KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

A Security Analysis of Interrupts in Embedded Enclaved Execution

Sven Cuyt

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. R. Strackx

Assessoren:
Ir. J. Van Bulck
Dr. A. Timany

Begeleiders:
Ir. J. Van Bulck
Dr. R. Strackx

Academiejaar 2018 – 2019

Preface

Before we jump into all the details of embedded enclaved execution and side-channel leakage, I would like to thank the following people. I would like to thank Prof. dr. ir. Frank Piessens, my supervisor, and Dr. Raoul Strackx, my co-supervisor for allowing me to work on an initially vague topic. It turned out to be far more interesting than I could have imagined. Also, my thanks goes to ir. Jo Van Bulck, my daily advisor who has taught me a lot over the last year. Thank you for all the interesting discussions and the valuable feedback. My thanks goes also to the Sancus team for making an open-source protected module architecture. It made working on this thesis considerably easier and I am happy to see my work being upstreamed. I also want to thank my friends: Wouter Baert, Kristof Achten, Michiel Bollen and Larisa Popescu for their support during the more difficult times. Last but not least, I would like to thank my parents for allowing me to pursue a degree in a field I really enjoy and their continuous support throughout my higher eductation.

Sven Cuyt

Contents

Abstract

With the rise of the Internet of Things and ubiquitous computing, small embedded devices are permeating our daily lives. They are increasingly being deployed in security-critical and privacy-sensitive tasks. However, to minimise production costs, these small embedded devices lack common hardware features to implement conventional security mechanisms, such as virtual memory isolation and processor privilige levels.

To respond to these concerns, recent research effort has shifted towards Protected Module Architectures (PMAs). PMAs allow security-critical code to be executed in an enclave isolated from the rest of the system. However, this software isolation enforced by PMAs only holds at the architectural level. Microarchitectural optimizations are still shared among enclaves and untrusted code.

As microarchitectural optimizations are commonly found in high-end CPU architectures, such as Intel x86, quite some research effort has gone into understand microarchitectural side-channel leakage in high-end enclaved execution. Comparatively little research effort has gone towards understanding microarchitectural side-channels against embedded enclaved execution. In this perspective, this master's thesis increases our understanding of microarchitectural side-channel leakage and automated exploitation in embedded enclaved execution. The main contributions of this master's thesis are twofold. First, Sancus-Step, an attacker framework for automatic side-channel exploitation, is proposed. Sancus-Step aims to be easy to use, while still being flexible enough such that developing side-channel attacks requires considerably less expertise. Second, side-channel leakage of compiler generated code is analysed, using Sancus-Step. We did an extensive analysis of the multiplication stub and an initial exploration of other stubs. We patch the multiplication stub and evaluate the performance and security of the patched stub.

This master's thesis shows considerable side-channel leakage in compiler generated code, which can easily be automatically exploited using our newly proposed attacker framework Sancus-Step. Additionally, this master's thesis shows that this side-channel leakage can be patched in software.

Samenvatting

Met de opkomst van het Internet der Dingen, doordringen kleine ingebedde apparaten ons dagelijks leven. Ze worden steeds vaker ingezet in kritische en privacy-gevoelige taken. Desondanks, om de productiekosten te drukken, ontbreken deze apparaten de gebruikelijke hardware-ondersteuning om conventionele beveiligingsmechanismen te implementeren, zoals virtueel geheugen en processorringen.

Om tegemoet te komen aan deze zorgen heeft recent onderzoek zich meer gericht op beveiligingsarchitecturen. Beveilingsarchitecturen laten toe om kritische code uit te voeren in een enclave, geïsoleerd van de rest van het systeem. Echter, de isolatie die beveilingsarchitecturen afdwingen is enkel geldig op architecturaal niveau. Microarchitecturale optimizaties worden nog steeds gedeeld tussen enclaves en niet-vetrouwde code.

Gezien microarchitecturale optimizaties meestal gevonden worden in *high-end* CPUs, zoals Intel x86, is er redelijk veel onderzoek gegaan naar het begrijpen van microarchitecturale *side-channel* lekken in *high-end* geënclaafde uitvoering. Echter, relatief weinig onderzoek richt zich op het begrijpen van microarchitecturale *side-channels* in ingebedde geënclaafde uitvoering. In dit perspectief verbetert deze masterscriptie onze kennis over lekken via microarchitecturale *side-channels* en de automatische exploitatie in ingebedde geënclaafde uitvoering. De belangrijkste bijdragen van deze masterscriptie zijn tweevoudig. Ten eerste wordt Sancus-Step, een aanvallersraamwerk voor automatische exploitatie van *side-channels*, voorgesteld. Sancus-Step streeft ernaar om gemakkelijk in gebruik te zijn, terwijl het flexibel genoeg blijft zodat nieuwe *side-channel* aanvallen ontwikkelen lukt zonder uitgebreide kennis van de aanvaller. Ten tweede, *side-channels* lekken in compiler gegenereerde code is geanalyseerd met behulp van Sancus-Step. We hebben uitvoerig de vermenigvuldingscode geanalyseerd en de andere gegenereerde code initieel verkend. We beveiligen de vermenigvuldingscode en evalueren de performantie en veiligheid van de beveiligde code.

Deze masterscriptie toont aan dat er aanzienlijke informatie door compiler gegeneerde code lekt via *side-channels*. Deze lekken kunnen gemakkelijk automatisch geëxploiteerd worden door gebruik te maken van ons aanvallersraamwerk Sancus-Step. Bijkomend toont deze masterscript dat deze *side-channels* in software gedicht kunnen worden.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations

Abbreviations

PMA	Protected Module Architecture
SPM	Self-Protecting Module
TOCTOU	Time of Check to Time of Use
TCB	Trusted Computing Base
ISR	Interrupt Service Routine
IoT	Internet of Things

Chapter 1

Introduction

In the last decade, the Internet of Things (IoT) has rapidly grown into a massive network of interconnect embedded devices. Often, IoT applications handle private data or perform safety critical operations. Thus, the security of these IoT applications is of paramount importance. However, commonly the low-end devices on which these IoT applications run, do not offer the same hardware security mechanisms [11], such as virtual memory and processor privilige levels, as their high-end counterparts. This lack of security mechanisms is caused by economic interests, as omitting these hardware security features, reduces cost and power consumption. Yet, as IoT applications often perform safety-critical actions or handle private data, it should be without surprise that they are interesting targets for malicious people. Of course, their weak security [8] only increases the interest of malicious people.

To respond to these concerns, researches have proposed in the last decade a variety of lightweight security architectures. These so called *protected module architectures* [23, 24, 16] provide strong software isolation, while keeping the trusted computing base to a minimum. They do not require hardware features, such as virtual memory, to operate, thus allowing them to isolate software in single address spaces.

PMAs isolate software at the architectural level. Thus, arguably, the attack surface for PMAs consists solely of (*i*) memory safety vulnerabilites, (*ii*) side-channel attacks and (*iii*) transient execution attacks. Classical memory safety vulnerabilies, such as the buffer-overflow [25] and return-oriented programming [26], are well understood. On the other hand, only recently has the research community raised our understanding of (microarchitectural) side-channels and transient execution attacks [6].

In Section 1.1 a brief summary on both conventional and state-of-the-art software isolation is given. Section 1.2 discusses the contributions of this thesis towards a better understanding of microarchitectural leakage in embedded enclaved execution. Section 1.3 outlines the rest of the thesis.

1.1 Software Isolation

Software isolation is a set of hardware and software features which disallow software to interact with one and other, unless through specific channels. Software isolation ensures that a program's execution will not be tampered with by untrusted software.

For conventional software isolation, such as those found in desktops and server, applications rely on a combination of hardware features and an omnipotent kernel to correctly isolate the application. The reliance on an omnipotent kernel for one's security guarantees is undesirable, as a kernel tends to be often huge in size. As to be expected, the bigger a piece of software is, the more likely it is vulnerable to well-known low-level exploits [25, 26].

Consequentially, recently researchers have directed their focus onto protected module architectures (PMAs) [23, 24, 16]. PMAs allow an developer to isolate software in a single address space. In short, a developer can create a enclave, which is responsible for its own security guarantees. PMAs often provide a minimal set of security guarantees, which allows an enclave to define and enforce its own security guarantees. The key security guarantee provided by any PMA is that the data of the enclave can only be accessed by the enclave itself.

PMAs have been developed for both high-end as low-end CPUs. Since 2015 PMAs for commodity CPUs entered mainstream market, as every Intel CPU since 2015 is equipped with Intel's Software Guard Extension (Intel SGX) [7, 19]. High-end PMAs have shown that they can act as an extra security mechanism on top of conventional software isolation found in kernels [4] or hypervisors [28].

Low-end PMAs however, often focus on enforcing strong security guarantees while relying only on hardware. This makes them particularly promising, as they minimise both cost and the TCB. As they are hardware only, they exclude any software, including the omnipotent kernel, from the TCB.

PMAs are thus capable of isolating software in single (and virtual) address spaces. However, this isolation is at the architectural level. The microarchitectural state of the CPU is thus shared among enclaves and untrusted code. The microarchitectural state may contain data of a previously executing process after a context switch. Through side-channels this left-over data can be leaked [12, 40]. Side-channels are information channels which come into existence from side-effects of computation, such as the amount of time code requires to execute. Additionally, side-effects of execution itself may affect the microarchitectural state of the CPU [5, 39], which can be inspected by the attacker. This allows an attacker to determine if e.g. a certain branch was executed. Even more complex attacks, known as transient execution attacks [17, 15, 32, 38], abuse microarchitectural optimizations to encode unauthorized data into the microarchitectural state of the CPU. The encoded data can later be retrieved through a side-channel, effectively leaking the unauthorized data. Recently, researchers have made a steady progression towards a better understanding of microarchitectural side-channel leakage and transient execution attacks against high-end enclaved execution, such as Intel SGX. This can be seen by the amount of attacks discovered in the last few years [21, 21, 36].

However, as most side-channel and transient execution attacks abuse microarchitectural optimizations, these attacks often only apply to high-end PMAs such as Intel SGX [32, 5]. Thus comparatively little effort went into researching microarchitectural side-channel leakage in embedded enclaved execution. Only recently was the first low-end microarchitectural side-channel discovered, named Nemesis [35]. Nemesis abuses a subtle timing difference in the interrupt logic of the CPU.

Often it is assumed that enclaves are uninterruptible [24, 28] when designing PMAs, which is arguably an incorrect assumption, as applications often run in a multi-threaded environment. While support for interruptible enclaves has been added to PMAs, the security implications of interrupts in embedded enclaved execution has been left as an open research question. Nemesis [35] showed that interrupts, indeed, introduce an exploitable side-channel, which was previously unknown. As such, researching the security implications of interrupts in embedded enclaved execution seems an promising area of research.

1.2 Contributions

The contributions of this thesis are three fold:

1. We design and implement Sancus-Step, an attacker framework which considerably reduces attacker's required expertise to mount side-channel attacks against embedded enclaves. Sancus-Step is in line with SGX-Step [34], another attacker framework targeting Intel-SGX. Hence, attacks against enclaves become easier. This also allows developers to easily test their own code against Nemesis and other side-channel attacks.

2. We show side-channel leakage in compiler generated code. This code is included at the compiler's choice, and thus can jeopardise applications, without the developer even being aware of the problem. Additionally we show the capabilities of Sancus-Step by extensively using Sancus-Step to mount proof-of-concept attacks in a micro-benchmark setting.

3. We patch the vulnerable stubs at assembly level and show emperically that the side-channels are closed. We evaluate our patched stub by comparing its performance with the vulnerable stub and give an informal security argument.

4. We provide an initial exploration of atomicity issues caused by interrupts in embedded enclaved execution. We demonstrate a proof-of-concept attack, abusing a time-of-check to time-of-use vulnerability in compiler generated code.

All code is open-sourced and available on github[1,2,3,4].

1.3 Outline

Chapter 2 provides an overview on the required security knowledge and provides an overview on relevant work. In Chapter 3 Sancus-Step is proposed and in Chapter 4 side-channel leakage of compiler generated code is analysed. Chapter 5 summarises the contributions of this thesis. Furthermore, its limitations are acknowledged and interesting open research questions are provided to guide new research.

[1]https://github.com/sancus-pma/sancus-compiler
[2]https://github.com/sancus-pma/sancus-support
[3]https://github.com/sancus-pma/sancus-examples
[4]https://github.com/jovanbulck/nemesis

Chapter 2

Background

To provide security guarantees, computer systems protect software from itself and other, potentially malicious or buggy, software. This is known as software isolation. For commodity and high-end processors, software isolation is a well researched and understood area of software security. Common solutions rely on hardware features to implement software isolation in a omnipotent operating system.

With the emergence of the internet of things, many embedded and interconnected devices do not support these common hardware features because of resource constraints, such as power consumption and chip size. This has triggered research towards lightweight protection mechanisms which can be deployed on embedded microprocessors to achieve the necessary security guarantees, while keeping overhead to a minimum [29, 23, 24]. As a result, many different protected module architectures (PMAs) have been proposed, each with their own advantages and disadvantages [18].

This chapter discusses briefly in Section 2.1 conventional software isolation techniques. In Section 2.2 Protected Module Architectures, and more specifically Sancus, will be discussed. Lastly in Section 2.3 microarchitectural side-channels, a class of powerful attacks, will be discussed.

2.1 Conventional Software Isolation

Overview

In high-end processors, software isolation is realised by a combination of hardware features, such as memory management units (MMU) and processor levels (also known as protection rings), and a trusted software layer, such as a hypervisor or operating system.

Memory management units introduce virtual memory spaces. A Virtual memory space is a mapping from virtual addresses to physical addresses. From a programmers point of view, memory is a large array of storage.

Whenever an application performs to access memory located at a virtual address, the virtual address is translated by the MMU into the corresponding physical address. In the case of paged memory, the most common virtual memory technique nowadays, translation of a virtual address to a physical address is called a page walk. To speed up this translation process, memory management units are equipped with a translation lookaside buffer, TLB, acting as a cache for recent page walk results. Upon each context switch the kernel ensures that the correct data is loaded (i.e. the correct page table is in memory).

Virtual memory allows software isolation. Virtual memory enabled processors ensure that different processes cannot name memory not belonging to them, let alone access them, effectively isolating the processes. Of course, this only works if programs are not allowed to alter their page tables or access the MMU.

Besides acting as core security block, virtual memory has other advantages as well. An application running on a processor equipped with a MMU, runs as though it is the only application running on that processor. Thus it can allocate its entire memory space if it is necessary, without having to worry that it will exhaust physical memory. The kernel ensures through the mapping that only valid memory accesses are made.

2.2 Self-Protecting Modules and Sancus

This section first gives an introduction into Self-Protecting Modules (SPM) [29] in Section 2.2.1. Then in Section 2.2.2, Sancus will be introduced as a concrete of a Protected Module Architecture, implementing SPMs.

2.2.1 Self-Protecting Modules

An SPM is a lightweight protection domain, co-existing with other SPMs and untrusted code in a shared address space. An SPM corresponds to three sections, an entry section `S_Entry`, a public section `S_Pub` and a private section `S_Priv`. The public section contains read-only data and the code of the SPM, while the private section contains data that should be private to the section. The entry section is a list of (function) pointers pointing into the public section. The entry section is the only way to enter an SPM. Of course, the entry section is read and execute only.

SPMs enforce a strict program-counter based access control policy to ensure that code can only be executed as intended and that private data stays private. Table 2.1 gives an overview of the access control policy that an SPM enforces. The "from" index is determined by the value of the program-counter. The "to" index is determined by the memory address being accessed. Code executing has different access rights depending on the value of the program counter.

Entry Section. The sole purpose of this section is to provide an entry into the code in `S_Pub`. To adhere to the principle of least privilege, `S_Entry` only needs to be able to jump into (execute) public section. Of course, this section is readable and executable from both inside and outside the SPM.

Public Section. This section contains both the SPM's code and read-only data. To execute the code within the public section, the client can only choose from the possible locations listed within the entry section. This security measure prevents code reuse attacks [26] or prevents clients from bypassing security sensitive code, such as access control checks or encryption code.

The SPM's code is considered trustworthy and secure. Thus it is assumed that it does not contain any exploitable code such as buffer overflows (on the stack or heap) and format string vulnerabilities.

Besides the code, read-only non-secret data can be stored in this section as well. This includes for example public keys used in public key cryptography.

Private Section. The private section is readable and writeable only by the SPM itself. The SPM can store data there to which it requires exclusive access, e.g. cryptographic keys. Note that this data is not only limited to application data, but also data necessary for a correct execution of the code, such as control flow data placed on the stack.

Since the SPM has exclusive access rights to this section of memory, any access policy can be enforced on the data residing in it. Thus, SPM's should at least protect the security sensitive data in the secret section, hence the name self-protecting modules.

Security Guarantees

SPM are realised by a PMA. PMAs generally provide SPMs with a set of security guarantees. This section provides an overview of the security guarantees generally provided, while Section 2.2.2 discusses in detail how Sancus provides these security guarantees.

Integrity of Control Flow. As mentioned above, the entry section is the only way for a client to enter an SPM. Since there are only a few possible entry points, the SPM can be assured that security sensitive code, e.g. access checks, cannot be bypassed, and that no code can be used to mount code reuse attacks [26].

Furthermore PMAs provide each module with its private stack to store control flow information, such as return addresses. It has been proven that such a private stack is secure in the absence of low level vulnerabilities [3]. Thus, control flow can only be subverted if and only if there exists a vulnerability in the high-level code of the SPM. Of course, it is up to the SPM's developer to ensure that the SPM's code is secure, e.g. by formal verification with VeriFast [37]. Thus, the TCB of the SPM consists of (*i*) the SPM's code and (*ii*) the TCB needed to implement the PMA.

From\To	S_Entry	S_Pub	S_Priv	Unprotected
S_Entry	---	--x	---	---
S_Pub	r-x	r-x	rw-	rwx
Unprotected	r-x	r--	---	rwx

Table 2.1: Overview of the program-counter based access control policy for SPMs[29], noted in traditional Unix style.

Secrecy of Private Data. The access control policy denies all memory accesses in the private section of an SPM, unless the memory access originates from code within the public section of the SPM. Since the SPM's code is trusted, it is assumed that it does not leak any private data and that it is impossible to influence control flow within the SPM. Thus, secrecy of private data immediately follows from the access control policy enforced by the PMA.

Authentication of an SPM. Reliable and efficient authentication of SPMs is essential, since its paramount to building modular, trustworthy systems. An SPM can be identified by the contents of its entry section, its public section and the start and end addresses of its entry, private and public section (since code might change based on these values, e.g. checking if a pointer points into unprotected memory). However, an SPM's public section might be arbitrarily large. To reduce this to a more mangeable size, a cryptographic hash of the public section and the start and end addresses of each section can be used instead. Once an SPM is provided with the hash of the module it wants to authenticate, it can calculate the hash himself and compare the result with the expected hash. This allows other code, both SPMs and untrusted code, to authenticate other SPMs.

Additionally, the SPM to be authenticated needs also to be correctly protected by the PMA. This is a guarantee only the PMA can provide. Section 2.2.2 discusses how Sancus provides this guarantee.

Note that Strackx et al. initially coined the term Self-Protecting Modules. However, for consistency reasons, throughout the rest of this master's thesis we will refer to SPMs as enclaves.

2.2.2 Sancus

Sancus is a protected module architecture (PMA) designed for embedded microprocessors. More specifically, Sancus is built on top of the openMSP430 CPU architecture. Sancus allows developers for isolating code on processors that commonly lack protection and security features found in high-end processors. Additionally, Sancus allows third parties to attest that an enclave is deployed and running untampered on the processor.

Sancus serves as the case study architecture of choice for this master's thesis, simply because it is an active research project at KULeuven [22] and is open source.

As such, all work presented in this master's thesis can be merged into the upstream.

System Model

Sancus is designed with the following system model in mind: a single infrastructure provider, *IP*, owns multiple embedded devices or nodes. We refer to nodes as N_i. Multiple third-party software providers, SP_j, want to deploy their software on the infrastructure provided by *IP*. The software modules that they deploy will be referred to as $SM_{j,k}$. This model is chosen because of its simplicity and abstractness.

Attacker Model

Sancus tries to protect enclaves from an attacker with two powerful capabilities [24]. First, the attacker has full kernel level control. This means an attacker can deploy his own enclaves, tamper with the operating system and all peripherals. Second, the attacker has full control over the network that the enclaves use to communicate. Sancus follows the standard Dolev-Yao model [10] [24]. The attacker cannot break cryptographic primitives. However, she can perform protocol level attacks, such as a man in the middle attack or replaying messages. Sancus assumes the attacker has no physical access to the nodes. Physical attacks, such as disconnecting nodes and placing probes on the memory bus are thus out of scope. Furthermore, side-channels are left out of scope .

Security Guarantees

Sancus aims to provide the following security properties towards the software providers:

- Software module isolation: enclaves are isolated from one another, from the operating system and all other software that may run on the node. The code and data of the enclave can only be read and modified by the enclave itself.

- Remote attestation: third-parties should be able to remotely attest that an enclave is running uncompromised on a specific node of *IP*.

- Secure communication: software providers should be able to securily communicate with its enclaves. Integrity, confidentiality, authencity and freshness of the communication should be guaranteed.

- Secure linking: enclaves should be able to communicate with each other securily.

- Confidential deployment: software providers can deploy encrypted enclaves, which will get authenticated before being deployed. This ensures the confidentiality of the code running on the nodes.

From/to	Entry	Text	Data	Unprotected
Entry	`r-x`	`r-x`	`rw-`	`rwx`
Text	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected / Other SM	`--x`	`---`	`---`	`rwx`

Table 2.2: Overview of the access control policy enforced by the Sancus hardware.

Cryptographic Primitives

To guarantee these security guarantees, Sancus utilizes multiple cryptographic primitives: (*i*) a hash function, (*ii*) a key derivation function, which takes as input a key and some data and returns a new key, and (*iii*) an authenticated encryption with associated data primitive. This primitive consists of an encryption and decryption function. The encryption function encrypts a given plaintext with a given key and calculates a MAC over the given plaintext and associated data. The decryption function does the reverse and fails if the tag is incorrect for the given ciphertext.

Key Management

The infrastructure provider *IP* has a single master key K_N, which it shares with all of its nodes. K_N can be used to distinguish malicious nodes from legitimate ones. For each software provider, a software provider key $K_{N,SP}$ can be derived from the master key K_N and a unique identifier for *SP*. For each enclave that software provider SP_i deploys, he can derive a enclave key $SM_{N,SP,E}$, which the module can use to do authenticated encryption and thus will use to securily communicate with third-parties.

Sancus hardware comes with a hardware implementation of the key derivation function, such that no software has to be trusted when deriving the keys.

Sancus Enclaves

A Sancus enclave is essentially a standalone executable, consisting of a text and data section. The text section is the code of the module, while the data section contains the code's runtime data. Sancus enforces the access control policy shown in table 2.2 for software running on it.

Simply said, Sancus enforces a policy where only a software module has exclusive access to its own text and data section. A module can be entered through only one entry point.

The identify of a Sancus enclave consist of a two parts: (*i*) a hash of the content of the text section and (*ii*) the start and end addresses of the text and data sections. This identity is used in the derivation of the enclaves specific key $K_{N,SP,SM}$. If the identity of an enclave and thus its key change, the software provider will be able to detect this, because the enclave will lack the correct key to attest itself.

When enabling protection for an enclave, the processor will zero out the data section of the enclave, as it is not part of the identity and thus not attested.

Sancus stores meta data in processor reserved memory. This metadata consists of the identity of each enclave, its cryptograpic key and a unique identifier. Additionally, the processor keeps track of the last executing enclave and the next unique identifier it will assign to an enclave.

Cryptographic keys are only exposed through a few instructions which indirectly access the cryptographic keys. Of course, the cryptograpic key that is used for the operation depends on which enclave is executing at the time of calling the instruction.

Remote attestation, secure linking and secure communication can all be achieved by using the enclave's specific key $K_{N,SP,E}$. Since this key is only availabe to the enclave, and is derived from the enclave's identity, the software provider can be assured that when this key is used, it is indeed communicating with the intended enclave.

Conclusion

Sancus is able to achieve strong security guarantees with minimal overhead for embedded microprocessors.

2.3 Microarchitectural Side-Channels

Low-level vulnerabilities which break software isolation, as discussed in Section 2.1, are a well researched and understood area of software security. In the last decade, researchers have focused on a new class of powerful attacks: side-channel attacks.

This section is divided as follows: Section 2.3.1 discusses microarchitectural optimizations, which are the rootcause of side-channel and transient execution attacks. Section 2.3.2 discusses transient execution and Section 2.3.3 discusses Nemesis.

2.3.1 Microarchitectural Optimizations

Since the beginning of hardware development, processor manufacturers have sought ways to increase the processing speed of their processors. Initially this was achieved by increasing the clock speed, while at the same time fitting more and more components on the same surface area. Eventually, physical limits have been reached, and thus hardware manufacturers have switched to other ways to increase performance.

Cache. A straightforward example is the processor's cache, which contains a subset of main memory. For each memory access, the cache is first checked. In case the address was found, the associated data is returned. This is called a cache hit. In the other case, a cache miss, the address is forwarded to main memory. Since the cache is a relatively small, but easy to access memory for the processor, access times can be up to five times faster [30]. If cache hits occur often enough, performance is increased significantly.

Reordering the Instruction Pipeline. Complex instruction set architectures (CISC) first decode instructions into multiple micro-ops (μ-ops), before processing each μ-op separately. This design choice has a few advantages. First, it allows updates of the CPU on the microarchitectural level by updating the way instructions are decoded in μ-ops. Second, μ-ops can often be processed independently of each other, allowing the processor to better use its full processing power, by ensuring that each of its executing units are busy as much as possible. To do this, the processor executes μ-ops out-of-order, keeping track of dependancies between different μ-ops in the reorder buffer (ROB). Of course, the processor will commit (i.e., make the results of the instructions visible at the architectural level) the instructions in order to ensure functional correctness at the architectural level.

In the case of a fault, any outstanding μ-ops results in the ROB will be squashed, thus discarding their results. However, side effects as a result of the out-of-order execution may change the microarchitectural state of the processor.

Speculative Execution. Plenty of software contains conditional branches where code may or may not be executed depending on some boolean condition. To prevent the processor from stalling the entire execution of the process, the processor will try to predict the outcome of the conditional branch and speculatively execute instructions along the predicted path. If the processor guessed correctly, the processor can commit the speculatively executed μ-ops, yielding a moderate performance gain. In the case of an incorrect guess, the processor will squash all μ-ops results in the ROB, before continuing on the correct path, paying a small performance penalty for the extra, unneccessary work that was done.

Similarly to the out-of-order execution, instructions that are executed and discarded because of a misprediction, can still cause a change in the microarchitectural state of the processor.

2.3.2 Transient Execution Attacks.

As discussed above, because of hardware optimizations in the microarchitecture of the processor, the processor might execute instructions ahead of time, expecting that they would be executed later anyway. However, in the case of a fault or misprediction, the processor has to disregard all executed instructions following the fault or misprediction. Transient instructions are instructions that are executed at the microarchitectural level, but are not committed to the architectural level. Transient execution attacks use transient instructions to encode unauthorized data in the microarchitectural state, such that they can be later recovered through a covert channel, such as Flush+Reload [40].

A recent exhaustive study [6] classified all known transient execution attacks, resulting in two main categories, Spectre-type and Meltdown-type. These categories are based on the cause of the transient execution. For Spectre-type attacks, the cause is misprediction of a branch, while for Meltdown-type attacks, faults cause the transient execution. In the next sections, spectre and meltdown will be briefly discussed.

Spectre

Spectre [15] allows an attacker to read, otherwise inaccessible, memory of a victim application, by transiently executing instructions found within the victim application. To do so, the attacker first finds the instructions in the address space, which will encode the unauthorized data in the microarchitectural state, e.g. the cache. We refer to these instructions as gadgets, similarly to ROP-gadgets [26]. Then the attacker tricks the processor into mispredicting a branch, e.g. whether a conditional jump is taken or the target address of a branch instruction, to transiently execute the gadget to encode data from the victim application into the microarchitectural state. The attacker can then recover the data through a covert channel. Note that the attacker can only read memory which the victim is allowed to access. The attacker does not increase his privileges and can thus only leak memory belonging to the victim application.

Meltdown

In Meltdown [17] the authors showed how to bypass the userspace kernelspace isolation and thus read memory belonging to the kernel. Since the attacker can read kernel memory, Meltdown is a privilege escalating transient execution attack. In this sense, Meltdown is more powerful than Spectre. However Spectre is more difficult to patch [2, 6]. To do so, the attacker executes an instruction that will always result in a fault, i.e. dividing by 0. The faulting instruction is followed by the transient execution which encodes the data located at an inaccessible address into the microarchitectural state, e.g. the cache. The attacker can then recover the data through a covert channel.

2.3.3 Nemesis

The previous sections showed how transient execution attacks allows an attacker to leak memory belonging to either a victim application or the kernel. Transient execution attacks exist because of hardware optimizations found in the microarchitecture of the processor. Another necessary component for a succesful transient execution attack is the covert channel that will be used to extract encoded data from the microarchitectural state. Most commonly, covert channels based on the cache are

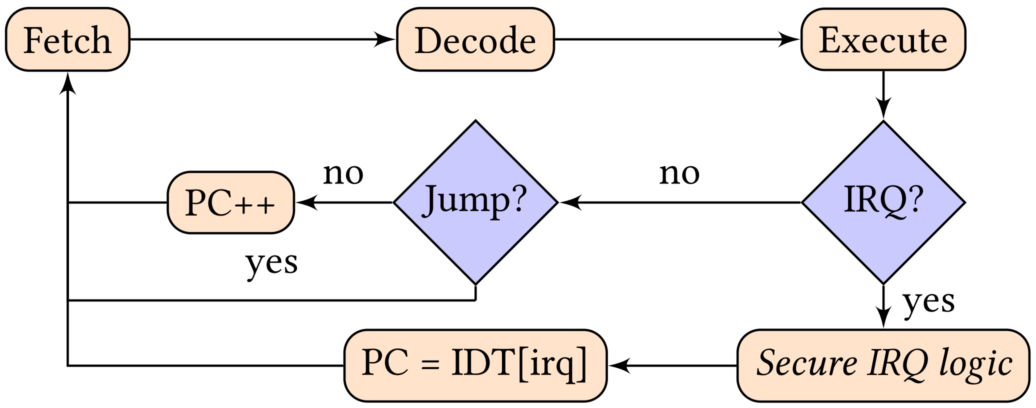

Figure 2.1: **Fetch-Decode-Execute Cycle:** CPU fetches, decodes and executes instructions from memory. Between instructions interrupts are handled. From: [35].

used, such as Flush+Reload[40]. Thus side-channels and transient execution attacks abuse the complexity of the instruction pipeline in most modern processors.

Since the root cause of side-channels and transient execution attacks are commonly microarchitectural optimizations, little research was done into side-channels for embedded microprocessors, as they lack those optimizations. However, Nemesis [35] is a recently found software only side-channel which is present in even the most simplistic instructions pipelines, such as those found in microprocessors intended for embedded devices. Nemesis is thus the first attacker controlled side-channel for both high-end and low-end processors.

Fetch-Decode-Execute Cycle

The simplest instruction pipeline found in processors is the fetch-decode-execute cycle, shown in Fig. 2.1 The processor executes instructions sequentially as they are provided by the instruction stream. First the processor fetches the next instruction from the memory location pointed to by the program counter. Then the processor decodes the instruction to understand what data is required to execute the instruction. In this step, the processor also fetches all required data from memory, if necessary. Lastly, the processor executes the instruction and starts the cycle again by fetching the next instruction.

This process is completely deterministic and predictable. However, it is desired to be able to interrupt the execution of a program to handle an urgent event, such as a timer firing or dealing with I/O. Since such events can occur at any time, processors delay checking for any of these events upon retiring an instruction. That is, upon completing the execution, but before fetching the next one, the processor checks if there are any pending interrupts. If so, the processor stores the (minimal) execution state, before handing control to the corresponding interrupt routine. If there is no pending interrupt, the processor continues as normal. This approach is the most common one [20, 13, 14], and makes handling of interrupts more predictable (as they are only handled at the retirement of an instruction).

Nemesis Side-Channel

Van Bulck et al. noticed that the common way of handling interrupts, i.e. at retirement of an instruction, introduces a delay equal to the amount of cycles left to execute. This delay propagates through the entire cycle, as can be seen in Fig. 2.2. Carefully timed interrupts allow an attacker to time this delay, and thus derive the execution length, i.e. the amount of cycles required to execute the instruction.

Van Bulck et al. showed how Nemesis [35] can be used to differentiate between conditional jumps being taken or not taken. By configuring a timer such that instructions with different execution lengths will be interrupted, depending on whether or not the jump was taken, allows the attacker to determine the outcome of the jump. If the conditional jump depends on secret data, the attacker can then use this knowledge to reverse control flow and leak secret data.

A Case Study. Van Bulck et al. considered the following case study application [35]: the Bootstrap Loader (BSL) of the MSP430. The BSL software is executed upon reset and allows device owners to reprogram their device. To ensure that only legitimate device owners can reprogram the device, the BSL software is password protected. The password is 32 bytes long.

```
   cmp.b @r6+, r12                 cmp.b @r6+, r12
   jz 1f                           jz 1f
   bis #0x40, r11                  bis #0x40, r11
1: ...                             jmp 2f
                               1:  nop nop nop nop 2: ...
```

Listing 2.1: Body of the (un)balanced BSL password comparison loop. From: [35].

The original password comparison implementation of Texas Instruments, shown on the lefthand side of Listing 2.1, is vulnerable to a start-to-end timing attack. Listing 2.1 only shows the main part of the password comparison loop, in which the value pointed to by r6 is compared with the value in r12. If any of the bytes differ, a bit is set in r11 to indicate an incorrect password. Note that the two-cycle bit set instruction (`bis`) is only executed in case of an incorrect password. An attacker could then bruteforce the password byte by byte by measuring the overall execution time. This vulnerability can be closed by balancing the else branch with no-ops to ensure a constant overall execution time.

However, a Nemesis attacker can still deduce the correctness of individual bytes. By timing interrupts such that they arrive one cycle after the execution of the conditional jump instruction, either a two-cycle `bis` instruction or a one-cycle `nop` instruction will be interrupted. The attacker can then measure the interrupt latency

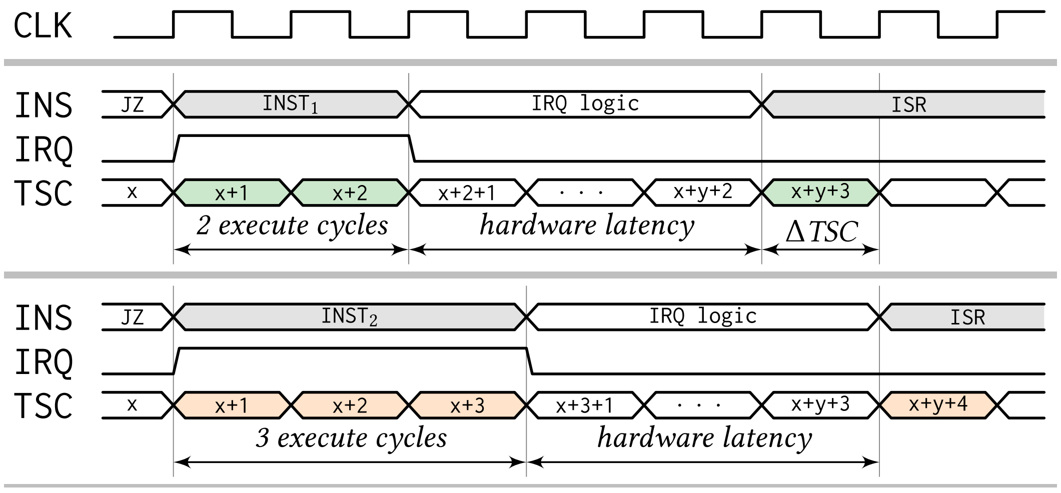

Figure 2.2: Interrupts arriving in a multi-cycle instruction are delayed proportional to the amount of cycles left to execute. From: [35].

to deduce the instruction length and thus whether the byte was correct or incorrect. This reduces the search space from exponential to linear.

2.4 Conclusion

In recent years, much research has been done into side-channels in high-end processors, which lead to new, dangerous classes of attacks: side-channel attacks and transient-execution attacks. However, our current understanding on side-channels in emdedded microprocessors is little. Thus, there are still many open research questions about microarchitectural side-channels, more specifically about their prevelance in low-end processors and their (automatic) exploitation.

Chapter 3

Sancus-Step

As PMAs offer strong security guarantees to their enclaves, few angles of attack are still possible. Hence, side-channel [40, 39, 35] attacks, transient execution attacks [32, 15, 17] and classical memory safety vulnerabilities [26] cover the leftover attack surface. Since memory safety vulnerabilities rely on bugs in the code, they are not a reliable way to attack enclaves. As such, most research effort goes towards side-channel attacks and transient execution attacks.

However, the power of side-channel attacks and transient execution attacks is determined by their information leakage granularity, both in time and in space. Page table based attacks [39, 5] have a spatial granularity of a page size (often 4 *KiB*) while cache based attacks [40, 12] have a spatial granularity of a cache line (often 64 B). Spatial granularity is difficult to increase, as it is inherent to the side-channel being used. On the other hand, temporal granularity can be increased. Researchers developed techniques to continuously monitor the side-channel to not miss any information leakage, e.g. by having a concurrent thread monitor the cache or page table entries [5].

To increase the temporal granularity of side-channel attacks even more, SGX-Step [34] was developed. SGX-Step allows an attacker to single step (i.e. interrupt after each instruction) a victim SGX enclave without relying on any application-specific knowledge. By single stepping a victim SGX enclave after each instruction, the side-channel can be inspected after each instruction. This allows an attacker to unambigously identify when e.g. something is brought into the cache or a certain page is accessed. Additionally, SGX-Step comes as a small library and kernel module, making it easy to use. Thus, SGX-Step both increased temporal granularity of side-channel attacks and the ease at which they can be mounted.

Sancus-Step is designed and developed with a similar goal in mind: reducing an attacker's effort to have maximal temporal granularity when doing side-channel attacks against Sancus enclaves. While maximal temporal granularity can be achieved against Sancus enclaves, it requires expertise and is highly error-prone [35]. As such, Sancus-Step focuses on the ease of use.

First, in Section 3.1 the necessary prerequisites for Sancus-Step (and Nemesis) are discussed. Section 3.2 discusses Sancus-Step's design and implementation. Then in Section 3.3 known atomicity issues and their corresponding attacks are discussed. This is an orthogonal area of research to Sancus-Step and side-channels. We conclude in Section 3.4.

3.1 Trusted Runtime

Originally, Sancus had no support for interruptible Sancus enclaves [24]. In 2016, Van Bulck et al. proposed a hardware mechanism to support secure interrupts of Sancus enclaves [33], similar to previous work on secure interrupts in microprocessors [16, 9]. However, this work was only partially upstreamed into the Sancus compiler repository on github[1]. Thus, to support Nemesis and Sancus-Step, a simple implementation of secure interrupts was implemented in this master's thesis, based on the work of Van Bulck et al. [33].

In Section 3.1.1 we discuss the secure interrupt engine that was proposed and upstreamed by Van Bulck et al [33]. Section 3.1.2 discusses the trusted runtime stubs that the Sancus compiler inserts to implement the secure compilation scheme [3] and the modifications necessary to support interruptable enclaves. These modifications are merged upstream into the Sancus compiler repository on github.

3.1.1 Secure Interrupt Engine

To securely interrupt enclaves the following an secure interrupt engine, similar to other secure interrupt engines [16], was implemented in the Sancus hardware (to maintain a hardware only TCB) by Van Bulck et al [33, 35]. The secure interrupt engine does the following whenever an enclave is interrupted: It (*i*) stores all registers on the stack, (*ii*) saves the stackpointer into a dedicated memory location and (*iii*) clears all registers before (*iv*) jumping to the corresponding ISR.

3.1.2 Trusted Runtime Stubs

The Sancus toolchain adheres to the secure compilation scheme, as proposed by [3]. Thus, upon entry and exit of an enclave, runtime checks are performed to ensure integrity of the control flow and confidentialty of the enclave's data.

Entry. Upon entry, the trusted runtime disables interrupts to switch the unprotected stack with the protected stack atomically. After switching the stacks, interrupts are enabled again. Then it checks if the call of the enclave is a return, i.e. the enclave previously called some other function which now returned. If it is, the enclave restores the program counter from the stack, such that the enclave continues after the call. Otherwise it calls the logical function called by the caller (recall that Sancus

[1] https://github.com/sancus-pma/sancus-compiler

has only one entry point and thus enclaves need to multiplex logical entry points on its one physical entry point).

To support interruptable enclaves, we modified the trusted runtime to check if the enclave was previously interrupted and is now being resumed. If its the case, the trusted runtime restores the saved execution context from the stack, including the program counter, thus resuming the enclave. The saved execution context is stored on the stack by the secure interrupt engine.

Note that the distinction between the stubs is rather vague in the source code. As such part of the changes to the entry stub (as described above) are located in the exit stub file (`__sm_exit.s`) to keep a coherent coding style. However, it makes more sense to think of the changes as part of the entry stub. As such we provide the modified code of both files in Appendix C and Appendix D.

Exit. Whenever an Sancus enclave exits through normal flow, i.e. calling code outside of the enclave, the trusted runtime checks saves the stackpointer of the enclave and clears all unused registers. Then the trusted runtime jumps to the code outside of the enclave. Nothing changed to support interruptable enclaves.

3.2 Design and Implementation of Sancus-Step

This section discusses the design of Sancus-Step in Section 3.2.1 and the implementation of Sancus-Step in Section 3.2.3 and Section 3.2.2. In Section 3.2.3 more low-level details are discussed and in Section 3.2.3 we evaluate Sancus-Step and discuss future work.

Sancus-Step is available as part of Sancus[2]. Its source code has been merged upstream into the offical Sancus github repository[3].

3.2.1 Main Design Goals

Sancus-Step is designed for achieving the following goals.

Easy to use

Sancus-Step reduces the required knowledge to mount succesfull attacks by being automated as much as possible and configurable (see below). Sancus-Step comes as a small library. Hence, using Sancus-Step to mount e.g. a Nemesis attack against a Sancus enclave requires just the use of a few predefined C macros and C functions. This produces a trace, which can later be visualised or post-processed to leak secret data. This will be further discussed in Chapter 4. Listing 3.1 shows a minimal working example to mount a Nemesis attack against a victim enclave.

[2] https://distrinet.cs.kuleuven.be/software/sancus/
[3] https://github.com/sancus-pma/

```c
#include <sancus_support/sancus_step.h>
// All functions starting with __ss are part of Sancus-Step

int main()
{
  // ... setup code
  __ss_start(); // Initialize Sancus-Step and start single stepping
  foo_enter(); // The entry point of the enclave Foo
  __ss_end(); // End single stepping
}

/*
 __ss_print_latency is the function
 containting the attacker's code to run after each instruction
*/
SANCUS_STEP_ISR_ENTRY(__ss_print_latency)
```

Listing 3.1: A minimal working example of mounting a Nemesis attack against a sancus enclave.

Sancus-Step comes with functionality to mount Nemesis out of the box. However, if desired, Sancus-Step can as easily be configured to execute an attacker's own code to mount a different attack. The attacker code will be executed after every instruction as shown in Fig. 3.1.

Automated Attacks

Sancus-Step supports Nemesis attacks out of the box by having predefined functions which return instruction latencies, making it easy for an attacker do real-time processing or post-processing. However, in the future new side-channels or attacks might be discovered, and thus it is desirable to be able to run arbitrary attacker code after each instruction of the victim enclave. As such, the `SANCUS_STEP_ISR_ENTRY` macro accepts a function pointer as an argument which it will call when it is running in single stepping mode. That is, when not determining the Sancus-Step parameters as explained above.

Automated Future Proof Configuration

To be easy to use, Sancus-Step fully automatically configures the necessary parameters, which might change across versions of Sancus and its libraries. Consequentially, Sancus-Step works across version changes without needing any information from the attacker. Sancus-Step sets up its own enclave, called the Configuration Enclave, shown in Fig. 3.1. The configuration enclave works together with untrusted code to automatically deduce the following parameters.

Enclave Exit Latency. The Enclave Exit Latency is the amount of clock cycles required for the Sancus hardware to securely interrupt an enclave. More specifically, this is the time between the last cycle of the currently executing instruction and the execution of first instruction of the corresponding ISR. Important to note here is the "last cycle of the currently executing instruction" as interrupts can arrive in any cycle of a multi-cycle instruction. This is the minimum amount of cycles required to interrupt a Sancus enclave, as the hardware only handles interrupts at instruction retirement. Although this time is entirely determined by the hardware, and thus unlikely to change, Sancus-Step still automatically determines this value to make it as future-proof as possible.

Enclave Resume Latency. The Enclave Resume Latency is the amount of clock cycles between the execution of the `reti` instruction (return from interrupt) at the end of the ISR and the start of execution of the next instruction in the victim enclave. This timing is used to configure the timer at the end of the ISR such that Sancus-Step reliably single steps the victim enclave. Upon entry of an enclave, the trusted runtime checks if it has been interrupted before. If it was interrupted before, it restores its state and returns to the next instruction, thus continuing execution right after where it got interrupted. It is exactly this time that Sancus-Step requires to single step enclaves. Since this time is determined by the execution time of the trusted runtime, it is likely to change.

3.2.2 The Configuration Enclave

The configuration enclave is an attacker controlled and deployed enclave, which allows the attacker to determine parameters of Sancus-Step automatically. Furthermore, some global variables are allocated. Sancus-Step uses some of these global variables to communicate with the configuration enclave and uses the rest to store the final values of the configuration parameters.

Configuration occurs in three steps:

1. Sancus-Step determines the amount of clock cycles between the moment the untrusted code calls the configuration enclave and the execution of the first instruction in the configuration enclave. Sancus-Step configures a timer to increment each clock cycle without generating any interrupts (continouos mode). Sancus-Step then calls the configuration enclave. On entry the configuration enclave stores the value of the timer a dedicated global variable.
2. Sancus-Step configures the timer to generate an interrupt such that the timer will interrupt the first instruction of the configuration enclave (using entry_delay). Then Sancus-Step calls the configuration enclave. When the enclave gets interrupted, the Sancus-Step ISR will configure the timer in continuous mode and resume the enclave. The second enclave instruction saves the value of the timer such that it can be returned to Sancus-Step. The returned

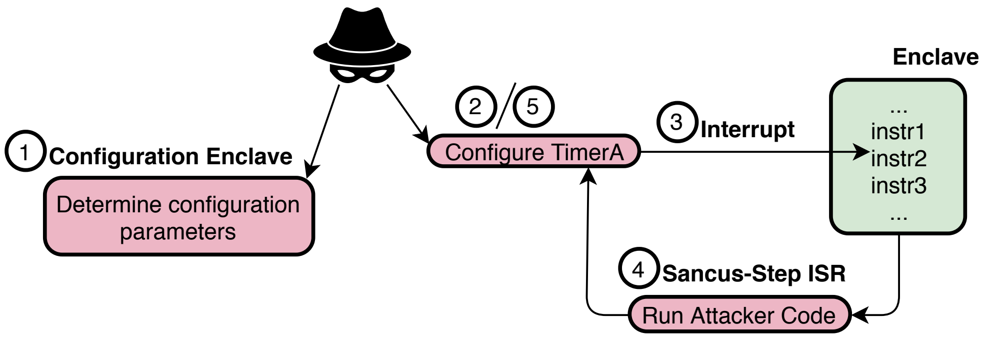

Figure 3.1: Sancus-Step: Overview of the attacker framework for single stepping sancus enclaves.

value is the enclave resume latency plus some constant, known overhead. Sancus-Step can thus compute the actual enclave resume latency.

3. To determine the enclave exit latency, Sancus-Step configures the timer to interrupt the enclave once more. This time, the timer will interrupt the enclave setting the timer in continouos mode. As such, while the timer is running, the Sancus hardware is busy securely interrupting the configuration enclave. Upon entry of the Sancus-Step ISR, it saves the value of the timer. Similarly, the measured latency is the actual enclave exit latency plus some constant, known overhead. As such, Sancus-Step can compute the actual enclave exit latency.

3.2.3 Automated Single-Stepping

Fig. 3.1 shows how an attacker performs automated single stepping of a victim enclave. Sancus-Step works in two phases: (*i*) the configuration phase (step 1 in Fig. 3.1), in which parameters of Sancus-Step are automatically configured, and (*ii*) the single stepping phase where attacker chosen code is executed after each instruction of the victim enclave (step 2-5 in Fig. 3.1). Note that step 1 and step 2 each only have to be executed once.

Step 1. In the first step, Sancus-Step configures its parameters by using the Configuration Enclave as described above.

Step 2. In step 2 the attacker configures the timer for the first time. This step is performed by the start function of Sancus-Step (`__ss_start`). Sancus-Step disables interrupts and configures timer A to fire an interrupt after zero cycles, i.e. timer A will immediately generate an interrupt upon enabling of interrupts. The trusted runtime disables interrupts upon entry so it can switch the unprotected stack with the protected stack securely. Afterwards they enable interrupts again, as to allow e.g. an operating system to suspend the enclave for scheduling purposes. Sancus-Step uses this knowledge to immediately interrupt the enclave once it has switched the stacks and enables interrupts again.

Step 3. The enclaves executes exactly one instruction before getting interrupted. This allows the attacker to measure the instruction length of the instruction. However, care has to be taken as not to interrupt the enclave before it executed a new instructed or after it executed multiple instructions. If no new instruction is executed, no progress is being made. If multiple instructions are executed, then some information is lost, as the measurement now covers multiple instructions.

However, compared to SGX-Step [34], Sancus-Step's timings are completely deterministic, as the MSP430 processor architecture does not contain any microarchitectural optimizations. Thus after determining the values of the latencies, as described above, no extra care has to be taken to ensure interrupts after every instruction.

Step 4. The MSP430 timer keeps track of time in a special register, TAR, and increments the value of TAR each clock cycle. Sancus-Step configures the timer to generate an interrupt upon reaching a certain value, which we will call the *trigger* value. Afterwards, the timer keeps on counting, i.e. it does not overflow and restart from zero. Thus, the ISR stores the value of TAR into a dedicated memory location to measure the delay. Sancus-Step can calculate the instruction latency by substracting the trigger value and the enclave exit latency from the measured delay. Then it checks if an enclave got interrupted. If no enclave was interrupted, it assumes the enclave is done executing and disables the timer, before resuming the unprotected code that was interrupted.

If an enclave was interrupted in step 3, the ISR checks if Sancus-Step is configuring the parameters. For a more detailed discussion of this part, we refer to Section 3.2.2. In the normal case of single stepping an enclave, the ISR calls the attacker's code.

Step 5. The ISR configures the timer to fire an interrupt on the first cycle of the next instruction in the victim enclave. The ISR then returns to the enclave to allow it to continue execution, hence repeating the cycle. Note that the configuration differs from the configuration in step 2.

Technical Implementation

Because Sancus-Step is heavily reliant on precise timings to work automatically, a lot of the library is written in assembly. By doing so, the compiler will not be able to mess up the timings, and thus the automatic configuration.

Furthermore, to rely as little as possible on the compiler and the Sancus entry protocol, the configuration enclave contains only one single entry point. As is evident from the above explanation, this entry point has multiple side-effects when called, even though in each call only one side-effect is of interest. This makes the code more difficult to understand. However, it ensures that Sancus-Step is not reliant on any compiler knowledge to be able to schedule interrupts with 100% accuracy. Indeed, Sancus-Step measures in the first step of the configuration the amount of clock cycles between calling the module and entering it. Since all other code of interest in the configuration enclave is contained within the entry function, it thus knows the two latencies required to configure the timer in step 2 and 3.

Discussion

General Discussion. This section proposed Sancus-Step, an attacker framework targetting Sancus. Sancus-Step allows automated side-channel attacks with maximal temporal resolution againt Sancus Enclaves. By being fully automated, Sancus-Step reduces the attacker's required knowledge to a minimum and is easy to set-up and use.

However, Sancus-Step is still a prototype and has its limitations as well. First, Sancus-Step currently assumes that when unprotected code gets interrupted while single stepping, the enclave has stopped executing. However, this assumption is not correct, as enclaves may call untrusted code during execution. We leave support for calls to untrusted code in enclaves as future work.

Second, the source code of Sancus-Step is mostly assembly. This makes it more difficult to understand and adapt to future needs. An approach would be to have multiple entry points for the configuration enclave, where each entry point has one designated task. However, to ensure correct automatic configuration of the timer for any specific interrupts which are required, each entry point has a corresponding `entry_delay` value. The timer can then be configured to interrupt after $entry\_delay + overhead$ amount of clock cycles, where the overhead is determined by the assembly code of the respective entry function.

Test Enclave. SGX-Step [34] uses debug enclaves to test the single stepping accuracy. SGX allows code to inspect the state of enclave, including the program counter. Thus, to test SGX-Step, one can single steps a debug enclave and check the value of the program counter after each interrupt to verify that at most a single instruction was executed.

However, Sancus does not provide such debug support (inspecting the state of a debug enclave), thus to test Sancus-Step, a specific test enclave has been developed. This test enclave has an entry function containing the code in Listing 3.2. The test enclave simply stores increasing values from zero to fifteen in the global variable counter. The test can thus check after each instruction either that the counter value is still at zero or is one higher than the previous one or is at the maximum value. This test effectively only test the correct single stepping of the entry function body. Hence, we extrapolate this to the entirety of the enclave, i.e. all code executed from the moment it is entered until it is exited.

```
void SM_ENTRY(testEnclave) test()
{
  asm (
    "mov #0, &counter"
    "mov #1, &counter"
    "mov #2, &counter"
    "mov #3, &counter"
    ...
    "mov #14, &counter"
    "mov #15, &counter"
  )
}
```

Listing 3.2: Entry function of the test enclave. The test enclave stores increasing values in the global variable counter.

3.3 Known Atomicity Issues

Researchers often left interrupts and atomicity related issues as future work [27, 31, 29]. However, they acknowledged that adding interrupts would not be without security issues. As such, atomicity related issues are not novel. However, to the best of my knowledge we are the first to practically demonstrate such an attack. Patching the current upstream implementation of interruptable Sancus enclaves is left as future work.

3.3.1 TOCTOU: Calling Another Module

Recall from Section 2.2.2 that when a Sancus enclave A wants to call another Sancus enclave B, A first verifies that B is indeed located at the expected address and correctly protected. This assures A that he will indeed call B. If the verification succeeds, A proceeds with the call of B, possible passing secret data in registers.

Proof of Concept Attack

Overview. By carefully configuring a timer, an attacker can interrupt the execution of A after B has been verified. The attacker can disable B, thus disabling Sancus' memory protection over the entire memory region spanned by B. Sancus only allows the Sancus enclave to destroy itself and subsequently disable the memory protection. However, B might have an entry point to allow clients to destroy itself or depending on the application, B could be tricked into destroying itself. E.g. a password manager containing usernames and their passwords is often protected by a master password. An attacker can try to bruteforce this master password. A possible defense against such a bruteforce attack, could be that the vault destroys itself after a small amount of incorrect guesses. For the remainder of this attack, we assume that the B can be disabled.

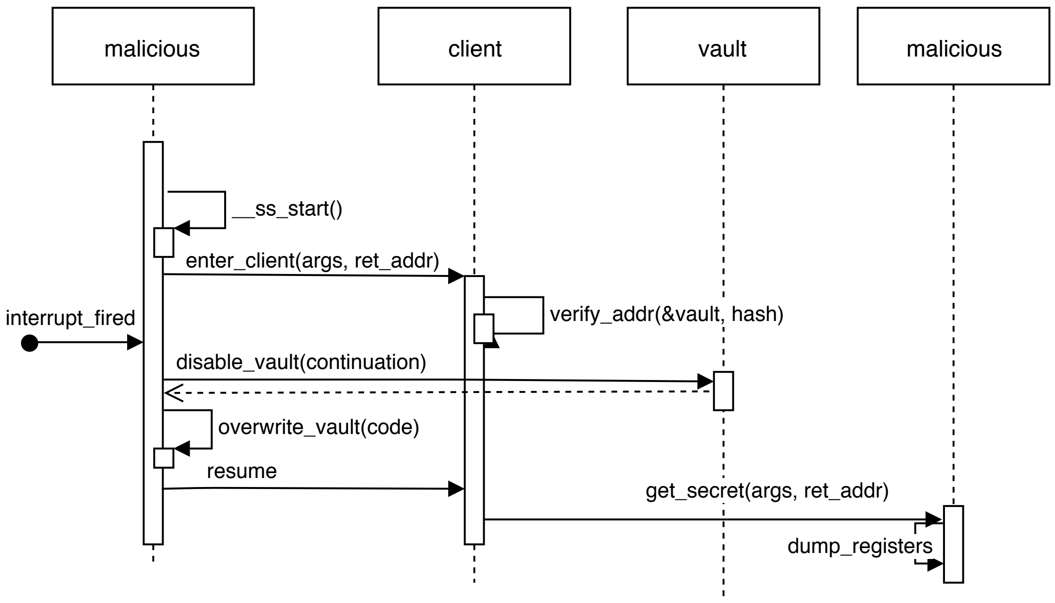
malicious
client
vault
malicious
__ss_start()
enter_client(args, ret_addr)
verify_addr(&vault, hash)
interrupt_fired
disable_vault(continuation)
overwrite_vault(code)
resume
get_secret(args, ret_addr)
dump_registers

Figure 3.2: A proof of concept attack against non-atomic verification of enclaves.

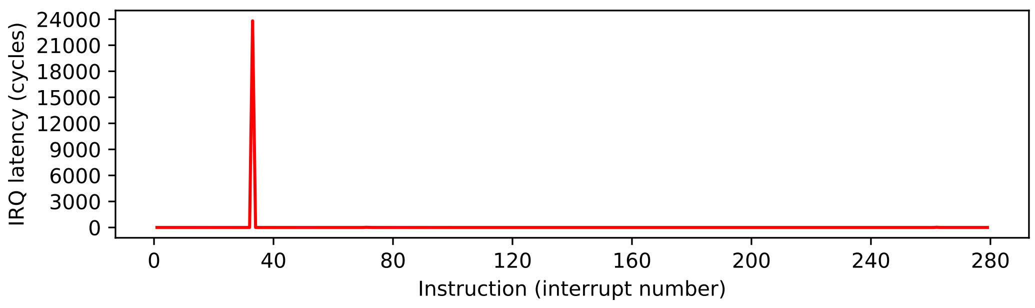

Figure 3.3: A trace extracted from a client enclave verifying another trusted enclave. The clear peak is the hardware instruction performing the verification.

After disabling B the attacker can now overwrite the public section of B with its own chosen code. The attacker then resumes A, which now unknowlingly calls the attacker's code instead of B. The attacker can now easily dump the content of each register, and thus easily leaks any, potentially secret, data A passes to B. This proof of concept clearly demonstrates the kind of issues arising with regard to interruptability of enclaves.

Proof of Concept Implementation using Sancus-Step. We demonstrate the feasibility of the aforementioned attack using the vault scenario from above. Additionally, by using Sancus-Step, we further showcase the flexibility and ease of use of Sancus-Step. Fig. 3.2 gives an overview of the proof of concept attack.

Sancus uses special hardware instructions for cryptographic operations, including the (first) verification of another enclave. Since generally cryptographic operations take considerable amount of time, they have a huge instruction latency. This is evident from Fig. 3.3. By single stepping the client and measuring the instruction latency of each instruction, it becomes simple for an attacker to automatically determine when the vault enclave can be destroyed. After each instruction, the attacker's code checks if the measured instruction latency is above a certain threshold value. If so, it triggers the destruction of the vault. The attacker can then overwrite the vault with its own code before resuming the client. Eventually the client will jump to the attacker's code, leaking any secret data passed in registers.

3.3.2 Atomic Exit

When a Sancus enclaves finishes executing it will save its private stack into a dedicated memory location, clear all registers, including status bits in the status register, before returning to the caller. However, when the last instruction of the enclave is interrupted, the Sancus hardware securely interrupts the enclave. During this secure interrupt, the hardware (*i*) stores all registers on the stack, (*ii*) saves the stackpointer into a dedicated variable and (*iii*) clears all registers before (*iv*) jumping to the corresponding ISR. Upon entry of the enclave, the enclave checks if it was interrupted. If so, it restores the old stack and pops all the register values of the stack, including the program counter. However, because the last instruction of the enclave is an unconditional jump to the outside, the stored program counter value on top of the stack is an address outside of the enclave. As such, when the enclave pops the program counter value of the stack, it jumps to the outside code. This in itself is fine behaviour.

However, the stored stackpointer in the enclave in the dedicated memory location still holds the value of the "filled" stack, i.e. the stack after the interrupt but before the resume. Thus there is now a whole execution context stored on top of the stack, which will never be popped again. An attacker repeatedly interrupting the last instruction of the enclave can thus store a new execution context on top of the stack. Eventually the stack will overflow into unprotected memory, allowing the attacker to read the execution context of the enclave right before the end. This is a clear breach of the secure compilation scheme, as proposed by [3].

3.4 Conclusion

This chapter presented Sancus-Step, an attacker framework enabling single stepping of Sancus enclaves. Sancus-Step requires interruptible enclaves. However, adding support for interruptible enclaves raises some security concerns, as was demonstrated by a proof of concept attack against the verification and calling of enclaves.

Sancus-Step considerably reduces attacker's expertise required to do automated side-channel analysis on Sancus enclaves. Sancus-Step achieves this by being easy to use, configurable and fully automated without relying on any compiler knowledge.

Chapter 4

Side-channel Leakage in Compiler Generated Code

The Sancus toolchain automatically generates and inserts code into the SM to aid the developer. The inserted code is SM-independent and can thus easily be generated by the toolchain, freeing the developer of the burden to write the same code for every SM. This code, commonly called stubs, includes common tasks such as protecting the integrity of the SM, the confidentiality of private data and functions to compute arithmetic operations. The inserted code is independent of the SM. Hence it can easily be generated by the toolchain and inserted whenever it is needed.

Since most developers will not verify the code inserted by the toolchain, it essentially becomes part of the TCB. It would be disastrous if a security-aware developer tries to develop a secure application on top of Sancus, while the Sancus toolchain unknowningly introduces vulnerabilities into his code.

4.1 Attacker Model and Goals

In this chapter, we will analyse the information leakage of the stubs inserted by the Sancus toolchain with regard to two different attacker models: (*i*) start-to-end attacker, capable of measuring the execution time of function calls and (*ii*) a Nemesis [35] attacker, capable of (repeatedly) interrupting the function call to measure the duration of the instruction that was interrupted. The Nemesis capable attacker has access to Sancus-Step. Furthermore, the attacker is capable of the following: (*i*) inspecting the source code of the victim application and (*ii*) running the victim application, possibly modified, with chosen inputs (if applicable). Both attacker models extend Sancus' attacker model. Thus the attacker has kernel-level control and can therefore ensure minimal noise. Interrupts are only caused by timers the attacker configures and there are no delays caused by scheduling. This master's thesis thus assumes single-threaded applications running on the openMSP430 microprocessor enhanced with the Sancus architecture.

The following stubs are inserted by the compiler:

- **Arithmetic stubs**: arithmetic stubs are used to securely handle arithmetic operations which are not supported by the MSP430 architecture, such as (unsigned) multiplication, (unsigned) division and (unsigned) modulo.

- **Entry and Exit Stubs**: The entry and exit stub ensure the integrity and confidentiality of the SM when entrying and exiting its code section. These stubs check arguments passed to the SM upon entry and clear unused registers upon entry and exit. By clearing unused registers upon entry, it becomes harder for the attacker to abuse flaws in the SM's code, while clearing unused registers upon exit ensures that no private data can be leaked.

We will extensively attack the multiplication stub and provide an initial exploration of the other stubs.

4.2 Compiler Runtime Library

The Sancus compiler is built on top of the Clang/LLVM infrastructure. According to the Clang/LLVM toolchain documentation, Clang automatically links against the necessary runtime libraries for C programs. The runtime library Clang uses by default is target-specific, but can be changed by the user[1].

Clang comes with its own runtime library (Compiler-rt), but can also use the runtime library of Gcc (Libgcc). Compiler-rt is developed to be compatible with Libgcc. For the MSP430 microprocessor, Clang defaults to the runtime library of Gcc (Libgcc), as Compiler-rt does not provide the necessary support for the MSP430 architecture.

Runtime libraries contain routines to handle operations that are not supported by the target architecture, such as integer multiplication and division, floating point and fixed point operations, exception handling and more. For Sancus specifically, the major amount of library calls to the runtime library are integer multiplication and division. However, the Sancus linker automatically replaces these calls to secure stubs, which are based on the source code of the stubs found in Libgcc. Thus all the code that the Sancus toolchain inserts, is based on Libgcc. The source code of Libgcc is handwritten, either in C or Assembly. Whether C or Assembly is used, depends on the actual function. C is preferred over Assembly, so only when C is impossible to use, Assembly is used. Since these stubs are based on handwritten code that varies from target to target, the analysis in this master's thesis has to be redone for each possible processor architecture. However, we expect that this analysis will have similar results for different processor architectures, as a quick inspection of the Libgcc source code revealed duplicated code for multiple processor architectures.

```c
#include <sancus/sm_support.h>

DECLARE_SM(SM_mul, 0x1234);

int SM_DATA(SM_mul) a = -63;
int SM_DATA(SM_mul) b = 6;

int SM_ENTRY(SM_mul) sm_multiply(void)
{
    volatile int c = a * b;
    return 0;
}
```

Listing 4.1: Code of SM_{mul}. This enclave is the running example for side-channel analysis in the multiplication stub in this master's thesis.

4.3 Analysis of the Multiplication Stub

More complex arithmetic operations, such as multiplication and division are not supported by the MSP430 microprocessor, and are thus emulated in software. Since an SM executes code in its own protection domain, it would have to make an *ocall* to the arithmetic function. However, the operands of the arithmetic function could be secret, so it is undesirable to pass them to the unprotected domain. To solve this, the developer can write a arithmetic function and include it in his module. Of course, as mentioned above, the arithmetic function code is independent of the enclave and can thus be safely inserted by the Sancus toolchain.

This master's thesis analyses side-channel leakage by using a simple module SM_{mul}. The source code of SM_{mul} is shown in Listing 4.1. This module has two hardcoded secret values, a and b, and one entry point. The entry point simply calculates the product of a and b, stores it in a local variable and then returns 0. For the caller, there is no way to determine the secret values of a, b and the product of a and b, by simply calling the entry point of the module. In this master's thesis, this simple module will serve as the basis for our analysis of the multiplication stub and the side-channel information leakage in it. We assume that the module is loaded using confidential deployment (see Section 2.2.2). Thus, the attacker cannot learn the secret constants when loading the module.

Important to note that execution times on the MSP430 CPU are completely deterministic and entirely determined by the instructions. Each instruction's execution time depends on the instruction and the address modes. For a full reference on the execution times of all instructions, we refer the reader to [13] and the appendices in [35].

4.3.1 Explanation of the Multiplication Stub

Pseudocode of the Multiplication Stub

Listing 4.2 shows the source code of the assembly stub as it is inserted by the Sancus toolchain into enclave. Listing 4.3 shows pseudocode of the assembly stub in c.

Note that the stub matches the exact assembly code found in the Libgcc library accompying the mspgcc compiler version 4.6.3 20120406.

```
__sm_mulhi3:
      mov      r15, r13
      clr      r15
1:    tst      r14
      jz       3f
      clrc
      rrc      r13
      jnc      2f
      add      r14, r15
2:    rla      r14
      tst      r13
      jnz      1b
3:    ret
```

Listing 4.2: Assembly code of the multiplication stub

```
unsigned int mul(unsigned  int a,
      unsigned int b)
{
    unsigned int rv = 0;

    while (a!=0)
    {
        if (b & 1)
            rv += a;
        b >>= 1;
        a <<= 1;
        if (b == 0)
            break;
    }
    return rv;
}
```

Listing 4.3: pseudocode of the multiplication stub

General Explanation

As can be seen from the pseudocode (Listing 4.3), multiplication is calculated as a series of additions and arithmetic shifts. Indeed,

$$a * b = \sum_{i=0}^{n-1} a * 2^i * b_i \; (1)$$

where b_i is the i-th bit of b. Multiplication by 2^i corresponds to a shift left by i places. Multiplication by b_i, which is either one or zero, can be implemented as a test. If b_i is zero, then the result is zero, otherwise it is the other operand of the multiplication. Thus as a whole, (1) can be implemented using only shift left, addition and a comparison operator.

Commonly, microprocessors only provide a shift left and shift right instruction. When a value needs to be shifted left or right by more than one, the shifting needs to be done in a loop. However, from the structure of (1) there are many shifts left needed by varying amounts. Thus, for performance reasons, the multiplication stub is optimized in the following way. At the beginning of each iteration, the current value of a is the original value of a (a_{orig}) shifted left by i and the current value of b is the original value of b (b_{orig}) shifted right by i. Because of these two invariants, the body of the for loop becomes simpler and more performant. First, checking if b_i is one or zero becomes a simple bitwise and operation with as second operand one (line 7). Second, if b_i is one, i.e. the test returns true, a already holds the correct value, i.e. it is equal to $a_{orig} * 2^i$, and thus it can be simply added to the intermediate result (line 8). After the for-loop, when all bits of b have been processed, the rv variable contains the final result.

4.3.2 Start-to-end Leakage

We first analyse the start-to-end side-channel leakage of the multiplication stub. This side-channel is applicable even against non-interruptable Sancus enclaves, as was assumed in Sancus 2.0[24].

Leak multiplicant equal to zero

From Listing 4.3 it is clear that the attacker can deduce whether a is zero or not by observing the overall execution time. This difference in overall execution time is caused by the condition of the for-loop. If a is zero, the body of the for-loop is never executed, while if a is not zero it is at least executed once. Hence, a constant overall execution time can be observed if a is zero, as can be seen in Fig. 4.1.

Naive Approach

Let us assume a is not zero, as leaking a equal to zero is trivial. Additionally, when a is zero, there is nothing else that an attacker can leak, as can be seen in Fig. 4.1. Thus the case when a is not zero, is far more interesting.

Fig. 4.2 zooms in on the graph shown in Fig. 4.1. More specifically, Fig. 4.2 shows the overall execution times for a small subset of b values. On Fig. 4.2, it can be clearly seen that the overall execution time of a multiplication increases or decreases in discrete amounts. Additionally, it is clear that multiple values for b yield the same overall execution time when multiplied with five.

Even without much expertise, an attacker can (pre)compute the overall execution times of the multiplication for each possible combination of multiplicand and multiplier in an offline phase. That is, she can generate a 2d matrix, where the overall execution time of $i * j$ is stored in row i and column j. Then, the attacker can execute the victim enclave once in the online phase, measuring its overall execution time and looking it up in the precomputed matrix. This results in a set of candidate pairs (a, b), which when multiplied take exactly the looked up execution time. Thus, the attacker can reduce the unknown, secret values of a and b to a set of candidate pairs (a, b).

This naive approach has a significant overhead, as computing the overall execution times for all possible multiplications of two 16 bit integers still requires 2^{32} measurements.

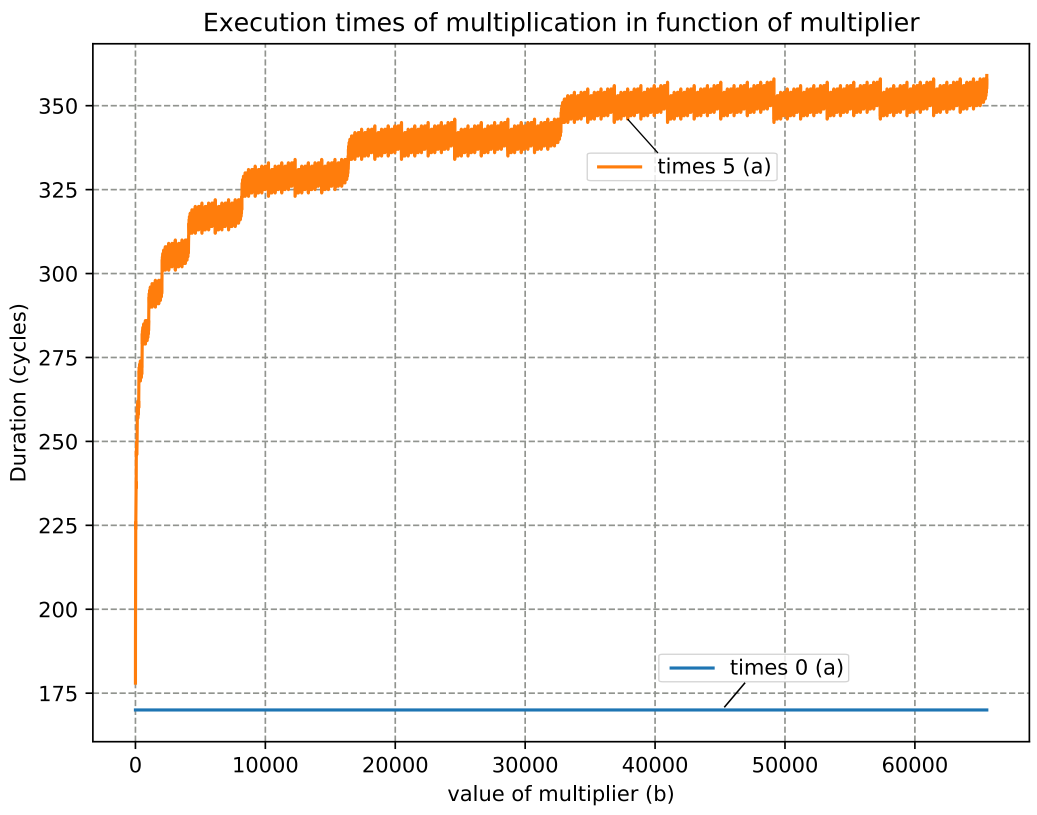

Figure 4.1: Overall execution time shown for two cases: (*i*) 0 * b and (*ii*) 5 * b for all 2^{16} b's.

Analytical Approach

The naive approach has a considerable amount of overhead. From the pseudocode, it is clear that the overall execution time depends on the amount of iterations of the while loop and the amount of times the add instruction (line 8 in Listing 4.3) is executed.

First, it is easy to note that for each 1 bit in b, an extra add instruction is executed. Regardless of the value of a and b, the add instruction always takes one cycle.

Second, the amount of iterations of the while loop depends both on a and b, as a is part of the condition, while b causes an early out if it is zero (the `break` statement on line 12). In each iteration a gets shifted over left by one and b gets shifted over right by one until either a or b becomes zero. Hence, the amount of iterations executed is equal to

$$min(pos\_msb(b) + 1, 16 - pos\_lsb(a))$$

where $pos\_msb(b)$ is equal to the position of the most significant 1 bit of b and $pos\_lsb(a)$ is equal to the position of the least significant 1 bit of a. $pos\_msb(b) + 1$ is the amount of shifts right it takes before b is zero and $16 - pos\_lsb(a)$ is the amount

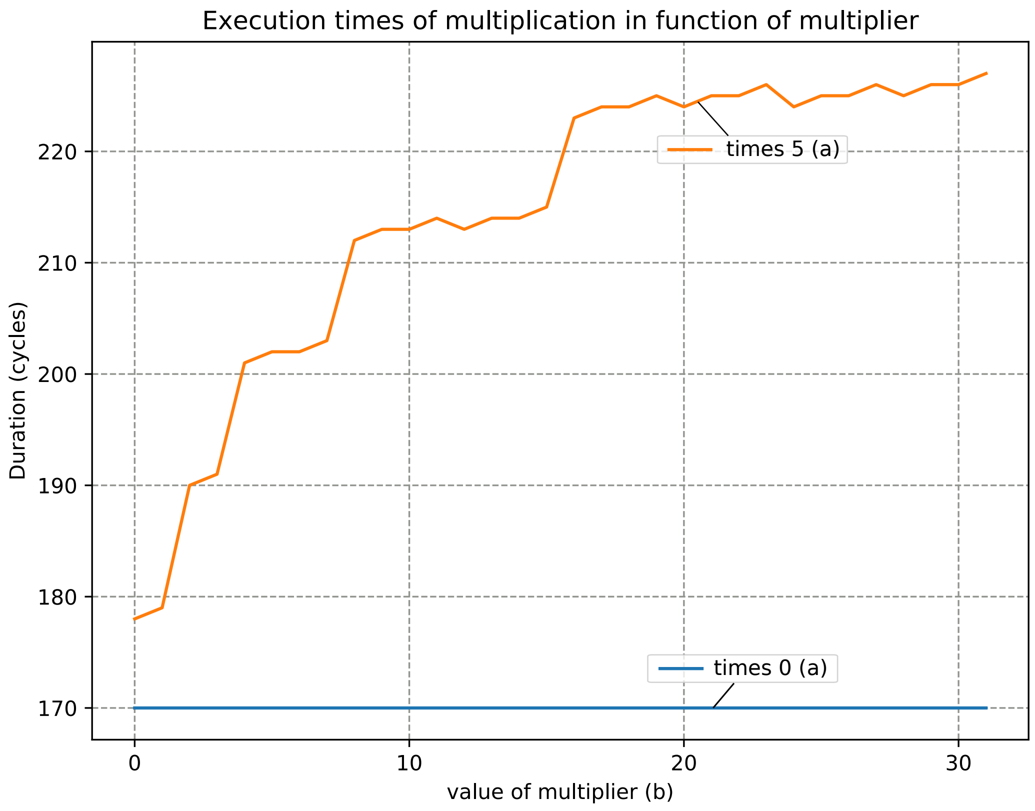

Figure 4.2: Overall execution time shown for two cases: (*i*) 0 * b and (*ii*) 5 * b for a subset (0 - 32) of all multiplier values.

of shifts left necessary before a will equal zero. To illustrate, consider the number

$$n = (0000010001000000)_2$$

$pos\_msb(n)$ is equal to 10, hence $m(n) + 1 = 11$. Indeed, shifting n eleven positions over to the right results in zero. Likewise, $pos\_lsb(n) = 6$, thus 10 shifts lefts are required to make n zero.

Hence, the overall execution time of a multiplication does not change as long as $16 - lsb\_pos(a)$ is bigger than $msb\_pos(b) + 1$. That is, a can be shifted more to the left than b can be shifted to right. Thus, the attacker can use this knowledge to reduce the amount of necessary precomputations.

E.g. if b is equal to $(1000)_2$, then the overall execution time of $a * b$ does not change as long as a does not equal zero modulo 2^{13}. That is, if at least one of a's bits is set in the 13 lowest significant positions. Fig. 4.3 shows for all possible different values of a the overall execution time of $a * b$, when b is equal to $(1000)_2$. From this graph, it is clear that, indeed, for the majority of the possible multiplicands (a), the overall execution time is the same. Thus the attacker does not need to measure all

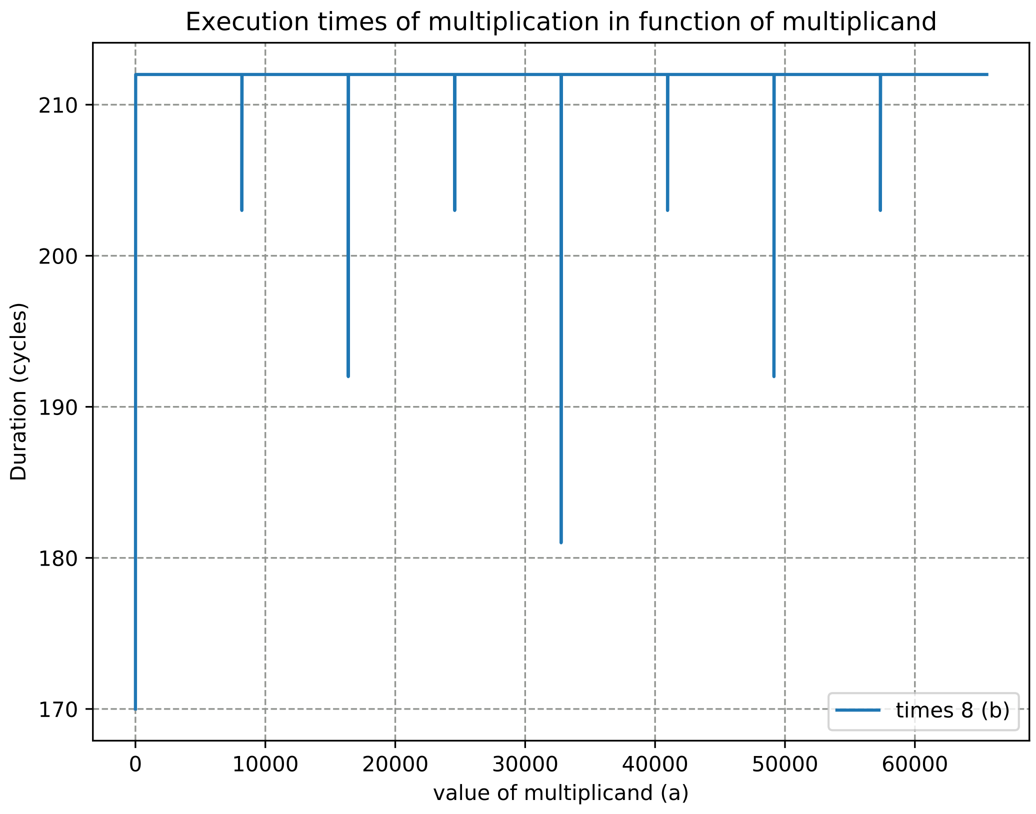

Figure 4.3: Overall execution time shown for $a * 8$ for all 2^{16} a's.

overall execution times of $a * (1000)_2$, but only for a smaller subset of values of a. By similarity, this observation holds for other values of b.

The attacker can then generate the 2d matrix using less computations, and compare the measured overall execution time of the victim multiplication with the generated matrix to deduce all possible combinations of a and b which lead to the measured execution time.

4.3.3 Nemesis Leakage

A start-to-end attacker can leak only so much about the multiplier (b) and multiplicand (a). As will be shown, a Nemesis capable attacker can leak up to the sixteen bits of the multiplier and a tiny bit of information about the multiplicand.

The analysis uses here extensive use of Sancus-Step to mount Nemesis against a victim application using multiplication. As mentioned in Section 3.2.1, Sancus-Step support Nemesis out of the box by making it easy to extract traces, such as the one shown in Fig. 4.4. Fig. 4.5 zooms in on the interesting part of the trace. In general, the analysis here consists of two phases: (*i*) an online phase, where the attacker single steps the victim enclave to extract such traces and (*ii*) an offline phase where

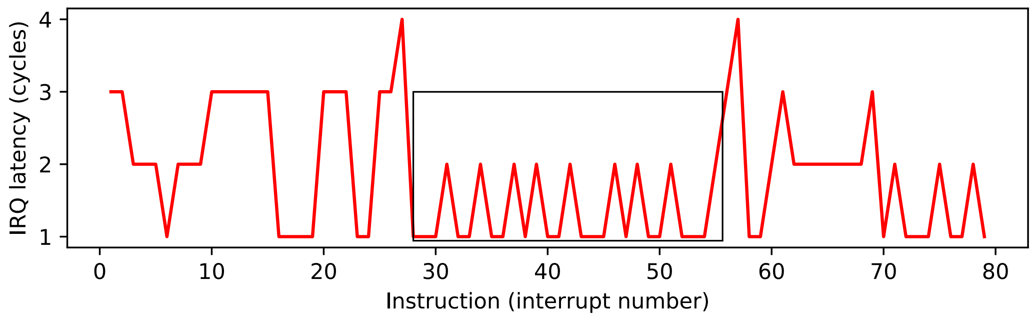

Figure 4.4: A trace extracted with Sancus-Step from the example enclave (see Listing 4.1).

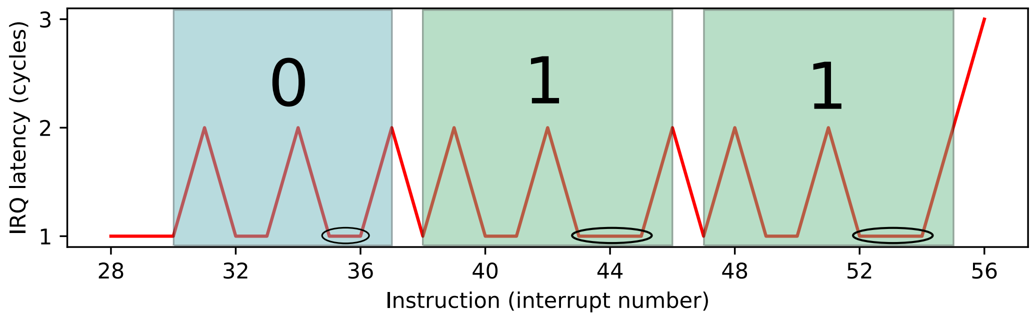

Figure 4.5: An annotated zoom of the black box on Fig. 4.4. The part of the trace shown corresponds to the call of the multiplication stub (instructions 28-56). The shaded areas correspond with the while loop. The marked parts show the difference for either a zero or one bit of *b*.

the attacker analyses the collected traces to leak data.

Leakage of Bits in Multiplier (*b*)

He can exactly determine which bits of *b* are set, and thus extract the value of *b*, the multiplier. An attacker can differentiate between the if-branch being executed and being skipped, by finding a delay such that the timer interrupts instructions with different execution times. From the annoted assembly code, shown in Listing 4.4, it is clear that interrupting the SM three cycles after the conditional jump on line 8 is executed, ensures that either a one cycle instruction or a two cycle instruction is interrupted respectively when the jump is taken and when it is not.

However, for convenience this master's thesis uses Sancus-Step to develop its attacks. By single stepping the protected code, an attacker can generate traces such as the one shown in Fig. 4.4. Such a trace allows an attacker to follow control flow extremely closely and thus determine exactly when the if-branch was executed and when it was skipped. In an iteration where the if-branch is executed, there is one

```
__sm_mulhi3:
    mov     r15, r13
    clr     r15
1:  tst     r14
    jz      3f
    clrc
    rrc     r13
    jnc     2f         // jump of interest
    add     r14, r15   // 1 cycle
2:  rla     r14        // 1 cycle
    tst     r13        // 1 cycle
    jnz     1b         // 2 cycles
3:  ret
```

Listing 4.4: The multiplication stub annotated with instruction execution times.

extra add instruction, which can be seen by the length of the marked parts in Fig. 4.5. This extra add instruction adds another timing of one cycle in the trace and thus creates a difference that can be automatically detected. Note that the bits are leaked from least significant to most significant, thus the multiplier (b) leaked from the trace in Fig. 4.5 is $(110)_2$ or 6.

Determining the stop condition

As mentioned in the start-to-end side-channel analysis, the amount of iterations in the multiplication algorithm depends on both a and b. The attacker can leak whether it was a or b that caused the code to exit the loop, as they happen at different times, hence different traces are generated when respectively a or b first becomes zero. This can be seen by comparing the following two traces: Fig. 4.6 and Fig. 4.7. The ending of the trace shows how in Fig. 4.6 there is first a test on b followed by a conditional jump. The program jumps to a test on a and then jumps to the end of the stub. This clearly differs with the ending of Fig. 4.7 where the conditional jump following the test on b is not taken and thus immediately returns.

Determining the stop condition of the while loop is important for two reasons. First, it shows whether a or b became zero after n iterations, thus leaking that no bit of a or b is set in respectively the $16 - n$ least significant bits or the $16 - n$ most significant bits. Second, if b becomes zero first, the attacker knows she recovered all bits of b. If a becomes zero first, she can only recover b partially, as she knows a is zero and not b. Hence, any of the remaining bits of b can still be one or zero.

Post-processing

To further reduce an attacker's effort, we provide a proof of concept program that takes a trace extracted with Sancus-Step and searches the trace for multiplication traces, using a regular expression. If it finds a multiplication trace, it returns as much information as possible, e.g. the value of b and whether a is zero or not. For the

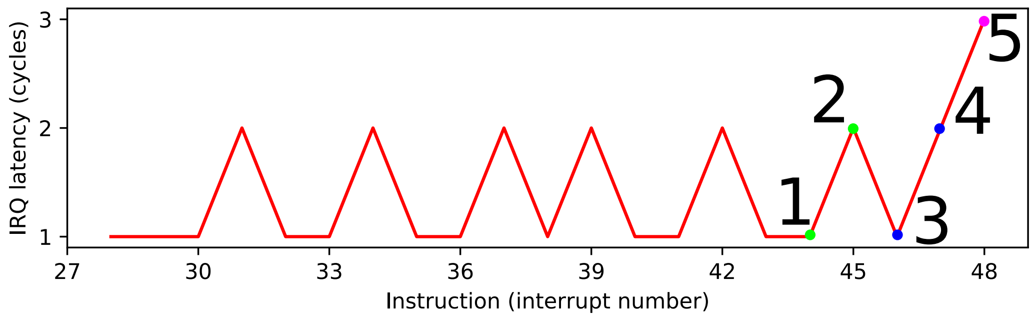

Figure 4.6: A part of a trace extracted with Sancus-Step from the execution of 0x4000 (a) * 0x4 (b). a becomes zero first. (1) The test of b equal to zero, followed by (2) a conditional jump, followed by (4) a test of a equal to zero, followed by (5) a return.

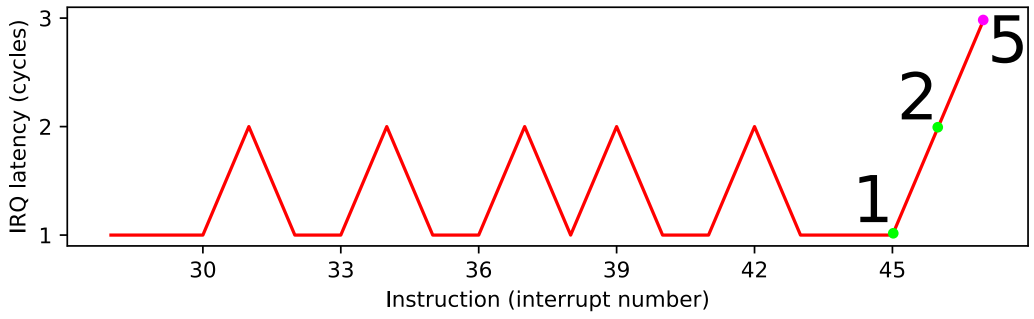

Figure 4.7: A part of a trace extracted with Sancus-Step from the execution of 0x4000 (a) * 0x2 (b). b becomes zero first. (1) The test of b equal to zero, followed by (2) a conditional jump, followed by (5) a return since b is zero.

example program, shown in Listing 4.1, we recover the full value of b and determine that one of the 14 least significant bits of a is set.

This program is just a proof of concept written and thus might return false positives or might not find all multiplication in bigger traces. However, it shows the feasibility for an attacker to extract information about the operands automatically. The source code of this program can be found in the Nemesis github repository[1] and in Appendix A.

[1]https://github.com/jovanbulck/nemesis

4.4 Hardening the Multiplication Stub

```
__sm_mulhi3:
    push    r12
    mov     #16, r12
    mov     r15, r13
    clr     r15
1:

    clrc
    rrc     r13
    jnc     2f
    add     r14, r15
    jmp     4f
2:  nop
    jmp     4f
4:  rla     r14
    sub     #1, r12
    jnz     1b
3:  pop     r12
    ret

```

Listing 4.5: Hardened version of the multiplication stub

```
__sm_mulhi3:


    mov     r15, r13
    clr     r15
1:  tst     r14
    jz      3f
    clrc
    rrc     r13
    jnc     2f
    add     r14, r15

2:

    rla     r14
    tst     r13
    jnz     1b
3:
    ret

```

Listing 4.6: Original multiplication stub

Listing 4.5 shows a hardened version of the multiplication stub. The stub is patched in such a way that for every possible input, single stepping generates the same traces. This is accomplished through inserting NOP instructions of different execution timings, which we will refer to as *compensation code*. As such they do not affect the overall behaviour of the stub. However, they do affect the microarchitectural side-channel leakage, as an attacker cannot differentiate between a NOP instruction of e.g. two cycles and another instruction of two cycles. Hence, it is impossible for the attacker to determine whether a conditional jump was taken or not.

The hardened stub is patched as follows. First, the unbalanced if which test if a bit of b is set is balanced with an accompying else branch. Second, there is no early out when either a or b is zero, thus ensuring a constant amount of sixteen iterations. These two countermeasurements remove all secret-dependent branches, thus removing the possibility for the attacker to use nemesis or start-to-end timing side-channel to infer data. The hardened stub is pending to be merged upstream in the Sancus-compiler github repository [2]

We empirically demonstrate using Sancus-Step that these two countermeasurements ensure that all extracted traces are always the same. Typically, a trace extracted from a victim enclave would look like the trace in Fig. 4.8. The repetitive part in

[2]https://github.com/sancus-pma/sancus-compiler

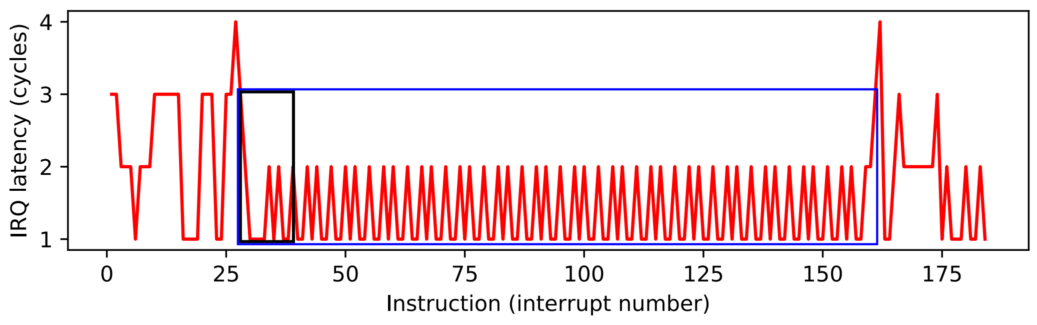

Figure 4.8: A trace extracted with Sancus-Step for the example enclave (see Listing 4.1). Note the repitition in the middle (approximately instructions 28-160).

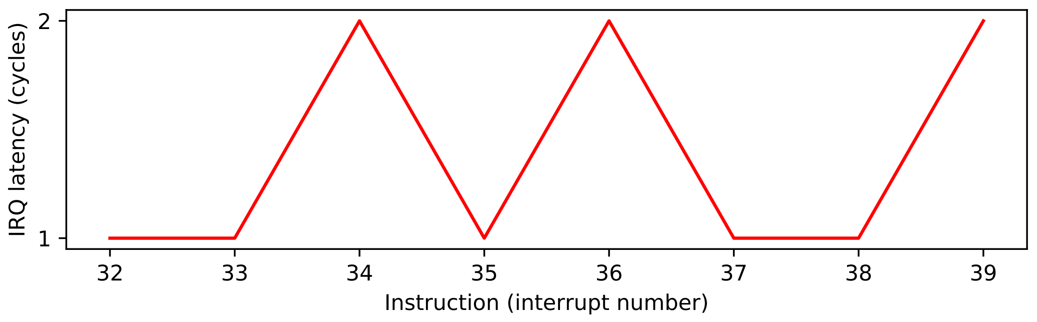

Figure 4.9: One iteration in the hardened multiplication stub always produces this trace. This corresponds to the small black box in Fig. 4.8.

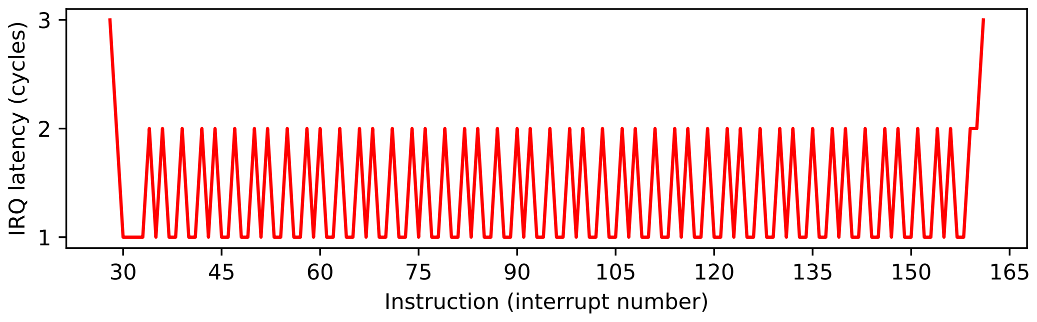

Figure 4.10: One iteration in the hardened multiplication stub always produces this trace. This corresponds to the big blue box in Fig. 4.8.

the middle is the trace extracted from the execution of the multiplication stub. Fig. 4.9 shows the constant trace from one iteration of the multiplication stub. As such, a trace extracted from the multiplication stub will be Fig. 4.9 repeated sixteen times plus some additional instructions at the beginning and end, as can be seen in Fig. 4.10.

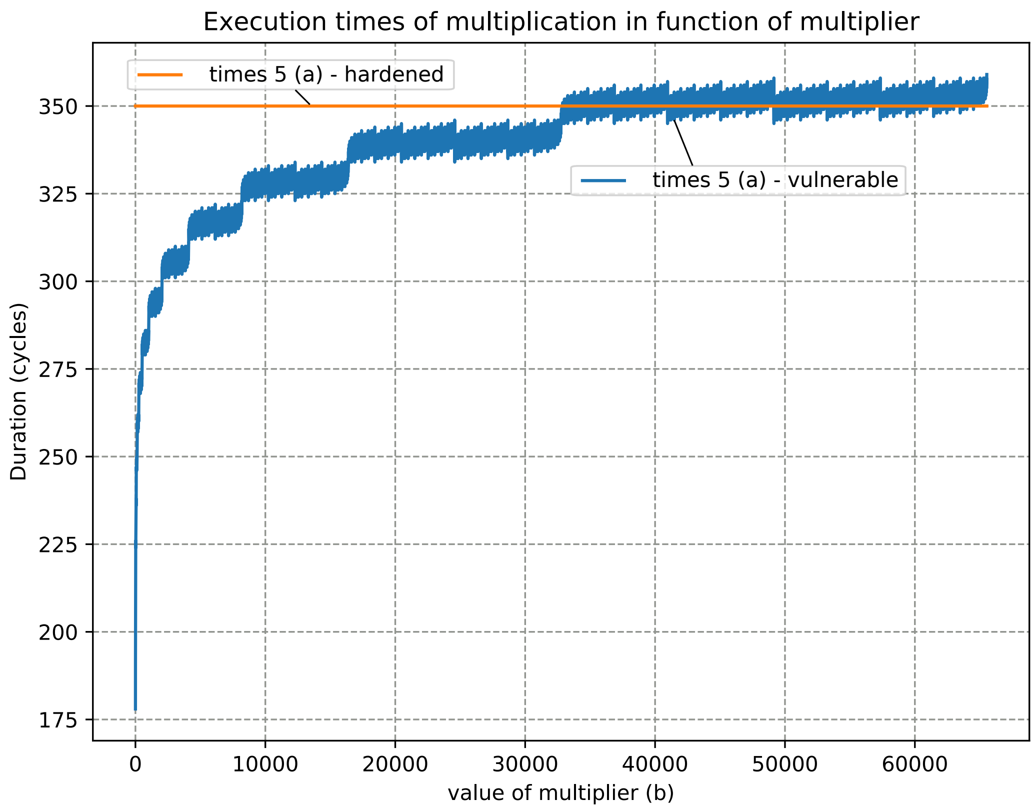

Figure 4.11: A comparison of the overall execution times of multiplication for 5 (*a*) * b for all 2^{16} b's. The vulnerable stub has a variable execution time, while for the hardened stub, the overall execution time is constant.

4.4.1 Performance Comparison

The hardened stub executes extra code to close the side-channels. However, it is evident that this comes at a performance cost, as unnecessary computations are done. First, if a bit of *b* is not set, an else-branch containing no-ops is executed. Second, when either *a* or *b* becomes zero, no useful work will be done anymore, as all intermediate results will be trivially zero. As such, the performance cost is higher if there are few useful iterations.

Fig. 4.11 compares the overall execution time of the vulnerable stub versus the hardened stub. For this case, 5**b*, if *b* is bigger, the performance cost becomes smaller and smaller. For large enough values of *b*, the hardened multiplication stub has a performance gain.

The hardened multiplication stub has an extra constant overhead of seven cycles compared to the original, vulnerable stub. These seven extra cycles are caused by the `push r12`,`mov #16, r12` and `pop r12` instructions at the beginning and the end of the stub.

Per iteration, if the bit of b is set, an extra `jmp` instruction of two cycles is executed. If the bit is not set, an extra `nop` of one cycle and `jmp` of two cycles instruction have to be executed. Thus, there occurs an overhead of two or three cycles per iteration, depending on the current bit of b.

Furthermore, the condition of the loop has been simplified. Instead of two conditions, a not equal to zero and b not equal to zero, there is only one condition now. Thus, a test with a corresponding conditional jump have been removed, resulting in a performance gain of three cycles.

This causes a performance gain if enough bits in b are set and if all sixteen iterations have to be executed, as per iteration the hardened stub can be one cycle faster. This can be seen on Fig. 4.11

4.4.2 Informal Security Argument

As mentioned, this hardened stub closes both start-to-end and Nemesis side-channels. First, the overall execution time for any input is now constant. This is achieved by performing always sixteen iterations, and implementing a balanced else-branch for the if-branch. Trivially, the start-to-end timing side-channel is thus closed.

Second, Nemesis is closed by making sure that in each iteration the same trace is produced, regardless of the current values of a and b. Indeed, the trace of an iteration does not depend on a as the hardened stub never branches on a. However, it branches on b. In the if-branch the hardened stub executes an `add` instruction of one cycle followed by a `jmp` instruction of two cycles, while in the else-branch the hardened stub executes a `nop` instruction of one cycle followed by a `jmp` instruction of two cycles. The last `jmp` instruction of each branch jumps to the same spot in the code, thus ensuring the same traces afterwards. Since in each iteration the same trace is extracted when the attacker single steps the stub, he can extract the same trace repeated sixteen times in a full run. Hence, there is no timer delay, such that a timer will interrupt instructions with different execution times depending on whether or not a conditional jump was taken or not. Thus the Nemesis side-channel is closed. Additionally, the compensation code ensures that the same amount of instructions are executed, regardless of the inputs. Otherwise, an attacker could simply count the amount of times she could interrupt the multiplication to infer data [35].

Recall that the attacker model assumes applications running on the openMSP430 architecture and thus instruction latencies are the only microarchitectural side-channel being analysed here. The proposed hardened stub might not suffice for patching code against Nemesis on Intel SGX enabled processors, as balanced if-else branches might be located on different code pages, and might thus be differentiated by monitoring page table entries [5].

```
__sm_udivhi3:
    mov.b    #16,     r12
    mov      r14,     r13
    clr      r14
1:  rla      r15
    rlc      r14
    cmp      r13,     r14
    jnc      2f
    sub      r13,     r14
    bis      #1,      r15
2:  dec      r12
    jnz      1b
    ret
```

Listing 4.7: Assembly code of the unsigned division stub

```
unsigned int udiv(unsigned int a, unsigned int b)
{
    unsigned int NB_BITS = sizeof(int)*8;
    unsigned int remainder = 0;
    unsigned int quotient = 0;
    for (int i = 0; i < NB_BITS; i++)
    {
        quotient = quotient << 1;
        remainder = (remainder << 1) + a & 0x1 << NB_BITS-1;
        if (remainder >= b)
        {
            remainder -= b;
            quotient |= 1;
        }
        a = a << 1;
    }
    return quotient;
}
```

Listing 4.8: pseudocode of the unsigned division stub

4.5 Future work

This section summarises our initial exploration of the other arithmetic stubs and discusses other potentially interesting stubs for further research.

4.5.1 Other Arithmetic Stubs

Listing 4.7 shows the assembly code of the division stub inserted by the Sancus toolchain. Listing 4.8 shows the pseudocode of the stub. The unsigned division stub repeatedly executes code in a loop until it executed it 16 times, as can be seen in Listing 4.8. As this is an initial exploration, we leave an extensive analysis, as performed for the multiplication stub, as future work.

However, from the pseudocode it is clear that there is an unbalanced if-branch (line 10-14). In this unbalanced if-branch, the bits of the quotient are set. Thus, through the Nemesis side-channel, an attacker can leak when the bits in the quotient are set, thus revealing the full 16 bits of the quotient. Appendix B contains the source code of our proof of concept program, similar to the one we developed for the multiplication stub.

The other arithmetic operations, division and (unsigned) modulo, all call the division stub. Thus, any data leakage contained within the unsigned division stub is also present in the division and (unsigned) modulo stub.

We provide the source code of the other stubs in Appendix E.

4.5.2 Entry and Exit Stub

Upon leaving an enclave, the enclave clears any unused registers to ensure that no private data is leaked through the registers. However, this is done with unbalanced branches. An attacker can thus leak with Nemesis how many used registers there are. This can be interesting, if e.g. an enclave foo calls bar with either one or two arguments depending if some secret is zero or not. By determining the amount of arguments foo passes to bar, one can thus deduce whether the secret value is zero.

Further work should investigate the extent to which side-channel information is leaked in the entry and exit stub. Additionally, other attack scenarios can be explored.

4.6 Conclusion

The analysis clearly shows that in both a start-to-end and nemesis side-channel, there is information leakage in the multiplication and unsigned division stub. In both cases, secret data can be leaked in a single run of the victim code. Furthermore, this code is present in any application that is built using the Sancus toolchain and uses multiplication or division anywhere inside an sm.

Chapter 5

Conclusion

Interconnected embedded devices are rising in popularity. Because of economic reasons, they lack common hardware security mechanisms. Yet, they are increasingly being deployed to handle safety critical operations and privacy sensitive data.

Lightweight PMAs, a novel research area, satisfy the required security guarantees, while ensuring minimal cost and power consumption. PMAs allow for efficient software isolation and provides strong security guarantees towards its enclaves.

However, our understanding of (microarchitectural) side-channel leakage in embedded enclaved execution is little. Arguably, side-channels are the only attack vector left, and thus it is of importance that we further investigate the occurences and exploitability of micro-architectural side-channels in microprocessors.

This master's thesis has shown that side-channel leakage even occurs in compiler generated code, thus introducing vulnerabilities into applications. We designed and implemented Sancus-Step, an attacker framework to automate side-channel attacks against Sancus enclaves, while minimising the required expertise to exploit side-channel vulnerabilities.

The remainder of this chapter is as follows: Section 5.1 acknowledges the limitations of the analysis in this thesis. Section 5.2 discusses open research questions and Section 5.3 summarises the contributions of this thesis.

5.1 Limitations and Challenges

First and foremost, this master's thesis builds on Sancus. As such, Sancus-Step is made specifically for Sancus. The core idea of Sancus-Step should be portable to other embedded PMAs [16], as Sancus-Step is at its core an easy to use library to automate single stepping of Sancus enclaves. As such, when porting Sancus-Step to another PMA, there are two requirements which need to be kept in mind: (*i*) interruptible enclaves and (*ii*) a timer or another way to precisely interrupt enclaved execution.

The side-channel analysis performed in this thesis is limited in scope, as the multiplication stub is only one of the many stubs the compiler generates and inserts into applications. Our choice for the multiplication stub is mostly a practical choice, as it was the smallest and easiest to understand. However, the techniques shown to analyse side-channel leakage are applicable to any other code.

The patch for the multiplication stub is specific to the MSP430 CPU architecture. Furthermore, it closes Nemesis. However, on more complex CPU architectures, other microarchitectural side-channels might exist, which are not closed by our hardened stub.

5.2 Future Work

We have done an extensive analysis of side-channel leakage in the multiplication stub. However, we acknowledge that there are other stubs as well, which require analysis as well. As such, which stubs leak what information is an interesting research question.

Atomicity related vulnerabilities, while not new, were never fully explored. In this thesis we provide a first step towards this direction. We show how Sancus-Step can be used to easily exploit a time-of-check to time-of-use vulnerability in the verification code of Sancus. However, there might exist many more time-of-check to time-of-use vulnerabilities in Sancus. Hence, an extensive analysis of Sancus stubs is required to determine potential vulnerabilities. Afterwards, they should be patched and merged upstream. We expect Sancus-Step to be useful for automating large-scale analysis of side-channel leakage in embedded enclaved execution.

5.3 Contributions

This thesis explored side-channel leakage in embedded enclaved execution. More specifically, this thesis analysed side-channel leakage of compiler generated code that is inserted by the Sancus toolchain. The contributions of this thesis are three-fold. First, we have proposed Sancus-Step in Chapter 3, reducing the expertise required to succesfully exploit side-channel leakage in compiler generated code. Additionally, because of the flexibility of Sancus-Step, it can also be used to exploit time-of-check to time-of-use vulnerabilities. Sancus-Step is easy to use, because it automatically determines its parameters. Furthermore, Sancus-Step can run any attacker's code to easily develop new exploits. Second, in Chapter 4 we have shown side-channel leakage in compiler generated code. We were able to extract all 16 bits of the secret multiplier value in our benchmark enclave. Additionally, we provide a patched stub and empirically show that is closes the Nemesis side-channel.

Appendices

Appendix A

Source Code of Post Processing Program for Multiplication

```python
#!/usr/bin/python3

def main():
    import sys
    import re

    NB_BITS_INT = 16

    if len(sys.argv) < 2:
        print("Usage: {} <file> [-p]".format(sys.argv[0]))
        exit()
    with open(sys.argv[1], "r") as inFile:
        lines = inFile.readlines()
        latencies = []
        for line in lines:
            if line.startswith("latency:"):
                _, latency = line.split(" ")
                latencies.append(latency.strip())
    latenciesString = "".join(latencies)
    hardened = False
    if len(sys.argv) > 2:
        if sys.argv[2] == "-p":
            hardened = True
    multiplicationRegex = "11123|11(12112(1?)112)+(12)?3"
    if hardened:
        multiplicationRegex = "3211"+"11212112"*16+"23"
    multiplicationPattern = re.compile(multiplicationRegex)
    multiplicationHit = multiplicationPattern.search(latenciesString)
    if multiplicationHit:
        multiplicationTrace = multiplicationHit.group()
        if hardened:
            print("Full trace", multiplicationTrace)
            print("Iteration trace", "11212112")
        else:
            print("Found multiplication")
            print("Full trace", multiplicationTrace)
            if multiplicationTrace == "11123":
                print("a = 0")
                print("b = unknown")
            else:
                multiplicationTrace = multiplicationTrace[2:] # consume the two known instructions at the beginning
                bitsOfB = ""
                bitEq1Trace = "121121112"
                bitEq0Trace = "12112112"
                aIsZeroTrace = "123"
                while multiplicationTrace.startswith(bitEq1Trace) or multiplicationTrace.startswith(bitEq0Trace):
                    if multiplicationTrace.startswith(bitEq1Trace):
                        bitsOfB += "1"
                        multiplicationTrace = multiplicationTrace[len(bitEq1Trace):]
                    elif multiplicationTrace.startswith(bitEq0Trace):
                        bitsOfB += "0"
                        multiplicationTrace[len(bitEq0Trace):]
                        multiplicationTrace = multiplicationTrace[len(bitEq0Trace):]
                if multiplicationTrace.startswith(aIsZeroTrace):
                    print("Result overflowed...incorrect result is returned")
                    print(("partial recovery of b\nb = " + "x"*(16-len(bitsOfB))+"".join(bitsOfB[::-1])))
                else:
                    print("At least one of the {} least significant bits of a is 1".format(NB_BITS_INT - len(bitsOfB) + 1))
                    print("b = {}".format(int(bitsOfB[::-1], 2)))
    else:
        print("Found no multiplication")


if __name__ == "__main__":
    main()
```

Listing A.1: Source code of the post processing program for multiplication traces

Appendix B

Source Code of Post Processing Program for Unsigned Division

```python
#!/usr/bin/python3

def main():
    import sys
    import re
    if len(sys.argv) != 2:
        print("Usage: {} <file>".format(sys.argv[0]))
        exit()
    with open(sys.argv[1], "r") as inFile:
        lines = inFile.readlines()
        latencies = []
        for line in lines:
            if line.startswith("latency:"):
                _, latency = line.split(" ")
                latencies.append(latency.strip())
        latenciesString = "".join(latencies)
        udivRegex = "211(1112(11)?12){16}"
        udivPattern = re.compile(udivRegex)
        udivHit = udivPattern.search(latenciesString)
        if udivHit:
            print("Found unsigned division")
            udivTrace = udivHit.group()
            udivTrace = udivTrace[3:] # consume the three known instructions at the beginning
            bitsOfQuotient = ""
            bitEq1Trace = "11121112"
            bitEq0Trace = "111212"
            while udivTrace.startswith(bitEq1Trace) or udivTrace.startswith(bitEq0Trace):
                if udivTrace.startswith(bitEq1Trace):
                    bitsOfQuotient += "1"
                    udivTrace = udivTrace[len(bitEq1Trace):]
                elif udivTrace.startswith(bitEq0Trace):
                    bitsOfQuotient += "0"
                    udivTrace[len(bitEq0Trace):]
                    udivTrace = udivTrace[len(bitEq0Trace):]
            print("quotient = {}".format(int(bitsOfQuotient, 2)))
        else:
            print("Found no unsigned division")


if __name__ == "__main__":
    main()
```

Listing B.1: Source code of the post processing program for unsigned division traces

Appendix C

Source Code of Modified Entry Stub

```
    .section ".sm.text"
    .align 2
    .global __sm_entry
    .type __sm_entry,@function

    ; r6: ID of entry point to be called, 0xffff if returning
    ; r7: return address
__sm_entry:
    ; === need a secure stack to handle IRQs ===
    dint

    ; If we are here because of an IRQ, we will need the current SP later
    . We do
    ; do not store it in its final destination yet (__sm_irq_sp) because
   we may
    ; not actually be called by an IRQ in which case we might overwrite a
     stored
    ; stack pointer.
    mov r1, &__sm_tmp
    # Switch stack.
    ; __sm_sp is our stackpointer
    mov #__sm_sp, &__sm_sp_addr
    mov &__sm_sp, r1
    ; initialize sp on first entry
    cmp #0x0, r1
    jne 1f
    mov #__sm_stack_init, r1

1:
    ; check if this is a return from a interrupt
    bit #0x1, &__sm_sp

    ; === safe to handle IRQs now ===
    eint

    jz 1f
    ; restore execution state if the sm was resumed
    br #__reti_entry ; defined in exit.s

1:
    ; check of this is an IRQ
    push r15
    ; sancus_get_caller_id()
    .word 0x1387
    cmp #0xfff0, r15
    jlo 1f
    ; SEMI-HACK: If we are not protected, and no other SM has ever been
    ; executed, the caller ID will be that of the last IRQ because entering this
    ; SM was no protection domain switch. This basically means that once an IRQ
    ; has occurred, we cannot call normal entry points anymore. Since it is nice
    ; to be able to use unprotected SMs during testing, and it is quiet common
    ; to have interrupts disabled then, the caller ID will always be that of the
    ; reset IRQ (0xffff). Since there is no valid use case of actually handling
    ; a reset inside an SM (since the reset will disable all SMs), we simply
    ; ignore it here so that normal entry points can still be used.
    cmp #0xffff, r15
    jeq 1f
    ; If we just do je __sm_isr we get a PCREL relocation which our runtime
    ; linker doesn't understand yet.
    br #__sm_isr
1:
    pop r15

    ; check if this is a return
    cmp #0xffff, r6
    jne 1f
    br #__ret_entry ; defined in exit.s

1:
    ; check if the given index (r6) is within bounds
    cmp #__sm_nentries, r6
    jhs .Lerror

    ; store callee-save registers
    push r4
    push r5
    push r8
    push r9
    push r10
    push r11

    ; calculate offset of the function to be called (r6 x 6)
    rla r6
    mov r6, r11
    rla r6
    add r11, r6

    ; function address
    mov __sm_table(r6), r11

    ; call the sm
    call r11

    ; restore callee-save registers
    pop r11
    pop r10
    pop r9
    pop r8
    pop r5
    pop r4

    ; clear the arithmetic status bits (0, 1, 2 and 8) of the status
    register
    and #0x7ef8, r2

    ; clear the return registers which are not used
    mov 4+__sm_table(r6), r6
    rra r6
    jc 1f
    clr r12
    clr r13
    rra r6
    jc 1f
    clr r14
    rra r6
    jc 1f
    clr r15

1:
    ; atomic leaving
    dint
    mov r1, &__sm_sp
    mov #0xffff, r6
    br r7

.Lerror:
br #exit
```

Listing C.1: Modified entry stub

Appendix D

Source Code of Modified Exit Stub

```
    .section ".sm.text"
    .align 2
    .global __sm_exit
    .type __sm_exit,@function

    ; r6: ID of entry point or function address to be called
    ; r7: register usage
    ; r8: entry point to call
__sm_exit:
    ; store and clear callee-save registers
    push r4
    clr  r4
    push r5
    clr  r5
    push r9
    clr  r9
    push r10
    clr  r10
    push r11
    clr  r11

    ; clear unused argument registers
    rra r7
    jc 1f
    clr r12
    rra r7
    jc 1f
    clr r13
    rra r7
    jc 1f
    clr r14
    rra r7
    jc 1f
    clr r15
1:
    ; atomic leaving of sm
    ;dint
    ; store sp
    mov r1, &__sm_sp

    ; pass the entry point as return address
    mov #__sm_entry, r7
    br r8

    .align 2
    .global __reti_entry
    .type __reti_entry,@function
__reti_entry:
    pop r4
    pop r5
    pop r6
    pop r7
    pop r8
    pop r9
    pop r10
    pop r11
    pop r12
    pop r13
    pop r14
    pop r15

    ; clear bit 0
    ; maybe we also should store stackpointer
    ; or we assume: reti never jumps out of sm
    bic #0x1, &__sm_sp

    reti

    .align 2
    .global __ret_entry
    .type __ret_entry,@function
__ret_entry:
    ; restore callee-save registers
    pop r11
    pop r10
    pop r9
    pop r5
    pop r4

    pop r8
    pop r7
    pop r6

    ret
```

Listing D.1: Modified exit stub

Appendix E

Source Code of the Other Arithmetic Stubs

```
__sm_divhi3:
    clr     r13
    tst     r15
    jge     1f
    mov     #3, r13
    inv     r15
    inc     r15
1:  tst     r14
    jge     2f
    xor.b   #1, r13
    inv     r14
    inc     r14
2:  push    r13
    call    #__sm_udivhi3
    pop     r13
    bit.b   #2, r13
    jz      3f
    inv     r14
    inc     r14
3:  bit.b   #1, r13
    jz      4f
    inv     r15
    inc     r15
4:  ret
```

Listing E.1: Assembly code of the division stub

```
__sm_umodhi3:
     call     #__sm_udivhi3
     mov      r14,    r15
     ret
```

Listing E.2: Assembly code of the unsigned modulo stub

```
__sm_modhi3:
     call     #__sm_divhi3
     mov      r14,    r15
     ret
```

Listing E.3: Assembly code of the signed modulo stub

Bibliography

[1] Clang/llvm toolchain documentation. https://clang.llvm.org/docs/Toolchain.html. Accessed: 2019-04-20.

[2] Spectre attack website. https://www.spectreattack.com/. Accessed: 2019-05-17.

[3] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, jun 2012.

[4] N. Avonds. Implementation of a state-of-the-art security architecture in the linux kernel. Master's thesis, KU Leuven, 2013. https://www.scriptiebank.be/sites/default/files/webform/scriptie/thesis_5.pdf.

[5] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, 2017. USENIX Association.

[6] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. *CoRR*, abs/1811.05441, 2018.

[7] I. Corporation. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Accessed: 2019-06-01.

[8] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.

[9] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. pages 147–152, 06 2014.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 2, 1983.

[11] M. O. Farooq and T. Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.

[12] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[13] T. Instruments. Msp430x1xx family user's guide. https://www.ti.com/lit/ug/slau049f/slau049f.pdf. Accessed: 2019-05-17.

[14] Intel. Intel 64 and ia32 architectures software developer's manual volume 3a: System programming guide, part 1. https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf. Accessed: 2019-05-17.

[15] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[16] P. Koeberl, S. Schulz, V. Varadharajan, and A.-R. Sadeghi. Trustlite: A security architecture for tiny embedded devices. 04 2014.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[18] 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*, PP(99), 2017.

[19] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. 06 2013.

[20] Microchip. megaavr data sheet. https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf. Accessed: 2019-05-17.

[21] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom. Fallout: Reading kernel writes from user space. 2019.

[22] J. Noorman. Sancus 2.0. https://distrinet.cs.kuleuven.be/software/sancus. Accessed: 2019.

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

[24] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7:1–7:33, September 2017.

[25] A. One. Smashing the stack for fun and profit. http://phrack.org/issues/49/14.html. Accessed: 2019-03-05.

[26] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[27] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. *Protected Software Module Architectures*, pages 241–251. Springer Fachmedien Wiesbaden, Wiesbaden, 2013.

[28] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[29] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. volume 50, pages 344–361, 09 2010.

[30] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Pearson, 2015.

[31] 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.

[32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [38].

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

[34] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. pages 1–6, 10 2017.

[35] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 178–195, New York, NY, USA, 2018. ACM.

[36] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[37] F. Vogels, B. Jacobs, and F. Piessens. Featherweight verifast. *Logical Methods in Computer Science*, 11(3), 2015.

[38] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [32].

[39] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[40] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

Fiche masterproef

Student: Sven Cuyt

Titel: A Security Analysis of Interrupts in Embedded Enclaved Execution

Nederlandse titel: Een Veiligheidsanalyse van Onderbrekingen in Ingebedde Enclaved Uitvoering

UDC: 681.3

Korte inhoud:

Small embedded devices are permeating our daily lives. However, to minimise production costs, these small embedded devices lack common hardware features to implement conventional security mechanisms, such as virtual memory isolation and processor privilige levels.

To respond to these concerns, recent research effort has shifted towards Protected Module Architectures (PMAs). PMAs allow security-critical code to be executed in an enclave isolated from the rest of the system. However, this software isolation enforced by PMAs only holds at the architectural level. Microarchitectural optimizations are still shared among enclaves and untrusted code.

Little research effort has gone towards understanding microarchitectural side-channels against embedded enclaved execution. In this perspective, this master's thesis increases our understanding of microarchitectural side-channel leakage and automated exploitation in embedded enclaved execution. The main contributions of this master's thesis are twofold. First, Sancus-Step, an attacker framework for automatic side-channel exploitation, is proposed. Sancus-Step aims to be easy to use, while still being flexible enough such that developing side-channel attacks requires considerably less expertise. Second, side-channel leakage of compiler generated code is analysed, using Sancus-Step. We did an extensive analysis of the multiplication stub and an initial exploration of other stubs. We patch the multiplication stub and evaluate the performance and security of the patched stub.

This master's thesis shows considerable side-channel leakage in compiler generated code, which can easily be automatically exploited using our newly proposed attacker framework Sancus-Step. Additionally, this master's thesis shows that this side-channel leakage can be patched in software.

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. R. Strackx

Assessoren: Ir. J. Van Bulck
Dr. A. Timany

Begeleiders: Ir. J. Van Bulck
Dr. R. Strackx