

Furthermore, `CFSan` found that Inclavare's bare-metal enclave assembly entry stub incorrectly uses a signed `JGE` x86 jump instruction, instead of a proper

unsigned `JAE` condition to sanitize the attacker-provided index into the `ecall` function-pointer table. Critically, this subtle oversight ultimately allows *arbitrary* control-flow hijacking by passing a large negative index into the `ecall` table and loading the function pointer from untrusted, attacker-controlled memory. Likewise, `CFSan` found that, depending on the optimization level, in-enclave relocation code for the `ecall` table was missing in the dispatcher of the Linux selftest enclave.

Finally, due to the lack of secure stack switching in GoTEE, `CFSan` reported unconstrained `RET` targets.

4.8 Discussion

We see Pandora as a mature prototype of an enclave-aware symbolic execution tool that can serve as a basis for future science. In particular, we designed Pandora with great care for usability, through a well-documented command line interface and detailed HTML reports, and reusability through our plugin-based approach that makes it easy to implement additional security analyses. Pandora has demonstrated its usefulness by automatically finding vulnerabilities in production runtimes. Hence, we believe that Pandora is a valuable step forward in vulnerability detection for enclaves.

Limitations Pandora, as any symbolic-execution tool, suffers from the well-known limitation of state explosion, which can make exhaustive exploration of larger binaries practically infeasible. Hence, vulnerabilities can still remain undetected in unexplored paths. Furthermore, angr [164] itself is not sound as it may concretize values during symbolic execution. To avoid missing program behavior, we adopted the most conservative approach whenever possible and tried to refrain from unnecessary concretization of symbolic values. Despite these limitations, angr is particularly powerful for rapid development of vulnerability plugins in comparison to fully fledged code verification tools. Moreover, Pandora is still able to automatically find vulnerabilities in production runtimes, which demonstrates the practicality of the approach. We implemented novel, enclave-aware performance optimizations, including uninitialized memory and state-uniqueness reductions (cf. Section 4.4.5), and we utilized both breadth-first and depth-first exploration in our evaluation to cover more enclave behavior.

Additionally, as any automatic vulnerability scanner, Pandora can report false-positive issues, which can potentially lead to overly exhaustive outputs. Pandora attempts to limit the strain on the human analyst via two steps. First, potential issues are classified into multiple levels of criticality, and the report outputs are

formatted in modern HTML forms that allow to filter away given criticality levels. Second, plugins may downgrade the severity of issues via sensible heuristics, e.g., Section 4.6.2 explained how `PTRSan` downgrades attacker-tainted pointers when they are constrained to a region entirely inside the enclave, closely resembling the benign pattern of an attacker-controlled index in a trusted enclave buffer.

Future Work Potential future extensions of Pandora concern novel vulnerability-detection plugins, as well as the investigation of transient execution access patterns in enclaves [30, 43, 67]. Furthermore, we see Pandora as a useful tool for a broad ecosystem analysis of the Intel SGX landscape and how fast vulnerability patches propagate across runtimes. Ultimately, future work could even explore automated exploit generation and binary patching using Pandora's precise vulnerability reports.

There are additionally some performance improvements that could allow Pandora to explore enclaves in even more depth. While we already implemented a depth-first extension to Pandora that severely limits the memory use necessary during exploration, `angr` still only utilizes one single CPU core. Future work could thus investigate how `angr` symbolic exploration can be split up onto multiple cores while retaining the same enclave-aware characteristics of Pandora that are necessary to e.g., identify enclave boundaries. Additionally, to mitigate path explosion, we could also adopt state-merging [85] or path prioritization strategies [20, 90].

4.9 Conclusion

In recent years, a sizable ecosystem of Intel SGX enclave shielding runtimes has emerged. However, writing secure SGX software has proven to be particularly challenging due to the moving nature of the threat landscape, and not even well-designed and vetted shielding runtimes have been immune to missing nuanced attack vectors or to reintroducing already known vulnerabilities into their code. The research community has only recently started to look into SGX-aware symbolic execution, but has focused on application logic only while largely skipping the crucial enclave shielding runtime itself. In this work, we presented Pandora, the first enclave-aware and pluggable symbolic-execution tool that allows *truthfully* validating arbitrary enclave binaries, including low-level runtime initialization and entry phases. With 4 diverse prototype plugins, we found 174 new and 69 known vulnerable code locations across a wide selection of 10 SGX runtimes. Ultimately, we envision Pandora not only as a practical

validation tool for real-world enclaves today, but also as a solid, extensible and open-source foundation for future science on SGX software validation.

Chapter 5

Conclusion

Modern computing faces many security challenges due to increased interconnectivity and sharing of resources. Hardware-based isolation primitives have established themselves as one viable path to secure those complex environments and their diverse stakeholders. Nowadays, trusted execution environments (TEEs) are maturing, and the research community understands microarchitectural and architectural nuances and their security impacts better than a few years ago. To that end, the most significant findings of the past years, such as Foreshadow [151], LVI [154], ÆPIC [26] and others, have been mitigated with either microcode updates or can be alleviated by following strict developer guidelines. At the same time, several research questions remain open before TEEs can be adopted broadly. For example, while the research community may have developed a clear understanding of the capabilities and limitations of the current TEEs, this understanding may not yet reach across different levels of developers' expertise. This dissertation has made contributions in two directions that can help bring modern trusted execution architectures closer to a broader adoption: availability guarantees for mutually distrusting enclaves and interface sanitization issues between enclaves and the untrusted environment.

In this chapter, we first summarize the contributions of this dissertation, then outline potential directions for future work and lastly conclude with final remarks.

5.1 Summary of Contributions

Availability Guarantees for Mutually Distrusting Enclaves In Chapter 2, we presented Aion, which tackles the issue of providing availability to mutually distrusting enclaves on the same system. By utilizing small hardware changes to be able to place a real-time scheduler inside an enclave, Aion can make this scheduler enclave privileged for availability only. As a result, Aion application enclaves can benefit from strong hard real-time availability guarantees if they trust the small scheduler enclave to provide this availability. At the same time, these application enclaves must not trust the privileged scheduler for confidentiality or integrity, nor do they need to trust any other enclave on the system. Aion shows that it is possible to carefully design modern enclave architectures to equip them for the complex domain of mixed-criticality and safety-critical systems.

Interface Sanitization in Intel SGX Software responsibilities for securing an enclave in the growing and diverse ecosystem of Intel Software Guard Extensions (SGX) are not sufficiently understood in practice. In parallel with related work, our contribution presented in Chapter 3 provided one building block to a clearer understanding of these software responsibilities of enclaves. We presented attacks on the rounding and precision modes of legacy x87 floating-point units (FPUs) and more modern Streaming SIMD Extensions (SSE) extensions and showcased that even these seemingly trivial modifications can impact the integrity of an enclave's computation. Our fault injection attacks on the Intel SGX FPU have led to two common vulnerabilities and exposures (CVEs) and patches in five enclave shielding runtimes. With the clear recommendations that resulted from our research, we aimed to help developers to sanitize their enclaves, and we were able to help developers implement our recommendations in their products.

Yet, these recommendations were partially refined in early 2023 when a new advisory by Intel unveiled more nuanced timing side channels, leading to an updated configuration of the FPU on enclave entry [73]. This is but one example of the fast-moving landscape of Intel SGX software responsibilities and what sanitizations have to be applied to any interaction with the untrusted world. To help both the research and developer community, we then presented Pandora as a means of automatically analyzing an enclave and validating that it contains no known vulnerabilities. With our approach of first creating a clear view of the enclave's memory at enclave creation time, Pandora is software development kit (SDK)-agnostic and can analyze arbitrary enclave binaries, irrespective of the developer's tools of choice. In our work, we developed plugins to find four classes of known vulnerability types, and we built Pandora to be highly extensible for more such plugins. As a result, Pandora found 174 new vulnerability instances

across 10 shielding runtimes, yielding 7 new CVEs. During this research, we worked with several vendors to mitigate the found vulnerabilities, and one vendor already notified interest in integrating Pandora into a continuous integration pipeline. Our work and several vendors' responses show that a high-quality tool to find known vulnerabilities automatically can help move the domain of Intel SGX development forward. We envision that a tool like Pandora can be used across vendors and projects, and establish a common understanding of the challenges enclaves face.

5.2 Future Work

Hardware-enforced isolation of software components has proven valuable and usable in modern computing systems. Some effort remains, however, to secure and ease the use of this latest addition of trusted execution technology that allows to dynamically spawn multiple enclaves on a system that may mutually distrust each other.

We see multiple directions for future work: evaluating availability on new generations of TEEs, building out Pandora to other classes of vulnerabilities and making it more practical, and investigating other TEEs-interfaces than that of Intel SGX.

Availability on Upcoming TEEs

Aion investigated availability on Sancus, a research architecture that is based on openMSP430. Related works exist that build on ARM TrustZone, such as RT-TEE [166] and MrTEE [161]. A crucial difference between Aion and the other availability works is, however, that the Aion architecture is the only architecture that allows mutually distrusting enclaves that additionally do not have to trust the underlying software stack (*i.e.*, the scheduler) for confidentiality and integrity. In TrustZone, the operating system (OS) in the secure world must be fully trusted by the trusted applications as it is the privileged entity that sets up and controls the secure world. While co-located applications in the secure world can be separated by means of virtual memory, the OS is trusted in terms of confidentiality, integrity, and availability by all applications.

An interesting research question arises with the upcoming ARM confidential compute architecture (CCA) [15]. ARM CCA introduces a new security mode called the realm management extension, which allows the creation of additional

so-called realms alongside the legacy TrustZone secure world. These realms have many similarities to enclaves, as they have been discussed in this dissertation. Importantly, different realms do not have to trust each other and must neither trust the underlying realm manager, *i.e.*, the hypervisor. At the same time, the TrustZone secure world does not cease to exist but continues to have several privileges without having direct access to the created realms.

Future work could explore how the ARM CCA architecture can be used to provide strong availability guarantees to multiple realms simultaneously. To a certain degree, we already discussed the capabilities of CCA regarding trusted timestamping [4]. In this context, the guarantees provided by the realm management software and hypervisor are similar to the trusted scheduler in Aion: The software is trusted to provide scheduling decisions but does not need to be trusted for confidentiality and integrity. Additional effort is necessary, however, to evaluate the capabilities of the CCA architecture and whether and what services a real-time scheduler in the secure world can provide to mutually distrusting mixed-criticality software separated into several realms.

A second orthogonal path for future work is to investigate and evaluate availability guarantees on the modern RISC-V architecture. RISC-V shows great promise due to its openness and several projects like Keystone [86] or CURE [19] have already demonstrated TEEs on RISC-V. Migrating Aion to a more modern RISC-V architecture would allow future work to better compare it to existing or upcoming works that similarly tackle enclave availability.

Increasing the Practicality of Pandora

In Chapter 4 we presented Pandora as a means of automatically checking known vulnerabilities in arbitrary Intel SGX enclaves. We strongly believe that our tool provides a reasonable trade-off between complex formal verification and constrained software tests. At the same time, the applied method of using symbolic execution to search for known vulnerabilities is not without disadvantages.

In the following, we address two of the major disadvantages and how future work can address them: the coverage of tools like Pandora in terms of vulnerabilities and in terms of completeness of the analyzed code.

Coverage of Vulnerabilities A reasonable criticism of tools like Pandora is that they can only detect already known vulnerabilities that the tool was designed to detect. At the same time, we see a usable symbolic execution tool like Pandora as a practical middle point. Although complex and time-consuming,

formal verification of enclaves would allow to receive clear guarantees on the security of a given enclave software. Yet, even formal verification is not without errors, as provable security is not a guarantee for the absence of flaws in the actual implementation. Any provably secure system relies on the correctness and completeness of the underlying model, the match between the formal model and actual implementation, and the completeness of the formalized and proven lemmas. Related work has already shown that provably correct trusted execution architectures are not necessarily always secure, as essential nuances can be missed when modeling a system [24]. On the other end of the abstraction spectrum lie software tests like unit tests in a continuous integration pipeline. These tests have the benefit of being run directly on the implemented software, nearly eliminating the danger of mismatching the checked model and reality. However, the downside of these tests is that it is difficult for them to provide any general statements about the absence of flaws. While specific undesired test cases can be evaluated, a general statement that a similar case will not happen is extremely difficult to generalize. One example of this is the case of application binary interface (ABI) sanitization in Open Enclave that we discuss in Section 4.7 where a unit test existed to verify the sanitization of the x86 direction flag but no test existed to verify the sanitization of the alignment check flag. This oversight led to a case of regression where a missing alignment check sanitization was reintroduced after this vulnerability had already been mitigated a few years prior.

We thus see symbolic execution at the reasonable intersection of analyzing the effective assembly-level code of the resulting enclave binary but also being able to check clear mathematical constraints of the values that we simulate. Still, a disadvantage of Pandora remains that it can only check for the absence of vulnerabilities that it was programmed to check for. In the presented work, we already carefully designed plugins like `PTRSan` so that it is able to detect any issue with pointer sanitization as a result of the constraints placed on the accessed pointer. Future work is necessary to address similar and related issues and carefully write plugins and detection methods. Interesting areas of extension for Pandora are the broad area of transient execution vulnerabilities for which a new Pandora execution model must be implemented and interruption-based attacks like SmashEx [41] and the new hardware-software co-design around Intel SGX AEX-Notify [39].

Coverage of Analyzed Code Symbolic execution suffers from a problem called state explosion, where each exploration step that contains a constraint branches the explored state into two states: one with the constraint being true and one with the constraint being false. This leads to an exponential increase in the number of to-be-evaluated states throughout an exploration of a binary.

While Pandora focused on the enclave shielding runtimes, *i.e.*, the small parts of an enclave that execute first, the guarantees provided by Pandora would also be useful for the applications that are executed afterward. Future work should explore how Pandora can be parallelized, how symbolic execution can be coordinated across machines, and how the special semantics of enclaved execution can be used as an advantage for symbolic execution. For example, after the initialization phase of an enclave succeeded and the enclave returned from this initial `ecall`, any subsequent `ecall`s can, in theory, be simulated in parallel as they can be seen as independent in a constraint system. Thus, a parallelized Pandora could take snapshots of specific states and ensure that a cohort of threads investigates each possible path into the enclave application. While this has its limit in regards to complex sequences of expected `ecalls` that are enforced by the global state, parallelization in execution would allow for a broader investigation of binaries in a short time. Similarly, an interesting extension is the exploration of multi-threaded applications as Pandora is currently not able to explore multi-threaded enclaves where different threads communicate with and wait on each other during execution. Furthermore, future work should also explore the coverage of Pandora so that a statement can be made of the exploration depth that was reached during a specific run.

Other TEE Interfaces

This dissertation investigated interface sanitization issues between Intel SGX enclaves and the untrusted world. While Intel SGX is exceptionally prone to these issues due to the memory design of trusted and untrusted world sharing an address space, other TEE architectures are not necessarily safe from similar vulnerabilities. Future work, and in fact, continuous effort, should be put into constantly reevaluating the intersection of trusted and untrusted domains to ensure that the interface does not open up an attack surface for malicious parties. Below, we discuss potential issues and future work directions for two types of such TEEs: interfaces with accelerator TEEs and interfaces with virtual machine (VM) TEEs.

Accelerator TEE Interfaces With the success of TEEs on central processing unit (CPU) architectures, increasing effort is being put into also protecting workloads on accelerators. Especially with the rise of machine learning, there is increasing interest in moving the computation-heavy predictions and learning onto accelerators that are explicitly designed for fast computations of these workloads. One example of such accelerators is the Nvidia Hopper architecture [110] that brings trusted execution technology to enterprise-grade graphics processing units (GPUs). To protect workloads being processed

inside Nvidia Hopper GPUs, explicit support for confidential computing allows establishing an attested and authenticated channel between the CPU and GPU TEEs. Interestingly, the Nvidia TEE has multiple operation modes to either flexibility reserve the whole GPU for a single CPU client or to access individually shielded VMs-based TEEs on the GPU from multiple, mutually distrusting CPU clients [110]. This complexity adds a second layer on top of the already challenging security CPU TEE landscape. Future work should investigate what security issues arise from this interface between CPU and GPU and whether and how a malicious party can impact the secure computations running inside the GPU VM of another client.

VM TEE Interfaces TEEs released after Intel SGX are following the VM-based protection approach that we introduced in Section 1.1.1, such as AMD Secure Encrypted Virtualization (SEV) [9] and Intel Trust Domain Extensions (TDX) [71]. While this dissertation focused on enclave architectures, future work should investigate interface sanitization issues with VM-based TEEs. The promise of these TEEs is that by drawing the isolation boundaries around the OS, the hypervisor and any software underlying the OS must not be in the trusted computing base (TCB) anymore. A side effect of trusting the OS is that the protected application does not reside in the same address space as any untrusted application, and thus the class of confused deputy attacks should not apply to the majority of VM-based TEE applications. Yet, nuanced use cases exist where the hypervisor may map shared pages into the address range of the trusted world to enable communication with the untrusted world or, for example, with peripheral drivers. These use cases then reintroduce the issue of confused deputy attacks to the realm of VM-based architectures which may be exceptionally prone to occurring issues as their key features allow running unmodified code inside the trusted environment. Future work should investigate potential issues arising from these and similar issues.

5.3 Concluding Remarks

The area of trusted execution environments is one promising building block to ensure security in modern computing. This dissertation contributed to the state of the art of enclave-based TEEs in two directions by proposing a hardware-software co-design to ensure enclave availability, and by securing the interface between untrusted and trusted environments. With the steady rise in interest from academia and industry, confidential computing, in general, is a growing domain that will, over the coming years, generate a larger variety of TEEs with more complex architectures. On this path, however, multiple challenges still

need to be overcome, and a better understanding of TEE security, capabilities, and limitations must be formed. This dissertation provides a step towards this understanding, and continued effort by the research community will be instrumental to the success of confidential computing across the computing spectrum.

Appendix A

Additional Resources for Aion

A.1 Atomicity State Machine

Figure A.1 shows the complete atomicity state machine with all edge cases. Upon boot of the system (initial state), the global interrupt enable bit is set. In this default state, legacy instructions like `eint` or `dint` have no effect on the interrupt enable bit. Only a `clix` instruction can disable interrupts for `x` cycles.

Switching into an enclave starts the atomic enclave entry counter that allows to atomically execute a `clix` instruction before being interrupted. Once the atomic enclave entry or a `clix` instruction inside an enclave expires, the state machine switches the enclave into an interruptable mode. In this mode, the enclave can be interrupted at any point by the scheduler, but it can also restart a `clix` instruction to enter a new atomic period. Crucially, however, the atomicity engine will ensure that the enclave will always spend one cycle in the interruptable enclave state to ensure that the scheduler will be executed if requested by a timer interrupt.

Upon switching to the scheduler, *i.e.*, the special enclave that has been spawned as the first enclave on the system, any remaining `clix` counter is reset to zero and the global interrupt enable bit is completely cleared. The scheduler is thus the only entity that has unbounded control over the interrupt bit. Illegal instructions like enclave entries from within an atomic enclave entry or nested `clix` instructions lead to violations that are handled by the scheduler.

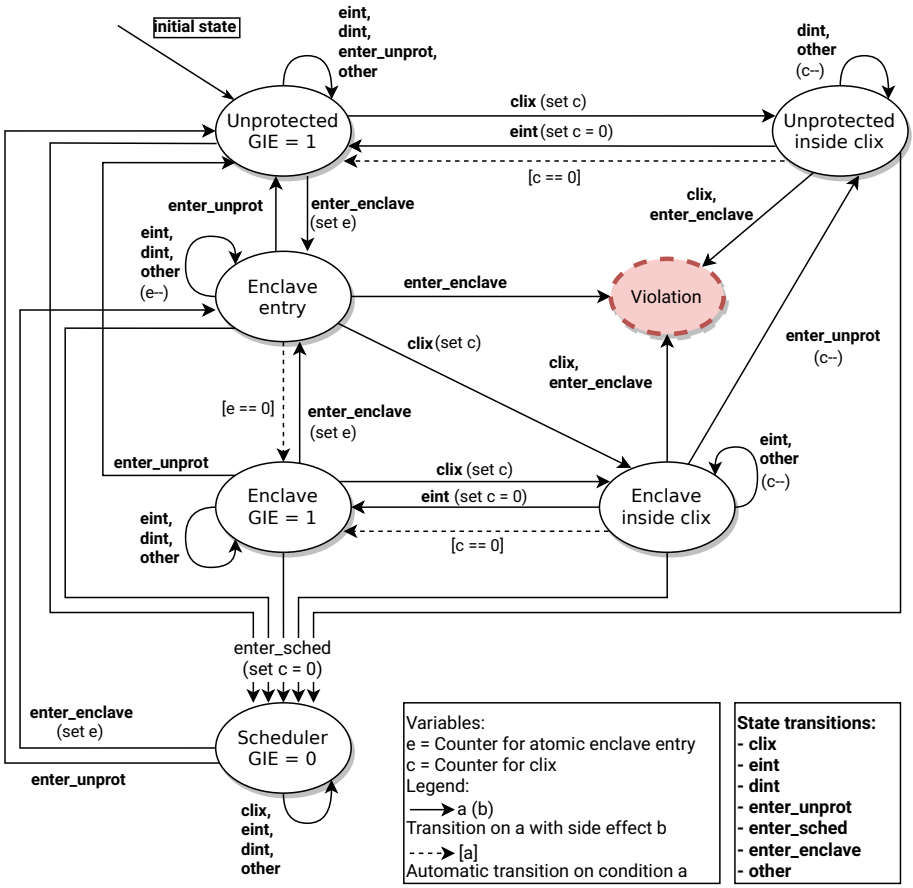


Figure A.1: Atomicity state machine showing explicit and implicit transitions from unprotected, enclaved, and scheduler states. Note that interrupt transitions are not explicitly shown in this figure but can be interpreted as `enter_sched` transitions.

A.2 Case Study Source Code in C

```
#include <msp430.h>
#include "uart.h"
#include "uart_hardware.h"
#include <stdio.h>
#include "kernel_defines.h"
#include "secure_mintimer.h"
#include "log.h"
```

```c
#include "sancus_helpers.h"

#define ___MACRO_CLIX(clix_length)  \
    __asm__("push r15");    \
    __asm__("mov.w %0, r15" : : "i"(clix_length));    \
    __asm__(".word 0x1389");    \
    __asm__("pop r15");

#define _HAVE_APPA            1
#define _HAVE_APPB            1
#define _HAVE_APP_SLEEP       1
#define _HAVE_IO_THREAD       1

/* --- IO Enclave ------------------------------------------------ */
DECLARE_SM(ioenclave,  0x1234);

#ifdef _HAVE_IO_THREAD
#define IO_BUFS        4
SM_DATA(ioenclave) unsigned char io_bufs[IO_BUFS] = {0, 0, 0, 0};
SM_DATA(ioenclave) bool io_ready[IO_BUFS] = {false, false, false,
    false};
#endif

// Output
bool SM_ENTRY(ioenclave) io_uart_write_byte(unsigned char b)
{
#ifdef _HAVE_IO_THREAD
    // Async I/O
    ___MACRO_CLIX(50);
    int caller = (int) sancus_get_caller_id();
    if (!caller || caller >= IO_BUFS) { caller = 0; }
    if (io_ready[caller]) {
      return (false);
    } else {
      io_bufs[caller]  = b;
      io_ready[caller] = true;
      return (true);
    }
#else
    // Sync I/O
    ___MACRO_CLIX(30);
    while (UART_STAT & UART_TX_FULL) {}       // !!
    UART_TXD = b;
    return (true);
#endif
}

// Read sensor
uint64_t SM_ENTRY(ioenclave) io_get_reading(void)
{
    ___MACRO_CLIX(30);
    return (secure_mintimer_now_usec64());
}
```

```
60
61 #ifdef _HAVE_IO_THREAD
62 static char sm3_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
63 // Async I/O thread
64 void SM_ENTRY(ioenclave) io_thread(void)
65 {
66     while (true) {
67       // this could implement *any* policy.
68       for (int i = 0; i < IO_BUFS; i++) {
69         if (io_ready[i]) {
70           ___MACRO_CLIX(30);
71           while (UART_STAT & UART_TX_FULL) {} // !!
72           UART_TXD = io_bufs[i];
73           io_ready[i] = false;
74         }
75       }
76 #ifdef _HAVE_APP_SLEEP
77       ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, ioenclave)
78 #endif
79     }
80     return;
81 }
82 #endif
83
84 /* --- APP A ------------------------------------------------- */
85 #ifdef _HAVE_APPA
86 static char sm1_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
87 DECLARE_SM(appa,       0x1234);
88
89 SM_DATA(appa) uint64_t reading_a = 0;
90
91 void SM_ENTRY(appa) a_entry(void)
92 {
93     printf2("A: ID %d, called by %d\n",
94         sancus_get_self_id(), sancus_get_caller_id());
95
96     while (true) {
97       reading_a = io_get_reading();
98       printf1("A: t is %lu\n", reading_a);
99       if (reading_a >= 50000) { io_uart_write_byte('A'); }
100 #ifdef _HAVE_APP_SLEEP
101       ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, appa)
102 #endif
103     }
104 }
105 #endif
106
107
108 /* --- APP B ------------------------------------------------- */
109 #ifdef _HAVE_APPB
110 static char sm2_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
111 DECLARE_SM(appb,       0x1234);
112
```

```
113 SM_DATA(appb) uint64_t reading_b = 0;
114
115 void SM_ENTRY(appb) b_entry(void)
116 {
117     printf2("B: ID %d, called by %d\n",
118         sancus_get_self_id(), sancus_get_caller_id());
119
120     while (true) {
121       reading_b = io_get_reading();
122       printf1("B: t is %lu\n", reading_b);
123       if (reading_b >= 50000) { io_uart_write_byte('B'); }
124 #ifdef _HAVE_APP_SLEEP
125       ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, appb)
126 #endif
127     }
128 }
129 #endif
130
131 /* --- Unprotected Job Creation ------------------------------ */
132 int main(void)
133 {
134     LOG_INFO("######## Riot on Sancus\n");
135     LOG_INFO("Case study with same prio levels\n");
136
137     while(sancus_enable(&ioenclave) == 0);
138 #ifdef _HAVE_APPA
139     while(sancus_enable(&appa) == 0);
140 #endif
141 #ifdef _HAVE_APPB
142     while(sancus_enable(&appb) == 0);
143 #endif
144
145 #ifdef _HAVE_APPA
146     thread_create_protected(
147         sm1_unprotected_stack,           // Unprotected stack for
    OCALLS
148         THREAD_EXTRA_STACKSIZE_PRINTF,   // size of the
    unprotected stack
149         1,                               // Priority to give
150         THREAD_CREATE_WOUT_YIELD,        // Thread create flag
151         SM_GET_ENTRY(appa),              // SM Entry address
152         SM_GET_ENTRY_IDX(appa, a_entry), // SM IDX address
153         "A");                            // Name for console
    logging
154 #endif
155 #ifdef _HAVE_APPB
156     thread_create_protected(
157         sm2_unprotected_stack,
158         THREAD_EXTRA_STACKSIZE_PRINTF,
159         1,
160         THREAD_CREATE_WOUT_YIELD,
161         SM_GET_ENTRY(appb),
162         SM_GET_ENTRY_IDX(appb, b_entry),
```

```
163         "B");
164 #endif
165 #ifdef _HAVE_IO_THREAD
166     thread_create_protected(
167         sm3_unprotected_stack,
168         THREAD_EXTRA_STACKSIZE_PRINTF,
169         1,
170         THREAD_CREATE_WOUT_YIELD,
171         SM_GET_ENTRY(ioenclave),
172         SM_GET_ENTRY_IDX(ioenclave, io_thread),
173         "IO");
174 #endif
175
176     LOG_INFO("Thread initialization done\n");
177     while(true){
178         secure_mintimer_usleep(300000);
179     }
180     LOG_INFO("Exiting main thread by shutting down CPU\n");
181     sched_shut_down();
182
183     UNREACHABLE();
184     return 0;
185 }
```

Listing A.1: Source code of our case study implementation in C. Note that two while-loops in l.48 and l.71 do have deterministic execution time unless there is a hardware fault.

Appendix B

Additional Resources for Faulty Point Unit Attacks

B.1 Proof-of-concept Enclave Code

This appendix lists the C source code (Listing B.1) and compiled assembly (Listing B.2) for the benchmark toy example enclave discussed in Section 3.3.2 and Table 3.1. The assembly code in Listing B.2 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 B.1: 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)
```

```
9   callq  4450 <acosl>
10  add    $0x10,%rsp
11  fstpt  -0x20(%rbp)
12  mov    -0x20(%rbp),%rax
13  mov    -0x18(%rbp),%edx
14  mov    %rax,-0x20(%rbp)
15  mov    %edx,-0x18(%rbp)
16  fldt   -0x20(%rbp)
17  leaveq
18  retq
19
20 <ecall_mul>:
21  push   %rbp
22  mov    %rsp,%rbp
23  fldt   0x10(%rbp)
24  fldt   0x20(%rbp)
25  fmulp  %st,%st(1)
26  pop    %rbp
27  retq
```

Listing B.2: Compiled assembly of Listing B.1.

B.2 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 3.4. We note that for brevity, both Algorithm 1 and Algorithm 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
```

Appendix C

Additional Resources for Pandora

C.1 Pandora CLI and Report Generation (*G4*)

We designed Pandora with great care for usability (*G4*), through a well-documented command line interface (CLI) and detailed HTML reports. Figure C.1 shows an example of a human-readable, interactive HTML report from the `PTRSan` plugin discovering unconstrained pointer dereferences in the Linux selftest enclave (cf. Section 4.7.1). Figure C.2 shows a part of the interactive command line interface of Pandora, which is intended to be highly usable for both rapid prototyping of new plugins and for long, unattended exploration runs. Issues are reported on the command line during a run, but are also logged in a JSON file that can later be expanded into the fully-fledged HTML reports visible in Figure C.1. If the human analyst wishes, multiple breakpoints regarding the exploration and the plugins are readily available from the command line, *i.e.*, to interrupt execution on interesting events and switch into a Python shell. This allows to quickly implement and troubleshoot plugins. Lastly, several options of Pandora are, in addition to the CLI, exposed via configuration files, allowing to define long-lasting analysis setups that can be controlled by changing few program options.

C.2 Static Analysis of Enclave Runtimes

This appendix describes optional support we added to Pandora to load enclave binaries from selected runtimes with purely static analysis only, *i.e.*, without first requiring the SGX-TRACER dynamic memory extraction phase described in Section 4.5. The difficulty in adequately supporting arbitrary enclave runtimes in this way lies in parsing opaque enclave memory layout metadata from the binary and loading SGX-specific data structures into the symbolic execution memory after the executable and linkable format (ELF) file has been loaded. Although inherently fragile and version-specific, we show that it is in principle possible to implement such support entirely statically for three exemplary runtime loaders.

While we consider some of these static loaders to be mature and satisfying our truthful initial memory layout criterion (*G1b*), we note that this highly labor-intensive static-analysis approach is evidently not runtime-agnostic (vs. *G2*). Furthermore, even for the individual runtimes that are supported, the static-analysis approach remains inherently fragile, as new versions of these runtimes may completely break or change the way runtime-specific data structures are utilized in the enclave.[1] Thus, we use our novel SGX-TRACER enclave memory extractor approach as the default runtime-agnostic and truthful loader in Pandora, as also used in the evaluation of Section 4.7.

Linux Selftest Enclave First, the Linux selftest enclave [145] is a minimal, self-contained enclave that has a fixed memory layout, with the thread control structure (TCS) always being stored at the start of the enclave range. This makes it an ideal baseline runtime as no enclave initialization is necessary and all relevant addresses are statically known at compile time. The Linux selftest enclave serves as the foundation for Pandora's unit-test validation framework, discussed in Section 4.7.1.

Intel SGX SDK Second, the Intel SGX SDK [78] encodes all information for the loading process in an additional ELF metadata section. Based on manual analysis of the open-source code of the Intel SGX SDK enclave loader, we added full support in Pandora to decode this opaque blob and extract the expected locations of TCSs, stack and heap regions, and patches to initialize enclave global data structures. We implemented mature support to perform these steps

[1] Examples of such changes in the past were versions 2.4, 2.14, and 2.17 of the Intel SGX SDK when the internal `_global_data_t` C structure was modified which resulted in altered offsets for the address of the enclave base address, a crucial piece of information to properly resolve addresses inside the enclave.

in Intel SGX SDK version 2.18.1 and also validated backwards compatibility and added support for version-specific fields in versions 2.18 and 2.17.1.

SCONE Lastly, we show that, in principle, the static enclave loading approach is even feasible without access to source code by implementing an elementary (incomplete) static loader for the proprietary SCONE [16] runtime. Specifically, we manually reverse engineered the enclave layout and location of TCS data structures and thread-local memory using a debugger. Based on this partial layout, our static loader inserts the required data structures into the symbolic memory layout when loading the SCONE runtime binary ELF file.

C.3 Pandora Breakpoints

Table C.1 lists all enclave-aware breakpoints added by Pandora. To accommodate various investigation scenarios, all breakpoints can be triggered before and after the event happened, *i.e.*, to investigate an event both before or after it had an impact on a Pandora state. For example, Pandora memory read breakpoint, similarly to angr memory read breakpoints, can be triggered before the read has happened, exposing, among other, its address and size; or after the read has happened, additionally exposing its value.

Table C.1: List of breakpoints added by Pandora. Plugins can hook these new breakpoints, in addition to all legacy angr breakpoints, to investigate specific events during exploration. All events can be hooked before and after they are explored. Individual breakpoints may additionally expose specific arguments, e.g., symbolic memory addresses and sizes.

Breakpoint event	**Triggered by**	**Description**
`eenter`	Enclave (Re)entry	A state is prepared to (re)enter the enclave
`eexit`	SGX Instructions	An `EEXIT ENCLU` is executed
`untrusted_mem_read`	Enclave Memory	Reads that fully lie in untrusted memory
`trusted_mem_read`	Enclave Memory	Reads that fully lie in enclave memory
`inside_or_outside_mem_read`	Enclave Memory	Reads that may lie in either region
`untrusted_mem_write`	Enclave Memory	Writes that fully lie in untrusted memory
`trusted_mem_write`	Enclave Memory	Writes that fully lie in enclave memory
`inside_or_outside_mem_write`	Enclave Memory	Writes that may lie in either region

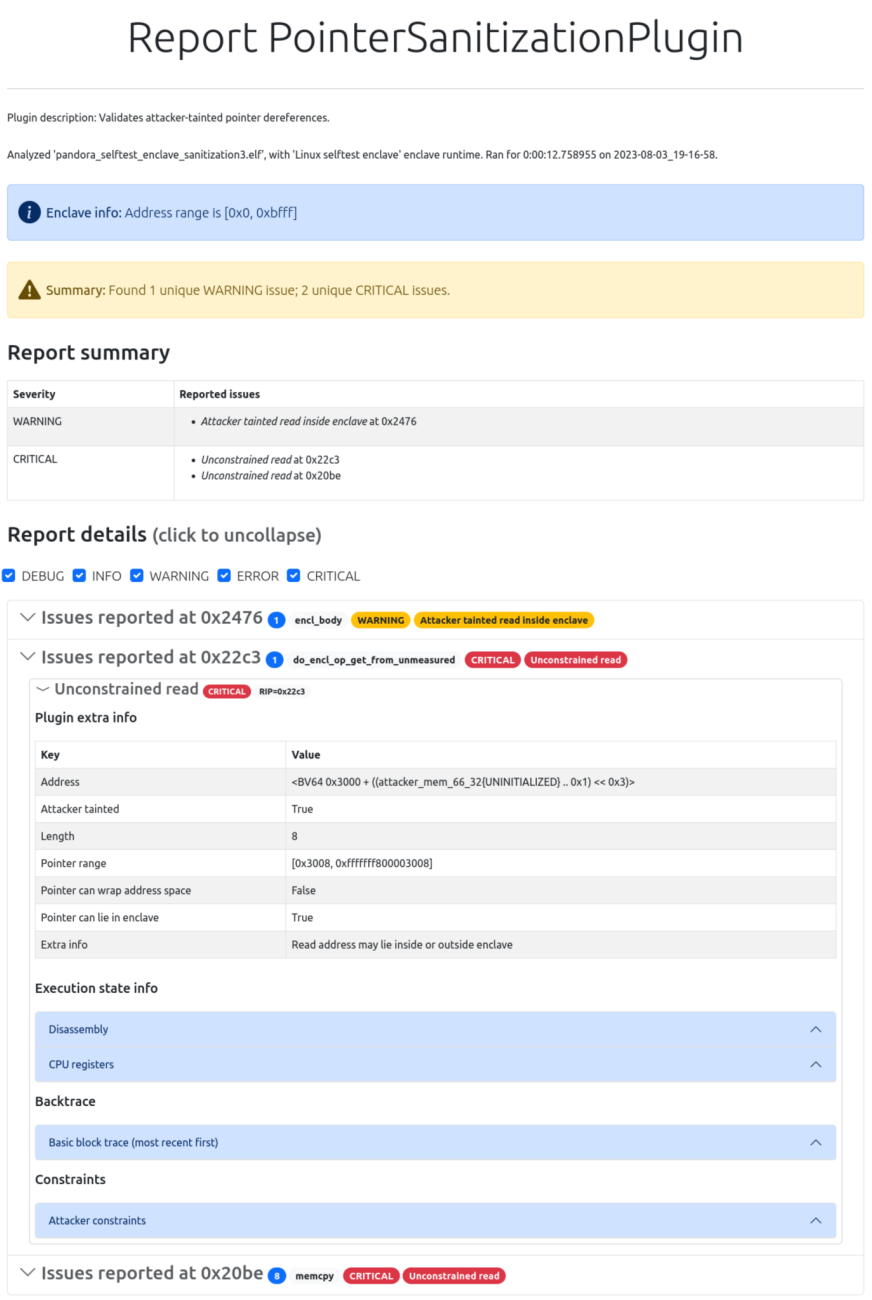
C.4 Vulnerability Details

Table C.2 provides a more detailed breakdown of the vulnerable code locations found by Pandora, as also summarized in Table 4.2 and discussed in Section 4.7.

Table C.2: Detailed evidence of Pandora finding and reproducing vulnerabilities both in production and research runtimes, where the "depth" column lists the number of basic blocks explored before the vulnerability (min–max); "L" indicates the location (Entry, Initialization, Application) of the vulnerability; and column "O" indicates whether the vulnerability could have been found by existing, state-of-the-art SGX symbolic-execution tools [12, 35].

Runtime	Version	Prod	Src	Plugin	L	Depth	Inst	Description	O
Newly found vulnerabilities in shielding runtimes (total 174 instances)									
EnclaveOS	3.28	✓	✗†	ABISan	E	8	1	MXCSR dependent timing	✗
EnclaveOS	3.28	✓	✗†	PTRSan	E	14–48	10	Compiler removed overflow check	✗
EnclaveOS	3.28	✓	✗†	PTRSan	I	15495–15521	5	`strlen` on unconstrained ptr (CVE-2023-38022)	✗
EnclaveOS	3.28	✓	✗†	#PICSan	I	14–100	33	Various SBDR issues (CVE-2023-38021)	✗
EnclaveOS	3.28	✓	✗†	CFSan	I	51	2	PIC jump before relocation	✗
GoTEE	014b35f	✗	✓	PTRSan	E/I	2–82	31	Various unconstrained pointers	✗
GoTEE	014b35f	✗	✓	#PICSan	E/I	2–82	18	Various SBDR/DRPW issues	✗
GoTEE	014b35f	✗	✓	CFSan	I	82	1	Unconstrained `RET` targets	✗
Gramine	1.4	✓	✓	ABISan	E	8	1	MXCSR dependent timing	✗
Intel SDK	2.15.1	✓	✓	PTRSan	I	29–30	2	Unconstrained pointer (CVE-2022-26509)	✗
Intel SDK	2.19	✓	✓	#PICSan	I	234	22	SBDR in enclave initialization	✗
↪ Occlum	0.29.4	✓	✓	#PICSan	I	17222	11	↪ SBDR inherited	✗
Linux selftest	5.18	✗	✓	ABISan	E	1	1	Unsanitized AC/DF, MXCSR, and FPU	✗
↪ Inclavare	0.6.2	✗	✓	ABISan	E	1	1	↪ Missing sanitization on entry	✗
Linux selftest	5.18	✗	✓	PTRSan	A	4–7	5	Various unconstrained pointers	✗
↪ Inclavare	0.6.2	✗	✓	PTRSan	A	8–539	2	Unconstrained src/dst addresses in `memcpy`	✗
Linux selftest	5.18	✗	✓	CFSan	A	5	1	PIC jump before relocation	✗
↪ Inclavare	0.6.2	✗	✓	CFSan	E	3	1	Unsigned jump target comparison in `ecall` array	✗
Open Enclave	0.19.0	✓	✓	ABISan	E	11	1	Unsanitized AC (regression) (CVE-2023-37479)	✗
Open Enclave	0.19.0	✓	✓	ABISan	E	11	1	MXCSR dependent timing	✗
Rust EDP	1.71	✓	✓	ABISan	E	7	1	MXCSR dependent timing	✗
SCONE	5.7.0	✓	✗	ABISan	E	3	1	Unsanitized FPU (CVE-2022-46487)	✗
SCONE	5.7.0	✓	✗	PTRSan	I	25–1827	10	Various pointer issues (CVE-2022-46486)	✗
SCONE	5.7.0	✓	✗	#PICSan	I	25–1827	11	Various SBDR/DRPW issues (CVE-2023-38023)	✗
Reproduced vulnerabilities in older versions (total 69 instances)									
GoTEE	014b35f	✗	✓	ABISan	E	3	1	Unsanitized FPU [5]	✗
Gramine	1.2	✓	✓	#PICSan	I	22–55	10	Various SBDR/DRPW issues	✗
Intel SDK	2.1.1	✓	✓	ABISan	E	3	1	Unsanitized DF/AC [157]; FPU [5]	✓
Intel SDK	2.13.3	✓	✓	#PICSan	I	207–6198	28	Various SBDR/DRPW issues	✗
Open Enclave	0.4.1	✓	✓	ABISan	E	4	1	Unsanitized DF [157]	✗
Open Enclave	0.4.1	✓	✓	PTRSan	I	402–1712	13	Unconstrained pointers [157]	✗
Open Enclave	0.4.1	✓	✓	#PICSan	I	442–1712	13	Various SBDR/DRPW issues	✗
Rust EDP	1.63	✓	✓	#PICSan	I	1041–1043	2	Various SBDR/DRPW issues	✗

Legend: † Not open source, but source code was made privately available; ↪ Based on above runtime.



Report PointerSanitizationPlugin

Plugin description: Validates attacker-tainted pointer dereferences.

Analyzed 'pandora_selftest_enclave_sanitization3.elf', with 'Linux selftest enclave' enclave runtime. Ran for 0:00:12.758955 on 2023-08-03_19-16-58.

Enclave info: Address range is [0x0, 0xbfff]

Summary: Found 1 unique WARNING issue; 2 unique CRITICAL issues.

Report summary

Severity	Reported issues
WARNING	• *Attacker tainted read inside enclave* at 0x2476
CRITICAL	• *Unconstrained read* at 0x22c3 • *Unconstrained read* at 0x20be

Report details (click to uncollapse)

DEBUG INFO WARNING ERROR CRITICAL

Issues reported at 0x2476 1 encl_body WARNING Attacker tainted read inside enclave

Issues reported at 0x22c3 1 do_encl_op_get_from_unmeasured CRITICAL Unconstrained read

Unconstrained read CRITICAL RIP=0x22c3

Plugin extra info

Key	Value
Address	<BV64 0x3000 + ((attacker_mem_66_32{UNINITIALIZED} .. 0x1) << 0x3)>
Attacker tainted	True
Length	8
Pointer range	[0x3008, 0xffffffff80003008]
Pointer can wrap address space	False
Pointer can lie in enclave	True
Extra info	Read address may lie inside or outside enclave

Execution state info

Disassembly

CPU registers

Backtrace

Basic block trace (most recent first)

Constraints

Attacker constraints

Issues reported at 0x20be 8 memcpy CRITICAL Unconstrained read

Figure C.1: Example of an HTML report generated by Pandora.

```
./pandora.py run --help
✅ Importing angr (this takes a second) 0:00:00

 Usage: pandora.py run [OPTIONS] FILE_PATH

 Shorthand for explore + report

╭─ Arguments ──────────────────────────────────────────────────────────────────────────────────────────────────────────╮
*    file_path      FILE  Path to the binary or log file to open [default: None] [required]                          
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────────────────────────────────────────────╮
--config-file     -c      FILE                                        Path to optional config file [default: None]
--log-level       -l      [trace|debug|info|warning|error|critical]   The log level for pandora [default: info]
--angr-log-level          [trace|debug|info|warning|error|critical]   The log level for angr [default: critical]
--help                                                                Show this message and exit.
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Report generation ──────────────────────────────────────────────────────────────────────────────────────────────────╮
--report-level  -L      [trace|debug|info|warning|error|critical]   The level for pandora reports. Set to debug to
                                                                    get all information.
                                                                    [default: info]
--report        -r      [html|log]                                  Define the format for all plugin reports.
                                                                    [default: html]
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Exploration options ────────────────────────────────────────────────────────────────────────────────────────────────╮
--num-steps       -n      INTEGER                                   Number of steps to execute in symbolic
                                                                    execution. 0 or negative allows to run to
                                                                    completion.
                                                                    [default: 100]
--plugins         -p      [default|all|abi|ptr|cf|dbg|aepic]        Define the plugins to activate, separated
                                                                    by a comma. Possible values for the plugin
                                                                    key are:
                                                                    ↪ default -- Shorthand for
                                                                    abi,ptr,cf,aepic
                                                                    ↪ all     -- Shorthand for all plugins
                                                                    ↪ abi     -- Validates CPU register
                                                                    sanitizations.
                                                                    ↪ ptr     -- Validates attacker-tainted
                                                                    pointer dereferences.
                                                                    ↪ cf      -- Detects attacker-controlled
                                                                    jump targets.
                                                                    ↪ dbg     -- Debug plugin.
                                                                    ↪ aepic   -- Validates MMIO buffer leaks
                                                                    when interacting with untrusted memory.
                                                                    [default: default]
--pandora-option          TEXT                                      Sets a specific advanced option via the
                                                                    format option=value. Default values shown
                                                                    below. Possible values for the option key
                                                                    are:
                                                                    ↪ PANDORA_ENCLAVE_MIXIN_ENABLE
                                                                    -- True
                                                                    ↪ PANDORA_EXPLORE_THREAD_COUNT
                                                                    -- 1
                                                                    ↪ PANDORA_EXPLORE_REENTRY_COUNT
                                                                    -- 0
                                                                    ↪ PANDORA_EXPLORE_DEPTH_FIRST
                                                                    -- False
                                                                    ↪ PANDORA_EXPLORE_USE_LOOP_SEER
                                                                    -- False
                                                                    ↪ PANDORA_EXPLORE_LOOP_SEER_BOUND
                                                                    -- 100
                                                                    ↪
                                                                    PANDORA_EXPLORE_ENABLE_SELFMODIFYING_CODE
                                                                    -- False
                                                                    ↪ PANDORA_REPORT_ONLY_UNIQUE
                                                                    -- False
                                                                    ↪ PANDORA_REPORT_OMIT_ATTACKER_CONSTRAINTS
                                                                    -- False
                                                                    [default: None]
--force-sdk       -s      [intel|linux-selftest|open-enclave|scone|du Define the sdk to use. Overrides auto
                          mp|auto]                                  detection if set to a specific SDK.
                                                                    [default: auto]
--action          -a      TEXT                                      Adds an action bound to a specific event
```

Figure C.2: Part of the command line interface of Pandora depicting helpful command options to the user.

Bibliography

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. "A survey on homomorphic encryption schemes: Theory and implementation". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–35.

[2] F. Alder, N Asokan, A. Kurnikov, A. Paverd, and M. Steiner. "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 (CCSW)*. ACM, 2019, pp. 185–199.

[3] F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck. "Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes". In: *In submission*. 2023.

[4] F. Alder, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg. "About Time: On the Challenges of Temporal Guarantees in Untrusted Environments". In: *6th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2023.

[5] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI poisoning attacks on Intel SGX". In: *Annual Computer Security Applications Conference (ACSAC)*. 2020, pp. 415–427.

[6] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg. "Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves". In: *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS'21)*. ACM, 2021, pp. 1357–1372.

[7] AliBaba. *Inclavare Containers: The Future of Cloud-Native Confidential Computing*. June 2022. URL: `https://www.alibabacloud.com/blog/inclavare-containers-the-future-of-cloud-native-confidential-computing_598992`.

[8] T. Alves and D. Felton. "TrustZone: Integrated hardware and software security". In: *ARM white paper* 3.4 (2004).

[9] AMD. “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More”. In: *White paper* (Jan. 2020).

[10] American National Standard (ANSI). “IEEE Standard for Binary Floating-Point Arithmetic”. In: *IEEE Std 754-1985* (1985).

[11] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. “Innovative technology for CPU based attestation and sealing”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[12] P. Antonino, W. A. Woloszyn, and A. W. Roscoe. “Guardian: Symbolic Validation of Orderliness in SGX Enclaves”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*. ACM, 2021, 111–123.

[13] ARM. *Arm Architecture Reference Manual Armv8*. ARM DDI: 0487G.a. 2023. URL: `https://developer.arm.com/documentation/ddi0487/gb/`.

[14] ARM. *FPSCR, the floating-point status and control register*. 2023. URL: `https://developer.arm.com/documentation/dui0068/b/Vector-Floating-point-Programming/VFP-system-registers/FPSCR--the-floating-point-status-and-control-register`.

[15] Arm Limited. *Arm CCA Security Model 1.0 DEN0096*. 2021. URL: `https://developer.arm.com/documentation/DEN0096/latest`.

[16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, et al. “SCONE: Secure Linux containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI 16)*. USENIX Association, 2016, pp. 689–703.

[17] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. “RIOT: An open source operating system for low-end embedded devices in the IoT”. In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4428–4440.

[18] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. “RIOT OS: Towards an OS for the Internet of Things”. In: *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE. Apr. 2013, pp. 79–80.

[19] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf. “CURE: A Security Architecture with CUstomizable and Resilient Enclaves”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[20] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), 50:1–50:39.

[21] A. Baumann, M. Peinado, and G. Hunt. "Shielding applications from an untrusted cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI 14)*. USENIX Association. 2014, pp. 267–283.

[22] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1213–1227.

[23] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. "Timing analysis of a protected operating system kernel". In: *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE. 2011, pp. 339–348.

[24] M. Bognar, J. Van Bulck, and F. Piessens. "Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures". In: *43rd IEEE Symposium on Security and Privacy (S&P'22)*. IEEE. 2022, pp. 1638–1655.

[25] M. Bognar, H. Winderix, J. Van Bulck, and F. Piessens. "MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling". In: *8th IEEE European Symposium on Security and Privacy (EuroS&P'23)*. July 2023.

[26] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz. "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture". In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.

[27] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. "DR. SGX: automated and adjustable side-channel protection for SGX using data location randomization". In: *Annual Computer Security Applications Conference (ACSAC)*. 2019, pp. 788–800.

[28] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. "TyTAN: Tiny trust anchor for tiny devices". In: *Design Automation Conference (DAC 2015)*. IEEE. 2015, pp. 1–6.

[29] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. "Provably secure isolation for interruptible enclaved execution on small microprocessors". In: *33rd IEEE Computer Security Foundations Symposium (CSF'20)*. June 2020.

[30] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe. "Constant-time foundations for the new spectre era". In: *PLDI*. ACM, 2020, pp. 913–926.

[31] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. “Unleashing Mayhem on Binary Code”. In: *33rd IEEE Symposium on Security and Privacy (S&P’12)*. IEEE. 2012, pp. 380–394.

[32] S. Checkoway and H. Shacham. “Iago attacks: Why the system call API is a bad untrusted RPC interface”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 2013, pp. 253–264. ISBN: 978-1-4503-1870-9.

[33] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. “SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution”. In: *4th IEEE European Symposium on Security and Privacy (Euro S&P’19)*. IEEE, 2019.

[34] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. “Linux kernel vulnerabilities: State-of-the-art defenses and open problems”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM. 2011, 5:1–5:5.

[35] T. Cloosters, M. Rodler, and L. Davi. “TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 841–858.

[36] T. Cloosters, J. Willbold, T. Holz, and L. Davi. “SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing”. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022, pp. 3147–3164.

[37] Confidential Computing Consortium. *A Technical Analysis of Confidential Computing v1.3*. 2022.

[38] Confidential Computing Consortium. *Common Terminology for Confidential Computing*. 2022.

[39] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein. “AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Aug. 2023.

[40] V. Costan and S. Devadas. “Intel SGX explained.” In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[41] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai. “SmashEx: Smashing SGX enclaves using exceptions”. In: *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS’21)*. ACM, 2021, 779–793.

[42] R. Cui, L. Zhao, and D. Lie. "Emilia: Catching Iago in Legacy Code." In: *28th Annual Network and Distributed System Security Symposium (NDSS'21)*. The Internet Society, 2021.

[43] L. Daniel, S. Bardin, and T. Rezk. "Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE". In: *28th Annual Network and Distributed System Security Symposium (NDSS'21)*. The Internet Society, 2021.

[44] F. Dewald, H. Mantel, and A. Weber. "AVR Processors as a Platform for Language-Based Security". In: *European Symposium on Research in Computer Security (ESORICS)*. 2017, pp. 427–445.

[45] S. Dinesh, N. Burow, D. Xu, and M. Payer. "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization". In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 2020.

[46] D. Dua and C. Graff. *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences. 2017. URL: http://archive.ics.uci.edu/ml.

[47] G. Duan, Y. Fu, B. Zhang, P. Deng, J. Sun, H. Chen, and Z. Chen. "TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation". In: *Future Generation Computer Systems* (2023).

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

[49] Edgeless Systems. *Edgeless RT*. 2023. URL: https://github.com/edgelesssys/edgelessrt.

[50] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. "SMART: Secure and minimal architecture for (establishing a dynamic) root of trust." In: *19th Annual Network and Distributed System Security Symposium (NDSS'12)*. Vol. 12. The Internet Society, 2012, pp. 1–15.

[51] Enarx Project. *Enarx: WebAssembly + Confidential Computing*. 2023. URL: https://enarx.dev/.

[52] A. Fog. *Calling conventions for different C++ compilers and operating systems*. 2023. URL: http://www.agner.org/optimize/calling_conventions.pdf.

[53] Fortanix. *Fortanix Enclave Development Platform – Rust EDP*. 2023. URL: https://edp.fortanix.com/.

[54] Fortanix. *Fortanix Runtime Encryption Platform and EnclaveOS*. 2023. URL: https://www.fortanix.com/platform/runtime-encryption.

[55] M. Fredrikson, S. Jha, and T. Ristenpart. "Model inversion attacks that exploit confidence information and basic countermeasures". In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*. ACM, 2015, pp. 1322–1333.

[56] Free Software Foundation. *GCC, the GNU Compiler Collection*. 2023. URL: `https://gcc.gnu.org/`.

[57] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[58] A. Ghosn, J. R. Larus, and E. Bugnion. "Secured routines: language-based construction of trusted execution environments". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019, pp. 571–586.

[59] L. Giner, A. Kogler, C. Canella, M. Schwarz, and D. Gruss. "Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX". In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022, pp. 3111–3127.

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

[61] O. Girard. *openMSP430 – a synthesizable 16-bit microcontroller core written in Verilog*. `https://opencores.org/project,openmsp430`. 2009.

[62] P. Godefroid, M. Y. Levin, and D. A. Molnar. "SAGE: whitebox fuzzing for security testing". In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[63] T. Goodspeed. "Practical attacks against the MSP430 BSL". In: *Twenty-Fifth Chaos Communications Congress*. 2008.

[64] Google. *Asylo: An open and flexible framework for enclave applications*. 2018. URL: `https://asylo.dev/`.

[65] E. Goossens. *Exploring schedulability on TEEs with availability guarantees*. Master Thesis, KU Leuven. 2022.

[66] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *12th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI 16)*. USENIX Association, 2016, pp. 653–669.

[67] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. "Spectector: Principled Detection of Speculative Information Flows". In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 2020, pp. 1–19.

[68] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. "Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution". In: *International Symposium on Engineering Secure Software and Systems*. ESSoS '18. Springer. June 2018, pp. 44–60.

[69] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd. "Mitigating branch-shadowing attacks on Intel SGX using control flow randomization". In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 2018, pp. 42–47.

[70] IBM. *Data-in-use protection on IBM cloud*. 2017. URL: `https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-ibm-intel-fortanix-partner-keep-enterprises-secure-core`.

[71] Intel. *Architecture Specification: Intel® Trust Domain Extensions (Intel® TDX) Module*. Specification. Version 344425-004US. June 2022.

[72] Intel. *Intel Product Specifications*. 2023. URL: `https://ark.intel.com/`.

[73] Intel. *MXCSR Configuration Dependent Timing*. 2023.

[74] Intel. *Processor MMIO Stale Data Vulnerabilities*. June 2022.

[75] Intel. *Stale Data Read from Legacy xAPIC*. Aug. 2022.

[76] Intel Corporation. *Deep dive: Load value injection*. 2020.

[77] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual – Combined volumes*. Reference no. 325462-062US. 2023.

[78] Intel Corporation. *Intel Software Guard Extensions – Get Started with the SDK*. 2023. URL: `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/get-started.html`.

[79] A. Karpathy. *Convnetjs: Deep learning in your browser*. 2014. URL: `http://cs.stanford.edu/people/karpathy/convnetjs`.

[80] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei. "COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 971–985.

[81] J. C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[82] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the 22nd Symposium on Operating Systems Principles*. ACM. 2009, pp. 207–220.

[83] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. “TrustLite: A security architecture for tiny embedded devices”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, 10:1–10:14.

[84] A. Kogler, D. Gruss, and M. Schwarz. “Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks”. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022, pp. 4147–4164.

[85] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. “Efficient state merging in symbolic execution”. In: *PLDI*. ACM, 2012, pp. 193–204.

[86] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. “Keystone: An open framework for architecting trusted execution environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[87] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. “Hacking in darkness: Return-oriented programming against secure enclaves”. In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 523–539.

[88] S. Lee and T. Kim. “Leaking uninitialized secure enclave memory via structure padding”. In: *arXiv preprint arXiv:1710.09061* (2017).

[89] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. “Inferring fine-grained control flow inside SGX enclaves with branch shadowing”. In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 557–574.

[90] Y. Li, Z. Su, L. Wang, and X. Li. “Steering Symbolic Execution to Less Traveled Paths”. In: *SIGPLAN Not.* 48.10 (Oct. 2013), 19–32. ISSN: 0362-1340.

[91] J. Liu, Y. Qin, and D. Feng. “SeRoT: A Secure Runtime System on Trusted Execution Environments”. In: *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE. 2020, pp. 30–37.

[92] LLVM. *The LLVM compiler infrastructure*. 2023. URL: `https://llvm.org`.

[93] H. Lu, D. L. Kreitzer, M. Girkar, and Z. Ansari. “System V application binary interface”. In: *Intel386 Architecture Processor Supplement, Version 1.1* (Dec. 2015).

[94] A. Lyons and G. Heiser. “Mixed-criticality support in a high-assurance, general-purpose microkernel”. In: *Workshop on Mixed Criticality Systems*. 2014, pp. 9–14.

[95] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna. "BOOMERANG: Exploiting the semantic gap in trusted execution environments". In: *24th Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, 2017.

[96] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation". In: *IEEE Transactions on Computers* 67.3 (2018), pp. 361–374.

[97] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. "Enabling trusted scheduling in embedded systems". In: *Annual Computer Security Applications Conference (ACSAC)*. 2012.

[98] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[99] Microsoft. *Introducing Azure confidential computing*. 2017. URL: `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`.

[100] Microsoft. *Microsoft Visual C++*. 2023. URL: `https://docs.microsoft.com/en-us/cpp/`.

[101] Microsoft. *Open Enclave SDK*. 2023. URL: `https://openenclave.io/`.

[102] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 469–486.

[103] J. T. Mühlberg, S. Cleemput, A. M. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. "Implementation of a high assurance smart meter using protected module architectures". In: *10th WISTP International Conference on Information Security Theory and Practice (WISTP'16)*. Vol. 9895. LNCS. Springer, 2016, pp. 53–69.

[104] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. "Plundervolt: Software-based fault injection attacks against Intel SGX". In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE. May 2020.

[105] A. Nilsson, P. N. Bideh, and J. Brorsson. "A survey of published attacks on Intel SGX". In: *arXiv preprint arXiv:2006.13598* (2020).

[106] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. "Sancus 2.0: A low-cost security architecture for IoT devices". In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), 7:1–7:33.

[107] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base". In: *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, 2013, pp. 479–494.

[108] J. Noorman, J. T. Mühlberg, and F. Piessens. "Authentic execution of distributed event-driven applications with a small TCB". In: *STM '17*. Vol. 10547. LNCS. Heidelberg: Springer, 2017, pp. 55–71.

[109] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation". In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 1429–1446.

[110] NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*. Specification. Version V1.04. 2023.

[111] OP TEE c/o Linaro. *Open Portable Trusted Execution Environment*. 2023. URL: `https://www.op-tee.org/`.

[112] M. Orenbach, B. Raveh, A. Berkenstadt, Y. Michalevsky, S. Itzhaky, and M. Silberstein. "Securing Access to Untrusted Services From TEEs with GateKeeper". In: *arXiv preprint arXiv:2211.07185* (2022).

[113] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. "Secure compilation to protected module architectures". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (2015).

[114] J. Pennekamp, F. Alder, R. Matzutt, J. T. Mühlberg, F. Piessens, and K. Wehrle. "Secure End-to-End Sensing in Supply Chains". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–6.

[116] S. Pinto and N. Santos. "Demystifying ARM TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys (CSUR)* 51.6 (2019), p. 130.

[117] S. Pouyanrad, J. T. Mühlberg, and W. Joosen. "SCF MSP: static detection of side channels in MSP430 programs". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*. 2020, pp. 1–10.

[118] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. "SGX-LKL: Securing the Host OS Interface for Trusted Execution". In: *arXiv preprint arXiv:1908.11143* (2019).

[119] B. D. Roeck. *Unified ABI shielding for Intel SGX runtimes*. Master Thesis, KU Leuven. 2021.

[120] J. Sakkinen and N. McCallum. *selftests/x86: Add a selftest for SGX*. Mar. 2020. URL: `https://lkml.kernel.org/lkml/04362c0cf66bf66e8f7c25a531830b9f294d2d09.camel@linux.intel.com/`.

[121] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. "RIDL: Rogue in-flight data load". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE. May 2019.

[122] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez. "Sok: Hardware-supported trusted execution environments". In: *arXiv preprint arXiv:2205.12742* (2022).

[123] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, Nov. 2019.

[124] Scontain. *SCONE – A Secure Container Environment*. 2023. URL: `https://scontain.com/`.

[125] G. Scopelliti, C. Baumann, F. Alder, E. Truyen, and J. T. Mühlberg. "Efficient and Timely Revocation of V2X Credentials". In: *31st Annual Network and Distributed System Security Symposium (NDSS'24)*. The Internet Society, 2024.

[127] G. Scopelliti, S. Pouyanrad, J. Noorman, F. Alder, C. Baumann, F. Piessens, and J. T. Mühlberg. "End-to-End Security for Distributed Event-Driven Enclave Applications on Heterogeneous TEEs". In: *ACM Transactions on Privacy and Security (TOPS)* (Apr. 2023).

[129] T. Sewell, F. Kam, and G. Heiser. "High-assurance timing analysis for a high-assurance real-time operating system". In: *Real-Time Systems* 53.5 (2017), pp. 812–853.

[130] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan. "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 955–970.

[131] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs". In: *24th Annual Network and Distributed System Security Symposium (NDSS'17)*. San Diego, CA: The Internet Society, Feb. 2017.

[132] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *37th IEEE Symposium on Security and Privacy (S&P'16)*. IEEE. 2016.

[133] P. N. Smart. *Cryptography made simple*. Springer, 2016.

[134] W. Snyder. *Verilator, the fastest Verilog/SystemVerilog simulator*. `https://www.veripool.org/wiki/verilator`. 2023.

[135] R. Strackx, F. Piessens, and B. Preneel. "Efficient isolation of trusted subsystems in embedded systems". In: *Security and Privacy in Communication Networks*. Springer, 2010, pp. 344–361.

[143] The Apache Software Foundation. *Apache Teaclave (Incubating)*. 2023. URL: `https://teaclave.incubator.apache.org/`.

[144] The Gramine Workgroup. *Gramine – A Library OS for Unmodified Applications*. 2023. URL: `https://gramineproject.io/`.

[145] L. Torvalds. *Linux operating system*. 2023. URL: `kernel.org`.

[146] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing machine learning models via prediction apis". In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 601–618.

[147] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge". In: *2nd IEEE European Symposium on Security and Privacy (Euro S&P'17)*. IEEE, 2017.

[148] Trusted Computing Group. *Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.59*. 2019.

[149] C.-C. Tsai, D. E. Porter, and M. Vij. "Graphene-SGX: A practical library OS for unmodified applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2017.

[150] S. Vaarala. "Duktape embeddable Javascript engine". In: *URL `https://duktape.org/`* (2023).

[151] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Aug. 2018.

[152] J. Van Bulck. *Secure Resource Sharing for Embedded Protected Module Architectures*. Master Thesis, KU Leuven. 2015.

[153] J. Van Bulck, F. Alder, and F. Piessens. "A Case for Unified ABI Shielding in Intel SGX Runtimes". In: *5th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2022.

[154] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. "LVI: Hijacking transient execution through microarchitectural load value injection". In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE. May 2020.

[155] J. Van Bulck, J. T. Mühlberg, and F. Piessens. "VulCAN: Efficient component authentication and software isolation for automotive control networks". In: *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2017.

[156] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. "Towards availability and real-time guarantees for protected module architectures". In: *Companion Proceedings of the 15th International Conference on Modularity (MASS'16)*. ACM, 2016, pp. 146–151.

[157] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, Nov. 2019, pp. 1741–1758.

[158] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic". In: *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*. ACM, Oct. 2018.

[159] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A practical attack framework for precise enclave execution control". In: *2nd Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2017.

[161] T. Van Eyck, H. Trimech, M. Salehi, T.-L. Ta, S. Michiels, D. Hughes, and H. Janjua. URL: `https://distrinet-tacos.github.io/documentation/`.

[162] S. Vanderhallen, J. Van Bulck, F. Piessens, and J. T. Mühlberg. "Robust authentication for automotive control networks through covert channels". In: *Computer Networks* 193 (2021), p. 108079.

[163] B. Wang and N. Z. Gong. “Stealing hyperparameters in machine learning”. In: *39th IEEE Symposium on Security and Privacy (S&P’18)*. IEEE. 2018, pp. 36–52.

[164] F. Wang and Y. Shoshitaishvili. “Angr-the next generation of binary analysis”. In: *2017 IEEE Cybersecurity Development (SecDev)*. IEEE. 2017, pp. 8–9.

[165] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin. “Towards Memory Safe Enclave Programming with Rust-SGX”. In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS’19)*. ACM, Nov. 2019, pp. 2333–2350.

[166] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang. “RT-TEE: Real-time system availability for cyber-physical systems using ARM TrustZone”. In: *43rd IEEE Symposium on Security and Privacy (S&P’22)*. IEEE. 2022, pp. 352–369.

[167] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*. RISC-V Foundation. Dec. 2023. URL: `https://riscv.org/technical/specifications/`.

[168] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi. “TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V”. In: *26th Annual Network and Distributed System Security Symposium (NDSS’19)*. The Internet Society, Feb. 2019.

[169] T. Werquin, M. Hubrechtsen, A. Thangarajan, F. Piessens, and J. T. Mühlberg. “Automated fuzzing of automotive control units”. In: *2019 International Workshop on Secure Internet of Things (SIOT)*. IEEE. 2019, pp. 1–8.

[170] H. Winderix, J. T. Mühlberg, and F. Piessens. “Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks”. In: *43rd IEEE European Symposium on Security and Privacy (Euro S&P’21)*. IEEE, 2021.

[171] L. V. Winkle. *Genann Neural Networks Library*. 2023. URL: `https://github.com/codeplea/genann`.

[172] J. Woodruff, R. N. Watson, D. Chisnall, S. W Moore, J. Anderson, B. Davis, B. Laurie, P. G Neumann, R. Norton, and M. Roe. “The CHERI capability model: Revisiting RISC in an age of risk”. In: *ISCA 2014*. 2014, pp. 457–468.

[173] Y. Xu, W. Cui, and M. Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *36th IEEE Symposium on Security and Privacy (S&P’15)*. IEEE. 2015, pp. 640–656.

[174] M. Yan, C. W. Fletcher, and J. Torrellas. “Cache telepathy: Leveraging shared resource attacks to learn DNN architectures”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2003–2020.

[175] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. Butler, and D. J. Tian. “ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs”. In: *IEEE Transactions on Dependable and Secure Computing* (2022).

Curriculum Vitae

Fritz Alder was born on 29th of May, 1992 in Lübbecke, Germany. From 2011 to 2015, he studied computer science at RWTH Aachen University in Germany. After receiving his bachelor's degree, he continued his studies at TU Darmstadt, Germany, in the master's program in IT Security. From 2016 to 2017, he spent an exchange year in Finland at Aalto University, Helsinki, under an Erasmus+ student grant and the Finland Scholarship by the Max Müller and Delphine Müller-Alewyn Foundation. In 2018 he stayed at Aalto University to work at the Secure Systems Group headed by Professor Asokan, under the supervision of Andrew Paverd. In early 2019 he completed his master's thesis under the title "TEE^2 – Combining Trusted Hardware to Enhance the Security of TEEs". He then joined the DistriNet research group in September 2019 under a grant from the research association Flanders (FWO) to pursue a Ph.D. degree. From May to July 2021, he was an intern at Microsoft Research in Cambridge, United Kingdom, working on formalizing changes to the Raft consensus algorithm made for the Microsoft Confidential Computing Framework (CCF).

List of Publications

Peer-reviewed international conference papers

- G. Scopelliti, C. Baumann, **F. Alder**, E. Truyen, and J. T. Mühlberg. "Efficient and Timely Revocation of V2X Credentials". In: *31st Annual Network and Distributed System Security Symposium (NDSS'24)*. The Internet Society, 2024
- **F. Alder**, J. Van Bulck, F. Piessens, and J. T. Mühlberg. "Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves". In: *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS'21)*. ACM, 2021, pp. 1357–1372
- **F. Alder**, J. Van Bulck, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI poisoning attacks on Intel SGX". in: *Annual Computer Security Applications Conference (ACSAC)*. 2020, pp. 415–427
- **F. Alder**, A. Kurnikov, A. Paverd, and N. Asokan. "Migrating SGX Enclaves with Persistent State". In: *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN'18)*. 2018, pp. 195–206

Peer-reviewed international journal articles

- G. Scopelliti, S. Pouyanrad, J. Noorman, **F. Alder**, C. Baumann, F. Piessens, and J. T. Mühlberg. "End-to-End Security for Distributed Event-Driven Enclave Applications on Heterogeneous TEEs". In: *ACM Transactions on Privacy and Security (TOPS)* (Apr. 2023)

- **F. Alder**, J. Van Bulck, J. Spielman, D. Oswald, and F. Piessens. "Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments". In: *Digital Threats* 3.2 (Feb. 2022)

Peer-reviewed international workshop papers

- **F. Alder**, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg. "About Time: On the Challenges of Temporal Guarantees in Untrusted Environments". In: *6th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2023
- J. Van Bulck, **F. Alder**, and F. Piessens. "A Case for Unified ABI Shielding in Intel SGX Runtimes". In: *5th Workshop on System Software for Trusted Execution (SysTEX Workshop)*. 2022
- J. Pennekamp, **F. Alder**, R. Matzutt, J. T. Mühlberg, F. Piessens, and K. Wehrle. "Secure End-to-End Sensing in Supply Chains". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–6
- **F. Alder**, N Asokan, A. Kurnikov, A. Paverd, and M. Steiner. "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 (CCSW)*. ACM, 2019, pp. 185–199

Papers in submission

- **F. Alder**, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck. "Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes". In: *In submission*. 2023

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET, SYSTEM SECURITY
Celestijnenlaan 200A box 2402
B-3001 Leuven