KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Validating Sancus Enclaves using Symbolic Execution

Gert-Jan Goossens

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige Software

Promotoren:
Prof. dr. ir. F. Piessens
Dr. ir. J. Van Bulck

Evaluator:
Dr. V. Rimmer

Begeleider:
Dr. ir. F. Alder

Academiejaar 2023 – 2024

Preface

First of all I want to thank the reader for taking the time and effort to read this thesis. I would like to thank my supervisors prof. dr. ir. Frank Piessens and dr. ir. Jo Van Bulck to allow me to work on this topic.

I especially want to show my gratitude towards dr. ir. Jo Van Bulck and dr. ir. Fritz Alder for all the insights and support they gave me throughout the last year. I learned a lot from them, and without them this work would not have been possible. Especially their genuine interest and excitement in my research provided me with the right motivation to bring this work to a successful conclusion.

This thesis is the culmination point of many years of hard work. Many thanks to my family and friends for their moral support are therefore in its place. I want to give a special thanks to my parents for their unconditional support throughout these years.

Gert-Jan Goossens

Contents

Preface i

Abstract iv

Samenvatting v

Listings vi

List of Figures and Tables vii

List of Abbreviations and Symbols ix

1 Introduction 1
- 1.1 Trusted Execution Environments . . . 1
- 1.2 Contributions . . . 2
- 1.3 Outline . . . 3

2 Background 5
- 2.1 Trusted Execution Environments . . . 5
- 2.2 Sancus . . . 7
- 2.3 Enclave Vulnerabilities . . . 9
- 2.4 Symbolic Execution . . . 13
- 2.5 Pandora . . . 15
- 2.6 IPE . . . 19
- 2.7 Conclusion . . . 19

3 Implementation 21
- 3.1 State of the MSP430 angr Backend . . . 21
- 3.2 Intel SGX vs. Sancus . . . 23
- 3.3 Symbolic Execution Engine Development . . . 24
- 3.4 Plugins . . . 28
- 3.5 Crashed States . . . 31
- 3.6 Conclusion . . . 34

4 Evaluation 35
- 4.1 Unit Test Framework . . . 35
- 4.2 Validating Compiler Stubs . . . 40
- 4.3 Reproducing 'Tale of Two Worlds' Vulnerabilities . . . 43
- 4.4 Conclusion . . . 46

5 Conclusion 47

5.1 Limitations & Challenges 47
5.2 Future Work 48
5.3 Contributions 49

A Minimal Sancus Example **53**

B The Sancus Entry Stub **55**

C An Elaborate Overview of the Unit Test Framework **59**

D Generated Reports for the Minimal Sancus Enclave **63**

E Generated Reports for Sancus Stubs v2.0.0 **69**

Bibliography **75**

Abstract

The need for security of data at rest and data in transit has long been acknowledged. However data in use, while it is processed in the CPU, has not nearly gotten as much attention. Especially since the rise of cloud computing and remote sensing networks, the importance of protecting data in use cannot be underestimated. Over the last decade, trusted execution environments (TEE) posed a solution to this problem by providing developers with a strong hardware enforced isolation mechanism. This mechanism is used to obtain an isolation in the software developed on top, and can thereby provide strong privacy and security guarantees to developers. Additionally, this software is often assumed to preserve the hardware provided guarantees. Research repeatedly pointed out that these assumptions do not necessarily hold. Since the use cases for TEEs are continuously evolving, the associated software will continuously be extended and developed. Therefore, the research for vulnerabilities in this evolving landscape is and will remain an ongoing process. This motivates the need for automated vulnerability discovery techniques that can relieve the human analyst from time-consuming and error-prone security auditing. Particularly, such automated tools, based on powerful symbolic-execution techniques, have already proven excellent in providing a thorough analysis on the diverse ecosystem for the popular high-end Intel SGX TEE. However, low-end architectures such as Sancus, a research TEE design for the MSP430 architecture, have so far been neglected by automated vulnerability discovery tools. This thesis bridges the gap by porting Pandora, a tool developed for automated discovery of control-flow and pointer vulnerabilities, to Sancus.

The functionalities of the novel MSP430-Pandora port are validated through an extensive unit test framework comprising of 36 crafted Sancus assembly programs. Furthermore, Pandora also proved its worth by autonomously rediscovering previously published vulnerabilities found in previous versions of the Sancus runtime and compiler infrastructure. In conclusion, this work provides a foundation for automated validation of TEEs developed on the MSP430 architecture. Looking forward, we discuss possible extensions such as validating interruptability and availability guarantees on Sancus, as well as automated validation of alternative MSP430 TEEs such as Texas Instrument's commercial Intellectual Property Encapsulation architecture.

Samenvatting

De noodzaak voor de beveiliging van gegevens in opslag en gegevens in transit wordt al lang erkend. Echter, gegevens in gebruik, terwijl deze in de processor worden verwerkt, hebben niet zoveel aandacht gekregen. Vooral sinds de opkomst van cloud computing en netwerken voor remote sensing, kan het belang van het beschermen van gegevens in gebruik niet worden onderschat. In het afgelopen decennium hebben trusted execution environments (TEE) een oplossing voor dit probleem geboden door ontwikkelaars een sterk hardware-afgedwongen isolatiemechanisme te bieden. Dit mechanisme wordt gebruikt om een afscherming te verkrijgen in de software die erop wordt ontwikkeld, en kan daarmee sterke privacy- en beveiligingsgaranties aan ontwikkelaars bieden. Bovendien wordt aangenomen dat deze software de hardware-geboden garanties behoudt. Onderzoek heeft herhaaldelijk aangetoond dat deze aannames niet noodzakelijkerwijs standhouden. Aangezien de gebruiksscenario's voor TEEs voortdurend evolueren, zal de bijbehorende software voortdurend worden uitgebreid en ontwikkeld. Daarom blijft het onderzoek naar kwetsbaarheden in dit evoluerende landschap een doorlopend proces. Dit motiveert de behoefte aan geautomatiseerde technieken voor kwetsbaarheidsdetectie die de menselijke analist kunnen ontlasten van tijdrovende en foutgevoelige beveiligingsaudits. In het bijzonder hebben dergelijke geautomatiseerde tools, gebaseerd op krachtige technieken zoals symbolische uitvoering, al bewezen uitstekend te zijn in het bieden van een grondige analyse van het diverse ecosysteem voor de populaire high-end Intel SGX TEE. Echter, low-end architecturen zoals Sancus, een onderzoeks-TEE-ontwerp voor de MSP430-architectuur, zijn tot nu toe verwaarloosd door geautomatiseerde kwetsbaarheidsdetectietools. Deze scriptie overbrugt deze kloof door Pandora, een programma ontwikkeld voor geautomatiseerde detectie van control-flow en pointer-kwetsbaarheden, naar Sancus te brengen.

De functionaliteiten van het nieuwe MSP430-Pandora programma worden gevalideerd door middel van een uitgebreid unit test-framework bestaande uit 36 zorgvuldig samengestelde Sancus-assemblyprogramma's. Bovendien bewees Pandora zijn waarde door zelfstandig eerder gepubliceerde kwetsbaarheden, die werden gevonden in eerdere versies van de Sancus runtime en compiler infrastructuur, opnieuw te ontdekken. Om samen te vatten biedt dit werk een basis voor geautomatiseerde validatie van TEEs ontwikkeld op de MSP430-architectuur. Vooruitkijkend bespreken we mogelijke uitbreidingen zoals het valideren van interruptibiliteit en beschikbaarheidsgaranties op Sancus, evenals geautomatiseerde validatie van alternatieve MSP430-TEEs zoals de commerciële Intellectual Property Encapsulation-architectuur van Texas Instruments.

Listings

3.1 Basic block crosses enclave boundary 30
3.2 Basic block ends at enclave boundary 30
3.3 Sancus entry stub error routine 32
4.1 Enclave jumps to attacker address 37
4.2 Enclave jumps to attacker address inside enclave 37
4.3 Enclave writes to non-tainted address outside enclave 38
4.4 Enclave reads attacker address inside enclave 38
4.5 Elaborate test enclave 39
4.6 Small enclave for stub runtime tests 41
4.7 Authentic execution vulnerability 44
4.8 Soteria loader enclave vulnerability 44
4.9 Vulnerable `sancus_is_outside_sm` function 45
A.1 A minimal Sancus enclave program 53
B.1 Sancus entry stub 55

List of Figures and Tables

List of Figures

2.1 Internal layout of a Sancus node 8
2.2 Overview of TEE vulnerabilities 10
2.3 Confused deputy attack 11
2.4 Incorrect buffer range checks resulting in buffer-enclave overlap. 12
2.5 Address space wrapping resulting from unsafe arithmetics 13
2.6 Symbolic execution 14
2.7 Overview of the Pandora architecture 17
2.8 Example control-flow sanitization report 18

3.1 Structure of the adapted Pandora codebase 25

4.1 Growth of enclave runtime stubs troughout different versions 41

D.1 The PTRSan report for the *minimal-sancus-example* enclave. 64
D.2 The PTRSan report for the *minimal-sancus-example* enclave. 65
D.3 The PTRSan report for the *minimal-sancus-example* enclave. 66
D.4 The PTRSan report for the *minimal-sancus-example* enclave. 67

E.1 The CFSan report for the Sancus stubs v2.0.0. 70
E.2 The CFSan report for the Sancus stubs v2.0.0. 71
E.3 The CFSan report for the Sancus stubs v2.0.0. 72
E.4 The PTRSan report for the Sancus stubs v2.0.0. 73
E.5 The PTRSan report for the Sancus stubs v2.0.0. 74

List of Tables

3.1 Changed lines of code 26
3.2 Sancus hooks 27

4.1 Issues reported in different Sancus stub versions 42

C.1 Unit test framework CFSan tests 60

C.2 Unit test framework PTRSan tests 60
C.3 C enclave tests overview 61

List of Abbreviations and Symbols

Abbreviations

TEE	Trusted Execution Environment
IoT	Internet of Things
MAL	Memory Access Logic
ISA	Instruction Set Architecture
ABI	Application Binary Interface
API	Application Programming Interface
ocall	Outgoing Call
ecall	Enclave Call
CFSan	Control Flow Sanitization plugin
PTRSan	Pointer Sanitization plugin
ABISan	ABI Sanitization plugin
AEPICSan	AEPIC Sanitization plugin
PoC	Proof-of-concept
IR	Intermediate representation
SSA	Secure State Area

Chapter 1

Introduction

The need for security of data in transit and data at rest has long been acknowledged. Especially in today's world where cloud-computing and Internet-of-Things (IoT) devices become omnipresent, enforcing isolation between different programs on these devices is crucial. However, security of data in use, while it is being processed on the CPU itself, has only received attention in the past decade. Trusted execution environments (TEE) provide a solution to this problem. TEEs provide developers with a hardware enforced isolation of protection domains. With this hardware adaptation, a separation between programs can be obtained in software. While this can provide strong security guarantees, it expects the software developed on top of this hardware to be bug free. This is however not a trivial task. This work will provide an answer to the question whether Pandora [5], a tool developed for automated vulnerability discovery, allows flexible extensibility towards new TEE architectures. Moreover it autonomously rediscovers vulnerabilities found [40] in the Sancus architecture through automated validation.

1.1 Trusted Execution Environments

Trusted execution environments [27, 34] provide developers with a hardware enforced isolation used to obtain strong privacy and security guarantees. This facilitates an isolation of trusted software, called 'enclaves' from other software running on the device. It even prohibits other enclaves and highly privileged software layers such as the underlying operating system from accessing these enclaves. TEEs provide the developer with multiple guarantees on the confidentiality and integrity of code and data. This is a crucial property for many programs, especially as concepts such as cloud computing and remote sensing become ubiquitous.

State of TEE Security TEEs provide an extra layer in the security-in-depth strategy. However, the security invariants it provides requires the software developed on top of this hardware to be flawless. A wide range of different TEE architectures exists [16, 28, 30], each often supporting multiple runtimes [4, 2, 9]. Given the extensive list of attack vectors these designs must address, numerous vulnerabilities

were already discovered throughout ecosystem. As the human analyst has limitations, automated vulnerability discovery could offer a solution to this problem. Automated solutions have been proposed [8, 15, 42, 24, 5] but low-end architectures such as Sancus have been neglected in this process.

Sancus As IoT devices become common in households and industry, the effects of compromising these devices grows every day. Sancus [28] is a low-end TEE design developed for the MSP430 architecture. It extends a 16-bit microprocessor and aims for usages on embedded devices such as pacemakers and sensor nodes.

Pandora The extensive ecosystem of Intel SGX runtimes necessitated the development of automated vulnerability discovery tools. Pandora [5] is an implementation of such a discovery tool developed to support validation of all Intel SGX runtimes. It uses symbolic execution, a technique that can be used to validate a program by traversing all its paths, to check programs developed with these enclaves on a specific set of vulnerabilities.

1.2 Contributions

This master's thesis delves deeper in automated validation for TEE architectures. The work of this thesis is based on Pandora, a TEE validation program for Intel SGX enclaves. This work adapts this tool with the goal to support automatic validation of enclaves developed for Sancus. More specifically, it provides the following contributions:

- An adaptation of Pandora towards the MSP430 Sancus architecture. Pandora targets automated enclave validation for Intel SGX runtimes. We adapt this program to support automated vulnerability discovery on Sancus. In detail, the absence of control flow and pointer vulnerabilities can be validated.
- Additions to the MSP430 angr backend. As the support for the MSP430 architecture in angr is not as well developed as support for popular architectures such as x86, this backend required adaptations to support seamless symbolic execution.
- An extensive unit test framework. The adaptation of Pandora can best be measured by its performance on crafted vulnerable enclaves. A unit test framework with 21 test cases to test the control flow detection and 15 crafted test cases for automated pointer vulnerability detection. In total also 19 enclaves written in C, were developed.
- Autonomous rediscovery of vulnerabilities found in previous versions of the Sancus runtime stubs and reproduced vulnerabilities described in earlier research on the Sancus codebase [40].

The code developed in this thesis is open-sourced and can be found at:
https://github.com/Gert-JanG/pandora-sancus/tree/main

1.3 Outline

This thesis is structured as follows:

Chapter 2 Background This chapter introduces the required background for this thesis. First the concept of a TEE and its security guarantees are discussed. Next a TEE developed for small embedded IoT devices, called 'Sancus' is described. Subsequently an elaboration of possible attack vectors in the landscape of TEE software is given. Then a basic introduction into to the concept of symbolic execution is given combined with an explanation of how this is already used in multiple tools developed towards automated vulnerability discovery. These topics all culminate in an explanation of Pandora. To finalize a small introduction to IPE is given. This section gives a measure of the state of IPE, a TEE design synthesized on MSP430 processors.

Chapter 3 Implementation This chapter sketches the developments and adaptations introduced to Pandora. It starts with an explanation of multiple fixes applied to the angr MSP430 backend. It continues with a comparison between Intel SGX and Sancus followed with a description of some major components and functionalities that have been developed and adapted. Next the different plugins for Pandora and its adaptations are described. The chapter ends with a description of a problem encountered in the symbolic execution of the entry stub and its solution.

Chapter 4 Evaluation This chapter describes the evaluation of Pandora on a wide variety of test enclaves. It starts with a description of test cases from an extensive unit test framework. Another part in the evaluation lets Pandora autonomously rediscover vulnerabilities described in previous research. It does this for older versions of the enclave runtime stubs, followed with vulnerabilities covered in previous research on enclave vulnerabilities.

Chapter 5 Conclusion This chapter gives a recap of the challenges encountered in this work and of the limitations that are still present in the current Pandora adaptation. It also gives some possible future directions that this research could further branch into. Then to finalize, a summary of the contributions from this thesis is given.

Chapter 2

Background

In the landscape of modern computing, security and privacy are key concepts. Safeguarding integrity and confidentiality are fundamental concepts within the domain of secure computing. One approach that has made strides into the secure computing domain is the concept of trusted execution environments (TEE). Section 2.1 explores the concept of TEEs and the guarantees it brings us. More specifically a TEE architecture for low-end devices, named 'Sancus' will be covered in Section 2.2. Although the development software developed on these TEEs reaches a more mature stage, a wide variety of vulnerabilities still occurs throughout these architectures and runtimes. This in research as well as in production grade TEEs. Section 2.3 elaborates further on this. While these vulnerabilities pose a significant threat to the security guarantees TEEs can provide, Section 2.4 and Section 2.5 elaborate on possible solutions that this thesis further delves into. Section 2.6 describes the current state of IPE, a small TEE synthesized on MSP430 processors.

2.1 Trusted Execution Environments

The need for security and privacy has given rise to the development of Trusted Execution Environments [27, 34]. Research behind the protection of data in transit and data at rest has been going on for decades, but the necessity for protection of data in use, while it is used in the processor, has not received as much attention. Especially in a world where devices and resources are shared between multiple users, protection of data in use is an essential feature. TEEs try to pose a solution to this problem. The goal of TEEs are to minimize the Trusted Computing Base (TCB) which can be defined as the combination of hardware and software components that are combined to obtain the security protections that TEEs offer [32]. In essence TEEs provide strong isolation between processes running on the same shared hardware. Another important feature that TEEs offer is attestation. This enables the user to verify that its code runs unmodified in the enclave. Section 2.1.1 and Section 2.1.2 go more in depth on those two properties.

2.1.1 Isolation

The main goal of TEEs is to provide isolation. Specifically, they ensure isolation from the operating system (OS) and other underlying software layers, allowing users to run their own enclaves on a system without having to trust any other software running on the machine. The use of a TEE requires a small hardware adaptation in the CPU. This adaptation can then limit the access to enclaves executed on the device. When examining the specific internals of trusted execution environments, they can generally be categorized into three distinct types.

A first TEE design relies on an untrusted host program sharing the address space with the trusted enclave. The enclave is thus embedded in the address space of the untrusted host. The CPU controls all accesses from the untrusted environment into the enclave and vice versa. This first design is used in TEEs like Intel SGX [16] and Sancus [28]. A second type of TEE obtains a separation between trusted and untrusted worlds with the help of a privileged software layer called the security monitor. A parallel can be drawn between the hypervisor in virtualization and the security monitor in TEEs. Both act as a mediator between two separated worlds. In a hypervisor these two worlds are the underlying OS and the virtualized OS while in the case of the security monitor these two worlds are evidently the trusted and untrusted worlds. This is the design for architectures like ARM TrustZone [30] and Keystone [25].

A third design arises in architectures such as AMD SEV-SNP [23] and Intel TDX [14]. These designs are oriented towards cloud and virtualized environments protecting the VMs from a potentially compromised hypervisor.

The rest of this thesis will focus on the first, shared address space design. This work will thus focus on enclaves that are enforced by the CPU through the means of access control. These are especially vulnerable to new types of attacks such as pointer vulnerabilities.

2.1.2 Attestation

As explained above, attestation is a feature enabling enclave users to verify that its code is loaded unmodified in the enclave. It can be done remotely and local. In remote attestation the TEE configuration is attested by a party on a different node than the enclave. Local attestation, as the name indicates, enables attestation between the enclave and a different party only on the same device. Note that the remote attestation mechanism can be used for local attestation as well, but often local attestation is added as a separate mechanism as it can be done with less overhead such as the use of symmetric encryption instead of asymmetric encryption [32].

Attestation can also be leveraged to support multiple enclaves attesting to one another and thus enabling secure code modularity. Secure code modularity can be defined here as obtaining code modularity where a secure module can verify that attested modules are trusted as well and cannot be interfered with. Attestation can be a necessity for enabling TEE support as an enclave can run on a device without trusting the underlying software. This inherently means that the loader on

the device itself cannot be trusted. As a consequence, the code loaded on the system is not necessarily the code that was intended to be loaded by the software provider. The attestation process can be done as follows: the TCB calculates a hash over the enclave memory the moment the security protections are enabled. In particular, from this moment on the untrusted code can no longer change the loaded enclave memory. If the hash calculated over the enclave memory does not correspond with the hash calculated by the user that loaded the code on the device, it is clear that the enclave program is not loaded correctly. The user can then take appropriate actions.

2.2 Sancus

The necessity for security in low-end, resource constrained devices has long been underestimated. Since Internet of Things (IoT) devices are becoming omnipresent in common households as well as in industry, the effects and consequences of compromising such devices become larger every day. A major limitation in the addition of security measures on IoT devices is their low-end nature and its resource limitations.

Noorman et al. [28] developed Sancus to bridge this gap between high-end TEE solutions and low-end resource constraints. Sancus is a research TEE developed at the KU Leuven. It extends the OpenMSP430 hardware which is a 16 bit microprocessor aiming for usage on small embedded devices such as pacemakers and sensor nodes. In Sancus the TCB is implemented entirely in hardware which means that no software components have to be trusted to safely deploy enclaves. This further contributes to the Sancus design goals as the (small) nodes do not have to run large trusted software layers. High-end TEEs often use advanced concepts such as virtual memory or hypervisor-like software layers as they are not resource constrained. Sancus's design is oriented towards a system with a single *Infrastructure Provider* that manages multiple Sancus enabled *Nodes* (N_i). *Software Providers* (SP_j) can then place one or more different *Software Modules* ($SM_{j,k}$) on these nodes. Sancus enhances systems with properties such as: software module isolation, remote attestation and secure linking. An overview of the memory layout of such an (N_i) is given in Figure 2.1. It indicates how all enclave layouts are kept in a *protected storage area* on the MCU. Each enclave is composed of a separate code and data section.

It must be noted that an adaptation in hardware only is not enough. Trivially, the software needs to be aware of this adaptation for the developer to make use of it. As a result, the Sancus toolchain contains an extension for the LLVM compiler. This extension ensures that entry and exit stubs are automatically added in the resulting binary. These stubs are used to prepare and clean the enclave environment on entry and exit. These stubs are crucial in Sancus' security as any vulnerability in it, will result in a vulnerability for all enclaves developed with this toolchain [40, 37]. Automatic validation of these stubs will be explored for the first time in Chapter 4.

2.2.1 Software Module Isolation

The property of software module isolation ensures that no software outside of a module can read or write the module's state or code. Memory protection forms a

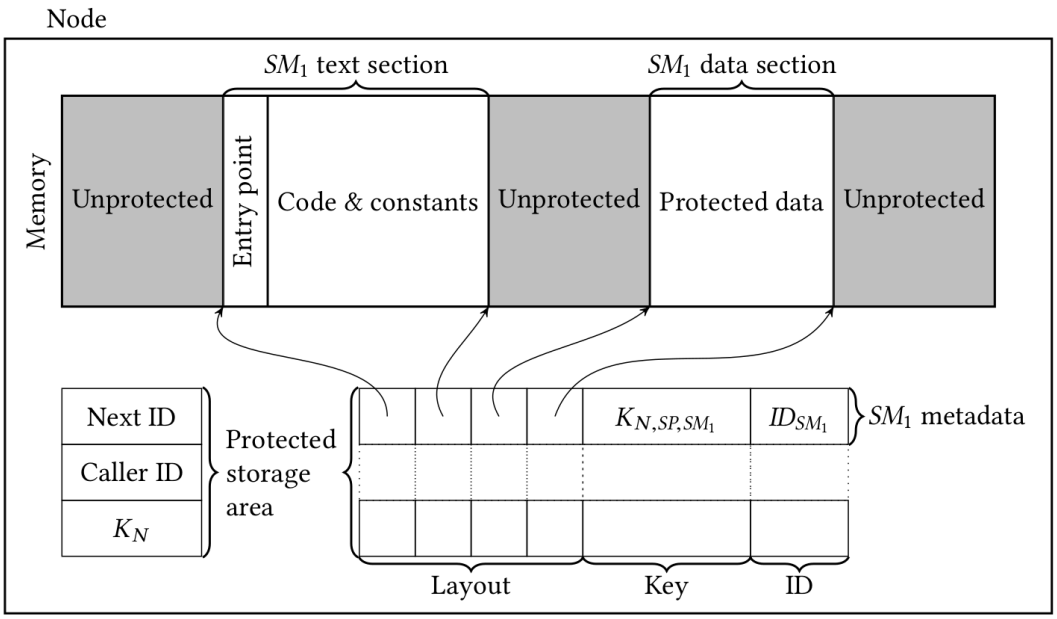


Figure 2.1: The memory layout of a Sancus module in a node N. Source: Sancus paper [28].

key concept in enhancing this isolation. Sancus uses *program counter-based memory access control* as proposed by Strackx et al. [35]. This form of access control restricts accesses to the data section of a module such that it is only read- or writable when the program counter (PC) is in the text section of the same module. At its core this means that some processor instructions using a specific key ($K_{N,SM,SP}$), can only access this key when they execute in the address range of the same enclave. $K_{N,SP,SM}$ is a key that is shared between the specific node N, the software provider SP and the software module SM. Note that SM is an identity of the specific software module depending on the module layout. This ensures that a software module can be loaded multiple times on the same N without having the same key. As implementation of this access control, the Sancus enabled processor is extended with a small *Memory Access Logic* (MAL) circuit. This MAL ensures that:

- The data section is only accessible when the PC is in the text section of the same module.
- The text section can only be executed by jumping to a specific physical entry point.
- The text section cannot be written. It can only be read while executing code in that section.

The memory access protections for a software module are enabled and disabled with the `protect` and `unprotect` instructions respectively.

2.2.2 Remote Attestation

Sancus provides the developer with remote attestation functionality. Remote attestation allows the software provider to verify that a software module is loaded unmodified on a node. With the help of authenticated encryption the software provider can always check whether the software module is loaded correctly. This authenticated encryption is available through the `encrypt` and the `decrypt` instructions. These instructions use the key $K_{N,SP,SM}$ which, as stated before, is only accessible when SM is executing. The last requirement for remote attestation is a freshness guarantee. This is provided by the SP that sends a nonce to SM on N. As this SM has exclusive access to $K_{N,SP,SM}$, it will encrypt the nonce with this key and send it back to the SP. This SP is then reassured that the enclave is loaded correctly.

2.2.3 Secure Linking

Secure linking enables software modules to link with and call other software modules on the same node. It also ensures high confidence in the fact that the intended module is called. Thanks to this secure linking, multiple enclaves can communicate with each other, enabling enclave modularity which also adds onto the intended isolation goal. Secure linking can be enabled through the `attest` instruction which checks the integrity and existence of a module at a specified address. Using this instruction a software module SM_1 can check if a certain software module SM_2 is present at the expected address. The identity of SM_2 should be known by SM_1 on deployment.

2.3 Enclave Vulnerabilities

Although a lot of research has been done towards enclave security, vulnerabilities are still being discovered. This can partially be attributed towards the large TEE ecosystem as multiple architectures exist, each with their own arrangement of runtimes. Van Bulck et al. [40] addressed these issues by mapping out 10 common vulnerabilities that arise in 8 different runtimes and architectures. Figure 2.2 gives an overview of these vulnerabilities and the affected runtimes. A full star indicates that the vulnerability has been successfully exploited, while a full circle indicates that the vulnerability has been confirmed but there was no proof-of-concept developed. The empty circle indicates that the vulnerability was not found or does not apply and half symbols indicate that improper sanitization leads to the possibility of side-channel leakages.

The vulnerabilities described here could be reproduced in multiple architectures and runtimes. The discovered vulnerabilities can be broadly categorized in ABI-level vulnerabilities and API-level vulnerabilities. Section 2.3.1 goes deeper into some ABI-level vulnerabilities while Section 2.3.2 elaborates on API-level vulnerabilities.

Vulnerability \ Runtime		SGX-SDK	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Asylo	Keystone	Sancus
Tier1 (ABI)	#1 Entry status flags sanitization	★	★	◐	●	◐	●	○	○
	#2 Floating-point register sanitization	★	★	○	★	★	●	○	○
	#3 Entry stack pointer restore	○	○	★	●	○	○	○	★
	#4 Exit register leakage	○	○	○	★	○	○	○	○
Tier2 (API)	#5 Missing pointer range check	○	★	★	★	○	●	○	★
	#6 Null-terminated string handling	★	★	○	○	○	○	○	○
	#7 Integer overflow in range check	○	○	●	○	●	○	●	●
	#8 Incorrect pointer range check	○	○	●	○	○	●	○	●
	#9 Double fetch untrusted pointer	○	○	●	○	○	○	○	○
	#10 Ocall return value not checked	○	★	★	★	○	●	★	○
	#11 Uninitialized padding leakage	[Lee17]	★	○	●	○	●	★	★

Figure 2.2: Overview of vulnerabilities discovered in different enclave architectures and runtimes. Source: J. Van Bulck, 2022 [38], based on [40].

2.3.1 ABI-level Vulnerabilities

Enclave vulnerabilities can arise through the Application Binary Interface (ABI). The ABI defines the communications of software on the binary level. It indicates for example who should clean parameters from the stack (as this can be done by the caller and by the callee) and how parameters will be passed (via registers or via the stack). Multiple ABI-level vulnerabilities arise in the context of enclaves. Some examples and a short explanation are given here.

Entry status flags sanitization

Many ISAs have a status register that contains status and control flags. When the untrusted application switches to the enclave context, the entry stub has to ensure that the status register is sanitized. An attacker can influence the execution of enclaves by controlling the values in the status register. It can for example set or unset control flags such as the 'direction flag' in x86, which controls the behaviour of loops. Therefore care must be taken as status and control flags must be set to safe values.

Floating-point register sanitization

An enclave can use the Floating-Point Unit (FPU). This implies that even the state in this FPU has to be sanitized. Alder et al. [6] discovered that sanitization of this state is not always done properly on enclave entry, leaving an opportunity for attackers to maliciously influence the behaviour of the FPU while executing on behalf of the enclave. This attack can therefore corrupt the results of this FPU. Moreover it enables a confidentiality breach as it can open a side-channel that leaks operands used in the calculations executed by this enclave.

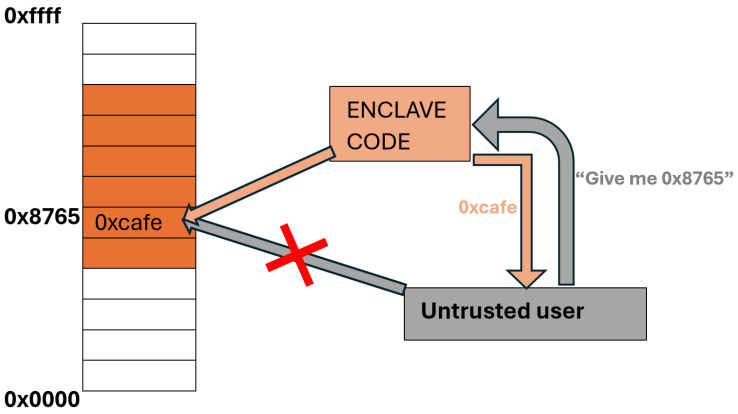


FIGURE 2.3: The confused deputy attack.

Entry stack pointer restore

On enclave entry the stack pointer should be stored, followed with a switch to a secure stack. This is an essential part in enclave security as the untrusted program can influence the call stack [17, 11]. By switching to a private stack, secrets used in the enclave that are pushed on the stack can never leak through this stack. Vulnerabilities can also arise when the conditions of this stack restore are not properly checked. This could lead to vulnerabilities of untrusted code executing an enclave reentry while no previous outgoing call has occurred.

Exit register leakage

Proper register sanitization should always precede an enclave exit. As the enclave can access secret values, these secrets could still be present in the registers when the enclave exits. The exit stub should thus always clear the used registers.

2.3.2 API-level Vulnerabilities

An important attack surface in TEE software arises through the API. A large part of the attack surface consists of an entire range of *Confused Deputy Attacks*. These attacks can occur when a victim and attacker program are placed in the same address space. This is visually represented in Figure 2.3. The attacker intends to fetch a secret memory address that it is not allowed to access. It therefore tries to trick the enclave, that can access the secret, to release the secret on behalf of the attacker. Some vulnerabilities arising through the API are described below.

Missing or incorrect pointer range check

When a pointer enters the enclave, the enclave should sanitize it by making sure the pointer is constrained to a specific area in the address space. If pointers are not properly checked, different attack vectors could arise. Untrusted code can for

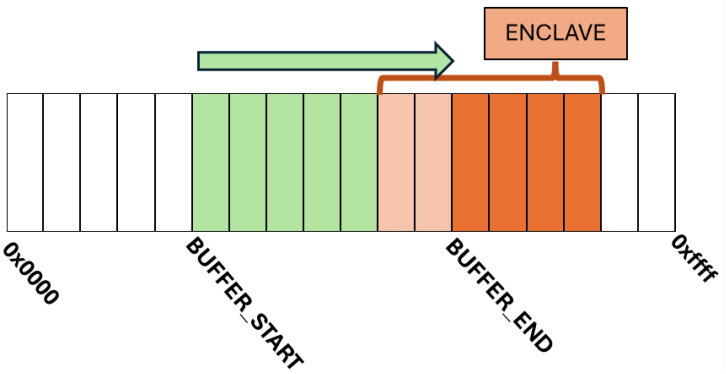


Figure 2.4: Incorrect buffer range checks resulting in buffer-enclave overlap.

example pass a pointer to a secret in enclave memory. Vulnerabilities could also arise when pointer ranges are checked incorrectly. The enclave checks the range of a pointer to determine if its range is entirely outside or inside the enclave. If this range is checked incorrectly, a possible overlap between a buffer and the enclave could result in the enclave leaking its own values when trying to access buffer's values as instructed by the untrusted code. This is depicted in Figure 2.4.

Null-terminated string handling

The enclave should always ensure that strings are null-terminated when copying them inside the enclave. Otherwise, the trusted functions inside the enclave could subsequently operate on this string. If the string is thus not properly terminated, these trusted functions could inadvertently operate on values in enclave memory beyond the string, as they expect to read until a null byte occurs. This could result in the enclave leaking sensitive information.

Integer overflow in range check

The use of unsafe arithmetics can result in subtle programming logic errors. When pointers are checked if they lie inside or outside the enclave address range, these overflows could result in faulty conditions. For example if the enclave end address is calculated by adding the size to the start address, this overflow results in a wrap around the address space. This can consequently lead to invalid range checks of the enclave address space. This is visually represented in Figure 2.5. The figure shows a buffer that wraps around the address space resulting in a lower end address than the start of the buffer. In this case, the enclave could wrongly assume that the buffer lies outside the enclave while the buffer actually contains the entire enclave.

Double fetch untrusted pointer

A subtle attack arises when a timing difference occurs between the validation of a pointer and the copying of the pointer inside the enclave memory. This timing

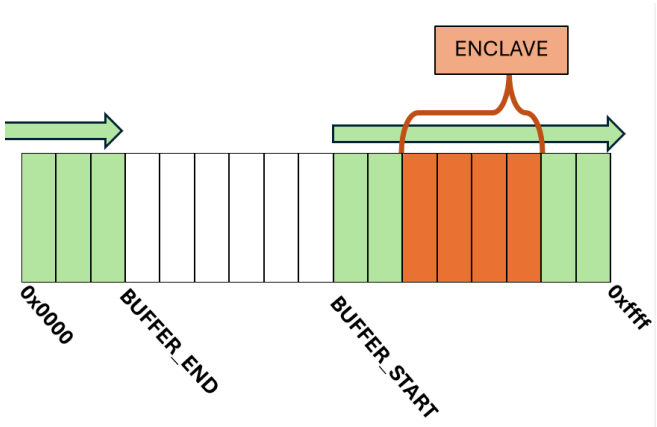


FIGURE 2.5: Wrapping of the address space through unsafe arithmetics resulting in incorrect range checks.

gives rise to '*time-of-check to time-of-use*' attacks. These attacks exploit the timing difference by changing the pointer during this short interval and hence result in malicious pointers entering the enclave.

Ocall return value not checked

Just like any input values into the enclave, return values from outgoing calls (ocall) have to be checked. These ocalls occur when the enclave calls a function that lies outside its own protected memory range. Exploitation of these vulnerabilities can be categorized as part of an entire class of 'Iago attacks' [13]. For example when an enclave wants to print some values to the standard output, the print function located outside the enclave will be called. The return value of this function can subsequently be influenced by the untrusted code which then enters the enclave.

Uninitialized padding leakage

A possible vulnerability arises through the use of padding, automatically added by the compiler, as a means of optimization [26]. The compiler does this to ensure that variables in memory always align to a multiple of the word size. This reduces the amount of memory reads when this value is fetched from memory. When an advanced data type such as a struct is used inside the enclave, the values of these data types can be automatically padded by the compiler. Consequently, if these padding bytes in the field are not overwritten with zero bytes, enclave secrets present in these bytes can inadvertently leak.

2.4 Symbolic Execution

Vulnerability discovery in enclaves can require a significant amount of time and effort. Although human analysts can extensively test software systems, manual testing

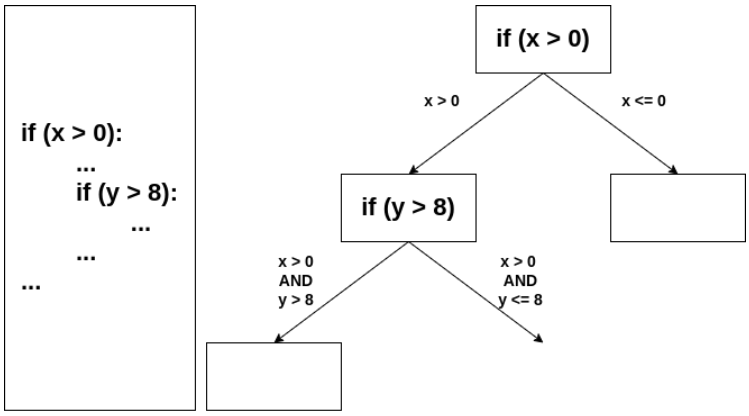


Figure 2.6: A visual representation of symbolic execution.

is inherently not a complete solution. Techniques such as fuzzing can reduce the required efforts, yet these do give any guarantees on the absence of vulnerabilities. Symbolic execution however tries to obtain this exact goal. Symbolic execution is a paradigm that allows developers to exhaustively trace all paths trough a program. In Section 2.4.1 the mechanism of symbolic execution is explained in more depth while Section 2.4.2 delves deeper in state-of-the-art applications of symbolic execution used to validate Intel SGX enclaves.

2.4.1 Symbolic execution mechanism

In symbolic execution a software system is exhaustively tested by traversing all possible paths in the program's control flow. While traversing a path, the system keeps all state symbolic (e.g. constraints, registers). When a branch, such as an if condition on a variable, occurs in the execution of a path, the program will not take a single path related to a concrete value. Instead the symbolic execution engine will simulate both paths. This means that on occurrence of a branch, a single state will split in multiple states. Each of the resulting states also keeps track of the constraint that is being put on the conditioned variable. An example can be seen in Figure 2.6. On the left the code that will be symbolically executed is represented. On the right a tree with all possible paths in the code section is shown. The initial state at the top branches on the value of x. This branch results in two new states of which the left branches again on the value of y. Constraints are accumulated and shown on the arrows.

angr [33] is an example of such a symbolic execution program. More specifically, it is an open-source binary analysis platform. angr provides effective techniques to trace all paths trough a binary program. It provides developers with an extensive stashing mechanism to keep track of interesting states, and which states for example are errored or finished. When a specific state is inspected, an SMT solver like Z3[18] can be used to resolve the constraints, if possible, into concrete values. These concrete values must satisfy all the constraints belonging to the state.

2.4.2 Symbolic Execution on Intel SGX

Despite Intel SGX's large ecosystem [4, 2, 36, 9] and the extensive efforts done to secure its runtimes, Section 2.3 demonstrates that vulnerability-free enclaves have not yet been achieved. Although many vulnerabilities were already found and patched, a systematic and efficient technique in vulnerability discovery has been lacking. That is until recently. Over the last few years, multiple solutions for automated vulnerability discovery have been developed for Intel SGX enclaves. Examples of these automated vulnerability discovery programs are: Guardian [8], COIN [24], SymGX [42], TeeRex [15] and Pandora [5]. It must be noted that while these systems have proven its worth, they all still contain limitations.

First and foremost all solutions except Pandora are limited to a single Intel SGX runtime. As a wide ecosystem of Intel SGX runtimes exists, each with their own characteristics [39], the effectiveness of these solutions is thus limited. Other limitations for TeeRex and Guardian are that they do not symbolically execute the enclave initialization in its entirety. TeeRex for example skips the enclave entry and immediately starts symbolic execution at the enclave application. Guardian on the other hand starts at the entry point, but skips the enclave initialization phase. COIN then requires the enclave source code to extract the parameters of enclave calls (ecall). In addition, SymGX only focuses on cross-boundary pointer vulnerabilities. What makes these tools similar is that they all use symbolic execution to validate enclaves. As a consequence, all these tools can suffer from path explosion which limits the coverage that can be achieved with it. Although some effort has been done to limit the amount of false positives, this is still a limitation present in tools like Pandora, TeeRex, SymGX. While the COIN paper does not discuss this aspect, the Guardian paper on the other hand explicitly states that it does not report any false positives. As made clear by now, significant efforts have been made to systematically and automatically validate enclaves developed for the Intel SGX hardware. However, these efforts are put in stark contrast when considering the efforts done to validate enclaves developed for Sancus. This architecture has been neglected thus far.

2.5 Pandora

Although many automatic enclave validation solutions exist, this work is based on the Pandora [5] tool. Not only because of its ability to include the enclave entry and exit stubs in the validation process, but also because of its extensible design as will be made clear in Section 2.5.1 and Section 2.5.2. Pandora is based on angr as mentioned in 2.4.1. It extends angr functionalities with a clear symbolic memory model and techniques such as taint tracking. This facilitates tracking of values, influenced by an attacker, through the enclave. Pandora can be run through a rich CLI interface. It enables setting multiple execution parameters such as the amount of symbolic execution steps that have to be taken, selecting which plugins have to be run. Moreover it allows to set a specific action on execution events such as opening an interactive shell on error.

2.5.1 Pandora's Architecture

An essential part of Pandora's ease of use is its modular architecture. A high-level overview of its architecture is depicted in Figure 2.7. The core of its architecture is the Pandora engine. This extensive engine executes for example the first state initialization, and starts the symbolic execution. It interacts with angr and augments it with techniques that help to keep track of enclave memory. This for example through different exploration techniques that determine which states have to be tracked. Pandora can support multiple Intel SGX runtimes by using a 'dynamic phase'. This phase can extract the necessary contents out of a binary and leverage these contents with useful debugging information. This phase returns a memory dump accompanied with a JSON file containing metadata about it.

Pandora allows extension to more runtimes through the development of a compatible SDK. Runtime specifics are thus added in the SDK. The plugins contain the real vulnerability detection logic. These plugins are independent of the underlying Pandora codebase and thus enable pluggable vulnerability detection. The plugins are discussed more in depth in Section 2.5.2. Another core part of its design: UI and Reports which facilitates the generation of extensive HTML reports. Each of the plugins will generate an extensive report describing all issues it found. It also reports context details such as the register contents of the state in which the issue was found. An example of such a generated report can be seen in Section 2.8.

Pandora reports issues and debugging information in six different criticality levels. These levels are, on increasing importance: Trace, Debug, Info, Warning, Error, Critical. These levels provide the developer with different levels of detail on the execution and issues. Even though Pandora is runtime agnostic as made feasible by the different SDKs, it is not architecture agnostic. Intel SGX is a high-end TEE solution and thus inherits high-end concepts such as a 'Thread Control Structure', a data structure that contains specific information used to manage a thread. This leaves its marks on the internal code of Pandora. To make an adaptation or extension to another architecture feasible, these places need to be pinpointed. Except for the dynamic phase and the plugins, Intel SGX specifics have made it into the Pandora codebase such as the abstraction between the Pandora engine and a specific SDK, the details in the generated reports and the implemented instruction hooks.

2.5.2 Pandora Plugins

Pluggable vulnerability detectors make it possible for the Pandora core to run without any plugins activated. In this scenario, it would only traverse the enclave code without checking for specific vulnerabilities. This is facilitated through the use of breakpoints. Each plugin can subsequently subscribe on any of these breakpoints. A breakpoint can be set for every interesting event such as a jump to an address that is attacker-tainted or for example every memory read and write. Pandora currently includes 4 plugins: CFSan, PTRSan, ABISan and AEPICSan each containing the logic for a distinct vulnerability category. In what follows the first three plugins will be discussed. The AEPICSan plugin is tailored towards a very specific AEPIC attack

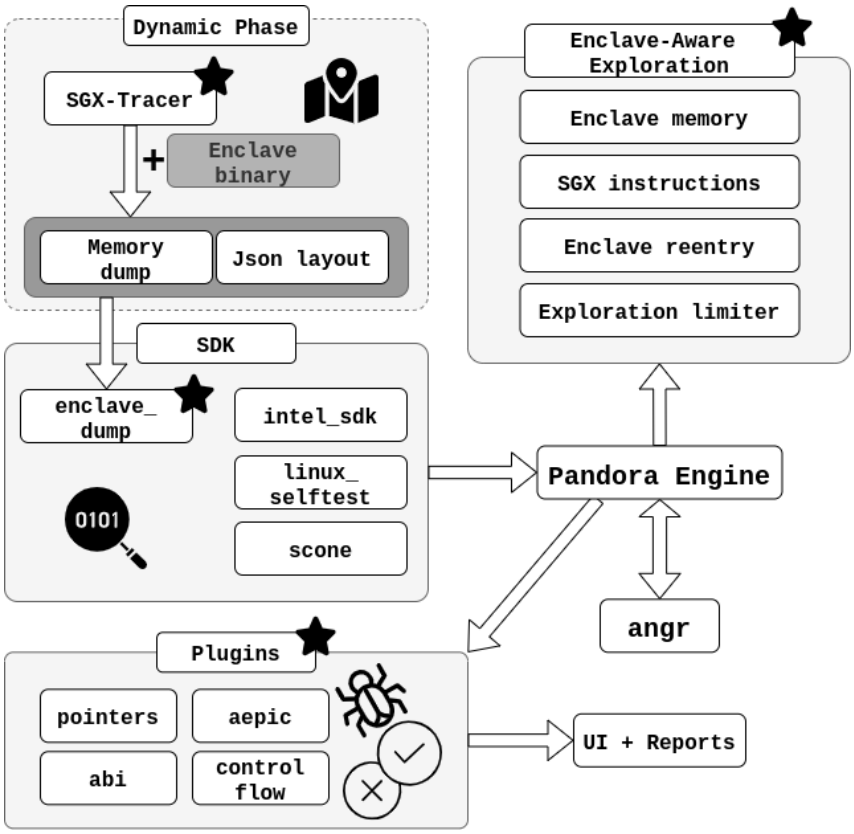


FIGURE 2.7: The Pandora architecture. Source: F. Alder, 2024 [5].

[12] targeting Intel SGX runtimes which requires the use of paging. As MSP430 does not support virtual memory this attack vector will not be present on Sancus.

CFSan

The ControlFlowSanitization plugin is essential in detecting unexpected changes in the control flow of the enclave execution. As an attacker can try to hijack the enclave execution, checks need to be in place to ensure that every instruction changing the control flow, for example a jump or a call, behaves as expected. This plugin will then accordingly check that first and foremost: jumping to an address inside unmeasured and uninitialized memory will be reported as a critical issue. This because an attacker has complete control over the contents of these pages. A second check done by the plugin validates the absence of arbitrary attacker-tainted jump targets. It is trivial that a jump to an attacker-controlled address could be undefined behaviour if this target address is not restricted to either the inside of the enclave or entirely outside the enclave. This is often the result of a programming logic error. A third and final check done by the CFSan plugin validates the absence of jumps to attacker-tainted target addresses that resolve entirely inside the enclave. This will be reported as a

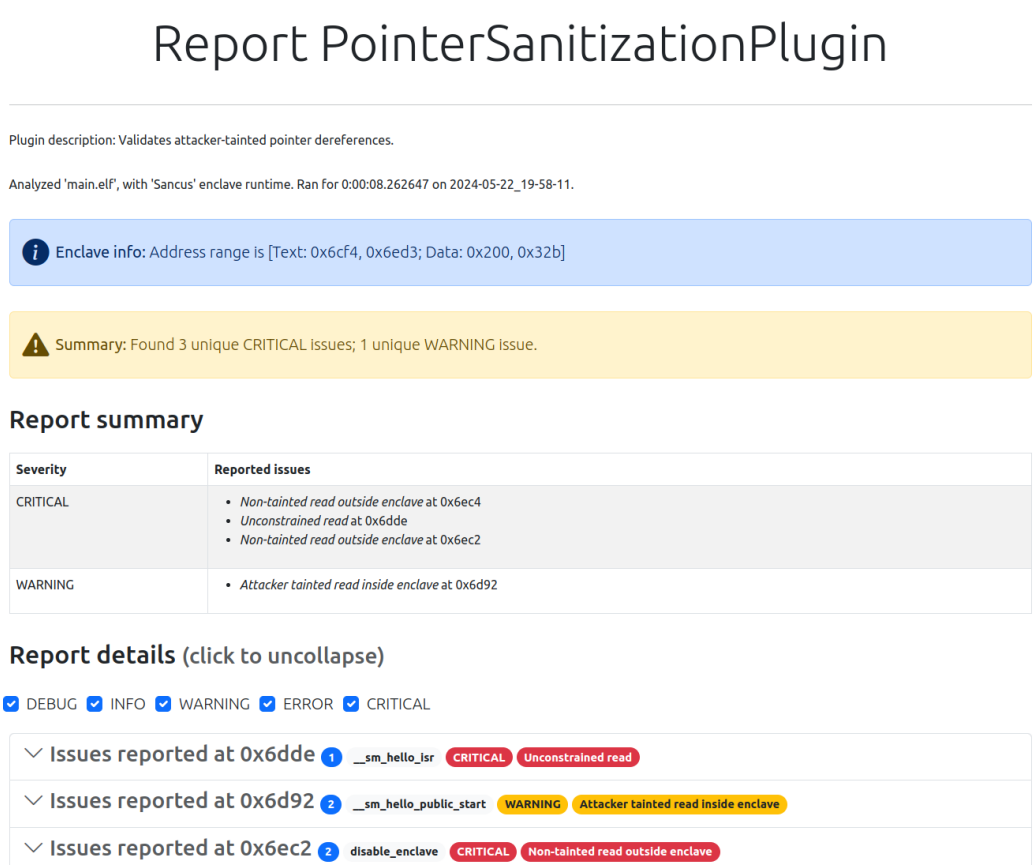


Figure 2.8: An example report generated by the CFSan plugin.

warning and requires further manual inspection as this can also be benign behaviour.

PTRSan

The PointerSanitization plugin contains logic that checks memory reads and writes. The plugin checks three different invariants. An initial validated invariant ensures that for every read and write the address must be restricted to either inside or outside the enclave. If this is not the case and the address can be resolved to any of the two, a critical issue is reported. A second invariant ensures that no read or a write is done to an attacker-tainted address that resolves inside the enclave. This is mainly a heuristic as attacker-tainted accesses in an enclave table are possible benign behaviour. If this issue does arise, a warning will be reported. The last invariant checked by the PTRSan plugin ensures that the enclave does not read or write at non-attacker-tainted addresses that lie outside the enclave. Otherwise a critical issue will be reported.

ABISan

The last plugin of interest, the ABISanitization plugin, checks that the low-level enclave state is sanitized on enclave entry and exit. The enclave itself can always call functions residing outside the address space of the enclave. Likewise an untrusted function can always call functions inside the enclave. When the enclave starts executing, it must be cleared of attacker-tainted values. The plugin checks three different invariants. It checks that the enclave does not read attacker controlled registers. If this is the case, a critical issue is reported. Furthermore, it checks that all registers are sanitized when the ABI sanitization is finished. The ABI sanitization is done by the entry stub inside the enclave. A jump to higher API-level code can be heuristically detected on the first call instruction in the entry stub. So if at this point registers remain unsanitized, a warning will be reported. A third and final check done by the ABISan plugin, confirms that on enclave exit the stack registers are attacker-tainted. This check ensures that no secrets leak through the registers when returning back control to the untrusted environment. Therefore these issues will be reported as critical.

2.6 IPE

Intellectual Property Encapsulation (IPE) [22, 21] provides isolation for code and data. It is developed by Texas Instruments and synthesized on the MSP430 processor architecture. IPE can protect a single memory segment against external access. With IPE in place, code as well as data is protected from read and write accesses originating from outside the IPE region. It does no call site verification and does not require the protected areas to have specific entry points. The untrusted environment can thus jump to any arbitrary address inside the IPE area. Moreover it does not clear state on context switches to the untrusted environment. As mentioned in [21], the goals of IPE are questionable and often lack concreteness. For example, IPE documentation states that: "poor code security practices can make the code more vulnerable even if it is within the encapsulated area" [22]. These unclear and basic specifications leave room for multiple vulnerabilities. This reflects through multiple hardware and software vulnerabilities discovered in IPE [31, 10]. The works in [10] however develop promising mitigations for many IPE vulnerabilities. These mitigations include the addition of `IPE_ENTRY`, `IPE_FUNC` and `IPE_DATA` preprocessor annotations and secure entry and exit stubs similar to Sancus. These could be validated in a similar fashion as done for Sancus in this work, but is however out of scope.

2.7 Conclusion

The need for protection of data in use gave rise to a new security paradigm called ‘trusted execution environments’. Although a lot of research has been done towards high-end TEE solutions. Resource efficient solutions for the IoT space remained absent. Sancus poses a solution to this gap. Its architecture obtains similar security

guarantees as high-end solutions, but does this without the need for high-end solutions such as virtual memory. While these TEEs are an extra defensive layer in the security of everyday devices, a large range of vulnerabilities are still present in the TEE ecosystem. This calls for adequate measures that leverages the human analyst from the burden of manually finding these vulnerabilities over and over again. Symbolic execution is a technique that can be used for this exact purpose. Over the years multiple programs were developed with the goal of automatic enclave validation in mind. While all these programs have proven its worth, Pandora is aimed at extensibility. It aims at validating all SGX runtimes with the help of symbolic execution. Its pluggable vulnerability detectors can generate extensive reports about (possibly) present vulnerabilities in the program. Pandora thus can validate any program developed for any SGX runtime and provide developers with extensive reports regarding its findings. It is clear that the state of IPE, a commercial TEE design for the MSP430 architecture, still lacks clear security goals. Multiple vulnerabilities were already discovered in its hardware and software indicating that its design still lacks fundamental measures. Validation of MSP430 enclaves through symbolic execution could pose a useful and efficient solution to these software vulnerabilities.

Chapter 3

Implementation

Many efforts have been done towards validation of Intel SGX runtimes. In contrast TEE architectures like Sancus were often neglected in this process. This work bridges that gap by adapting the existing Pandora codebase for Intel SGX to a codebase tailored towards validating Sancus enclaves. Section 3.1 starts with the development of a proof-of-concept (PoC). This PoC tries to early on estimate the state of MSP430 support in angr and describes solutions to challenges encountered in this backend. Next a comparison between Intel SGX and Sancus will be given in Section 3.2. This comparison drives the changes implemented to the codebase as described in the rest of this chapter. While the adaptation includes a lot of work under the hood, some larger parts during the development of the symbolic execution engine are described in Section 3.3. With a functioning symbolic execution engine the vulnerability detection plugins can be added one by one. The addition of the plugins and its developments are described in Section 3.4. Another challenge that arose during the development of the symbolic execution engine was that for compiler-generated enclaves, the symbolic execution kept running indefinitely. The cause of this problem and the implemented solution will be described in Section 3.5.

3.1 State of the MSP430 angr Backend

The work in this thesis relies heavily on the support for MSP430 in angr. Before adapting the more complex Pandora codebase, a small scale PoC was developed. Section 3.1.1 describes this PoC. Section 3.1.2 elaborates on issues that arose in this backend and discusses some adaptations that had to be done to be able to obtain seamless MSP430 angr support. First things first, a small enclave file is required to test the symbolic execution. This enclave best be minimal to not be flooded with minor details that are not important for the initial core functionality. This minimal enclave is described in Appendix A.

3.1.1 Proof-of-Concept

The goal of the PoC is to traverse an enclave using angr's symbolic execution without advanced concepts, such as taint tracking, to be able to estimate the state of the MSP430 support in angr. The PoC starts the simulation at the entry point of the enclave as only the code inside the enclave has to be validated. It symbolically executes the enclave using angr, but without for example instantiating symbolic values for the registers and memory locations. Before traversing the enclave, the program ensures to parse and hook all Sancus related instructions. These instructions are hooked to a skip function which thus ensures that execution continues at the address past the Sancus instruction. This because the angr framework does not support Sancus instructions. When control returns to the untrusted context, the PoC program finishes execution. Note that this PoC is not a simplified version of Pandora, rather, it is an essential step in finding limitations and challenges early on in the entire development process.

3.1.2 angr's MSP430 Support

The PoC thus aims at early discovery of limitations of the MSP430 support in angr. Although basic support exists, it is unfortunately less mature than support for popular architectures such as x86. MSP430 support in angr is enabled by the *angr-platforms* repository[1] which implements a lifter to lift the binary code to VEX intermediate representation (IR) [41], an IR that abstracts away architecture specifics. With this IR, angr can reason about programs in an architecture agnostic way. Support for multiple other architectures such as AVR and RISC-V is implemented here as well. First and foremost, the development of the PoC made clear that disassembly functionality is not properly developed for MSP430. Therefore instead of using Capstone [1] for disassembly, parsing of the object dump is used to generate useful debugging output. Furthermore, some bugs were still present in the code that supports MSP430 instructions in angr. To be more specific, the `br`, `rra` and `rrc` instructions required fixes for the code to run seamlessly.

As part of this research I implemented and merged patches upstream to the *angr-platforms* backend. A first small fix pushed upstream fixed an unused 'writeout' argument that was passed to the `compute_result` function of the `rra` instruction. Another fix I pushed upstream unifies the behaviour of the `br` and `ret` instruction with the behaviour of relative jump instructions such as `jmp` and `jc`. This unification creates an equal behaviour for all jump type instructions. Before this fix, the target **ip**, was immediately committed to the state by the `br` and `ret` instruction. For the other jump types, these effects would however not immediately be committed to the state. The effects of this can be noticed in for example the CFSan plugin as depending on which of the two different instructions would execute, a different behaviour was observed by Pandora.

Moreover, I discovered that a specific assertion was triggered in the `pyvex` backend. `pyvex` is the implementation of VEX IR used by angr. By analysing the stacktrace it

[1] https://github.com/angr/angr-platforms

became clear that the assertion was triggered at the occurrence of the `rra` instruction, which is present in the entry stub of Sancus enclaves. A next step was to isolate this instruction in a small handwritten enclave that only contains this instruction. This made clear that the assertion was not only triggered by the `rra` but also by the `rrc` instruction.[2] Another bug discovered later on resided in the behaviour of the `br` instruction. As many states would end up in this instruction and stayed at this specific address, it was an indication that the effects of the `br` did not affect the state.[3]

3.2 Intel SGX vs. Sancus

From the PoC onwards, the extension to a real Sancus enclave validation program was the next step. As starting point, two possible development paths were under consideration. A first option was a gradual extension of the PoC to a fully functional program obtaining the same results for Sancus enclaves as Pandora obtains for Intel SGX enclaves. This working codebase could then gradually be merged with the Pandora code itself. The other option was to start from the existing Pandora codebase and from there gradually adapt away from the Intel SGX oriented code and step-by-step include the Sancus intrinsics. After some examination the second direction was deemed the most efficient path to take, and is thus the path pursued in this work. This immediately initiates a starting point for the adaptation: creating an oversight into the differences and similarities between Intel SGX and Sancus.

3.2.1 Similarities

As both Intel SGX and Sancus are TEE architectures, a parallel between them can be drawn. A lot of similarities can be drawn but this section focusses on the similarities important for the developments in this work. The most important similarity between them is the single address space. Both architectures have a single address space shared between the trusted and the untrusted processes. The enclave is therefore embedded in an untrusted host process. This single address space can be a root for many vulnerabilities as already elaborated in 2.3. Another similarity is that both Intel SGX and Sancus support multiple enclaves. This means that at any point in time there can be more than one single enclave present in the address space. An additional similarity between Intel SGX and Sancus is that they both only allow for the untrusted code to jump to specific entry points in the enclave. Multiple entry points can be present in a single enclave. Jumps from outside the enclave to an arbitrary location inside the enclave are thus not allowed. Upon enclave entry, developers depend on the enclave entry routine to scrub registers.

[2] After disclosing this bug to my supervisor Jo Van Bulck, he discovered that some values are not properly typed. His patch has been pushed upstream.

[3] After disclosing this bug to my supervisor Jo Van Bulck, he discovered that the `br` instruction does not properly branch to the address in the source operand, but to its own address. His patch has been pushed upstream and incorporated in the codebase.

3.2.2 Differences

Naturally, there are also differences between the Sancus and Intel SGX architecture. The most significant difference can be found in the underlying architecture. While Intel SGX is developed for the x86 architecture, Sancus is developed for the MSP430 architecture. This inherently separates them into different use-cases as x86 is widespread in high-end computing environments while MSP430 is commonly found in low-end environments such as low-power embedded applications. This also results in multiple high-end features present on Intel SGX, such as virtual memory or a 'thread control structure', a data structure that contains specific information used to manage a thread, being disregarded when developing for Sancus. Furthermore Intel SGX is a production-grade TEE while Sancus is still a research TEE. This also results in there being a larger ecosystem built for SGX compared to Sancus. Moreover Intel SGX enclaves must be entered and exited through specific instructions. Enclaves can be entered through `EENTER` and `ERESUME` instructions while `EEXIT` and `AEX` can be used to leave the enclave. This unlike in Sancus as there are no special instructions developed to enter or exit the enclave. Sancus enclaves can be entered and exited using regular control flow instructions such as `call` or `br`. As stated above, both Intel SGX and Sancus support multiple enclaves. An important note here is that Intel SGX enclaves are completely isolated from each other. This is not necessarily the case for Sancus enclaves. In Sancus, enclaves can securely link with and call each other allowing secure code modularity. A last important difference is the fact that in Intel SGX an enclave has to be a contiguous area in memory. This is not the case for Sancus as the text and data section do not have to be placed next to each other in memory. A Sancus enclave can thus be split into two separate parts in memory.

3.3 Symbolic Execution Engine Development

As made clear by now, there are a lot of differences but also similarities between Sancus and Intel SGX. The fact that they are both TEEs using a single address space makes an extension of Pandora to Sancus useful. To incorporate the differences between the two, unique features of SGX have to be removed from the codebase, and unique features of Sancus have to be added. An initial goal for the adaptation of Pandora was to enable it to symbolically execute Sancus enclaves. The development process was therefore split into two disjunct parts. First Pandora had to able to symbolically execute Sancus enclaves, and only then the plugins could be added and adapted. In what follows the amount of changes and additions will be described in lines of code (LOC) added and removed as reported by *git diff*. This measurement does include documentation. Although a very noisy and often miss representative measurement, it is added for sake of completeness.

Figure 3.1 gives a general overview of the Pandora codebase. Some details and modules are not depicted here as the figure is meant to give a quick overview of the structure of the codebase. Note that the *pandora.py* file is the entry point of the Pandora codebase. It is thus called through the CLI and interacts with the internal components of the system such as the `SDKManager`, `HookerManager` and

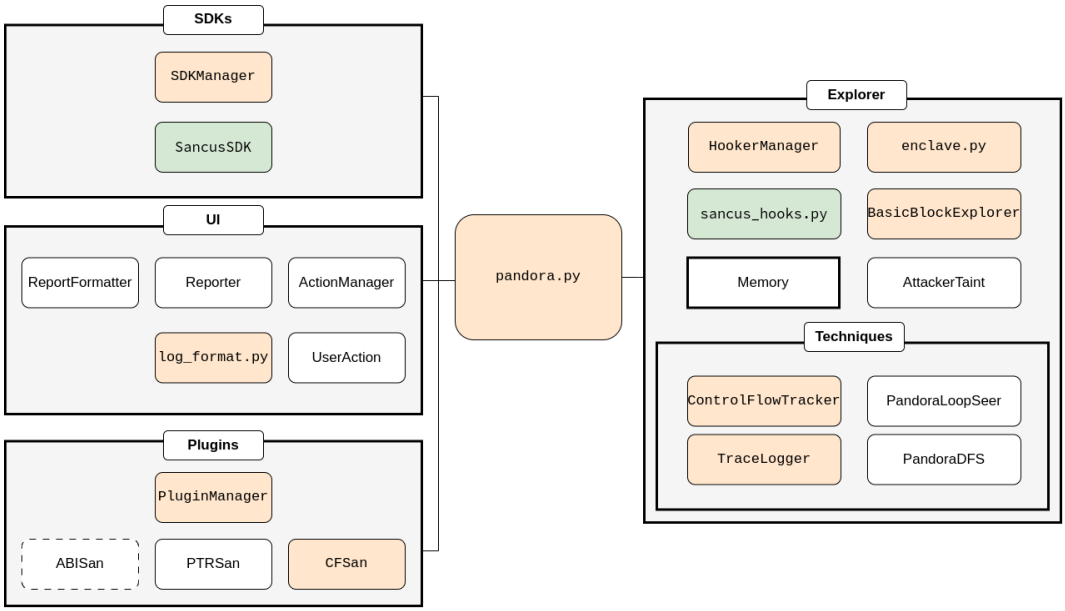


FIGURE 3.1: Structure of the new Pandora codebase. Green indicates that these files/classes were implemented from scratch while orange indicates that adaptations were made.

`BasicBlockExplorer`. The figure depicts 4 main packages. The **SDKs** package contains all logic and abstractions for facilitation of different runtimes. The **Explorer** package contains core functionalities such as exploration of the enclave, tainting and symbolizing memory, and hooking of instructions. Combined they form the core of Pandora's symbolic execution engine. The **UI** package contains logic for reporting and logging and the **Plugins** package trivially controls the plugins. The green coloured files and classes indicate that they are added from scratch while the orange colour indicates that adaptations have been made. Every rectangle with a bold border illustrates a package. Note that **Memory** is also a package as is indicated with a bold border, but this package deals with the underlying memory management and its details are not important for the overall view.

Table 3.1 shows the amount of changed LOC per package. The first column of *added LOC* does not include the newly added files. The second column *Added LOC (incl new)* does include the sizes of newly added files and is left empty for clarity reasons if this is not applicable. Note that the column *Removed LOC* also does not contain obvious removed lines of code such as entirely removed SDK classes.

3.3.1 Sancus-SDK

A good starting point for the adaptation of Pandora was the development of a new SDK class. Pandora can support multiple runtimes by adding on new SDK classes. An SDK class is initialized by and interacted with through the `SDKManager`. This `SDKManager` class facilitates an abstraction between the *pandora.py* entry point and

Table 3.1: Changed LOC split in added and removed LOC. The column (incl new) includes the lines of code added by entirely new files.

Package	Added LOC	Added LOC (incl new)	Removed LOC
SDKs	26	221	169
Explorer	268	412	266
Plugins	53	-	16
UI	125	-	28
pandora.py	8	-	6

the wide range of possible SDKs. For this purpose the Pandora codebase is extended with a `SancusSDK`. The SDK class takes care of SDK detection, fetching enclave addresses and sizes, providing a textual representation of the executing basic block and fetching specific instruction addresses. A basic block is a block of consecutive instructions that always ends on the occurrence of an instruction that breaks the normal control flow such as a `call` or `jmp` instruction. While the SDKs developed for Intel SGX runtimes all use Capstone for disassembly, Capstone does not support the MSP430 architecture. A possible solution here could be to extend the Capstone framework with MSP430 support. While this solution would possibly accelerate future developments that for example are made possible by this work, it would be an orthogonal line of work. Therefore, the `SancusSDK` overcomes this issue by mainly using an object dump of the binary as a substitute. The object dump is used in multiple ways. It is used for parsing basic blocks that are added as context during debugging and in generated reports, but also for the parsing of Sancus instruction addresses and the parsing of enclave boundaries. The Sancus-SDK class itself is implemented in 195 lines of code.

3.3.2 Hooks

An important part of the implementation concerns the hooking of Sancus instructions. Hooking is necessary as the *angr-platforms* repository only supports the default MSP430 instructions [3]. Another option here could be to extend angr's support for MSP430 by adding support for Sancus instructions to the backend. Although it is an interesting path to take, it would require delving deep into the intrinsics of angr and even more challenging into the VEX IR. Just like developing an extension of Capstone this would be a more orthogonal line of work. Sancus instructions are hooked by the `HookerManager`. This `HookerManager` iterates over the enclave section and replaces every Sancus instruction with a hook that simulates the intended effects of the instruction on the symbolic execution state. In the current implementation, most of the hooks merely implement a skip functionality. This means that the effects of the instructions on the symbolic execution state are not present and left for future work. It could be stated that the effects of the Sancus instructions itself are of less importance in the entire line of developments. Table 3.2 gives an overview of the Sancus instructions and their corresponding hook and state.

Table 3.2: Table with all Sancus instructions and hooks.

Wrapper	Instruction	Hooked method	Hook Implemented
sancus_disable	SM_DISABLE	SimUnprotect	Partial
sancus_enable	SM_ENABLE	SimProtect	✗
sancus_verify_address	SM_VERIFY_ADDR	SimAttest	✗
sancus_wrap	SM_AE_WRAP	SimEncrypt	✗
sancus_unwrap	SM_AE_UNWRAP	SimDecrypt	Partial
sancus_get_id	SM_ID	SimGetID	Hardcoded: return 0 & constrain r7
sancus_get_caller_id	SM_CALLER_ID	SimGetCallerID	Hardcoded: return 0
sancus_stack_guard	SM_STACK_GUARD	SimNop	✗
N/A	SM_CLIX	SimNop	✗

The `sancus_enable` instructions are not used in common implementations of Sancus enclaves. Creation of enclaves is typically done inside the untrusted environment and supporting this instruction is therefore not a major concern. The `sancus_disable` instruction on the other hand does occur inside the enclave as enclaves disable its own security protections. Currently its hook is only partially implemented. At the moment this instruction sets a global variable in the state indicating that the protections are disabled. This will enable filtering in the `BasicBlockExplorer` and ensure that the state is removed from the stash of active states as these states should no longer be tracked. This hook is thus only partially implemented as in a full implementation for example also a breakpoint should be triggered on which the ABISan could inspect the state of the ABI. The `sancus_verify_address` instruction is the instruction used for enclave attestation. Its implementation is not an immediate concern within the scope of this work. This because validating the secure linking process is not included in this work. Moreover this instruction provides attestation functionality for the *SP* to ensure that the module is loaded unmodified on the node *N*. This instruction will thus not typically be called from inside the enclave itself.

The instructions `sancus_wrap` and `sancus_unwrap` provide a module with encryption and decryption functionality. It thus provides the enclave with confidentiality guarantees. These functionalities are used for secure communication and data confidentiality. A limitation of this work however is that it only supports validation of a single enclave. Therefore secure communication does not have to be validated in this work. Furthermore encryption and decryption of data is not the core of this work. The decryption instruction is however partially implemented to support part of the evaluation in Chapter 4. This hook reads the values passed to it in the required registers, but instead of decrypting the data with the key, it simply stores these values inside the foreseen buffer. Full instruction support can be implemented in future extensions given that the key is present, but is however not the main goal of this work.

Finally the `sancus_get_id` and the `sancus_get_caller_id` instructions are hard-coded in this work. The `sancus_get_id` instruction is hooked to a function that will put the zero value in the return register as this function should return the ID of the module that lies at the address in the return register. This hook can be hard-coded as in the current work as it only occurs in the entry and exit stub. It is

used to check that the attacker-provided address indicating where execution should continue at enclave exit does not lie inside the enclave. Furthermore this instruction also ensures that this symbolic continuation address in register **r7** is constrained to an address outside the enclave. This constraining of the continuation register is crucial to avoid reports from the CFSan plugin as the enclave will at a certain point jump to this attacker-provided target address. The hook for `sancus_get_caller_id` can be hard-coded to put a value of zero in the return register. This because we validate the enclave which will always be called by the untrusted world as secure linking cannot yet be validated in the current work. The ID of the caller will thus always be zero.

Up to this point all except two instructions have been discussed. These instructions were added later onto the basic Sancus functionality described in the Sancus paper [28]. An instruction to set stack guards is added along with an instruction to support Aion [7], in particular the `clix` instruction. This instruction will disable interrupts for a certain amount of cycles to enable atomic operations. Both of these instructions are hooked to a function that simply skips the instruction. This is left as an interesting possible future direction that would cover validation of Sancus's availability and interruptability guarantees supported by Aion [7].

3.4 Plugins

With a working symbolic execution engine, the plugins can be added one by one. These plugins are developed based on the breakpoint functionality provided by angr and should therefore be able to work independently of the runtime. In an ideal scenario, they would require no adaptations when adding a new runtime. Yet this is not entirely the case when extending to a new architecture. Next the incorporation of the CFSan plugin and the PTRSan plugin will be discussed. An overview of the features provided by ABISan, that could provide useful vulnerability detection for Sancus is given. Note that the adaptation of this plugin is not included in current implementation. Also note that Pandora originally provides four plugins, but as discussed in 2.5.2 the MSP430 architecture has no support for an elaborate paging mechanism, this plugin is thus irrelevant for the current work.

3.4.1 CFSan

The CFSan plugin validates three invariants for Intel SGX as explained in Section 2.5.2. Not an invariant, but an extra check done by the CFSan plugin validates that an enclave does not jump to non-executable memory. This should not occur in well-developed enclaves and Sancus will even terminate execution when this happens. The CFSan plugin checks for such issues and every occurrence of such a jump will be reported as a critical issue. The `ControlFlowTracker` will subsequently remove these states from the stash of active states. The first invariant checked by CFSan validates that there are no jumps in the enclave to a target address that is inside unmeasured and uninitialized memory. This invariant is specifically developed for Intel SGX

which supports unmeasured and uninitialized pages and is therefore redundant for Sancus enclaves. The remaining two invariants are however important for Sancus.

The second invariant ensures that there are no enclave jumps to an attacker-tainted address when the target address is restricted to lie entirely inside the enclave. If this does occur, the plugin will report this jump as a warning issue. The third invariant confirms that there are no enclave jumps to an attacker-tainted address when this address is restricted to lie outside the enclave. Every occurrence of such a jump will be reported as a critical issue. The invariants above only required a minor change. When an enclave jumps to a target address, it must be checked whether this target instruction lies inside the enclave itself. Instead of calculating these lengths for every jump instruction, the CFSan plugin developed for Intel SGX safely overapproximates this instruction length to 15 bytes, which is the maximum instruction length in x86. MSP430 instructions have a maximum length of 6 bytes, so this parameter has to be changed to avoid issues being overestimated as a critical issue which should actually result in a warning issue. The changes to the CFSan plugin were done in 53 added LOC and 16 removed LOC.

Implicit Enclave Exit

An important finding of this work is the discovery of a new class of possible vulnerabilities that could arise in enclaves developed on the Sancus architecture. This vulnerability emerges from the fact that Sancus enclaves are not enforced to be left through explicit instructions. Unlike for Intel SGX where enclaves have to be left through specific instructions (as discussed in 3.2.2), Sancus enclaves depend on the developer or runtime to explicitly leave enclaves with a jump instruction. If improperly done, this could result in a single basic block crossing the enclave boundary. In other words, from one instruction to another the executing program would be left without CPU protections. In the Pandora code alone this already gives rise to possible improper enclave validation. This can be explained as follows: The different plugins are based on the presence of breakpoints at certain events. If an enclave is left within a single basic block, so without the use of an explicit jump outside the enclave, the breakpoint for a control flow change will for example not be triggered. The validation code would therefore still assume it is executing inside the enclave and for example not trigger exit breakpoints on which the ABISan plugin checks the ABI state.

An example of such an implicit enclave leave is given in Listing 3.1. Note that the labels `__sm_foo_public_start` and `__sm_foo_public_end` indicate the start and end of the text section of the enclave. The `nop` instructions here indicate a 'no-operation' which does nothing. These are added for clarity. The enclave stores the hexadecimal value of **0xdead** in the **r15** register on line 4. Register **r15** is used for the return value so this enclave returns the value **0xdead**. As there is no explicit jump to the end of the enclave (the `__sm_foo_public_end` label), all six instructions in the example belong to a single basic block. As Pandora symbolically executes the program per basic block, this results in an unclear transition from enclave code to non-enclave code. Listing 3.2 on the other hand indicates what is the

expected behaviour of enclaves. On line 5 the enclave will jump explicitly to the label `__sm_foo_public_end`. The example thus contains two separate basic blocks of four and three instructions where the former is contained entirely inside the enclave and the latter (starting at line 8) exists entirely outside the enclave.

This implicit enclave exit was however not (yet) discovered in any compiler-generated enclave. This could be attributed to the fact that the current compiler infrastructure for Sancus does introduce explicit enclave entry and exit routines. Nonetheless this vulnerability can still be introduced for handwritten enclaves and in possible future extensions and adaptations.

```
1 __sm_foo_public_start:
2     nop
3     nop
4     mov #0xdead, r15
5
6 __sm_foo_public_end:
7     nop
8     nop
9     ret
```

Listing 3.1: A basic block crossing the enclave boundary.

```
1  __sm_foo_public_start:
2      nop
3      nop
4      mov #0xdead, r15
5      jmp __sm_foo_public_end
6
7  __sm_foo_public_end:
8      nop
9      nop
10     ret
```

Listing 3.2: An explicit jump outside the enclave.

3.4.2 PTRSan

Just like the CFSan plugin, the PTRSan plugin was developed to be independent of the underlying architecture and runtime. It validates three invariants for both read and write operations. The first invariant ensures that a read or write to an attacker-tainted address is restricted to either inside or outside the enclave. A symbolic pointer that is not constrained to any of the two (inside or outside enclave) could clearly result in undesired behaviour and could be the result of improper constraining of a pointer. This will thus be reported as a critical issue. The PTRSan plugin's second invariant ensures that attacker-tainted read or write targets are not constrained entirely within the enclave. While these type of pointers might indicate benign behaviour, such as an attacker-tainted index access in a data structure within the enclave memory, they are reported as warnings. From here they require further manual inspection. The third invariant validates that there is no non-tainted memory access outside of the enclave. This could result in undefined behaviour and is therefore also marked as a critical issue. All of the issues detected with the PTRSan plugin can also arise for Sancus enclaves. The plugin in itself did not require any changes. This again shows Pandora's strength as the port of the PTRSan plugin to an entirely new architecture requires no changes in the plugin itself.

The plugin depends however on underlying logic that determines whether a symbolic address is constrained to the inside or outside of the enclave address range. As discussed in 3.2.2 a Sancus enclave is not a single contiguous memory block as is the case in Intel SGX. Therefore this underlying logic had to be adapted to

incorporate these differences.

3.4.3 ABISan

The last plugin that could be developed for Sancus is the ABISan plugin. Note that this plugin is left for future developments and the adaptation to Sancus is thus not included in this work. However an explanation of which plugin invariants are useful for Sancus and which are not, is given here. In contrast to the CFSan plugin and the PTRSan plugin, this plugin is not architecture agnostic and is inherently intertwined with many Intel SGX and x86 specifics.

The plugin checks three different invariants. The first invariant ensures the enclave does not read attacker-tainted configuration registers. If this is the case, a critical issue is reported. This invariant would not provide a useful extension for Sancus as the setup of a Sancus enclave does not depend on configuration registers. In Sancus, enclaves are set up using only the general purpose registers. The second invariant checks that at the end of the ABI sanitization phase all registers are properly sanitized. If this is not the case a warning will be reported for every unsanitized register. This invariant would be a useful check for Sancus enclaves. Furthermore they could use the same heuristic as implemented for Intel SGX. According to this heuristic the switch from the low-level ABI sanitization code to the high-level API code occurs at the first `call` instruction inside the enclave. As can be seen in the Sancus `sm_entry` stub in Appendix B, the first (and only) `call` instruction jumps to the specified enclave function.

The last invariant provided by ABISan plugin ensures that on enclave exit, a specified set of registers contains attacker-tainted values. Moreover the data registers should only contain symbolic values if they are attacker-tainted. For example the stack pointer should be attacker-tainted at the moment the enclave exits. If this is not the case, this would indicate that the address of the private enclave stack is still present. In Intel SGX the **rbx** register should also be attacker-tainted at the moment of enclave exit as at this point it should contain the continuation address. For Sancus this is the **r7** register. If these data registers do not contain attacker-tainted values at the point of enclave exit, a warning would be reported. The data registers that are not tainted should also not contain symbolic values as this could indicate that enclave secrets are still present in these registers. If these data registers would not contain untainted symbolic values a warning would be reported as well.

3.5 Crashed States

During development it became clear that symbolic execution of the *minimal-sancus-example* enclave did not behave as expected as the symbolic execution always had to be restricted to a certain number of execution steps. If not restricted, the symbolic execution would run indefinitely. Some tracing of symbolic execution paths made clear that this was intended behaviour and was caused by the entry stub. A precise analysis of the cause is given in Section 3.5.1 and the implemented solution is described in Section 3.5.2.

3.5.1 Sancus Entry Stub

As discussed in Section 3.2, entering an enclave can only be done through specific entry points to which the enclave can be entered. As it is not advised to impose the developer with the task of managing these entry points, a compiler generated entry stub lifts the burden from the developers and takes on this task. An initial version of this entry stub is described in [37]. Over time this entry stub has been extended, and compared to the initial version (v1.0.0) of this entry stub, the current version (v2.1.0) contains a lot of added functionality such as the addition of Aion [7] which allows for availability and real-time guarantees in Sancus. These extensions also include the addition of a proper error handling routine in case violations occur during the execution of the entry stub or when invalid arguments were provided to the stub.

The error routine is shown in Listing 3.3. This routine first and foremost clears the base address of the Secure State Area (SSA). The SSA is the secure location in memory where the data of the enclave is stored. Clearing this address ensures that the private enclave memory can no longer be accessed. Next the stub will try to overwrite the text section of the enclave. This is a violation that will be detected by the MAL. Consequently it will result in a reset of the processor. From here the execution will come to a halt as the processor is reset. For safety precautions an infinite loop is placed at the end of this routine such that the control flow would never be able to proceed any further.

```
1 .Lerror:
2     ; caller provided poisoned arguments -> trigger an intentional
      violation by illegally writing to the SM text section
3     ; NOTE: we don't support nested ecalls, so we can simply leave the
       SM internal state untouched and ready for a new ecall
4     ; NOTE: clear SSA adddress so enclave-internal state is not
      touched
5     mov #0, &__sm_ssa_base_addr
6     mov #1, &__sm_entry
7     ; should never reach here
8 1:
9     jmp 1b
```

Listing 3.3: The error routine in the Sancus entry stub (v2.1.0).

3.5.2 Removing Infinite Looped States

The Sancus entry stub leverages the developers with functionality such as scrubbing registers. As clear by now, errors can occur during the execution of this stub. These errors are handled by the error routine shown in 3.3. During the symbolic execution of compiler-generated enclaves, some states will thus effectively end up in this error state. This because as explained in 2.4 instead of simulating a single path through the enclave, the symbolic execution will branch and split a state in multiple states, each traversing a separate path. In the entry stub, every conditional jump to this routine will effectively result in a symbolic state that ends up simulating this error

stub. As Pandora is not aware of the hardware crashes, a write to the text section will not automatically result in an errored symbolic state. Accordingly has as effect that some states in the symbolic execution end up looping indefinitely.

Pandora uses angr's stashing mechanism to keep track of all states executing inside the enclave. If for example states leave the enclave they can be moved from the active stash to another stash indicating that these states do not need to be simulated any further. Consequently, the symbolic execution of these enclaves will never end as the states stuck in an infinite loop will never be removed from the stash of active states. To overcome this, multiple solutions are possible:

Parse the backtrace A possible way to overcome this issue, can be to devise a mechanism that automatically detects when an infinite loop is executing. There are multiple ways in which this can be implemented. A first possible method could check the backtrace of basic blocks for repeating occurrences. This backtrace contains a list of basic blocks that have executed up until that point. An infinite loop would thus result in multiple equal blocks at the top of the backtrace. A challenge with this solution is quantifying how many equal blocks indicate an infinite loop. Concrete: how to distinguish a finite loop from an infinite loop based on the amount of iterations it has done. As this can be non-trivial other solutions might be better suited.

Parse and hook object dump A second solution for automated detection of infinite loops, and the original solution implemented, is to identify the infinite loops in advance and accordingly anticipate on them. This was initially implemented as follows: before the start of the symbolic execution, the object dump will be parsed and checked for the occurrence of infinite loops. Manual inspection of the object dump indicated that the infinite loop is always executed as a `jmp` instruction with a relative offset of zero. Therefore the code could parse out all the addresses where this operation occurs. Next the hooker would hook a dedicated function to these addresses. This hooked function marks the state with a global boolean variable `infLoop`. The `BasicBlockExplorer` executing steps for all states would then filter all states to a separate stash based on the presence of this global variable.

Detect write inside text section In contrast to automatic loop detection, another possibility is to automatically detect writes to the text section of the enclave and move these states to the stash with errored states. This can be done with the help of angr breakpoints. This solution is implemented in the current design. Similar to how the plugins use angr's breakpoint functionality, a breakpoint can be triggered for every write inside the enclave. This executes a callback function that checks if the current write is to the text section of the enclave. If it is, it marks the state with global variable `trusted_mem_write`. Just like in the scenario

above, the `BasicBlockExplorer` will filter these marked states from the active stash to the errored stash.

Note that a choice between the latter two options is a trade-off. Parsing and hooking the object dump is an effective and efficient solution for the problem of infinite loops in symbolic execution. This solution would immediately resolve all infinite loops (of this specific 1-instruction format) everywhere inside the enclave. However this solution does not adhere to one of the main goals of Pandora: *truthful* symbolic execution. This can be defined as aiming to mimic the hardware as close as possible. The latter solution, using breakpoints on writes inside the enclave, does achieve this goal and is therefore the solution implemented in current design.

3.6 Conclusion

Adaptation and porting of a codebase like Pandora to a new architecture is not trivial, especially considering the fact that angr's support for MSP430 is not as mature as for an architecture like x86. Starting from a proof-of-concept, some limitations in the MSP430 angr backend could be found early on. This also resulted in some contributions to the public *angr-platforms* repository. The next step was to start developing the symbolic execution for Sancus enclaves. Based on the differences between Intel SGX and Sancus, the codebase of Pandora could meticulously be adapted.

For the development itself a lot of changes had to be made to the Pandora codebase. Some major adaptations were illustrated such as the development of a Sancus-SDK that contains all runtime or architecture specific logic. Another step required the hooking of all Sancus instructions as these are not supported by the angr backend. While it would be a nice feature to have the hooks implemented, a lot of the work would go unnoticed for the goal of validation as the enclaves itself hardly make use of these instructions itself. With a working symbolic execution engine the plugins could be added. For the CFSan plugin a new type of issue is introduced when a basic block crosses the enclave boundary. These types of issues could in the first place undermine the validation done by other plugins such as the ABISan plugin. While the addition of the PTRSan plugin did not require many changes, the ABISan plugin is inherently intertwined with Intel SGX specifics. Therefore, the development of this plugin is left for future work. However, an overview of its functionalities that would be useful for Sancus is provided.

Another interesting development arose from the observation that in the symbolic execution some states would end up looping indefinitely. It was found that this behaviour could be expected for paths of the symbolic execution that ended up in the error routine of the Sancus entry stub. By adhering to the goal of *truthful* symbolic execution, states that write to the text section of the enclave will automatically be removed from the actively tracked states as these states would result in a crash on real hardware.

Chapter 4

Evaluation

Evaluating the Pandora's performance on test enclaves is crucial to understand its effectiveness. This chapter discusses its vulnerability discovering abilities and limitations on the basis of a wide range of test enclaves. Pandora's plugins cover an extensive variety of vulnerabilities, but to write test cases for all different issues can be a challenging task when the compiler automatically introduces entry and exit stubs that aim to prevent vulnerabilities. Handwritten assembly enclaves provide a solution for this challenge. An extensive unit test framework has been developed thanks to these handwritten enclaves as they are used to achieve a large coverage of all issues discoverable with the plugins. Some examples are discussed in Section 4.1. This unit test framework enforces trust in Pandora's functionality but it is not the goal of this work. Pandora is developed to automatically discover vulnerabilities in higher-level Sancus enclaves developed in C. Hence, validating compiler-generated enclaves is an essential goal. Section 4.2 will demonstrate how Pandora can autonomously rediscover vulnerabilities from previous versions of the enclave entry stub. Section 4.3 then reproduces some vulnerabilities discovered throughout different Sancus versions and extensions. Pandora proves to autonomously detect and reports on all these vulnerabilities.

4.1 Unit Test Framework

Sancus has already been around for over 10 years. Meanwhile its toolchain has reached a certain level of maturity. The compiler generated stubs facilitating enclave entry and exit have accordingly become more complex. This makes testing Pandora functionalities a challenging task. Handwritten enclaves offer a solution to this challenge. In contrast to the compiler-generated enclaves, handwritten enclaves enable swift crafting of specific vulnerabilities. Moreover for the development of the symbolic execution engine and the discovery of bugs in the angr backend, small test enclaves are a useful resource as they do not necessarily include the complex stubs. This section discusses some basic assembly enclaves that give a more intricate insight into the vulnerabilities discoverable with Pandora. First some example test cases for the CFSan plugin are discussed in Section 4.1.1, continued with test cases the

PTRSan plugin in Section 4.1.2.

As the examples presented in this section are written in MSP430 assembly, the most important details and instructions to follow along will be described. A complete summary covering the entire ISA can be found at [3].

For compiler-generated enclaves, appropriate labels indicating the endpoints of the enclave will be automatically generated. These labels are:

`__sm_[SM_NAME]_public_start` and `__sm_[SM_NAME]_public_end`

for the begin and end of the enclave text section. For the enclave data section the labels

`__sm_[SM_NAME]_secret_start` and `__sm_[SM_NAME]_secret_end`

are used. In what follows, these labels will thus indicate the enclave boundaries. An overview of all test cases for each plugin, combined with a small description on the tested behaviour is given in Appendix C.

4.1.1 CFSan Tests

One of the checks done by the CFSan plugin ensures that an enclave does not jump to an attacker tainted address that is not restricted to either the inside or the outside of the enclave. Listing 4.1 is an example of such a vulnerable enclave. The `br` instruction results in a jump to the provided address. The enclave will thus jump to **r15**. As there are no constraints imposed on the value of register **r15**, this will result in a jump to an attacker provided address. The CFSan plugin reports this as a critical issue: *Symbolic unconstrained tainted jmp target.*

Listing 4.2 illustrates an example of a jump constrained within the enclave. The example uses absolute addresses, although relative addresses (using labels) could be used as well. A subtlety in the CFSan plugin is that it always overapproximates the target instruction to be six bytes long, as this is the maximum length of a MSP430 instruction. This approach avoids the need to compute the size of the target instruction for each jump. The absolute addresses can be obtained with the help of the object dump tool. On line 4 the address **0x6c1a** is compared against the register **r15**, using the `cmp` instruction. All paths that assume **r15** lower than this address will leave the enclave, with the `jl` instruction. The remaining paths will compare the register **r15** to value **0x6c1e** which is also an address inside the enclave. Similarly, all paths that assume **r15** greater than or equal to this address will leave the enclave (with instruction `jge`). The remaining paths will thus have **r15** constrained to values between **0x6c1a** and **0x6c1e**. Next the `br` instruction will jump to the address inside **r15**. This jump is thus constrained inside the enclave itself as all paths that assumed **r15** to be lower or greater than these addresses have left the enclave. This will be reported by the CFSan plugin as a warning: *Symbolic jmp tainted target in enclave memory.*

For the CFSan plugin in total 21 assembly test cases were written of which 8 cases test normal enclave behaviour (so without reports generated by the CFSan

plugin). The 13 other cases test specific vulnerabilities detectable with the CFSan plugin. For all cases the CFSan plugin behaves as expected.

```
1  .text
2  __sm_foo_public_start:
3  enter_foo:
4      br r15
5
6  __sm_foo_public_end:
7      ret
8
9  .data
10 __sm_foo_secret_start:
11 __sm_foo_secret_end:
```

LISTING 4.1: An enclave jumping to an attacker provided address.

```
1  .text
2  __sm_foo_public_start:
3  enter_foo:
4      cmp #0x6c1a, r15
5      jl end
6
7      cmp #0x6c1e, r15
8      jge end
9
10     br r15
11
12     nop ;Address 0x6c1a
13     nop
14     nop ;Address 0x6c1e
15     nop
16     jmp __sm_foo_public_end
17
18 __sm_foo_public_end:
19 end:
20     ret
21
22 .data
23 __sm_foo_secret_start:
24 __sm_foo_secret_end:
```

LISTING 4.2: An enclave jumps to an attacker tainted address constrained to the inside of the enclave.

4.1.2 PTRSan Tests

The PTRSan plugin checks invariants for every memory read and write. Listing 4.3 gives an example of a write to a non-attacker-tainted target address. On line 4 the address **0x1000**, an address outside the enclave, will be put in **r15**. The `mov` instruction can move values between registers and memory locations. Then on line 5 the value of **0x1** will be written to the address inside **r15**. As this address lies outside the enclave, a critical issue will be generated: *Non-tainted write outside enclave.*

Listing 4.4 presents an instance of an enclave that reads an attacker tainted address inside the enclave. This example acts in a similar way as Listing 4.2 where the attacker-tainted value inside **r15** will first be constrained to lie within the address range of the enclave. On line 10, the value at the address stored in register **r15** will be moved to the register **r14**. This will be reported as a critical issue: *Unconstrained read.*

```
.text
__sm_foo_public_start:
enter_foo:
    mov #0x1000, r15
    mov #0x1, @r15
    nop
    nop
    nop
    jmp __sm_foo_public_end

__sm_foo_public_end:
    ret

.data
__sm_foo_secret_start:
    .space 64
__sm_foo_secret_end:
```

Listing 4.3: An enclave writing to a non-tainted address that may lie outside the enclave.

```
.text
__sm_foo_public_start:
enter_foo:
    cmp #0x6c0c, r15
    jl end

    cmp #0x6c24, r15
    jge end

    mov @r15, r14
    jmp end

__sm_foo_public_end:
end:
    ret

.data
__sm_foo_secret_start:
__sm_foo_secret_end:
```

Listing 4.4: An enclave reading an attacker tainted address inside the enclave.

For the PTRSan plugin in total 15 assembly test cases have been developed of which 4 test the normal enclave behaviour. The remaining 11 cases test specific vulnerabilities detectable with the PTRSan plugin. For all these test cases the PTRSan reported vulnerabilities as expected.

4.1.3 C Enclaves

While handwritten assembly enclaves confirm the functionalities of the plugins, the goal of Pandora's adaptation is to validate *'real'* compiler-generated enclaves. This section explains some basic enclaves written in C that can be validated due to this work. It is also an appropriate place to highlight some existing limitations. As a starting point, the *minimal-sancus-example* enclave is validated. This minimal enclave was mentioned before in Section 3.1.1 and is elaborated on in Appendix A. Next a more complex example is validated. This enclave also highlights some limitations that can be met during validation.

Minimal Sancus Enclave

A perfect starting point for validating enclaves written in C is the *minimal-sancus-example* enclave. This enclave contains only a single entry point function which will simply return the integer, passed as an argument, incremented with one. For this example no issues are found by the CFSan plugin. There are however two warnings reported by the PTRSan plugin, which both are *attacker tainted read inside enclave* issues. To get a deeper understanding of these reported issues, Appendix D contains the report generated for this example. Both of the issues reported by the PTRSan

plugin originate from the `__sm_basic_enclave_entry` label which is the entry stub generated by the compiler.

As indicated in the report, these warnings require further manual inspection. The specific issues both refer to a basic block where the register **r6** is used as an index into the `__sm_table`. This table enables multiple logical entry points through a single physical entry point. The report also indicates that the value of register **r6** is constrained to the value of zero. The enclave does contain an entry point, which will be stored at index zero in the table. The second generated warning again indicates an index into the `__sm_table`. These indices into the `__sm_table` can be seen in the entry stub shown in Appendix B.

An Elaborate Enclave Example

While the *minimal-sancus-example* enclave already gave a proper first indication of the power of Pandora, validating a more challenging enclave would naturally give a more realistic view into its abilities. An example of such an enclave is given in Listing 4.5. This example has two defined enclave entry points, an enclave function and some enclave private data.

The example also indicates some limitations that are present in the current implementation of Pandora. A first limitation is that enclave reentry is not supported in the current implementation. What this means is that for each path in the code where an ocall occurs, the validation will not continue past this ocall. The validation process thus stops at the point where the exit stub jumps back to the untrusted context. An example of this limitation can be seen on line 8 where the function `pr_info` is called. This function is not defined within the trusted context of the `basic_enclave` and thus the exit stub will be executed at which the validation process stops. The return statement on the following line and all the other instructions following it, will be left unvalidated. Another limitation can be seen in the `do_loop` function. As indicated in Section 2.4 symbolic execution can quickly become intractable when phenomenons such as path explosion occur. When a loop is present in a program, the symbolic execution process can quickly run into time and memory limitations such as unsoundness of the validation.

```
1  int SM_DATA(basic_enclave) priv_data = 3;
2
3  int SM_FUNC(basic_enclave) set_priv_data(int number)
4  {
5      if (number > 0) {
6          priv_data = number;
7      } else {
8          pr_info("Not possible to set priv_data lower than 0");
9          //no path in the validation will reach here as reentry is not
       yet supported
10         return -1;
11     }
12     return 0;
13 }
14
15 int SM_ENTRY(basic_enclave) do_loop(int *buffer, size_t length)
```

```
16 {
17     int result = 0;
18     for (size_t i = 0; i < length; i++) {
19             result +=  buffer[i];
20     }
21     return result;
22 }
23
24 const char* SM_ENTRY(basic_enclave) set_number(int amount)
25 {
26     int res = set_priv_data(amount);
27     if (res >= 0) {return "Number set";}
28     else {return "Number not set";}
29 }
```

Listing 4.5: A more elaborate test enclave.

The validation of this enclave results in a warning generated by the CFSan plugin and two warnings and a critical issue generated by the PTRSan plugin. The CFSan warning indicates a *symbolic call tainted target in enclave memory* which arises from the fact that there are multiple entry points in this enclave. In warning arises because the enclave contains multiple entry points. Just like in the warnings generated for the *minimal-sancus-example*, the register **r6** is used as an index into the `__sm_table`. In this case the table will contain more than one entry. This means that instead of a concrete value, the address in register **r6** will be a symbolic constrained value. This means that the value will be restricted to a range of values spanning the different entry points in the enclave. The enclave will subsequently jump to the address fetched from this `__sm_table`. Therefore a warning is generated.

For the PTRSan plugin both generated warnings originate from the entry stub. Just like in the *sancus-minimal-example* they both refer to an attacker-tainted read inside the enclave. This is due to the entry point ID provided by the untrusted runtime in register **r6**, which is used as an index into the `__sm_table`. The critical issue on the other hand originates from the `do_loop` function. This function executes a read of arbitrary memory locations, passed by the `buffer` pointer, without doing any checks on this pointer. Line 19 can thus read any arbitrary value including private enclave secrets. Therefore a critical issue is generated.

In total 19 enclaves developed in C were written, including the ones described in the sections below.

4.2 Validating Compiler Stubs

While the current implementation encounters some limitations, it nonetheless provides a useful tool in enforcing the security of Sancus runtimes. Figure 4.1 plots the size of the entry and exit stub throughout different Sancus versions. It clearly indicates a significant increase in size. As shown in [40], previous versions of the enclave entry stub contained critical vulnerabilities. Manually tracing these stubs requires a lot of insights and time. Pandora leverages the developer of this burden. This section

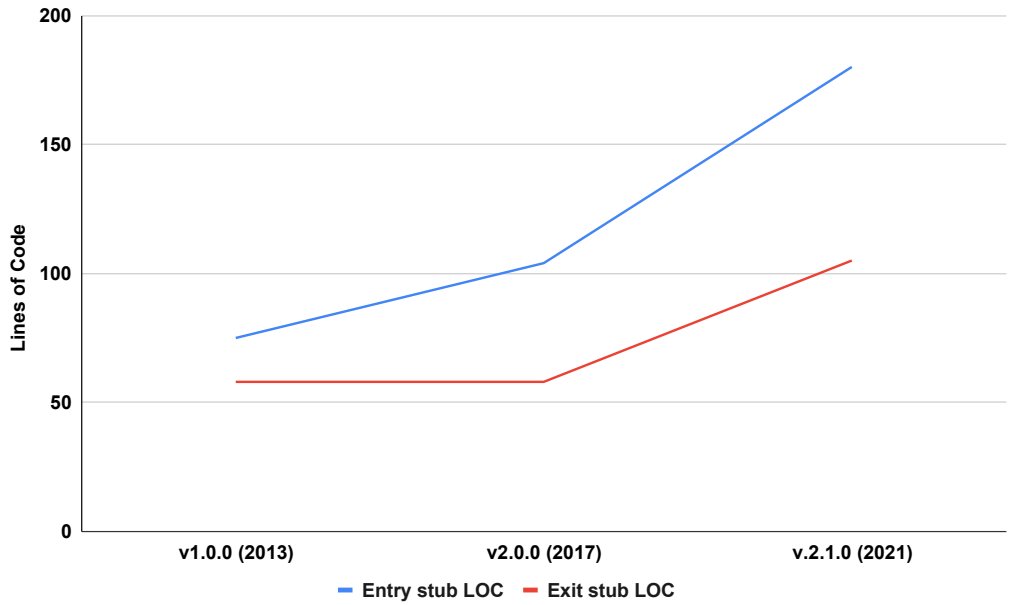


FIGURE 4.1: Enclave stub sizes for different Sancus versions measured in lines of code.

will test multiple older versions of the Sancus entry and exit stub with the aim to rediscover vulnerabilities in these versions.

Validating the Sancus stubs requires a test enclave in which all stubs can be encountered in the symbolic execution. This enclave is given in Listing 4.6. Line 5 will ensure the exit stub is included in the validation process as this ocall will leave the enclave through the exit stub.

```
1 int SM_ENTRY(basic_enclave) basic_function(int arg)
2 {
3     if (arg < 0)
4     {
5         pr_info("This branch will take the exit stub");
6         return 0;
7     }
8     return 1;
9 }
```

LISTING 4.6: Small enclave to test multiple versions of the entry and exit stubs.

As a starter, version 2.1.0 is validated. For this version, only two warnings are generated by the PTRSan plugin. An inspection of these generated warnings indicates that these warnings are the same as discussed in Section 4.1.3 for the *minimal-sancus-example* enclave. As can be seen in Table 4.1, these warnings are reported for every version, and will not be further elaborated upon. It must be noted that although no real vulnerabilities were found by Pandora for version 2.1.0, this

does not necessarily mean that there are no vulnerabilities present in these stubs. This not only because the ABISan plugin is not yet implemented, but also because other vulnerabilities, not covered by the plugins of Pandora could be present in the stubs. It does however guarantee that the invariants covered by the CFSan plugin and the PTRSan plugin hold for these stubs.

Table 4.1: Reported issues with different versions of Sancus stubs in the validation of Listing 4.6.

Stub version	#CFSan warning	#CFSan critical	#PTRSan warning	#PTRSan critical
1.0.0	1	1	2	1
2.0.0	1	1	2	1
2.1.0	0	0	2	0

In what follows, the stubs in the compiler are replaced with their previous version 2.0.0 and version 1.0.0. It is clear that both versions encounter the same amount of vulnerabilities. Reviewing them has shown that these issues address the same issues. Furthermore all vulnerabilities discovered arise in the Sancus entry stub. Following explanations thus apply to both stub versions. The reports generated for version 2.0.0 are shown in Appendix E.

The critical issue indicated by the CFSan plugin indicates a *symbolic unconstrained tainted jump target*. This indicates a possible jump to an arbitrary target address within (or outside) the enclave. Moreover the jump is not constrained and the target range can span the entire memory range of the MCU: **[0x0,0xffff]**, thus including the address range of the enclave. This vulnerability confirms the findings addressed by Van Bulck et al. [40]. A deep dive into the generated report indicates that a return from an earlier ocall is simulated as the **r6** register contains the value **0xffff**. The enclave entry stub documentation indicates that this is used to indicate a return from an earlier ocall in contrast to normal entry where the **r6** register should contain the ID of the targeted entry point in the enclave. The special value **0xffff** is used to indicate reentry. The issued path in question simulates this reentry, but does not place any constraints on the **r7** register. This register should contain the continuation point where the code should resume at enclave exit. As the stub does not check whether this address belongs to the caller itself, a jump to this register **r7** is a vulnerability. The attacker can thus trick the enclave into jumping to any arbitrary address. The enclave itself executes the jump, so an attacker can even specify addresses within the enclave as target address.

Both the warning generated by the CFSan plugin and the critical issue generated by the PTRSan plugin originate from the `__sm_foo_ret_entry` stub which is executed to return from a previous ocall. Just like the jump to an unsanitized **r7** register value, Van Bulck et al. [40] discovered that both version 2.0.0 and version 1.0.0 of the stubs suffer from a vulnerability where a thread waiting for a return from an ocall can be entered by setting specific register values before entry. In this reported case, a reentry is simulated where no previous ocall occurred. The routine that facilitates this return tries to restore the register values as before the ocall. However as there did not occur any ocall, the enclave will pop under the stack and

arbitrary values from these memory locations will be placed inside the registers. At the end of this return stub a `ret` instruction will be executed, but as the enclave previously already popped under the stack, a value under the stack will be fetched. The enclave will try to return to this address, which is simulated by **0x0** as the secret enclave memory will be initialized with zeroes.

The warning generated by the CFSan plugin indicates a *concrete ret target in non-executable memory* which can be explained by the fact that the enclave tries to return to the address **0x0** as popped underneath the stack. This also explains the critical vulnerability indicated by the PTRSan plugin. The enclave will execute a *non-tainted read outside enclave* as the enclave will try to read at this address **0x0**.

4.3 Reproducing 'Tale of Two Worlds' Vulnerabilities

Section 2.3 made clear that a wide range of vulnerabilities arises in different enclave architectures and runtimes. A lot of them were applicable on the Sancus architecture. An ideal way to show Pandora's power is to let it autonomously rediscover these vulnerabilities. This section will reproduce the vulnerabilities described in [40] and let Pandora autonomously detect and report on these vulnerabilities. The first vulnerability for Sancus discovered in this paper has already been reproduced as these vulnerabilities were found in the enclave entry stub and these are described in Section 4.2. The concrete vulnerability allows enclaves to reenter an enclave thread that was not waiting for ocall return. Also by setting registers to trick an enclave into returning does not properly sanitize the continuation address enabling an attacker to provide an enclave address such that on enclave exit the enclave jumps to this specific attacker-tainted address inside the enclave. The last vulnerability applicable to Sancus described in this paper will however not be covered as this vulnerability should be detected with the ABISan plugin. This vulnerability allows to leak enclave secrets through padding bytes introduced in structs.

4.3.1 Authentic Execution

Another vulnerability was discovered in a Sancus extension developed to support authenticated message passing. The enclave has to decrypt encrypted payloads while copying them inside the enclave. This payload was not sanitized. Listing 4.7 shows the vulnerable enclave code. This listing does not exploit the compiler-generated code, but accurately recreates the vulnerability. On line 12 the `sancus_unwrap_with_key` is called with the unsanitized `payload` argument. This method will consequently decrypt the values from this `payload` and write them into the `input_buffer` inside the enclave. This instruction uses the `sancus_unwrap` instruction and as described in Section 3.3.2 this hook was only partially implemented. This partial implementation copies the required arguments inside the enclave `input_buffer` but does not yet decrypt and checks these arguments. This thus effectively triggers the required breakpoints for memory reads and writes. Pandora was able to report unconstrained reads and unconstrained writes inside the enclave and can thus autonomously detect these vulnerabilities.

```
void SM_ENTRY(basic_enclave) __sm_handle_input(uint16_t conn_id,
                const void* payload, size_t len)
{
    if (conn_id >= SM_NUM_INPUTS)
        return;

    const size_t data_len = len - AD_SIZE - SANCUS_TAG_SIZE;
    const uint8_t* cipher = (uint8_t*)payload + AD_SIZE;
    const uint8_t* tag = cipher + data_len;

    uint8_t* input_buffer = alloca(data_len);
    if (sancus_unwrap_with_key(__sm_io_keys[conn_id], payload, AD_SIZE
    ,
                               cipher, data_len, tag, input_buffer))
    {
        __sm_input_callbacks[conn_id](input_buffer, data_len);
    }
}
```

Listing 4.7: The vulnerable code that decrypts an unsanitized payload.

4.3.2 Soteria loader enclave

An additional vulnerability was discovered in a trusted loader enclave [20] developed for supporting lightweight code confidentiality and integrity. The vulnerable enclave is shown in Listing 4.8. This code does not exploit the vulnerability, but rather accurately recreates the original vulnerable code. The vulnerability in this enclave arises in lines 3 through 6. The untrusted context passes an enclave layout struct to this enclave which is then immediately accessed inside the `sm_loader_load` method. This method does not execute any bounds checking on the values fetched from this struct. Pandora was able to discover these vulnerabilities indicating four unique critical *unconstrained read* issues. Manual inspection confirmed that these issues arise indeed from the accesses to the `SancusModule` argument.

```
int SM_ENTRY(sm_loader) sm_loader_load(struct SancusModule *scm)
{
    size_t pstart  = (size_t)scm->public_start;
    size_t pend    = (size_t)scm->public_end;
    size_t pcstart = (size_t)scm->public_start;
    size_t pcend   = (size_t)scm->public_end;
    size_t i;
    int ret;

    // check boundaries
    if (pend < pstart  pcend < pcstart)
        return 0;

    // check sizes
    if ((pend - pstart) != (pcend - pcstart))
        return 0;

    //...
```

```
19 }
```

Listing 4.8: The vulnerable Soteria loader enclave.

4.3.3 Vulnerable `sancus_is_outside_sm`

A last reproduced vulnerability arose in a previous version of the `sancus_is_outside_sm` function. Listing 4.9 shows an example enclave using this vulnerable function. Line 1 to line 6 shows the vulnerable macro that defines whether a buffer lies entirely outside an enclave. The vulnerability arises as the macro only checks for the endpoints to lie inside the enclave. However a buffer that includes the entire enclave address range, so a buffer that starts before and ends after the boundaries of the enclave will properly pass this `sancus_is_outside_sm` function. Furthermore, passing a length argument that lets the calculation for the buffer end address wrap around the address space will not properly be checked. The enclave first ensures that the buffer has no overlap with this vulnerable function on line 11. If no overlap is reported, the enclave will copy data from that buffer inside the enclave on line 16. Pandora's PTRSan plugin was able to report this vulnerability autonomously. Moreover a similar enclave using the patched version of the `sancus_is_outside_sm` was also validated with Pandora. Pandora did confirm that the vulnerabilities were properly patched as these vulnerabilities did no longer arise in the patched version.

```
1  #define __OUTSIDE_SM( p, sm ) \
2      ( ((void*) p < (void*) &__PS(sm))  ((void*) p >= (void*) &__PE(sm)
       ) ) && \
3      ( ((void*) p < (void*) &__SS(sm))  ((void*) p >= (void*) &__SE(sm)
       ) )
4
5  #define sancus_is_outside_sm_vulnerable(sm, p, len) \
6      ( __OUTSIDE_SM(p, sm) && __OUTSIDE_SM((p+len-1), sm) )
7
8  void SM_ENTRY(basic_enclave) copy_data_from_buffer(int *buffer, int
       length)
9  {
10     //vulnerable function
11     if (!sancus_is_outside_sm_vulnerable(basic_enclave, buffer, length
       *2)) return;
12
13     for (int i = 0; i < length; i++)
14     {
15         //access the data
16         int result = buffer[i];
17     }
18     return;
19 }
```

Listing 4.9: An enclave checks arguments with the vulnerable `sancus_is_outside_sm`.

4.4 Conclusion

To evaluate Pandora, we developed an extensive Unit Test framework. Testing the entire range of vulnerabilities that Pandora can autonomously discover can be non-trivial. Therefore a total of 21 handwritten assembly enclaves extensively test the CFSan plugin while 15 assembly enclaves confirm that the PTRSan plugin delivers to its promises. Pandora validated not only handwritten assembly enclaves but also higher-level enclaves. Although Pandora still encounters limitations when faced with more elaborate enclaves such as no support for enclave reentry, this does not impede even more in-depth testing of Pandora. Pandora autonomously rediscovered vulnerabilities from older versions of the Sancus entry stub. Moreover Pandora proved to autonomously detect earlier discovered vulnerabilities throughout different Sancus extensions and versions. Just like in the previous tests, Pandora confirmed the presence of these vulnerabilities.

Chapter 5

Conclusion

Trusted execution environments have seen considerable developments in recent years. Among these developments, the ecosystem for many TEE architectures has been widely extended. Although TEEs provide many security guarantees, research has pointed out that enclaves can still be vulnerable to multiple attack vectors. While a lot of research has been directed towards the development of automated validation techniques that leverage the developer from the burden of manual vulnerability discovery, a lot of this research has been directed towards high-end TEE architectures such as Intel SGX. Meanwhile the landscape of TEE architectures for low-end embedded devices such as Sancus has often been neglected in this research. This thesis tries to bridge this gap by porting Pandora, a tool developed for automatic validation of Intel SGX enclaves, to Sancus.

Section 5.1 examines the limitations that this work still encounters in greater detail. It also highlights some challenges encountered throughout the development that did require substantial effort which is possibly not represented in the text. The discussed limitations also form a segue into Section 5.2 which further explores possible future research directions. To conclude, Section 5.3 highlights the contributions of this master's thesis.

5.1 Limitations & Challenges

I first want to highlight some challenges that were encountered in this thesis that did not shine through in the writing of this thesis. Many parts of the implementations might seem trivial, but a lot of work occurred under the hood. For example the issues discovered in the MSP430 angr backend required many hours of debugging. These debugging efforts only resulted in some small lines of code. The importance of these additions however cannot be underestimated and were pushed upstream, available for future research. Furthermore, although symbolic execution is an extremely powerful technique, a lot of time was spent manually tracing symbolic execution paths at the binary level which is a non-trivial task.

In what follows, some limitations present in this work are acknowledged:

Validate single enclaves only A first limitation of the adapted Pandora codebase is that only files containing a single enclave can be validated. When encountering binaries containing multiple enclaves, the Pandora code will no longer work as expected. For entirely isolated enclaves this does not immediately form a problem. For enclaves that depend on other enclave code modules however, this is a real limitation. This also has as a consequence that the secure linking process cannot be validated. The original Pandora also allows validation of a single enclave only, but as Intel SGX enclaves are entirely isolated, this is not a problem there.

No enclave reentry A second limitation present is that enclave reentry is not supported. All enclaves containing ocalls to the untrusted environment will not be validated beyond the outgoing call.

Incomplete implementation of hooks A minor limitation is that the hooks for Sancus instructions are not implemented. When encountering Sancus instructions in the enclave itself, Pandora might no longer adhere to the principle of *truthful* symbolic execution depending on the specific instruction. However as discussed in 3.3.2 this is only a minor limitation.

ABISan validation Another limitation is the fact that the ABISan plugin is not implemented. An initial analysis about which ABISan invariants would provide a useful feature for validation of Sancus enclaves is however given.

Pandora's limitations To finalize, as this work is an adaptation of the Pandora program, it also inherits its limitations. This means that incomplete code coverage can be a side effect when encountering the path explosion phenomenon. This is however less of an issue for smaller embedded Sancus enclaves than for the original implementation towards Intel SGX. Moreover, angr in itself is not guaranteed to be sound. Another limitation inherited from Pandora arises when encountering encrypted code. This can be overcome when provided with the decryption key, but this is not the main goal of Pandora as the developer would run Pandora on the non-encrypted binary. Another limitation is that although Pandora has proven its worth by discovering and rediscovering vulnerabilities, there is only a limited ground truth regarding enclave bugs to measure the completeness of Pandora.

5.2 Future Work

The work in this thesis opens a pathway for multiple future research directions. One limitation of the current implementation is that it does not include the ABISan plugin. This can be part of future extensions of the Pandora-Sancus codebase. The Pandora-Sancus adapted codebase is currently not merged into the Pandora codebase for Intel SGX enclaves. This can also be done as an initial step to include even more different enclave architectures. Furthermore an interesting future research direction

could target to validate the secure linking process of Sancus enclaves. This would also require an adaptation to support multiple enclaves in the validation process. In contrast to secure linking, another direction could target the validation of Aion [7], which extends Sancus with interruptability and availability guarantees. Possible future directions are not limited to the realm of Sancus enclaves. As this work has done some essential efforts in supporting the symbolic execution of the MSP430 architecture, future developments could aim to support other enclave architectures developed on top of the MSP430 architecture such as IPE [22, 21], VRASED [29] and SMART [19]. Extension to the IPE architecture might be similar to the current work [10] while VRASED and SMART, oriented towards remote attestation, might provide interesting results when validating the remote attestation process. VRASED's remote attestation process is formally verified which might provide interesting results as it is indicated that gaps between formal models and the real-world design might exist [11].

5.3 Contributions

This master's thesis targeted the validation of Sancus enclaves. The path taken to reach this goal encountered many challenges. The following contributions were thereby made:

- The Pandora codebase is ported to support validation of Sancus enclaves. This reinforces trust in the security guarantees that Sancus provides. More specifically it guarantees the absence of well-known control flow and pointer issues.
- To enable symbolic execution of Sancus enclaves, a proper support for the MSP430 architecture in angr was required. For this thesis I contributed to the *angr-platforms* backend by fixing present issues. These contributions were pushed upstream.
- An extensive unit test framework with 21 assembly enclave tests specifically directed towards testing the functionality of the CFSan plugin and 15 assembly enclave tests for the PTRSan plugin. In total 19 enclaves written in C were developed.
- Autonomous rediscovery of previously found vulnerabilities in older versions of the Sancus runtime stubs. Moreover Pandora is able to autonomously detect reconstructed vulnerabilities that arose throughout the Sancus codebase described in earlier research.

The code developed for this thesis, including the unit test framework is open-sourced and can be found in: https://github.com/Gert-JanG/pandora-sancus/tree/main

Appendices

Appendix A

Minimal Sancus Example

This appendix contains a minimal Sancus enclave example (Listing A.1) as mentioned in 3.1.1. This example starts with a MACRO `DECLARE_SM` that initializes a struct containing the layout of the module such as the start and end address of the text and data section. Next an enclave function and entry point is defined. This can be done with an annotation `SM_ENTRY(name)` to indicate to which enclave this function belongs. In the main function the protections for the secure module are enabled with the `sancus_enable` instruction. This activates the hardware protection for the module with the defined layout. Then finally the enclave function `plus_one` will be called. This module is minimal in that it does only essential basics of enclave functionality. Enclave destruction, enclave reentry. etc are therefore not present in this enclave. This makes it a perfect first coding example such as often is the case for a 'Hello World' program. Note that a 'Hello World' program would not be minimal in an enclave scenario as it would require writing to the standard output. In this case these output functions reside in the untrusted context and therefore require functionalities such as enclave reentry etc. To give an insight in how enclave programs are developed, this example contains the entire main file. Pandora however only validates the enclave itself.

```
1  #include <msp430.h>
2  #include <stdio.h>
3  #include <sancus/sm_support.h>
4  #include <sancus_support/sm_io.h>
5
6
7  DECLARE_SM(basic_enclave, 0x1234);
8
9  /* ======== ENCLAVE FUNCTION ======== */
10
11 int SM_ENTRY(basic_enclave) plus_one(int parameter)
12 {
13     return parameter + 1;
14 }
15
16 /* ======== UNTRUSTED CONTEXT ======== */
17 int main()
```

```
18 {
19     //Setup UART etc.
20     msp430_io_init();
21
22     //Initialize enclave
23     sancus_enable(&basic_enclave);
24
25     int enclave_result = plus_one(1);
26     return(enclave_result);
27 }
```

Listing A.1: A minimal working example of a Sancus enclave.

Appendix B

The Sancus Entry Stub

This appendix contains the Sancus `sm_entry` stub version 2.0.0 in Listing B.1. This stub makes a jump from ABI sanitization code to API code on line 72. Note that version 2.0.0 of the entry stub is given here while at the moment version 2.1.0 already exists. This version is however more complex and contains extensive logic for the facilitation of Aion which is out of scope for this work.

As discussed in Section 4.1.3, lines 69 and 86 contain accesses into the `__sm_table` which will be reported by the PTRSan plugin.

```
1     .section ".sm.text"
2     .align 2
3     .global __sm_entry
4     .type __sm_entry,@function
5
6     ; r6: ID of entry point to be called, 0xffff if returning
7     ; r7: return address
8 __sm_entry:
9     ; If we are here because of an IRQ, we will need the current SP
      later. We do
10    ; do not store it in its final destination yet (__sm_irq_sp)
      because we may
11    ; not actually be called by an IRQ in which case we might
      overwrite a stored
12    ; stack pointer.
13    mov r1, &__sm_tmp
14    # Switch stack.
15    mov &__sm_sp, r1
16    cmp #0x0, r1
17    jne 1f
18    mov #__sm_stack_init, r1
19
20 1:
21    ; check of this is an IRQ
22    push r15
23    ; sancus_get_caller_id()
24    .word 0x1387
25    cmp #0xfff0, r15
26    jlo 1f
27    ; SEMI-HACK: If we are not protected, and no other SM has ever
```

```
    been
28      ; executed, the caller ID will be that of the last IRQ because
    entering this
29      ; SM was no protection domain switch. This basically means that
    once an IRQ
30      ; has occurred, we cannot call normal entry points anymore. Since
    it is nice
31      ; to be able to use unprotected SMs during testing, and it is
    quiet common
32      ; to have interrupts disabled then, the caller ID will always be
    that of the
33      ; reset IRQ (0xffff). Since there is no valid use case of actually
     handling
34      ; a reset inside an SM (since the reset will disable all SMs), we
    simply
35      ; ignore it here so that normal entry points can still be used.
36      cmp #0xffff, r15
37      ; If we just do je __sm_isr we get a PCREL relocation which our
    runtime
38      ; linker doesn't understand yet.
39      jeq 1f
40      br #__sm_isr
41  1:
42      pop r15
43
44      ; check if this is a return
45      cmp #0xffff, r6
46      jne 1f
47      br #__ret_entry ; defined in exit.s
48
49  1:
50      ; check if the given index (r6) is within bounds
51      cmp #__sm_nentries, r6
52      jhs .Lerror
53
54      ; store callee-save registers
55      push r4
56      push r5
57      push r8
58      push r9
59      push r10
60      push r11
61
62      ; calculate offset of the function to be called (r6 x 6)
63      rla r6
64      mov r6, r11
65      rla r6
66      add r11, r6
67
68      ; function address
69      mov __sm_table(r6), r11
70
71      ; call the sm
72      call r11
73
```

```
74      ; restore callee-save registers
75      pop r11
76      pop r10
77      pop r9
78      pop r8
79      pop r5
80      pop r4
81
82      ; clear the arithmetic status bits (0, 1, 2 and 8) of the status
        register
83      and #0x7ef8, r2
84
85      ; clear the return registers which are not used
86      mov 4+__sm_table(r6), r6
87      rra r6
88      jc 1f
89      clr r12
90      clr r13
91      rra r6
92      jc 1f
93      clr r14
94      rra r6
95      jc 1f
96      clr r15
97
98  1:
99      mov r1, &__sm_sp
100     mov #0xffff, r6
101     br r7
102
103 .Lerror:
104     br #exit
```

Listing B.1: The Sancus entry stub version 2.0.0.

Appendix C

An Elaborate Overview of the Unit Test Framework

This appendix contains an overview of all test cases developed for the unit test framework. Table C.1 shows the tests developed for the CFSan plugin and a short description of what it does. It also shows whether the test passes the plugin (✓) without triggering the vulnerability detectors or whether it is reported (✗).

Table C.2 shows the tests developed for the PTRSan plugin and a short description of what it does. It also shows whether the test passes the plugin (✓) without triggering the vulnerability detectors or whether it is reported (✗).

Table C.3 shows the tests developed in C and a short description of what it does. It also shows whether the test passes each plugin (✓) without triggering the vulnerability detectors or whether it is reported (✗). False positives are marked as a ✓.

Table C.1: An overview of the assembly test cases written for the CFSan plugin and whether they pass (✓) the plugin without triggering vulnerability detectors or whether they are reported (✗) by the CFSan plugin.

CFSan Test	**Description**	✓ | ✗
br_within_encl_0	Normal behaviour	✓
br_within_encl_1	Normal behaviour	✓
call_c_func_with_priv_stack	Normal behaviour	✓
example_baseEnclave	Normal behaviour	✓
if_else	Normal behaviour	✓
jmp_within_encl	Normal behaviour	✓
ret_int	Normal behaviour	✓
test	Normal behaviour	✓
bb_cross_enclave_boundary_0	Basic block crosses enclave boundary	✗
bb_cross_enclave_boundary_1	Basic block crosses enclave boundary	✗
bb_cross_enclave_boundary_2	Basic block crosses enclave boundary	✗
bb_cross_enclave_boundary_3	Basic block crosses enclave boundary	✗
bb_cross_enclave_boundary_4	Basic block crosses enclave boundary	✗
bb_cross_enclave_boundary_5	Basic block crosses enclave boundary	✗
br_to_attacker_addr	Branch to an attacker address	✗
call_func_in_encl_no_priv_stack_0	Call enclave function without private stack setup	✗
call_func_in_encl_no_priv_stack_1	Call enclave function without private stack setup	✗
jmp_to_non_exec_mem_0	Jump to non-executable memory	✗
jmp_to_non_exec_mem_1	Jump to non-executable memory	✗
ret	Return from inside the enclave	✗
tainted_br_inside_encl	Jump to address constrained within enclave	✗

Table C.2: Assembly test cases written for the PTRSan plugin and whether they pass (✓) the plugin without triggering vulnerability detectors or whether they are reported (✗) by the PTRSan plugin.

PTRSan Test Name	**Description**	✓ | ✗
test_0	Normal behaviour	✓
test_1	Normal behaviour	✓
test_2	Normal behaviour	✓
call_c_func_with_priv_stack	Normal behaviour	✓
mov_secret_to_attacker_addr	Write a value to attacker address	✗
non_tainted_read_outside_encl	Read from concrete address outside enclave	✗
non_tainted_write_outside_encl	Write to concrete address outside enclave	✗
tainted_read_inside_encl_absaddr	Read tainted address constrained within the enclave	✗
tainted_read_inside_encl_reladdr	Read tainted address constrained within the enclave	✗
tainted_write_inside_encl_absaddr	Write tainted address constrained within the enclave	✗
tainted_write_inside_encl_reladdr	Write tainted address constrained within the enclave	✗
unconstrained_read	Read from address not constrained in or out the enclave	✗
unconstrained_read_1	Read from address not constrained in or out the enclave	✗
unconstrained_rw	Read and write from/to address not constrained in or out the enclave	✗
unconstrained_write	Write to address not constrained in or out the enclave	✗

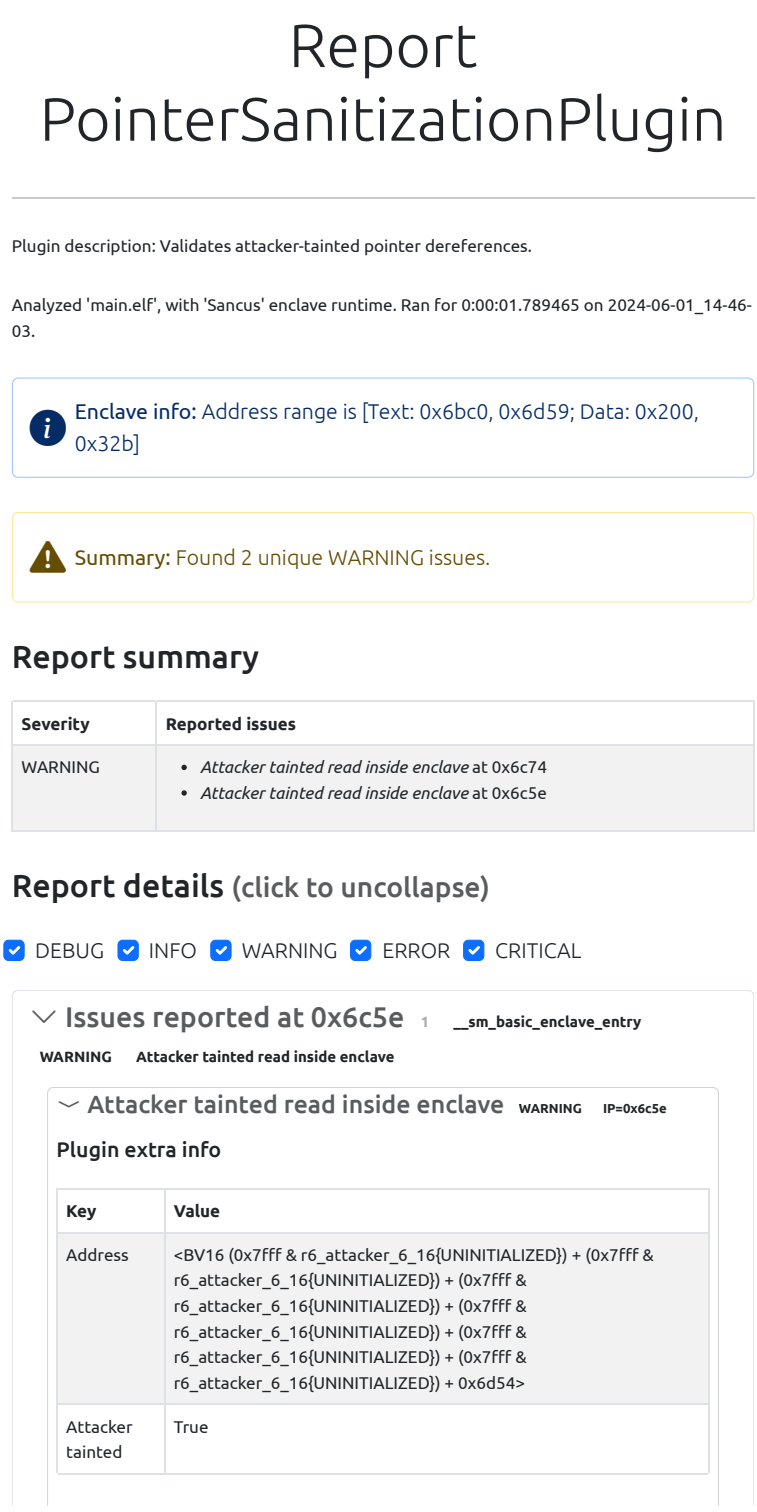
TABLE C.3: C enclaves with a small description and whether they pass (✓) the plugin without triggering vulnerability detectors or whether they are reported (✗). False positive reports are marked as a ✓.

C Test Name	Description	CFSan	PTRSan
comp_gen_test_1	Basic enclave with normal functionality	✓	✓
comp_gen_test_2	Basic enclave with a function reading from unconstrained buffer	✓	✗
fib	A recursive fibonacci calculating enclave	✓	✓
fib2	An iterative fibonacci calculating enclave	✓	✓
hello-library	Hello-world with library function from the Sancus directory	✓	✓
hello-world-basic-sancus	Hello-world from the Sancus directory	✓	✓
sancus-minimal	See Appendix A	✓	✓
sancus-minimal_3entries	minimal-sancus-example but with multiple entry points	✓	✓
sancus-minimal_v1.0.0	minimal-sancus-example compiled with stubs v1.0.0	✗	✗
simple_encl_sanc_v1.0.0	See Section 4.2	✗	✗
simple_encl_sanc_v2.0.0	See Section 4.2	✗	✗
simple_encl_sanc_v2.1.0	See Section 4.2	✓	✓
t2w_with_programming_error	fixed sancus_is_outside, but with a programming error	✓	✗
t2w_1_fixed_improper_r7_sanitization	See Section 4.3	✓	✓
t2w_1_improper_r7_sanitization	See Section 4.3	✗	✗
t2w_2_authentic_execution	See Section 4.3	✓	✗
t2w_3_soteria_loader	See Section 4.3	✓	✗
t2w_4_fixed_sancus_is_outside_sm	See Section 4.3	✓	✓
t2w_4_vulnerable_sancus_is_outside_sm	See Section 4.3	✓	✗

Appendix D

Generated Reports for the Minimal Sancus Enclave

This appendix contains the complete PTRSan report generated for the *minimal-sancus-example* enclave. Figure D.1 to D.4 depict the report for the PTRSan plugin. At the top of the report, some general information is given about the plugin and for example the address range of the enclave. Next a report summary is given with all found issues and the addresses at which they were found. At the bottom an extensive documentation containing the details for each issue is generated. For each issue the header indicates in which section the issue was found. Each issue can also be analysed in depth. For each issue, some extra info is presented such as: the address at which the issue occurred, a boolean indicating if the address is attacker tainted, the possible range of the target in case it is symbolic. Next a **Disassembly** block is given which contains the basic block at which the issue occurred followed with an overview of the **CPU registers** indicating the contents of each register at that moment. The **Backtrace** indicates the path that was taken in the symbolic execution to reach this state and to round up the **constraints** imposed on the state are shown.



Report PointerSanitizationPlugin

Plugin description: Validates attacker-tainted pointer dereferences.

Analyzed 'main.elf', with 'Sancus' enclave runtime. Ran for 0:00:01.789465 on 2024-06-01_14-46-03.

Enclave info: Address range is [Text: 0x6bc0, 0x6d59; Data: 0x200, 0x32b]

Summary: Found 2 unique WARNING issues.

Report summary

Severity	Reported issues
WARNING	• *Attacker tainted read inside enclave* at 0x6c74 • *Attacker tainted read inside enclave* at 0x6c5e

Report details (click to uncollapse)

DEBUG INFO WARNING ERROR CRITICAL

Issues reported at 0x6c5e 1 __sm_basic_enclave_entry

WARNING Attacker tainted read inside enclave

Attacker tainted read inside enclave WARNING IP=0x6c5e

Plugin extra info

Key	Value
Address	<BV16 (0x7fff & r6_attacker_6_16{UNINITIALIZED}) + (0x7fff & r6_attacker_6_16{UNINITIALIZED}) + (0x7fff & r6_attacker_6_16{UNINITIALIZED}) + (0x7fff & r6_attacker_6_16{UNINITIALIZED}) + (0x7fff & r6_attacker_6_16{UNINITIALIZED}) + 0x6d54>
Attacker tainted	True

Figure D.1: The PTRSan report for the *minimal-sancus-example* enclave.

Key	Value
Length	2
Pointer range	[0x6d54, 0x6d54]
Pointer can wrap address space	False
Pointer can lie in enclave	True
Extra info	Issue downgraded to a warning since read is strictly constrained to memory region inside enclave. Disclaimer: This is a heuristic only, please double check manually!

Execution state info

Disassembly

```
    6c4a:       04 12           push    r4
    6c4c:       05 12           push    r5
    6c4e:       08 12           push    r8
    6c50:       09 12           push    r9
    6c52:       0a 12           push    r10
    6c54:       0b 12           push    r11
    6c56:       06 56           rla     r6
    6c58:       0b 46           mov     r6,     r11
    6c5a:       06 56           rla     r6
    6c5c:       06 5b           add     r11,    r6
    6c5e:       1b 46 54 6d     mov     27988(r6),r11   ;0x6d54(r6)
    6c62:       8b 12           call    r11
```

CPU registers

```
                pc       : 0x6c4a
                sp       : 0x2f4
        *       sr       : <BV16
((((((((((((((((((((((((((((((((sr_attacker_2_16{UNINITIALIZED} &
0xfff5 | 0x2) & 0xfefa | 0x1) & 0xfffd | 0x2) & 0xfefa | 0x1) &
0xfef8 | 0x2) & 0xfef8 | (0#15 .. (if (0x7fff &
r6_attacker_6_16{UNINITIALIZED}) == 0x0 then 1 else 0)) << 0x1) &
...
        *       zero     : <BV16 zero_attacker_3_16{UNINITIALIZED}>
        *       r4       : <BV16 r4_attacker_4_16{UNINITIALIZED}>
        *       r5       : <BV16 r5_attacker_5_16{UNINITIALIZED}>
                r6       : 0x0
                r7       : 0x5c3e
        *       r8       : <BV16 r8_attacker_8_16{UNINITIALIZED}>
        *       r9       : <BV16 r9_attacker_9_16{UNINITIALIZED}>
        *       r10      : <BV16 r10_attacker_10_16{UNINITIALIZED}>
                r11      : 0x0
        *       r12      : <BV16 r12_attacker_12_16{UNINITIALIZED}>
        *       r13      : <BV16 r13_attacker_13_16{UNINITIALIZED}>
```

FIGURE D.2: The PTRSan report for the *minimal-sancus-example* enclave.

Figure D.3: The PTRSan report for the *minimal-sancus-example* enclave.

Key	Value
Pointer range	[0x6d58, 0x6d58]
Pointer can wrap address space	False
Pointer can lie in enclave	True
Extra info	Issue downgraded to a warning since read is strictly constrained to memory region inside enclave. Disclaimer: This is a heuristic only, please double check manually!

Execution state info

Disassembly

```
6c64:       3b 41           pop     r11
6c66:       3a 41           pop     r10
6c68:       39 41           pop     r9
6c6a:       38 41           pop     r8
6c6c:       35 41           pop     r5
6c6e:       34 41           pop     r4
6c70:       32 f0 f8 7e     and     #32504, r2      ;#0x7ef8
6c74:       16 46 58 6d     mov     27992(r6),r6    ;0x6d58(r6)
6c78:       06 11           rra     r6
6c7a:       08 2c           jc      $+18            ;abs 0x6c8c
```

CPU registers

Backtrace

Basic block trace (most recent first) - Length: 17

Constraints

Attacker constraints

FIGURE D.4: The PTRSan report for the *minimal-sancus-example* enclave.

Appendix E

Generated Reports for Sancus Stubs v2.0.0

This appendix contains the complete CFSan and PTRSan reports generated for the Sancus entry and exit stub version 2.0.0. Figures E.1 to E.3 depict the report for the CFSan plugin and Figures E.4 to E.5 contain the report for the PTRSan plugin.

Report ControlFlowSanitizationPlugin

Plugin description: Detects attacker-controlled jump targets.

Analyzed 'main.elf', with 'Sancus' enclave runtime. Ran for 0:00:03.088532 on 2024-06-01_19-03-28.

Enclave info: Address range is [Text: 0x6c64, 0x6d8d; Data: 0x200, 0x32b]

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

Report summary

Severity	Reported issues
WARNING	• *Concrete ret target in non-executable memory* at 0x6d34
CRITICAL	• *Symbolic unconstrainted tainted jmp target* at 0x6cde

Report details (click to uncollapse)

DEBUG INFO WARNING ERROR CRITICAL

Issues reported at 0x6d34 1 __sm_basic_enclave_ret_entry WARNING

Concrete ret target in non-executable memory

Concrete ret target in non-executable memory WARNING

IP=0x6d34

Plugin extra info

Key	Value
Target	0
Attacker tainted	False
Symbolic	False
Target range	[0x0, 0x0]

Figure E.1: The CFSan report for the Sancus stubs v2.0.0.

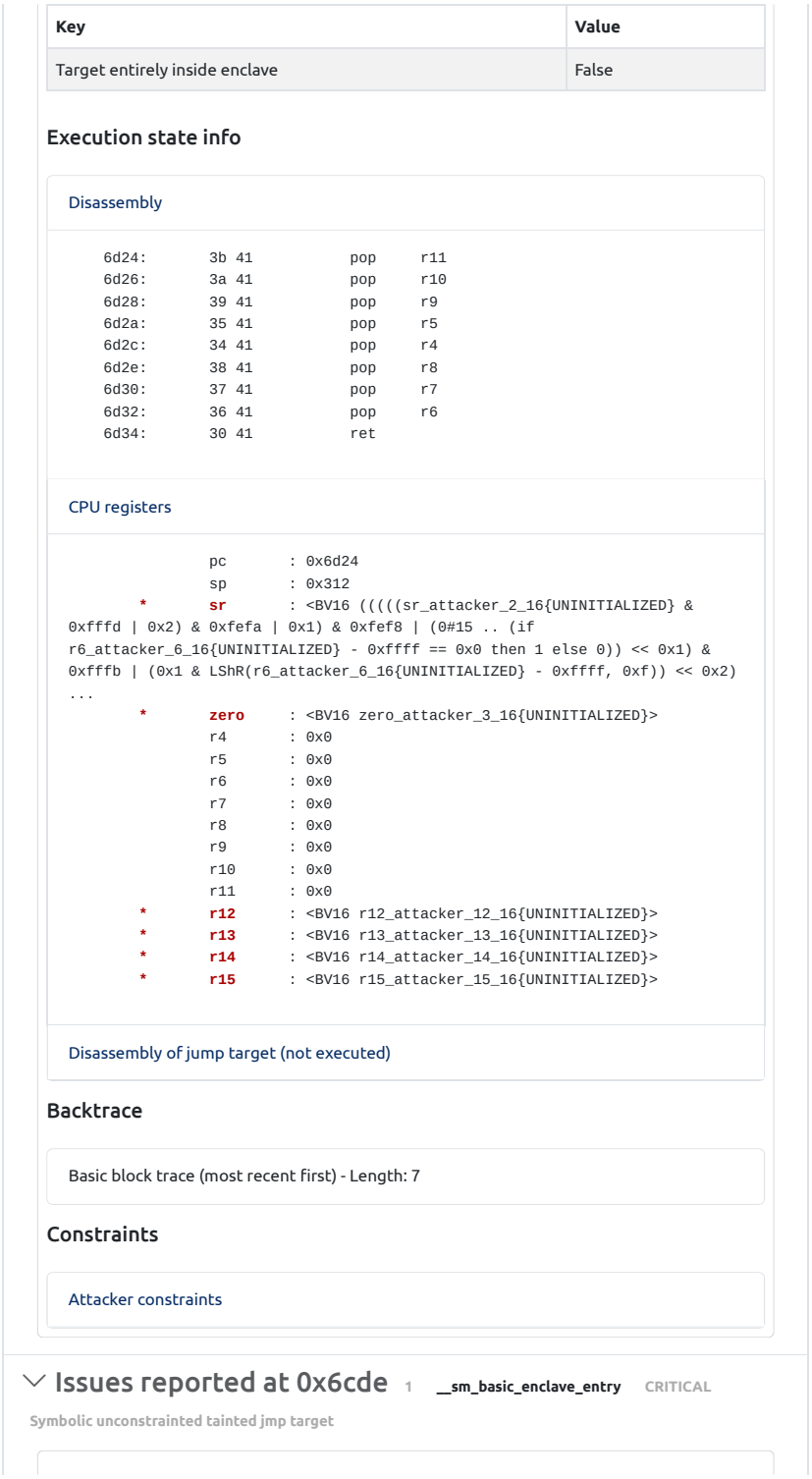
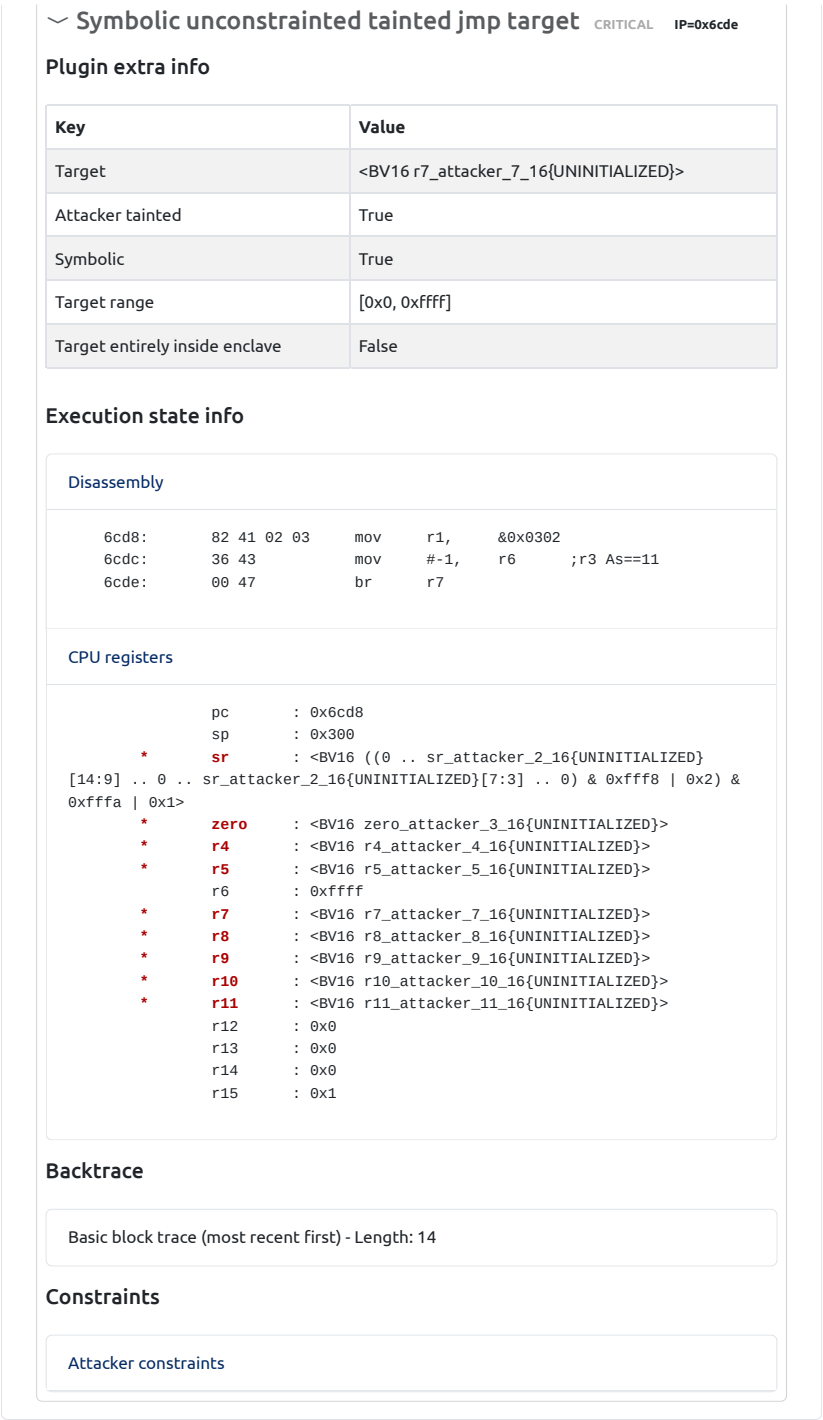


FIGURE E.2: The CFSan report for the Sancus stubs v2.0.0.



Symbolic unconstrainted tainted jmp target CRITICAL IP=0x6cde

Plugin extra info

Key	Value
Target	<BV16 r7_attacker_7_16{UNINITIALIZED}>
Attacker tainted	True
Symbolic	True
Target range	[0x0, 0xffff]
Target entirely inside enclave	False

Execution state info

Disassembly

```
    6cd8:       82 41 02 03     mov     r1,     &0x0302
    6cdc:       36 43           mov     #-1,    r6      ;r3 As==11
    6cde:       00 47           br      r7
```

CPU registers

```
                pc       : 0x6cd8
                sp       : 0x300
        *       sr       : <BV16 ((0 .. sr_attacker_2_16{UNINITIALIZED}
[14:9] .. 0 .. sr_attacker_2_16{UNINITIALIZED}[7:3] .. 0) & 0xfff8 | 0x2) &
0xfffa | 0x1>
        *       zero     : <BV16 zero_attacker_3_16{UNINITIALIZED}>
        *       r4       : <BV16 r4_attacker_4_16{UNINITIALIZED}>
        *       r5       : <BV16 r5_attacker_5_16{UNINITIALIZED}>
                r6       : 0xffff
        *       r7       : <BV16 r7_attacker_7_16{UNINITIALIZED}>
        *       r8       : <BV16 r8_attacker_8_16{UNINITIALIZED}>
        *       r9       : <BV16 r9_attacker_9_16{UNINITIALIZED}>
        *       r10      : <BV16 r10_attacker_10_16{UNINITIALIZED}>
        *       r11      : <BV16 r11_attacker_11_16{UNINITIALIZED}>
                r12      : 0x0
                r13      : 0x0
                r14      : 0x0
                r15      : 0x1
```

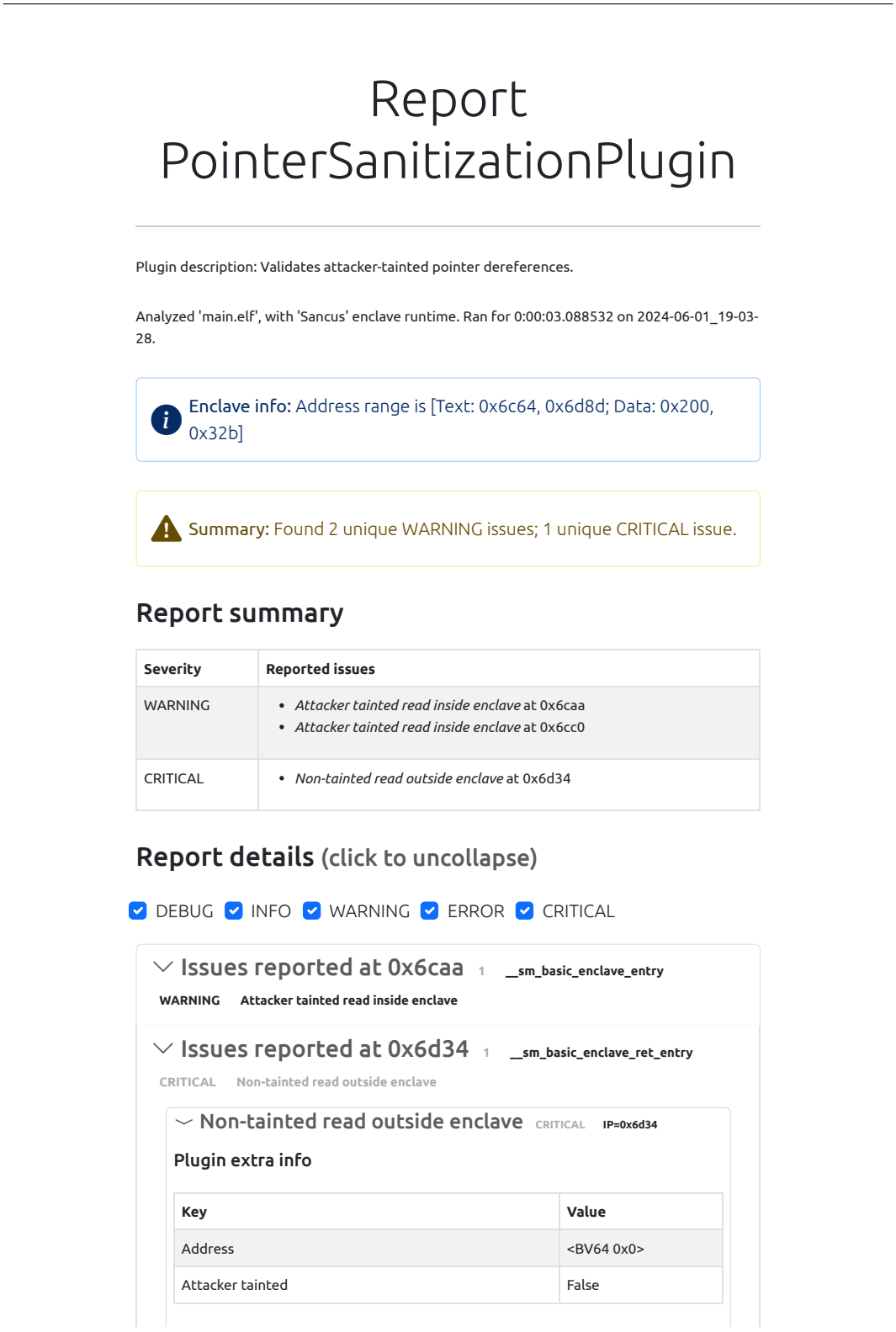
Backtrace

Basic block trace (most recent first) - Length: 14

Constraints

Attacker constraints

Figure E.3: The CFSan report for the Sancus stubs v2.0.0.



Report PointerSanitizationPlugin

Plugin description: Validates attacker-tainted pointer dereferences.

Analyzed 'main.elf', with 'Sancus' enclave runtime. Ran for 0:00:03.088532 on 2024-06-01_19-03-28.

Enclave info: Address range is [Text: 0x6c64, 0x6d8d; Data: 0x200, 0x32b]

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

Report summary

Severity	Reported issues
WARNING	• *Attacker tainted read inside enclave* at 0x6caa • *Attacker tainted read inside enclave* at 0x6cc0
CRITICAL	• *Non-tainted read outside enclave* at 0x6d34

Report details (click to uncollapse)

DEBUG INFO WARNING ERROR CRITICAL

Issues reported at 0x6caa 1 __sm_basic_enclave_entry

WARNING Attacker tainted read inside enclave

Issues reported at 0x6d34 1 __sm_basic_enclave_ret_entry

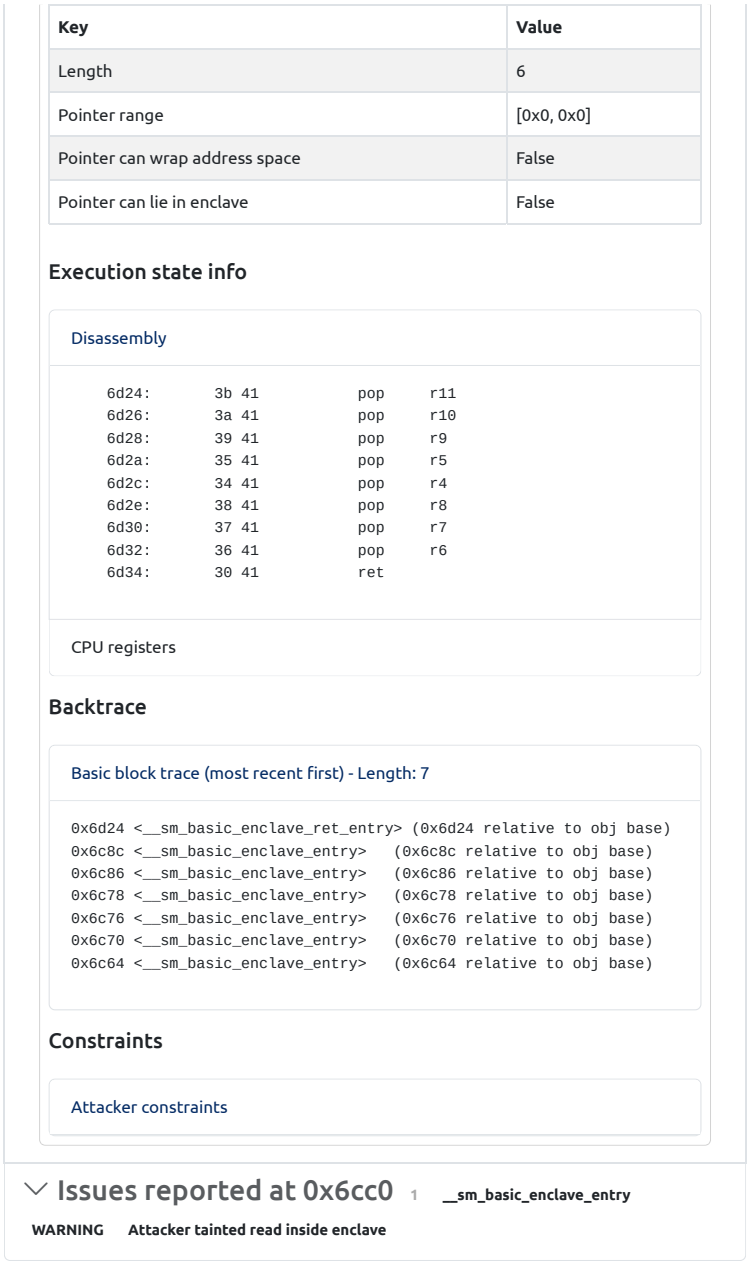
CRITICAL Non-tainted read outside enclave

Non-tainted read outside enclave CRITICAL IP=0x6d34

Plugin extra info

Key	Value
Address	<BV64 0x0>
Attacker tainted	False

FIGURE E.4: The PTRSan report for the Sancus stubs v2.0.0.



Key	Value
Length	6
Pointer range	[0x0, 0x0]
Pointer can wrap address space	False
Pointer can lie in enclave	False

Execution state info

Disassembly

```
6d24:       3b 41           pop     r11
6d26:       3a 41           pop     r10
6d28:       39 41           pop     r9
6d2a:       35 41           pop     r5
6d2c:       34 41           pop     r4
6d2e:       38 41           pop     r8
6d30:       37 41           pop     r7
6d32:       36 41           pop     r6
6d34:       30 41           ret
```

CPU registers

Backtrace

Basic block trace (most recent first) - Length: 7

```
0x6d24 <__sm_basic_enclave_ret_entry> (0x6d24 relative to obj base)
0x6c8c <__sm_basic_enclave_entry>   (0x6c8c relative to obj base)
0x6c86 <__sm_basic_enclave_entry>   (0x6c86 relative to obj base)
0x6c78 <__sm_basic_enclave_entry>   (0x6c78 relative to obj base)
0x6c76 <__sm_basic_enclave_entry>   (0x6c76 relative to obj base)
0x6c70 <__sm_basic_enclave_entry>   (0x6c70 relative to obj base)
0x6c64 <__sm_basic_enclave_entry>   (0x6c64 relative to obj base)
```

Constraints

Attacker constraints

Issues reported at 0x6cc0 1 __sm_basic_enclave_entry

WARNING Attacker tainted read inside enclave

Figure E.5: The PTRSan report for the Sancus stubs v2.0.0.

Bibliography

[1] Capstone disassembly framework. https://www.capstone-engine.org/. Accessed: 2024-05-09.

[2] Intel sgx sdk. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html. Accessed: 2024-05-05.

[3] Msp430 family instruction set summary. https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf. Accessed: 2024-05-09.

[4] Open enclave sdk. https://openenclave.io/sdk/. Accessed: 2024-05-05.

[5] F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck. Pandora: Principled symbolic validation of intel sgx enclave runtimes. In *45th IEEE Symposium on Security and Privacy-IEEE S&P 2024*. IEEE, 2024.

[6] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens. Faulty point unit: Abi poisoning attacks on intel sgx. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, pages 415–427, New York, NY, USA, 2020. Association for Computing Machinery.

[7] 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 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1357–1372, New York, NY, USA, 2021. Association for Computing Machinery.

[8] P. Antonino, W. A. Woloszyn, and A. W. Roscoe. Guardian: Symbolic validation of orderliness in sgx enclaves. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, CCSW '21, pages 111–123, New York, NY, USA, 2021. Association for Computing Machinery.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, Nov. 2016. USENIX Association.

[10] M. Bognár, C. Magnus, F. Piessens, and J. Van Bulck. Intellectual property exposure: Subverting and securing intellectual property encapsulation in texas instruments microcontrollers, 2024-01-12.

[11] M. Bognar, J. Van Bulck, and F. Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1638–1655, 2022.

[12] 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)*, pages 3917–3934, Boston, MA, Aug. 2022. USENIX Association.

[13] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. Association for Computing Machinery.

[14] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley. Intel tdx demystified: A top-down approach, 2023.

[15] 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)*, pages 841–858. USENIX Association, Aug. 2020.

[16] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[17] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai. Smashex: Smashing sgx enclaves using exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 779–793, New York, NY, USA, 2021. Association for Computing Machinery.

[18] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In ISOC, editor, *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, 2012. Copyright ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA and is available at :.

[20] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, pages 241–250, New York, NY, USA, 2015. Association for Computing Machinery.

[21] J. Hofmans. A comparative analysis of security features between Sancus and TI MSP430 IPE. Master's thesis, KU Leuven, 2022. https://distrinet.cs.kuleuven.be/software/sancus/publications/hofmans22thesis.pdf.

[22] T. Instruments. Msp code protection features. https://www.ti.com/lit/an/slaa685/slaa685.pdf?ts=1717443926531&ref_url=https%253A%252F%252Fwww.google.com%252F, 2015. Accessed: 2024-06-01.

[23] D. Kaplan, J. Powell, and T. Woller. Amd sev-snp: Strengthening vm isolation with integrity protection and more, 2020.

[24] 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*, ASPLOS '20, pages 971–985, New York, NY, USA, 2020. Association for Computing Machinery.

[25] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[26] S. Lee and T. Kim. Leaking uninitialized secure enclave memory via structure padding (extended abstract), 2017.

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

[28] J. Noorman, J. V. 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. *ACM Trans. Priv. Secur.*, 20(3), jul 2017.

[29] 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)*, pages 1429–1446, Santa Clara, CA, Aug. 2019. USENIX Association.

[30] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), jan 2019.

[31] M. Schink and J. Obermaier. Taking a look into Execute-Only memory. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[32] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez. Sok: Hardware-supported trusted execution environments, 2022.

[33] 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 *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[34] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures, 2013-01-01.

[35] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In S. Jajodia and J. Zhou, editors, *Security and Privacy in Communication Networks*, pages 344–361, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[36] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: a practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658, USA, 2017. USENIX Association.

[37] J. Van Bulck. Secure resource sharing for embedded protected module architectures. Master's thesis, KU Leuven, 2015. https://distrinet.cs.kuleuven.be/software/sancus/publications/vanbulck15thesis.pdf.

[38] J. Van Bulck. The hitchhiker's guide to subverting intel sgx enclaves. https://jovanbulck.github.io/files/circuit22.pdf, 2022. Accessed: 2024-05-01.

[39] J. Van Bulck, F. Alder, and F. Piessens. A case for unified abi shielding in intel sgx runtimes, 2022-03-01.

[40] 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 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1741–1758, New York, NY, USA, 2019. Association for Computing Machinery.

[41] VEX. libvex_ir.h documentation. https://github.com/vex/pub/blob/master/libvex_ir.h, 2024. Accessed: 2024-05-25.

[42] Y. Wang, Z. Zhang, N. He, Z. Zhong, S. Guo, Q. Bao, D. Li, Y. Guo, and X. Chen. Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 2710–2724, New York, NY, USA, 2023. Association for Computing Machinery.