KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Analyzing side-channel leakage in secure DMA solutions

Márton Bognár

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Secure Software

Thesis supervisor:
Prof. dr. ir. Frank Piessens

Assessors:
Dr. Pieter Philippaerts
Ir. Jo Van Bulck

Mentor:
Ir. Jo Van Bulck

Academic year 2019–2020

Preface

First of all, thank you for taking the time to read this thesis.

I would like to thank my advisor Jo Van Bulck, with whom we have spent countless hours discussing the details of this thesis. Without him, this work could not have been completed.

I appreciate prof. Piessens' efforts to help me find an interesting thesis topic, thank you for being so approachable. Thank you to everyone at DistriNet for working on such great projects that allow theses like this one to exist.

Additionally, I would like to thank the people at the (late) Scademy who initially introduced me to the world of security and got me interested in it.

I'm also grateful for my parents for not complaining too much about me moving to a different country just to study security.

Shoutout to the cubes, David, and the people from CPP FTW.

Márton Bognár

Contents

Preface i

Abstract iv

List of Figures v

List of Tables vi

Listings vii

1 Introduction 1
- 1.1 Trusted Execution Environments . . . 2
- 1.2 Contributions . . . 4
- 1.3 Outline . . . 5

2 Background 7
- 2.1 Trusted Execution Environments . . . 7
  - 2.1.1 Commercial solutions . . . 8
  - 2.1.2 Sancus . . . 9
- 2.2 Full abstraction . . . 11
- 2.3 Secret-dependent control and data flow . . . 11
- 2.4 Side-channel attacks . . . 12
- 2.5 Nemesis . . . 13
- 2.6 Direct Memory Access . . . 16
  - 2.6.1 Implementation details . . . 17
  - 2.6.2 Security of DMA . . . 18
- 2.7 Conclusions . . . 19

3 DMA-based side-channel attacks 21
- 3.1 Basic attack . . . 22
- 3.2 An end-to-end example . . . 23
- 3.3 Vulnerable enclaves . . . 26
- 3.4 Attacking the Sancus platform . . . 27
- 3.5 Attacker peripheral . . . 29
- 3.6 Instruction analysis . . . 30
  - 3.6.1 Instruction fetching . . . 30
  - 3.6.2 Different instructions . . . 31
  - 3.6.3 Addressing modes . . . 32
  - 3.6.4 The value of the operands . . . 33
  - 3.6.5 The influence of the previous instruction . . . 35
- 3.7 Comparison with interrupt-based attacks . . . 36

3.7.1 Stronger DMA leakage . . . 37
3.7.2 Stronger interrupt leakage . . . 37
3.7.3 A combined attacker framework . . . 38
3.8 Attack limitations . . . 42
3.8.1 Limitations of the experimental setup . . . 42
3.8.2 Fundamental limitations . . . 43
3.9 Conclusions . . . 44

4 Defenses **45**
4.1 Code balancing . . . 46
4.2 Hardened compiler . . . 48
4.3 Giving the DMA priority . . . 48
4.4 Vulnerability in Sancus with DMA priority . . . 49
4.5 Conclusions . . . 51

5 Conclusions and future work **53**
5.1 Future work . . . 53
5.2 Contributions . . . 54

A Memory traces of different instruction formats **59**
A.1 Jump instructions . . . 59
A.2 Single operand instructions . . . 60
A.3 Double operand instructions . . . 63

Bibliography **79**

Abstract

With the advent of the Internet of Things, industrial control systems, and increasing automation in all areas of life, networked embedded devices have become ubiquitous. These systems are often running safety-critical software or handle sensitive data, but attacks against them are discovered regularly.

Trusted Execution Environments (TEEs) have been introduced to provide strong security guarantees for executing programs, both on high-end systems and on small embedded platforms. These TEEs execute programs in so-called enclaves, which provide isolation from unprotected code and other enclaves.

One important class of attacks against TEEs are the so-called side-channel attacks. While the enclaves isolate their contents on the architectural level, side-channel attacks exploit microarchitectural implementation details of the target platform to extract information from the enclaves. These attacks often target optimization features or shared resources on the system, such as caches or the shared memory. Side-channel attacks are well-researched in the context of high-end processors, but attacks against small embedded systems have not received much attention.

In this thesis, a new side-channel attack is presented that is effective against enclaves running on a TEE. The attack targets the shared memory bus and uses the timing of Direct Memory Access requests issued to unprotected addresses in the memory to reconstruct the memory accesses of the processor running the enclave.

The attack is demonstrated on the Sancus platform, which is based on the 16-bit TI MSP430 microcontroller. An experimental peripheral is presented that is capable of capturing the memory access patterns of the processor. Examples in the thesis show how this leakage can lead to the extraction of secrets from an enclave.

An analysis of the leakage is also provided; the factors that influence the memory access pattern of an instruction are described, and the leakage of the attack is compared to Nemesis, the current state-of-the-art attack on the Sancus platform that utilizes interrupt latency measurements to leak secrets.

A number of defense methods are also proposed both on the software and the hardware level. One of the possible defense methods on Sancus is experimentally shown to contain the same leakage as the original attack.

List of Figures

2.1 Simplified figure of the openMSP430 architecture. 10
2.2 Timing diagram of a correct guess 15
2.3 Timing diagram of an incorrect guess 15
2.4 Instruction timing traces with a correct guess 15
2.5 Instruction timing traces with an incorrect guess 16

3.1 Memory contention on a system with DMA 22
3.2 Memory traces for `mov` and `add` 24
3.3 Timing diagram for a correct and an incorrect password guess 25
3.4 Timing diagrams for an incorrect and a correct guess in the secret-dependent data flow 28
3.5 Memory traces for `add` and `mov` 31
3.6 Memory traces for `jmp` and `jne` 32
3.7 Memory traces for different source operands of the `mov` instruction 33
3.8 Memory traces for `mov` with different absolute addresses as its target operand 34
3.9 Memory traces for `mov` with different registers as its target operand 35
3.10 Memory traces with different addresses as the parameter of SWPB 36
3.11 Timing diagram for the two enclaves 38
3.12 Hypothetical enclave example 41

4.1 Timing diagram for a correct and an incorrect password guess with the DMA priority setup 51

List of Tables

3.1 Addressing modes in the MSP430 architecture 32

Listings

2.1 Program 1 11
2.2 Program 2 11
2.3 A secret-dependent control flow 12
2.4 A secret-dependent data flow 12
2.5 A password comparison program 14
3.1 The running enclave example 24
3.2 A secret-dependent data flow 28
3.3 Enclave 1 38
3.4 Enclave 2 38
3.5 The target enclave 40
3.6 The interrupt handler routine 40
4.1 Naive padding of the example enclave 47

Chapter 1

Introduction

Throughout history, most computing devices have been designed to support multiple programs and potentially multiple users. The users of the system and the programs they run do not necessarily trust each other. A malicious user or program – or even a malfunctioning benign one – should not be able to interfere with other programs on the system, influence their execution, or collect any information that is not explicitly offered by them.

Isolating programs from each other is a well-researched field in software security. Many solutions have been proposed that address this challenge at multiple different levels.

Close to the hardware, operating systems and processors implement protection rings to separate privileged code (typically the operating system kernel and device drivers) from user-level code. Virtual memory allows the separation of the memory space of different processes, also through the cooperation of the operating system and hardware in the CPU or a Memory Management Unit. Processes can be executed inside virtual machines or containers that monitor, translate, and if necessary, modify or block accesses to resources outside the container (e.g. memory and file accesses, network communications).

These techniques can increase the security of shared systems and are often used on mainstream systems. An issue with most of these approaches is that they require a sizeable trusted software and hardware layer.

The subset of software and hardware components that are required to work correctly for a protection mechanism to provide security guarantees is called the *Trusted Computing Base* (TCB). For most of the solutions outlined above, this TCB is large enough (e.g. it contains a hypervisor or the operating system) that its correctness is practically impossible to verify, and it possibly contains bugs or vulnerabilities that in turn compromise the protection mechanism itself.

The more complex the protection technique, the harder it is to verify the correctness of its implementation and the level of security it provides. Protection mechanisms that rely on the correct functioning of the operating system have no guarantees for the security they provide in case the operating system is compromised. Numerous new bugs are discovered regularly in operating systems and the complex software solutions such as containers, and formally verifying the absence of bugs is infeasible due to these systems' size and complexity.

Another issue with these approaches is the amount of resources they require, either in terms of extra hardware components or computational overhead. While the top of the line computers are still increasing in performance more or less in line with Moore's law, the area of applications for computers has grown much larger in the past decade.

The majority of commercial computing devices manufactured today are not personal computers or servers, but small embedded devices. Some examples include personal mobile devices such as smartphones or watches, but also components used in the Internet of Things and industrial control devices. These devices are much more constrained in terms of the resources they can use (importantly they need to keep both physical size and power consumption low), so implementing the above protection mechanisms is inconvenient, if not impossible.

1.1 Trusted Execution Environments

Recent research has aimed at improving the low security of embedded devices [50], and as a result *Trusted Execution Environments* (TEEs) have appeared [29, 41]. Some examples include Intel SGX [11], ARM TrustZone [7], and Sancus [34], the target platform of this thesis. The goal of these systems is to provide sufficient security guarantees (comparable to the previously mentioned mechanisms in high-end systems) while relying only on a minimal Trusted Computing Base and producing little overhead.

The minimal TCB also enables a more thorough security analysis, which can make even formal verification efforts possible [9, 16].

TEEs execute programs in so-called secure *enclaves*. The TCB – which includes the trusted processor – enforces isolation between an enclave and other, untrusted code and data. Enclaves generally consist of a protected code and data section in memory. The TEE enforces that the enclave's data can only be accessed by the code of the same enclave, and the enclave's code can only be entered from predefined entry points to reduce the risk of Return Oriented Programming based attacks [37]. The isolation guarantees require a cooperating and bug-free enclave. If the enclave voluntarily – or involuntarily, for example as a result of a buffer overflow [3] – exposes protected data to the outside world, the isolation guarantees cannot hold.

TEEs often assume a privileged attacker who can control the operating system and connect peripherals to the device. This is possible because the TEEs do not include the operating system in their TCB in order to minimize it, so they do not rely on the operating system for isolation-critical tasks (but often do for resource allocation or handling communications with connected devices or a network).

Side-channel attacks A powerful class of attacks against both traditional computing systems and TEEs is the so-called side-channel attacks. These attacks do not directly exploit a vulnerability in the executing enclave's code to extract information from its output or the values of public memory or registers that it modified, but rather use microarchitectural implementation details of the underlying system, such as measuring the time it takes for an operation to complete to infer some secret information from the isolated process.

Some side-channel attacks require physical access to the target device. Examples include the use of measuring equipment to analyze the power consumption of the

device [22] or placing a probe on the memory bus the analyze the memory accesses of an enclave [27].

Side-channel attacks can also be executed from software, Yuanzhong Xu et al. presented an attack [52] that makes use of the advanced attacker model of TEEs to compromise the operating system and induce page faults in the victim process to leak information about its control flow and data accesses.

Other software side-channel attacks measure the timing of accesses to shared resources that can leak information about the accesses of the victim process. Flush+Reload [53] measures cache access timings, while DRAMA [35] uses the timing of main memory accesses.

Side-channels can also be used to create covert channels between an isolated process and the outside world. The Spectre [23] and Meltdown [28] attacks that exploit speculative and transient execution in modern processors both use Flush+Reload [53] to leak secrets from isolated processes.

Increasing attacker surface In the past decades, the attacker surface has increased greatly. Most modern computing devices are connected to the internet, which allows attackers to target them remotely, over the network. Remote attackers can exploit vulnerabilities in the code handling network traffic, and then gain control of other parts of the system. Even side-channel attacks have been identified that can effectively extract information from a process running on a remote device by measuring the time it takes for certain network requests to complete [26].

Remote versions of the Rowhammer attack [21] have also been presented [43]. These attacks exploit the physical properties of DRAM chips to modify bits in stored value belonging to a victim process by issuing rapid memory requests to nearby memory locations.

The software running on the devices is also often extendable, third parties have the option to run their code on the device together with other tenants. Additionally, computing devices and especially high-end CPUs have been extended with features that increase the performance of the system. Such features include instruction and data caching, Direct Memory Access, and speculative execution. These changes affect conventional computing systems as well as TEEs, as there are numerous examples of TEEs employing these technologies to increase performance. As shown by the examples in the previous section, these features, while increasing the performance of the system, also allowed new classes of attacks to emerge.

Direct Memory Access Direct Memory Access (DMA) enables components in the system to issue read or write requests to the memory while bypassing the CPU, thus increasing its performance by not having to coordinate memory reads and writes originating from peripheral devices.

Enabling memory access to untrusted peripherals gave rise to a new class of attacks, which modified the data of running (in some cases, privileged) processes in the memory [8]. As a response, countermeasures such as IOMMU devices have appeared, that regulate the incoming DMA requests, but recently multiple attacks have surfaced that can bypass the protection offered by these countermeasures [25, 30].

DMA has also been extended to support remote machines, Remote DMA is a protocol that allows connected devices to easily access each other's memory. This

protocol has been used to execute the remote attacks NetCAT [26] and Throwhammer [43] mentioned in the previous section.

Numerous TEEs allow components to issue DMA requests, although with restricted access to avoid trivial attacks against protected memory regions: on Intel SGX [11] and on the current version of Sancus [40], DMA requests cannot access addresses that belong to an enclave's protected memory to preserve the isolation guarantees.

1.2 Contributions

In this thesis, we present a novel DMA-based side-channel attack that is effective even against processes running in a TEE's enclave. Our attack uses timing measurements of DMA requests to unprotected memory regions to infer the memory access patterns of the CPU. These measurements can be used to reconstruct the control and data flow of enclaves and can lead to the leakage of security keys or other sensitive information from the victim enclaves.

The detailed contributions of this thesis are the following:

- We describe our attack in which an unprivileged attacker with control over a connected peripheral with DMA capabilities can infer the memory access patterns of an executing enclave. The attack can be executed by measuring the latency of DMA requests to unprotected memory if the memory accesses of the processor are prioritized over the DMA requests.
- Using practical examples, we show that using the collected memory traces and some knowledge of the source code or the inputs to the enclave, an attacker can reconstruct control and data flows in some contexts, thus breaking the isolation guarantees of the enclave. Our examples also show how this can lead to the extraction of sensitive information from the enclaves.
- We highlight how certain features of Sancus' [34] underlying openMSP430 platform [17] contribute to even more severe leakage. These features of the architecture make the attack completely deterministic and accurate to a single clock cycle. The partitioning of the memory with different interfaces also makes the attack more fine-grained and provides more information about the memory accesses than a system with a single memory interface.
- On top of the theoretical explanation and the analysis of the leakage, we present a minimal, programmable peripheral device implemented on the current, unmodified Sancus platform. This peripheral conforms to the attacker model of the architecture and is capable of capturing these cycle-accurate traces.

  We also describe some limitations of both the peripheral and the attack in general.
- Lastly, we discuss potential defenses against the attack and highlight some challenges. We explain how balancing the instructions of conditional branches could eliminate the leakage, but also show some of the issues with this approach.

  A possible hardware defense is also proposed, where DMA requests have priority over the CPU's memory requests and always take a fixed, constant time to complete, which would also eliminate the leakage.

Using a modified attacker peripheral, we experimentally show that the current implementation of this strategy on Sancus is flawed, and enables the attacker to capture the same information leakage through a small modification in the malicious peripheral. An alternative solution is discussed which would eliminate the current leakage, but introduce a different (Nemesis-type) leakage.

The main code contributions of this thesis are in the process of being upstreamed to the Sancus repositories and are available on the following pages:

- https://github.com/sancus-pma/sancus-examples/pull/13
- https://github.com/sancus-pma/sancus-core/pull/19
- https://github.com/sancus-pma/sancus-support/pull/10

1.3 Outline

Chapter 2: Background An overview of Trusted Execution Environments is provided with a focus on Sancus, the target architecture of this thesis. This is followed by an introduction to side-channel attacks and a showcase of Nemesis, the current state-of-the-art side-channel attack on Sancus. Finally, a discussion of Direct Memory Access and the security concerns with its implementation is provided, together with examples of DMA-based attacks.

Chapter 3: DMA-based side-channel attacks We start this chapter with a theoretical explanation of the attack, then show how it can be implemented on the Sancus platform, and why certain design features of the base openMSP430 platform amplify the severity of the leakage.

We describe the concrete implementation of a peripheral device that can execute the attack, and show a real-world example of using it.

A detailed discussion is provided that showcases the factors that influence the memory access pattern of an instruction.

We conclude the chapter by comparing the leakage to that of Nemesis and listing the limitations of the attack.

Chapter 4: Defenses Following some discussion on attack detection and trivial ways of prevention, this chapter proposes two defense methods and describes some challenges with their implementation; one on the software level (code balancing), and one in hardware (prioritizing DMA requests over the CPU).

We experimentally show that in the current Sancus implementation, assigning the priority to DMA devices is not a sufficient defense, and in fact provides the same information leakage as before, only requiring a slight modification in the attacking methodology.

Chapter 5: Conclusions and future work Finally, in the last chapter we summarize the thesis and propose a few directions for future research related to the presented attacks.

Chapter 2

Background

2.1 Trusted Execution Environments

Trusted Execution Environments became a topic of research in the past two decades, and have since also appeared in commercial and industrial systems. The term TEE does not have a universal definition [39], but Maene et al. [29] list the following five features as being fundamental to most TEEs:

Isolation One of the crucial features of most TEEs is the ability to execute a process in isolation, meaning that no other running process (privileged or not) can interfere with its execution or extract sensitive data from it. These processes are said to run in an enclave, which provides the protection mechanisms.

More concretely, enclaves usually consist of a protected code and a protected data section. Other processes are not allowed to read or modify the protected code, and can only jump to predefined entry points that represent public functions of the enclave. This prevents some Return Oriented Programming [37] based exploits, which would chain snippets of the enclave's existing code together to divert the control flow and ultimately lead to the leakage of secrets.

Other processes are also prevented from reading or writing the protected data section of the enclave. This isolation is often – for example in the case of Sancus [34] – provided by program counter based memory access control. This is an inexpensive, but effective way of enforcing the memory boundaries between the enclaves and other unprotected code [42].

Attestation Most TEE architectures provide a mechanism for third parties to verify the integrity of the enclaves in the system. The enclave is said to have an identity that is dependent on the contents of its code and static data values. If the enclave produces the expected value after it has been loaded, the third party knows that the code and the initial values of the enclave have not been tampered with. The actual implementation of this mechanism depends on the TEE, on Sancus for example the calculation involves an enclave-dependent key that is only accessible from hardware, to prevent a malicious enclave falsely reporting the expected identity [34].

Attestation can take place either locally between enclaves, or remotely between an enclave and a third party over the network.

Sealing Enclaves might need to save their state between executions. Sealing allows enclaves to save data on the device that is only accessible to them, potentially with more constraints, for example binding it to a certain state of the device.

Dynamic Roots of Trust When building trust chains on a TEE e.g. for attestation, these chains need to contain an inherently trusted root item (Root of Trust). A Dynamic Root of Trust can be generated at runtime by the TCB, using the properties of an enclave after initialization, similar to the process for calculating the identity for attestation.

Code confidentiality In some cases, programs need to be protected even when they are not executing. If the code is protected intellectual property or it contains sensitive static values, such as identifiers or keys, it should be inaccessible on the storage device. Confidential loading enables programs to be stored encrypted, and be decrypted when the loading is performed – after which the isolation guarantees will continue to protect its contents.

TEEs usually strive to have a minimal and hardware-based TCB to be able to guarantee these security features. Relying on a complex software layer – such as a commercial operating system – with potential vulnerabilities for security-critical tasks could easily compromise the guarantees offered by the TEE.

Attacker model The attacker model used by most TEE systems is fairly universal [29] and assumes an attacker with strong capabilities.

Adversaries are assumed to have control over all untrusted software on the platform, including the operating system. In most models, they are also able to create new enclaves containing their malicious code.

The attacker is also assumed to have control over the network; they can inject, remove, modify packets traveling on the networks the TEE is connected to. However, in line with the Dolev-Yao attacker model [14], they are not able to break cryptographic primitives.

Usually, TEEs offer no guarantees regarding the availability of enclaves, Denial-of-Service attacks are out of scope. Side-channel attacks are often also not directly addressed in the attacker model of TEEs, but for example Sanctum [12] guarantees the ineffectiveness of certain side-channel attacks against enclaves on the platform.

In the following subsections we will briefly introduce Intel SGX and ARM TrustZone, two commercial TEE solutions and their security models, then focus on Sancus, which is the target architecture of this thesis.

2.1.1 Commercial solutions

Intel's Software Guard Extensions (SGX) [11, 31] extend the instruction set and modify the hardware components of modern high-end Intel processors to offer TEE capabilities. In SGX, enclaves run in the address space of their host application, which is a regular user-level process.

SGX includes both the processor and certain software components like predefined Intel-provided enclaves in the TCB. In contrast to some other TEEs like Sancus, the system memory is not included in the TCB; when writing or reading enclave data

from the memory, a hardware component (Memory Encryption Engine) is responsible for automatic encryption and decryption of the data.

SGX allows attackers to issue DMA requests to the memory, but any request that targets an address belonging to an enclave is treated as invalid.

The attacker model of SGX explicitly places side-channel attacks out of scope [11, 20].

ARM TrustZone [36, 7] is a collection of hardware and software components on an ARM System-on-Chip. These components separate the processor into a "secure world" (protected) and a "normal world" (unprotected) region. The components enforce that "normal world" components cannot access "secure world" resources. Therefore the TrustZone architecture does not support running separate enclaves that do not mutually trust each other.

TrustZone does not claim to defend against software side-channel attacks, and in contrast to SGX considers the memory to be part of the TCB. TrustZone also utilizes a secure-world operating system, which increases the size of the software layer of the TCB considerably.

2.1.2 Sancus

Sancus [34] is a research-focused TEE with a hardware-only TCB. It utilizes program counter based memory access control [42] to provide isolation to the enclaves that are executed on the platform.

Sancus features a von Neumann memory architecture with a single address space. Enclaves on Sancus have two designated memory sections, a code and a data section. Memory accesses to these areas are restricted by the memory backbone. The data section of an enclave can only be accessed if the program counter of the CPU is pointing to the text section of the same enclave. The text section has a designated entry point, the only operation permitted for untrusted code is jumping to this entry point.

Sancus was designed specifically for networked embedded devices and allows third-party software providers to deploy enclaves over the network and perform remote attestation on them. It also enables secure communication both between two enclaves and between an enclave and a software provider.

Since its introduction, Sancus has been used to implement a smart metering application [33] and a secure inter-component communication framework in automotive applications [45].

The architecture is based on the openMSP430 platform [17], which is an open-source implementation of TI's MSP430 architecture [19]. Since its 2.0 release [34], Sancus has been extended with multiple new features. In connection with the Nemesis attack [49], interrupt handling was enabled and secure interrupt handling routines were added to the compiler.

Seminara [40] performed a security analysis of adding Direct Memory Access support to Sancus, and subsequently, the DMA capabilities of openMSP430 were added to Sancus with some modifications to preserve the isolation guarantees (see section 2.6 for the implementation details).

While these features extend the capabilities of Sancus and make it more user-friendly and performant, they also increase the attack surface, as already shown by the Nemesis attack [49], and hinted at by the thesis extending Sancus with DMA [40].

Attacker model The attacker model of Sancus is similar to the general model introduced earlier. The main memory is part of the TCB, attackers cannot directly access the data of enclaves by probing the memory bus or performing other physical attacks on the memory chip.

In the original attacker model [34], peripherals were also considered part of the TCB and attackers were not allowed any physical access to the device, but with the addition of DMA support and the related protections, the attacker model has been modified [40]; peripherals have been moved outside of the TCB, and adversaries are now allowed to connect peripherals to the system or compromise and replace existing ones.

Sancus also considers availability guarantees out of scope, a malicious party can cause memory violations by addressing an enclave's memory, which causes the processor to reset, thus resulting in a low-effort Denial-of-Service attack. However, efforts have been made to extend the architecture to support real-time guarantees with a trusted scheduler [46].

openMSP430 architecture As mentioned above, Sancus is based on the openMSP430 architecture [17], an open-source architecture based on and compatible with TI's 16-bit MSP430 [19]. The design can be flashed to a Field-Programmable Gate Array (FPGA) or be implemented as an Application-Specific Integrated Circuit.

Figure 2.1 shows the schematic of the architecture.

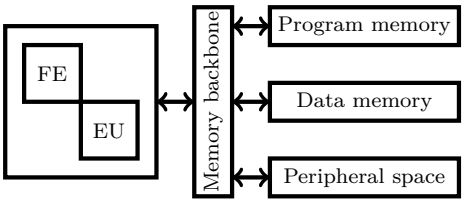


Figure 2.1: Simplified figure of the openMSP430 architecture.

The CPU features a two-stage pipeline implemented as two separate units. The *frontend* (FE) fetches the instructions from the program memory and handles the instruction decoding. The *execution unit* (EU) is responsible for performing arithmetic operations, loading and storing values from the memory or the register file. The frontend handles the instruction fetch and decode in a single cycle, the execution unit starts the execution in the next clock cycle. The frontend starts fetching and decoding the next instruction while the current instruction's execution is in its last cycle.

All memory accesses from the CPU and the DMA interface are handled by the memory backbone. The memory address space is divided into three partitions: program memory, data memory, and memory-mapped I/O (denoted by peripheral space on the figure). The memory backbone provides separate interfaces for these regions, and it also handles the address translation from the global address space to these separate partitions.

The frontend only fetches instructions from the program memory, the documentation of openMSP430 [17] lists not being able to execute instructions from data memory

as a known limitation. The execution unit can read and write addresses in all three memory partitions.

2.2 Full abstraction

When analyzing the security of a system, the notion of full abstraction [2, 1] can be used to provide a framework to reason about information leakage.

Two programs are contextually equivalent if – on some abstraction level – the programs produce the same observable output when given the same input [2]. The notion of an oracle can be used, which can execute the programs with different pairs of inputs and can observe any outputs the program might have – including indirect outputs, such as modifying publicly accessible memory locations.

If no input exists for which the oracle observes different outputs from the two programs and is thus unable to differentiate between the two, we say that the programs are contextually equivalent.

Listing 2.1 and 2.2 show two programs that are contextually equivalent (based on [2]): the `secret` field is not accessible from outside of the class instance.

```
class C {
    private int secret = 0;

    void m() {
        secret = 0;
    }
}

c = new C();
c.m();
```

Listing 2.1: Program 1

```
class C {
    private int secret = 0;

    void m() {
        secret = 1;
    }
}

c = new C();
c.m();
```

Listing 2.2: Program 2

A compilation or translation scheme is said to satisfy full abstraction if it translates contextually equivalent programs in one abstraction level to contextually equivalent programs in the other abstraction level. And vice versa, non-equivalent programs never translate to equivalent programs on the lower abstraction level.

Using full abstraction can be beneficial when proving whether an extension to a system (such as enabling DMA) introduces any information leakage. If we can show two programs that are contextually equivalent on the source code level, but their translations are no longer contextually equivalent after adding the extension, we know that new information leakage has been introduced to the system.

For example, interrupts on Sancus have been proven to break full abstraction when implemented naively [9], which is also the root cause of the Nemesis attack [49].

2.3 Secret-dependent control and data flow

Two patterns in software that are prime candidates for leaking information are secret-dependent control and data flows. A sequence of instructions is called a secret-dependent control flow if a conditional jump is executed that is dependent on some

sensitive value. An example of this is comparing a stored password to a user-provided guess and executing some instructions in case of a match.

```
if (password == guess) {
    open_admin_menu();
} else {
    log_attempt();
}
```

Listing 2.3: A secret-dependent control flow

A secret-dependent data flow happens if depending on the value of a secret a memory read or write is executed with different locations or values. An example of this would be an enclave that takes a password guess as an input and sets different flags depending on whether the guess was correct.

```
if (password == guess) {
    is_admin = 1;
} else {
    is_guest = 1;
}
```

Listing 2.4: A secret-dependent data flow

2.4 Side-channel attacks

Traditionally, the *architecture* of a system refers to the properties that define the functionality of the system [6]. This information encompasses for example the instruction set, addressing modes, and register set. In contrast, the *microarchitecture* of the system refers to the implementation details of this system, such as caching, instruction reordering, and other features that influence the behavior of the system, but not its functionality.

Put differently, a program written for a certain architecture will give the same result on different implementations of the same architecture, regardless of the microarchitectural implementation of the systems.

Side-channel attacks are a recent class of attacks in the security literature that target these implementation details instead of vulnerabilities in the application. The attacks observe the microarchitectural properties and state of the system to extract secret information from the victim processes. They are especially interesting in the context of TEEs, which usually only guarantee isolation on the architectural level.

While traditional attacks target vulnerabilities such as buffer overflows [3] in an application's code to extract information, side-channel attacks use either physical measurements such as power analysis [22], memory probing [27], or attributes of the system that can be measured from software.

These software side-channel attacks have targeted a large variety of microarchitectural features to extract information from a process. Many of these attacks are based on accessing shared resources, and on the fact that one process can affect the microarchitectural behavior (for example execution time) of another if they use the same constrained resource on the system.

One of the earliest side-channel attacks used the differences in execution time of cryptographic operations to extract the keys used for the calculations [24].

Caching attacks such as Flush+Reload [53] measure the timing difference between cache hits and misses to infer the memory access of the victim process. DRAMA [35] measures the timing difference caused by a reloaded row buffer in the DRAM to infer whether a victim process has activated the same row as the attacker. Yuanzhong Xu et al. presented an attack [52] that induces page faults to examine the memory access patterns of a victim process. Nemesis [49] and Copycat [32] target the interrupt handling mechanism to infer the types and number of instructions executed on the processor for a victim process.

Some of these side-channel leakages can also be used for building a covert channel between processes running in different security contexts. As an example, Flush+Reload is used for the Spectre [23] and Meltdown [28] attacks that target speculative and transient execution in high-end processors to leak information from a privileged process.

Numerous side-channel attacks are either applicable on a TEE [47] or have been ported to show their effectiveness on these platforms with a stronger security model than most conventional systems. Alongside Nemesis and Copycat, attacks on Intel SGX include Foreshadow [44], which is an attack targeting enclaves based on an idea similar to Meltdown [28], and SgxPectre [10], which exploits the idea of Spectre [23] on SGX.

In the next section, we will focus on Nemesis, because this is the only attack that has been shown to work on Sancus, the target of this thesis. The other side-channel attacks are ineffective on Sancus due to its minimal design and lack of optimization features such as caching or speculative execution.

2.5 Nemesis

The state-of-the-art software side-channel attack on the Sancus platform is Nemesis [49]. Nemesis is a side-channel attack that targets the secure interrupt handling mechanism of the processor. Although the interrupt handling code in the compiler has been hardened to prevent directly leaking any data from the executing enclave (such as register values or the stack pointer's position), an attacker can still gain information about the process through the timing of the interrupt requests.

The main idea behind the attack is that during interrupt request handling, the CPU first finishes executing the instruction that is in the execution phase when the interrupt request arrives. Because in the openMSP430 architecture different instructions take different numbers of clock cycles to execute [17], the time it takes for the interrupt request to be handled is also different based on which instruction was executing at the time the request came in. In other words, a well-timed interrupt request leaks the number of cycles the currently running instruction in an enclave takes to execute.

This timing difference enables the detection of certain secret-dependent control flows, and even secret-dependent data flows in some cases. If the instructions in the two branches have different execution lengths, an attacker can tell which one is executing based on how many cycles it takes for an interrupt request to be handled that was issued exactly in the cycle when one of the two instructions starts executing.

To illustrate this attack, we use the small program in listing 2.5, based on TI's bootstrap loader code from the Nemesis paper. On the left, the assembly code of the enclave can be seen, on the right, we listed a high-level representation of the code. This simple program takes a guess for the password in the `guess` variable (`r6` register) and compares it to the predefined value of `0x42` (`password` variable). If the values do not match, the `incorrect` variable is set to true (a bit is set in the `r8` register). If the guess was correct, nothing happens. At the end of the execution, the `incorrect` variable is set to false again (`r8` register is zeroed out), so a full abstraction oracle cannot differentiate the control flows based on the value of `r8`.

```
    cmp.b @r6, r7          ; if (guess != password) {
    jz 1f
    bis #0x1, r8           ;     incorrect = true;
    jmp 2f                 ; } else {
1:  nop                    ;     // do nothing
    nop
    nop
2:  nop                    ; }
    mov 0x0, r8            ; incorrect = false;
```

Listing 2.5: A password comparison program

Looking at the assembly code, we can see that this code has been hardened against a simple end-to-end time measurement attack. If the `else` branch of the conditional (lines 6-8) had not been padded with the `nop` instructions, it would have been possible to tell which branch was taken based on the time it takes for the entire program to execute: the `if` branch would take longer because it would include the time it takes to set the `incorrect` variable to true.

Figure 2.2 shows the timing diagram of some of the internal signals of Sancus for the execution of the enclave with the correct input (`0x42`) provided as the password guess. [1]

The `exec_done` signal deserves special attention here. This is an internal signal used by the CPU to signal when the final cycle of an instruction's execution takes place. In other words, this signal is high one cycle before a new instruction starts executing. The delay between two cycles with `exec_done` high equals the number of cycles the given instruction takes to execute. This is exactly the information the Nemesis attacker is capable of inferring from interrupt requests. This also corresponds to the length of the instruction boxes in the figure.

On figure 2.3 the same signals are shown for the execution of the enclave when the user supplies an incorrect value in the `guess` variable. Notably, in the 6th clock cycle, the `exec_done` signal is high in the first figure, but not in the second one. This is because in the first case, two subsequent `nop` instructions are executed, while in the second case the `jmp` instruction is running due to the guess being incorrect.

Because the `incorrect` variable (the `r8` register) is cleared at the end of the program, for a source-code level observer, the outputs are exactly the same, regardless

[1] These traces were generated using the vcdvis framework, which was also developed as part of this thesis: https://github.com/martonbognar/vcdvis
When comparing timing diagrams in this thesis, we will always position them vertically to make the comparison of signals in a given clock cycle easier.

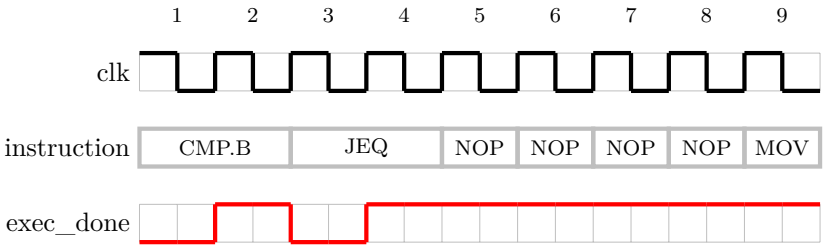


FIGURE 2.2: Timing diagram of a correct guess

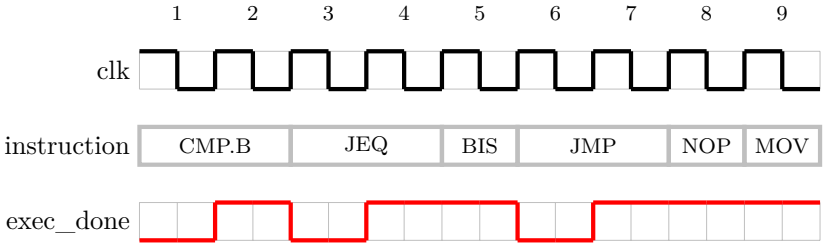


FIGURE 2.3: Timing diagram of an incorrect guess

of whether a correct input is provided or not. For an observer at the microarchitectural level however, the modules have a different observable output based on the input. This is because at this level, the leakage provided by the Nemesis attack is also considered to be part of the output, and as we have seen in the traces, those are not equivalent.

To make the attack more clear, figures 2.4 and 2.5 provide graphs of the observed interrupt latencies when executing the enclave with a correct and an incorrect guess as inputs. It can be observed that in one case, we have multiple interrupts with the delay of one cycle, which corresponds to the `nop` instructions, while in the other case, we have a longer interrupt delay of two cycles that is caused by the `jmp` instruction after setting the `incorrect` variable. Because of this difference, an observer (the oracle) is capable of differentiating the two executions with the different inputs, thus breaking the notion of full abstraction.

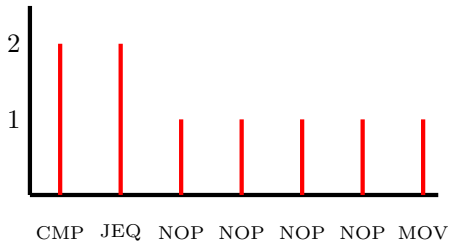


FIGURE 2.4: Instruction timing traces with a correct guess

The vertical axis represents the interrupt latency in clock cycles

A number of attacker frameworks have been implemented to enable easy exploita-

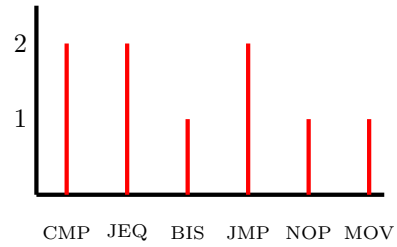


Figure 2.5: Instruction timing traces with an incorrect guess

The vertical axis represents the interrupt latency in clock cycles

tion of the vulnerability on different systems. SGX-Step [48] is a framework that allows attackers to single-step enclaves (execute instructions one-by-one) on Intel's SGX TEE. The complexity of Intel SGX and the optimization techniques it employs can make latency measurements less deterministic, but some of these features can be turned off by privileged attackers, and statistical methods can compensate for measurement noise.

Similarly, Sancus-Step [13] is the equivalent framework for Sancus. Sancus-Step enables software attackers to define their own interrupt routine, then generate an interrupt and run the handler routine after the execution of every instruction of a victim enclave.

As part of this thesis, a number of pull requests were submitted to Sancus-Step to improve its robustness and fix some issues. [2]

An effective software defense against Nemesis (other than disallowing interrupt requests) would either eliminate secret-dependent control flows entirely or balance them in a way that both branches have sequences of instructions where the nth instruction in one branch has the same execution length as the nth instruction in the other branch.

On the microarchitectural level, a provably correct padding scheme has been proposed that eliminates the information leakage [9].

To summarize, Nemesis is a side-channel attack against the Sancus and SGX platforms that enables differentiating between certain secret-dependent control and data flows. The collected traces have a granularity of individual instructions and are accurate to the cycle level.

2.6 Direct Memory Access

In systems with no DMA support, the processor needs to facilitate data transfers between peripherals and the memory. To store sensor readings in memory for example,

[2] The contributions are available on the following pages:

- https://github.com/jovanbulck/nemesis/pull/2
- https://github.com/sancus-pma/sancus-support/pull/8
- https://github.com/sancus-pma/sancus-compiler/pull/25
- https://github.com/sancus-pma/sancus-examples/pull/10

the CPU needs to periodically poll the sensor for new data, read it from the peripheral, then write the values to memory, or the peripheral needs to interrupt the processor whenever a new reading is available, which then the processor can write into memory.

This is especially an issue in high-performance systems where a memory access operation can take multiple hundreds of clock cycles, during which the processor cannot perform other operations. With DMA support in the system, these data transfers can take place completely independently of the processor or can be initiated by the processor, but run in parallel with its execution, generating an interrupt when it finishes. In general, DMA can offer a significant performance increase, especially in systems with a high volume of I/O traffic with peripheral devices.

2.6.1 Implementation details

Systems only have a limited number of DMA interfaces connected to the memory, usually a component known as the DMA Controller is responsible for arbitrating between the DMA requests originating from different devices.

This DMA Controller can be programmed by the processor by modifying its registers (or potentially by other peripherals through dedicated control buses) to handle the data transfers independently of the processor.

In some systems where only one device needs DMA capabilities, the DMA Controller can also be omitted, and a peripheral device can be directly connected to the memory to handle DMA transfers.

Modern commercial systems usually offer DMA capabilities to peripheral devices through a Firewire, Thunderbolt, or USB-C connection and internally through PCI Express. More recently, some systems also provide Remote DMA capabilities that allow components that connect to the system remotely – over a conventional network connection – to issue DMA requests.

Another emerging trend is heterogeneous CPU-FPGA platforms, even as an offering from cloud providers [5, 4] which enable a flexible way to offload computations to a dedicated programmable hardware component. In these systems, the FPGA platform often communicates with the main memory using DMA requests.

openMSP430 The openMSP430 system offers DMA capabilities [17], but does not provide a DMA Controller out of the box, meaning that only one peripheral device can issue DMA requests.

Because the main memory can only handle one operation at a time, the memory operations issued by the processor and the DMA requests need to be arbitrated. In openMSP430 the priority is controlled by a dedicated signal in the system. Originally this signal was meant to be driven by the DMA Controller or the DMA device [17], but in the Sancus implementation it has been hardwired to give the processor priority [40].

This arbitration only takes place if a DMA request and the processor try to access the same memory partition in the same clock cycle, requests to different memory partitions can be handled in parallel on openMSP430.

2.6.2 Security of DMA

In the modified Sancus attacker model the adversary is assumed to control the peripherals in the system, which includes the DMA Controller if the system is equipped with one. The priority between the CPU and the DMA Controller was given to the CPU to prevent a potential Denial-of-Service attack [40]: in case the DMA device has priority over the memory requests issued by the CPU, a malicious peripheral can continuously issue DMA requests to the program memory, thus denying the program executing on the CPU from loading its instructions and executing.

This is far from the only issue with including DMA on a system. As DMA accesses the memory while bypassing the CPU, it also bypasses the protections the CPU provides. With a naive implementation of DMA, peripheral devices have unlimited access to the entire memory address space. This leads to trivial attacks by reading or overwriting the code or data structures of executing programs [8].

For memory requests issued by the CPU, a component called the Memory Management Unit provides bounds checking that handles virtual memory accesses and prevents processes from accessing each other's memory region. To achieve a similar effect, modern systems include a component usually referred to as the IOMMU (Input-Output Memory Management Unit), which regulates the DMA requests generated by peripherals to only target appropriate memory regions.

While the IOMMU is a step in the right direction and does protect against certain attacks, it is not a silver bullet. Several attacks have been found that can circumvent the IOMMU or exploit a bug in its implementation to still access memory belonging to other processes [30, 25].

Vulnerabilities have also been found in the underlying Thunderbolt protocol [38] that allow an attacker to for example clone the identity of a trusted peripheral and thus bypass the manual approval the user would have to grant to peripherals before they can issue DMA requests.

Apart from accessing the memory space of victim processes, DMA requests have also been utilized in other attacks.

The original Rowhammer attack modifies the data stored by a victim process on a DRAM chip by exploiting the physical properties of a chip and flipping bits by rapidly accessing nearby locations in the memory. This attack has been modified to be executed over the network using RDMA [43], and on a heterogeneous Intel Aria platform using DMA requests from an FPGA [51].

NetCAT [26] is an attack that can recreate a cache side-channel attack on modern Intel processors over the network using RDMA requests that can be served from the cache (dubbed Direct Cache Access).

DMA in TEEs Implementing DMA on TEEs without breaking the isolation guarantees poses several challenges. Allowing unrestricted memory access to peripheral devices as it existed on early systems and as it does now on embedded architectures – such as the openMSP430 – would trivially break the isolation guarantee by allowing peripheral devices to access the private memory of enclaves.

According to the security analysis conducted on introducing DMA to the Sancus architecture [40], program counter based memory access control cannot effectively regulate DMA requests in a way that allows trusted peripherals or enclaves to issue

DMA requests targeting their protected memory section due to the inherently parallel nature of the operation with regards to the processor.

The program counter can change during the DMA transfer if another process takes the priority on the system, or alternatively, a malicious program could initiate a DMA transfer to copy data from an enclave's protected memory, and immediately jump to the enclave to have the program counter point to a location that would enable the transfer.

Both on Sancus [40] and on Intel SGX [11] DMA transfers have been restricted to unprotected memory sections to avoid the possibility of leaking information from enclaves.

Vulnerabilities related to DMA have already been discovered in a TEE context, Mathieu Gross et al. [18] present a vulnerability in a TrustZone enabled heterogeneous CPU-FPGA platform that breaks the isolation guarantees and allows untrusted DMA requests to access secure memory.

2.7 Conclusions

Recent research has introduced Trusted Execution Environments (TEEs) to improve the security guarantees for both high- and low-end systems. TEEs offer architectural isolation for programs among other security features, and have already found some commercial success.

Side-channel attacks target the microarchitectural implementation details of a system. This makes them especially relevant for TEEs, as they are capable of breaking the isolation guarantees that are guaranteed on an architectural level. These attacks often target optimization features and shared resources on a system, such as the caching mechanism or a shared main memory unit.

Direct Memory Access is an optimization technique that allows peripherals in the system to access the main memory without the processor having to handle the transfer, thus freeing up execution time. DMA has been shown to also increase the attacker surface, a number of attacks exist in the literature that utilize DMA accesses in traditional systems to leak or modify information of a victim process. The microarchitectural security implications of introducing DMA has not received much attention so far.

Chapter 3

DMA-based side-channel attacks

Side-channel attacks often exploit a shared resource to leak information about an isolated process. This resource can be the cache, as in the case of Flush+Reload [53] and other attacks, where cache eviction leads to some memory accesses served by the main memory which causes a measurable delay compared to a cache access. It can be the execution unit of the processor, as in the case of Nemesis [49], where interrupt requests are served with a delay that depends on the currently executing instruction, or the main memory, as in the case of DRAMA [35].

While many side-channel attacks have been discovered on high-end systems, this field has been relatively unexplored in the context of embedded TEEs, partly due to their lack of optimization features that are often the source of the information leakage.

To date, the only side-channel attack on the Sancus platform is Nemesis, which exploits the implementation details of the execution pipeline. It has also been shown that Nemesis can be patched by implementing a padding scheme for interrupts [9].

In this chapter, we present a memory-based side-channel attack with a leakage comparable to that of Nemesis. Our attack is experimentally shown to work on Sancus and can be mounted by an unprivileged attacker with DMA capabilities.

Outline of the chapter First, in section 3.1 we explain the basic information leakage that results from the contention of the shared memory bus. Section 3.2 shows a practical example of extracting a password from an enclave using the attack technique.

Section 3.3 explains what makes an enclave vulnerable against this attack. Section 3.4 details how the attack can be executed on the Sancus platform and shows why the leakage is more severe on that platform, then section 3.5 explains the implementation of our malicious peripheral.

We analyze the leakage by systematically discussing the factors that influence the memory trace of an instruction in section 3.6. We continue the analysis by comparing the leakage to interrupt-based attacks in section 3.7 and present our extension of the Sancus-Step framework to simplify the extraction of the memory traces from software.

Finally, we discuss some limitations of our attack in section 3.8.

3.1 Basic attack

Our attack targets the shared memory bus. The property we exploit for the attack is that the memory unit is only able to serve a single request at a time. If a component of the system issues a memory request at the same time as another component, or while another request is being served, one of the requests needs to be delayed until the other request finishes. This contention is shown in figure 3.1 with a concrete example of a system where both the processor and a DMA-capable device can issue memory requests.

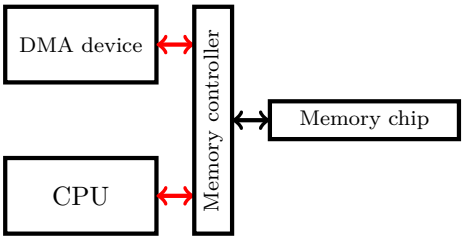


Figure 3.1: Memory contention on a system with DMA

By measuring the time it takes to complete a memory request, it can be inferred whether that request was delayed or not. As a result, the issuer also gains information about whether or not other memory requests were issued at the same time.

It is important to consider the question of priority. If different components issue a memory request at the same time, the memory controller needs to make a decision about which request to serve first. If memory requests `A` and `B` are issued at the same time, and `B` is served first – delaying `A` until `B` completes – only the issuer of `A` notices a delay, thus only they are able to infer that a resource contention happened, `B` is executed in the same time as it would have taken with no contention.

Enclaves in a TEE are executed on the trusted procesor, so an attacker trying to break the isolation of an enclave needs to trace the memory requests issued by the CPU. In our simplified model we consider a CPU that is capable of issuing a single memory request at a time, so in order to cause resource contention in the memory, another component in the system needs to also be capable of issuing memory requests, for example via DMA requests.

Attacker model Our attacker model is in line with the modified Sancus threat model introduced in 2.1.2.

In our attack, we consider an attacker that is capable of issuing DMA requests either by connecting a peripheral to the system, or by some other means, for example by compromising the firmware of an already connected peripheral, such as a network controller. This approach is desirable because many existing TEE systems such as Intel SGX [31] and the current version of Sancus [40] allow untrusted parties to issue DMA requests, and (at least on Sancus, SGX was not examined in this thesis) the CPU's requests are prioritized over DMA requests in the memory controller.

The attacker also needs access to a cycle-accurate clock that allows the precise measurement of the time the DMA requests take to complete. Using this measurement,

the attacker can infer whether the CPU was accessing the memory at the same time they issued a DMA request. If the request is delayed, the CPU accessed the memory, otherwise it did not.

We also assume knowledge of the source code of the executing enclave. This enables more precise attacks that require a single execution of the enclave with a given input and a limited number of DMA requests, but in theory any vulnerable enclave's isolation guarantees can be broken by a dedicated attacker, even without knowledge of the source code.

Crucially, the attacker does not have physical access to any of the componenets in the TCB such as the memory controller or the memory module, and cannot access the protected memory regions of the enclaves in any way, including with DMA requests.

Noise and undeterminism An additional concern is noise and undeterminism. If the timing difference between delayed and not delayed requests cannot be distinguished, the attacker cannot reliably discern when the CPU accessed the memory. Another issue could be the memory activity of other components in the system. If other peripherals issue DMA requests at the same time, or the memory is accessed in some other way and the attacker's DMA request is delayed, they could falsely conclude that the request was delayed as a result of the CPU accessing the memory.

Furthermore, systems making use of caching mechanisms often serve memory requests from the cache, avoiding the expensive operation of accessing the main memory. In this case, no activity is observed at the memory, even though the CPU issued a memory request.

These are not a concern on the Sancus architecture, which only allows a single DMA peripheral to be connected and has no caching capabilities. On other, more advanced platforms, such as Intel SGX [31] and ARM TrustZone [7], these effects can be minimized as shown for example in the DRAMA attack [35], but the implementation of this attack on these platforms is left as future work (refer to chapter 5).

For more discussion on the attacker model and the limitations of the attack, see section 3.8.

3.2 An end-to-end example

Assuming that the attacker is capable of accurately measuring when the CPU accesses the memory, we still need to establish that this leakage contains meaningful information.

As a minimal example, let us first examine two single instructions. For this example we will use two openMSP430 instructions, `mov` and `add`. `mov` stores a byte in memory, while `add` adds a byte to a value stored in memory. Figure 3.2 shows timing diagrams extracted from the Sancus simulator.

The Sancus simulator allows running simulations on the Verilog implementation of the framework and examining the internal signals. This same code is synthesized to the FPGA for the real implementation, so the behavior of the simulator and the real system is equivalent on a cycle-accurate level. Thus any insight gained by analyzing the simulations is also applicable in the real implementation.

This figure shows that `add` issues two memory requests (one to read the initial value in the 2nd cycle, and one to write the sum in the 4th), while `mov` only issues

one (to move the value into memory in th 4th cycle). Based on this information, it is clear that the two instructions differ on a microarchitectural level.

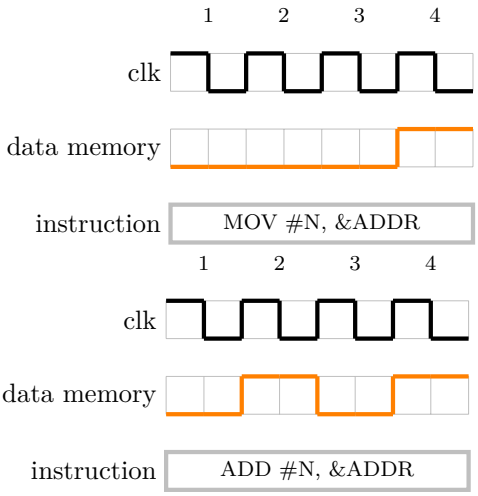


Figure 3.2: Memory traces for `mov` and `add`

To show the value of this information in a real-life scenario, we use the notion of full abstraction introduced in section 2.2. If as a result of this information leakage, two contextually equivalent programs at the source code level can be distinguished at the microarchitectural level, we can conclude that a DMA attacker is capable of breaking the isolation guarantees and as a result can extract information regarding the enclave's state.

In this model, which follows the openMSP430 architecture [17], memory requests issued by the processor and DMA requests both take a single cycle to complete. If a memory request is issued by the CPU in the same cycle as a DMA request, the issuer of the DMA request will experience a measurable delay (an extra cycle) before the request completes.

Memory requests are issued by the processor while executing instructions. If executing different intructions results in different memory requests issued by the CPU, the attacker is able to determine which instruction sequence was executed based on the collected memory trace.

```
    mov &password , r7
    cmp r6 , r7                 ; if (guess == password) {
    jnz 1f                      ;
    mov #0x00 , &guesses        ;     guesses = 0;
    jmp 2f                      ; } else {
1:  add #0x01 , &guesses        ;     guesses += 1;
    jmp 2f                      ; }
2:  mov #0x0 , r7               ; password = 0;
```

Listing 3.1: The running enclave example

For our running example, we will use the instruction set of openMSP430, on which the Sancus TEE is based. Consider the enclave in listing 3.1. This program checks whether the user's supplied password guess is correct. If the user's guess is incorrect, the `guesses` counter is increased, but if it is correct, the counter is reset to zero. Both the `guesses` and the `password` variables are located in the enclave's protected memory section.

This code is considered secure in the current Sancus implementation. The conditional control flow is balanced; both the `mov` and `add` instructions take 4 clock cycles to complete as shown on figure 3.2. This makes the two branches indistinguishable both for a start-to-end timing measurement and a Nemesis attacker.

Because the `guesses` variable is in the protected section of the memory and the `r7` register is cleared before exiting the enclave, this program has no observable output on the source code level that allows differentiating between providing a correct and an incorrect password guess.

If we consider two instances of this enclave initialized with different passwords, E_1 with the password `42` and E_2 with the password `41`, these enclaves are contextually equivalent on the source code level. There is no input the attacker can choose that will result in different observable outputs by the two enclaves.

Figure 3.3 shows the complete timing diagram of executing the enclave with different inputs. Even though on the source level running the enclave with a correct and an incorrect guess are indistinguishable – they are contextually equivalent – these traces show a difference in the 8th clock cycle. This means that an attacker can observe different outputs from the enclaves E_1 and E_2 by providing `41` or `42` as input. More precisely, we can see that the difference is caused by the `add` and the `mov` instructions' memory access patterns that were previously shown in figure 3.2.

This finding means that the attacker with the DMA request measuring capabilities can break full abstraction on the system.

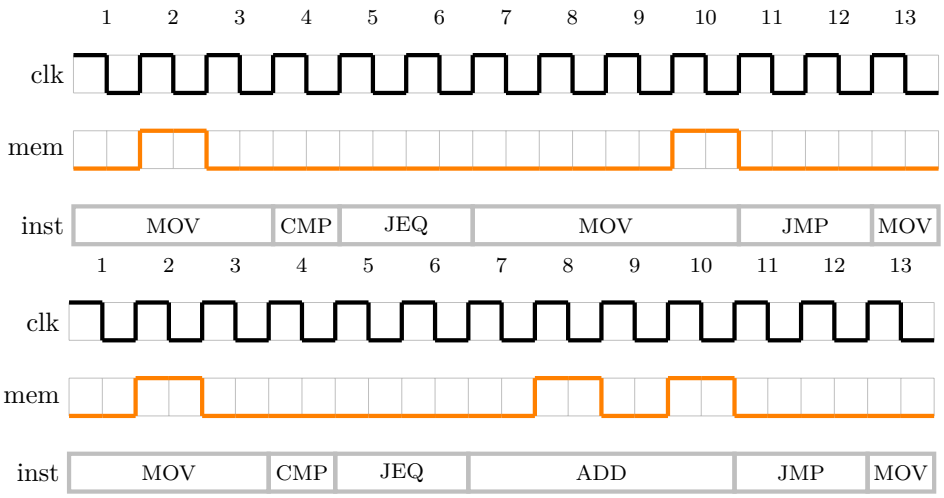


FIGURE 3.3: Timing diagram for a correct and an incorrect password guess

This also means that the attacker is capable of extracting the password from the

enclave by bruteforcing all possible passwords. For whichever password the memory access pattern is different – showing the `mov` pattern instead of the `add` pattern – that is the correct guess.

In a more realistic example (such as the password comparison routine presented for the Nemesis attack [49]) the password would be a multi-byte string, and the user of the enclave would get some indication about whether their password guess was correct or not. Bruteforcing a multi-byte password where only the final result is known (correct or incorrect) takes an exponential effort in the number of bytes.

If the loop that decides whether a given byte of the password matches or not is vulnerable against the attack, that reduces the required complexity from an exponential to a linear effort.

3.3 Vulnerable enclaves

The previously described attack works by detecting a secret-dependent control flow. The executed instructions depend on the user-provided password guess. We have shown that the attacker can determine which branch was executed, and that information also implicitly contains information about the value of the password – we know whether the guess was correct or not.

The requirement for the attack to work is that the two branches of the conditional have different memory access patterns that the attacker can distinguish. This is similar to the case of Nemesis, where the instructions in the different branches need to have a different execution length for the attack to work. For a more detailed comparison of our attack and Nemesis, see section 3.7.

These requirements cover a wide range of enclaves, but it is important to note that not every such enclave was secure before the introduction of our attack, there are different ways of detecting secret-dependent control flows and even secret-dependent data flows. Different execution lenghts of instructions is what Nemesis exploits, but a simple end-to-end timing can reveal information about conditional branches where the total execution length of the branches is different.

If an enclave writes data into an unprotected memory location, that value can be recovered by an unprivileged attacker. Even if the enclave clears the value before exiting, an attacker with interrupt capabilities can interrupt the execution of the enclave before the cleanup happens and can read the value.

Memory reads from unprotected memory addresses can also be detected. An attacker can set up their enclave with boundaries that place the target address in its protected region. When the victim enclave tries to read the value of the target address, it will cause a memory violation that can be detected.

In all of our examples we will focus on enclaves that operate on memory addresses that belong to their protected region and show that even with this condition the isolation guarantees can be broken.

Detecting secret-dependent data flows requires the ability to differentiate between memory accesses to different addresses in the memory. In the next section, we show how the implementation details of the Sancus TEE in some cases enables this for addresses that lie in the protected region of the victim enclave.

3.4 Attacking the Sancus platform

As described in section 2.1.2, the Sancus TEE [34] is implemented on the openMSP430 [17] platform. This architecture has certain features that make the attack described above very effective. There is no caching system, so every memory read and write is served from the main memory. Each memory request – both by the processor and via a DMA request – takes precisely one clock cycle to complete.

The openMSP430 platform uses a single global address space for all executing programs, but importantly, this address space is divided into three distinct partitions; program memory, data memory, and peripheral space (memory-mapped I/O). These partitions are served by three separate memory interfaces. As a consequence, memory requests to these three memory regions can be served concurrently.

If there are multiple memory requests in the same clock cycle for one of the three memory partitions, the serving priority is decided based on a dedicated priority signal. In the current implementation of Sancus, this signal is hardwired to give the CPU priority to avoid denial-of-service attacks carried out by untrusted peripherals [40].

This means that if both the processor and a DMA device want to read or write the same memory partition, the DMA request will be delayed until the processor is done accessing the resource. This can last multiple cycles if the CPU issues multiple consecutive memory requests. For example, if the DMA device and the CPU issue a memory request in the same cycle, and the CPU issues two more memory requests in the following two cycles (so three consecutive accesses by the CPU in total), the DMA request will be completed only after the fourth cycle.

For a more detailed discussion on the priority signal and the consequences of reversing it, see section 4.3.

The modified attacker model of Sancus introduced earlier allows untrusted peripherals to be connected to the platform. These peripheral devices have access to the same clock as the CPU and the memory module, so timings can be measured with single cycle accuracy. The untrusted peripherals are allowed to issue DMA requests to unprotected memory regions [40]. There are no other components in the system that can issue memory requests.

Considering the very deterministic nature of the openMSP430 architecture; namely the single cycle memory requests, the lack of caching, and the lack of other components that can influence how memory requests are served, we can conclude that a DMA attacker can precisely detect in which cycles the victim enclave was accessing which memory partition.

In the remainder of the thesis, we will show memory accesses to the three memory partitions separately on the timing diagrams, because the attacker has the ability to detect them separately.

Detecting secret-dependent data flows Using the implementation details of Sancus – specifically the different memory partitions – an attacker can detect some secret-dependent data flows in addition to secret-dependent control flows. Listing 3.2 contains a program that writes a value into its protected data section.

If the user guessed the key correctly, the value of a sensor is written to a protected memory location. If the guess was incorrect, a random value is read from data memory and that is written to the target address. Because the dynamic value is contained in peripheral space, while the random is stored in data memory, the DMA attacker can

again detect a difference between the two executions, as demonstrated by figure 3.4. Both the data memory and the peripheral bus accesses are different in the 8th clock cycle.

```
    mov &password, r7
    cmp r7, r6
    jz 1f
    mov &random_data, &target
    jmp 2f
1:  mov &periph_data, &target
    jmp 2f
2:  nop
```

Listing 3.2: A secret-dependent data flow

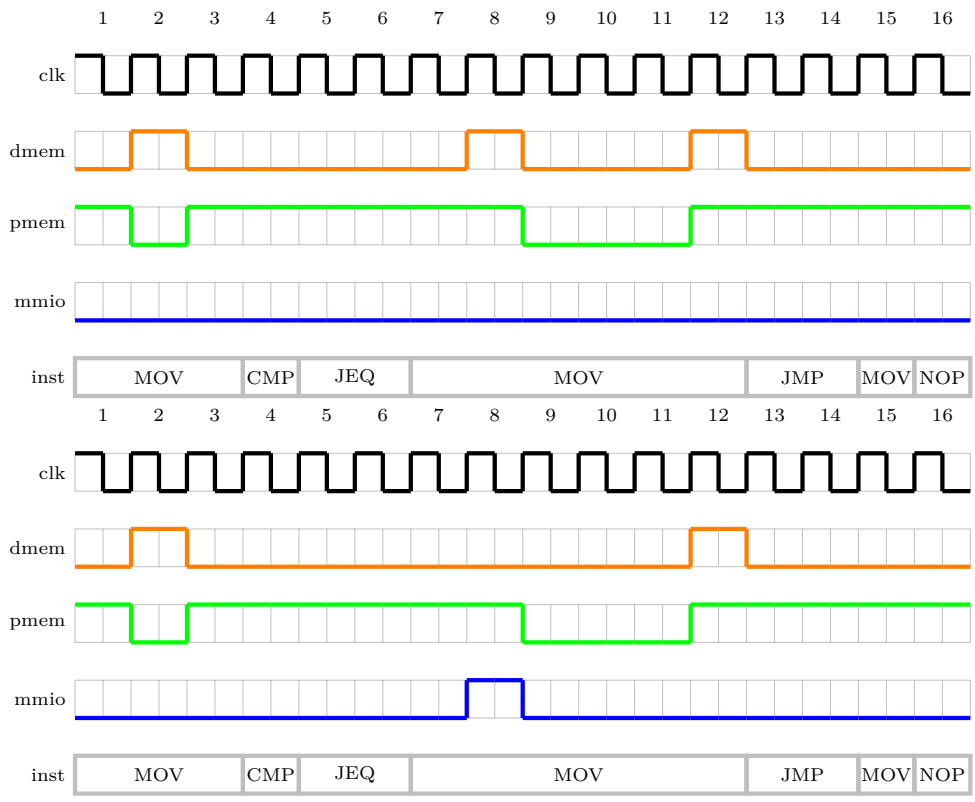


Figure 3.4: Timing diagrams for an incorrect and a correct guess in the secret-dependent data flow

3.5 Attacker peripheral

In the previous sections, we have shown the potential of the attack using timing diagrams extracted from the Sancus simulator. In this section, we explain our implementation of a programmable malicious peripheral that is capable of extracting the same memory access pattern traces from enclaves.

Our peripheral is implemented in about 100 lines of Verilog and we have tested its capabilities in both the Sancus simulator and flashed onto the Xess XuLA2 FPGA. The peripheral can be controlled from software through memory-mapped I/O registers. The peripheral can record and store the memory access patterns of the CPU, based on the timing of issued DMA requests.

Implementation details The peripheral's design is very straightforward: it exposes three registers to the memory-mapped I/O address space. These are the `countdown`, `address`, and `trace` registers.

The `address` register is modified by the attacker. This is the address to which the DMA requests will be issued. It should be an unprotected memory address in one of the three memory partitions.

The `countdown` register is also set by the attacker. Its default value is `0xFFFF`, this indicates that the counter is switched off. If `countdown` is set to any other value, this number will be decreased in every subsequent clock cycle. The peripheral starts issuing the DMA requests and capturing the memory access pattern after the countdown reaches zero.

Once the capturing starts, in each cycle the peripheral issues a read DMA request to the defined address. In every cycle, the peripheral checks whether the request issued in the previous cycle has completed (meaning no memory access by the CPU) or not (indicating resource contention). The value of the DMA request (the value read from memory) is discarded.

The captured memory access pattern is stored in the 16-bit `trace` register. The capturing period lasts 16 cycles, after which `countdown` is reset to `0xFFFF`. The value of the `trace` register corresponds to the memory access pattern of the CPU for the given memory partition during the capturing period. Each bit in the register represents one clock cycle in the capturing period, the most significant bit the first cycle, the least significant bit the last cycle. If the value of the given bit is 0, that means the processor did not access the given memory partition in the given cycle. The value 1 indicates a memory access.

The peripheral shares a clock signal with the processor and the memory unit, which makes the implementation straightforward. Once the peripheral is connected to the system, the attacker can execute an arbitrary number of memory access pattern captures from unprotected code using the memory-mapped registers of the peripheral.

In our implementation, the capturing period is fixed to 16 cycles to simplify the implementation and to avoid generating large amounts of DMA requests which could degrade the performance of the system and make detection easier.

For a discussion on the limitations of the peripheral, see section 3.8.

Executing the attack Using the peripheral to show the breaking of full abstraction and to extract the password from the enclave introduced in listing 3.1 requires only

minimal code.

First the peripheral is set up by pointing the `address` register to an unprotected data memory address. Then the correct timer value is determined that will start the capture when the CPU starts executing the first instruction of the enclave after jumping to its entry point.

Then a loop is set up which iterates through every possible byte value, writes it to the `r6` register (which stores the password guess), writes the determined delay value to the `countdown` register, then jumps to the entry point of the enclave.

After the enclave finishes executing, the attacker examines the value of the `trace` register. If the 8th bit is 0, that indicates that the correct password has been found and the branch has been taken with the `mov` instruction. If the 8th bit is 1, the loop continues with the next possible byte value for the password.

For reference, the content of the `trace` register after a correct password guess is the binary value `0100000001000000`, and after an incorrect guess `0100000101000000`. When comparing this to the data memory accesses in figure 3.3, we can see that they match perfectly, and the values only differ in their 8th bit.

3.6 Instruction analysis

We have already shown multiple examples of enclaves that featured a secret-dependent control or data flow with different memory access traces for its branches.

To understand what causes the differences in the memory accesses, we have conducted a systematic overview of the factors that influence what memory accesses are issued by the CPU when executing different instructions.

Appendix A contains an extensive (but not exhaustive) mapping of instructions to memory access patterns, while this chapter will explain the theory behind the different patterns.

This analysis is especially useful for constructing defenses against this attack or analysis tools that can detect vulnerable enclaves. For a further discussion on defenses, see chapter 4. It can also be used by attackers to reconstruct instruction sequences as accurately as possible from a captured memory trace with no knowledge of the enclave's source code.

To plot the different instruction traces, the attacker peripheral could be utilized with three executions of the enclave containing the instruction of interest, each time capturing the memory trace of one of the three partitions of memory, but the Sancus simulator can also be used to plot the memory accesses more conveniently.

3.6.1 Instruction fetching

The openMSP430 architecture features a simple two-stage pipeline for executing instructions. As mentioned in section 2.1.2, the CPU consists of two modules, the frontend and the execution unit. The frontend fetches and decodes the instructions, it only accesses the program memory to load the instructions and other constants for the instructions. The execution unit can access all three memory partitions.

The core always fetches the next instruction in the last cycle of the execution of the current one. This means that during the execution of any instruction on the platform, the program memory will always be accessed in the last cycle to fetch the next instruction.

If the loaded instruction contains constants, such as memory addresses or immediate values, these are loaded separately in the first cycles of the execution. A 3-byte instruction (that contains two constants) will have program memory accesses in – at least – its first two cycles of execution, and in the last to fetch the next instruction.

3.6.2 Different instructions

In the attack description, we have already shown that some instructions' traces differ. We used the example of the `add` and `mov` instructions with a data memory address as a target and a constant value as the first parameter. In figure 3.5 we show the complete timing diagrams of these instructions for all three memory partitions. We can see that the data memory access patterns are different for these instructions in the second cycle.

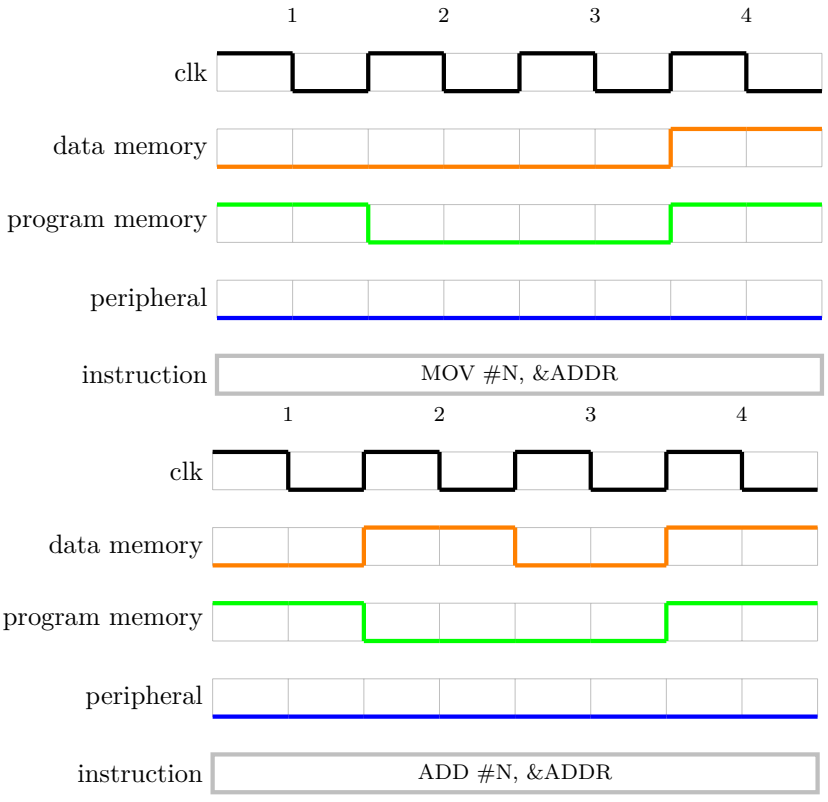


FIGURE 3.5: Memory traces for `add` and `mov`

In figure 3.6 the memory access plots of the `jmp` and the `jne` instructions are shown. These instructions both take one parameter that points to a program memory location, the target of the jump. The traces of these instructions are equivalent, so a DMA attacker could not differentiate between them.

As a conclusion, some instructions' traces differ from each other, but there are also different instructions that produce the same memory access patterns.

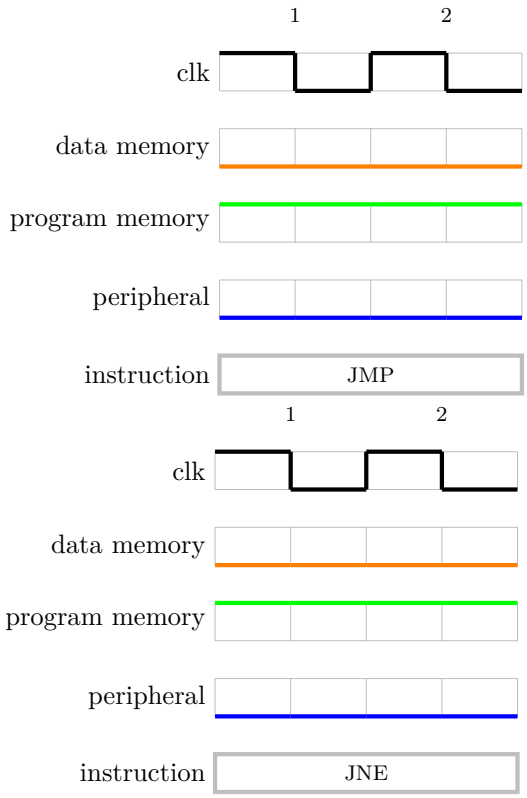


FIGURE 3.6: Memory traces for `jmp` and `jne`

	Example	Source	Destination
Register mode	r8	x	x
Indexed mode	42(r8)	x	x
Symbolic mode	ADDR	x	x
Absolute mode	&ADDR	x	x
Indirect register mode	@r8	x	
Indirect autoincrement	@r8+	x	
Immediate mode	#0x42	x	

TABLE 3.1: Addressing modes in the MSP430 architecture

3.6.3 Addressing modes

The openMSP430 architecture allows 7 different addressing modes for source operands and 4 for destination operands. These addressing modes are summarized in table 3.1.

The addressing modes used for a given instruction also influence the memory access pattern. In figure 3.7 we have the same `mov` instruction, but in the first case its source operand is a register, in the second case it takes an absolute (data) memory address

as its first parameter.

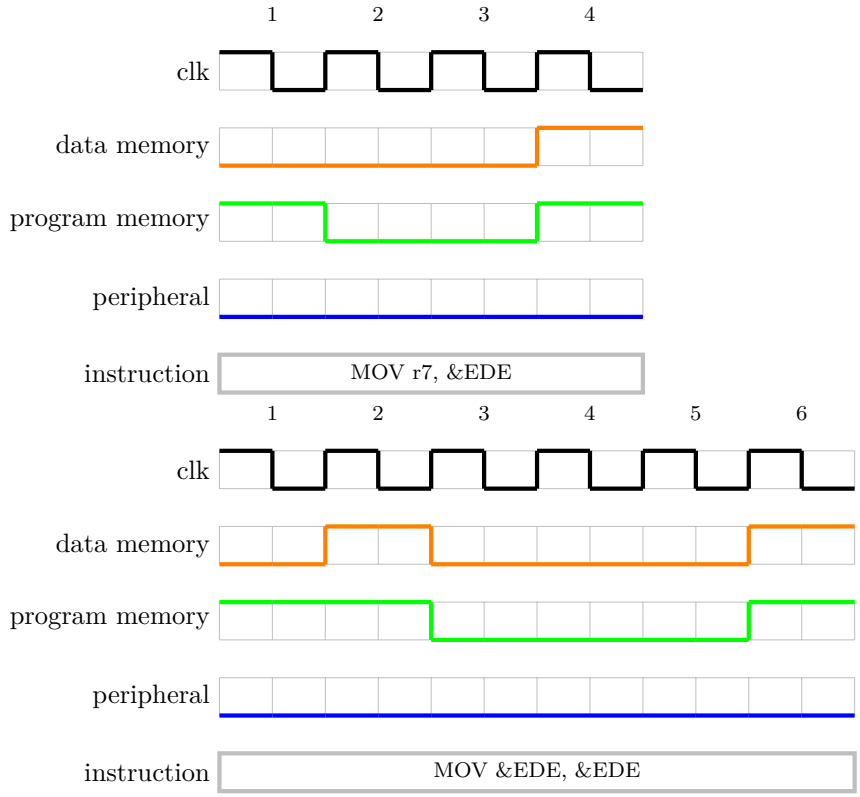


FIGURE 3.7: Memory traces for different source operands of the `mov` instruction

In general, register mode does not result in any memory accesses. Immediate mode loads a constant value from program memory. Indexed mode loads an offset value from program memory, then the target address is calculated by adding the offset to the register's value. For symbolic and absolute mode, first the target address is loaded from program memory. The subsequent memory access (also for indexed mode) depends on the location of the target address, it can be located in either of the three partitions of memory. The same is the case for indirect register and autoincrement modes, but here the target address is stored in the register, so there is no preliminary program memory access to load the address from the program memory.

3.6.4 The value of the operands

Some of the addressing modes refer to registers, others to memory locations. In the case of the latter, another factor we need to take into account is in which memory partition the provided address resides. In figure 3.8 we see the same `mov` instruction executed, in all cases taking an absolute memory address as its destination operand.

The only difference is that in the first case, the address points to data memory, in the second case, the supplied address points to the peripheral region, and in the third

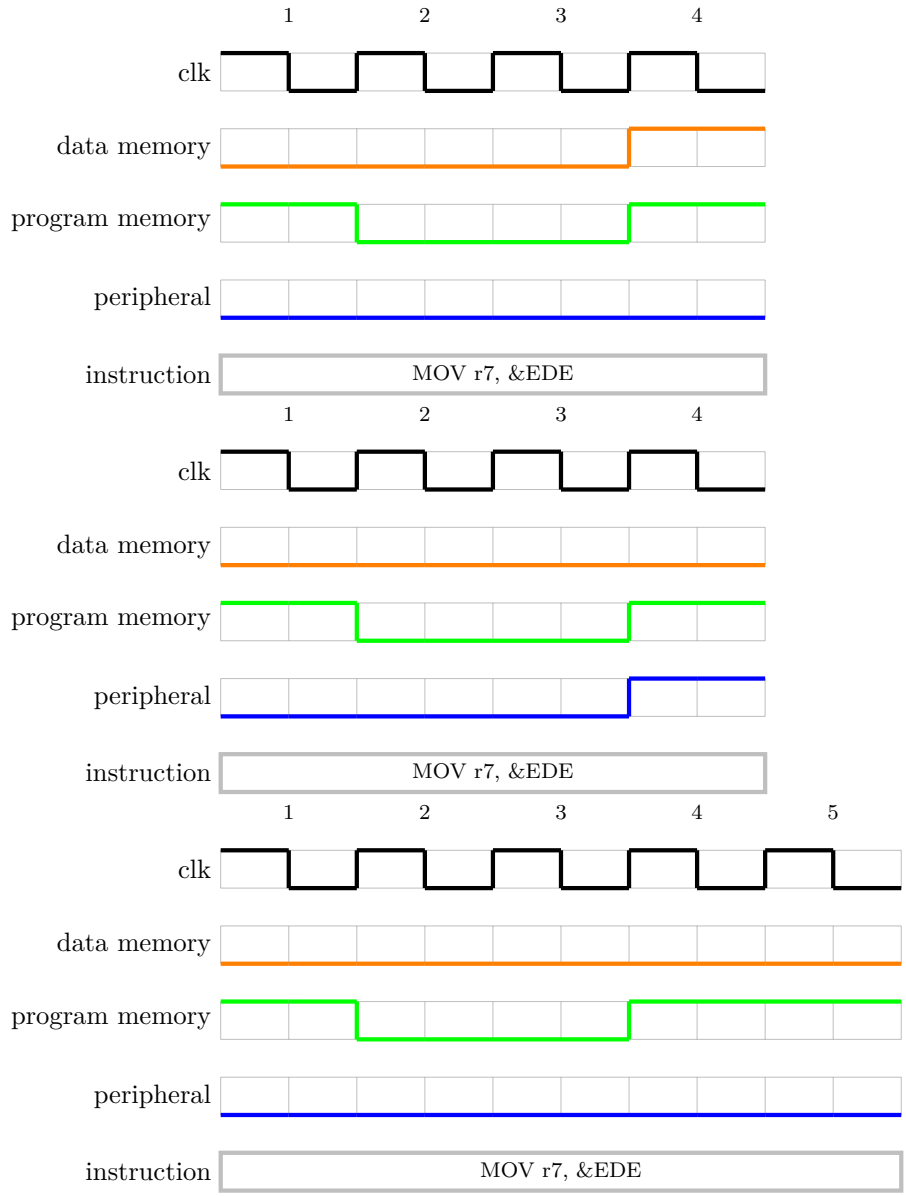


Figure 3.8: Memory traces for `mov` with different absolute addresses as its target operand

case, it points to program memory. The differentiating factor is the access in the 4th cycle: in the first case, the data memory is being accessed, in the second case, the peripheral bus is used, and in the third, the program memory is being accessed.

This distinction was previously used to differentiate between the secret-dependent data flows.

It is also interesting to note that the instruction writing to program memory takes an extra cycle to complete. This is presumably to accommodate the instruction fetch and the write to program memory in the same execution. This way overwriting the next instruction can also be supported, and the fresh version will be fetched in the next cycle.

Even in the case when the destination is a register, we need to make a distinction between the values. If the target register is the program counter (the `r0` register), the platform makes the execution one cycle longer, because the fetching of the next instruction will be delayed, as the value of the PC is not known until the current instruction finishes executing. This can be seen in figure 3.9.

This is documented in the openMSP430 specification [17], jump instructions always take 2 cycles to execute, and writing to the program counter is considered a jump.

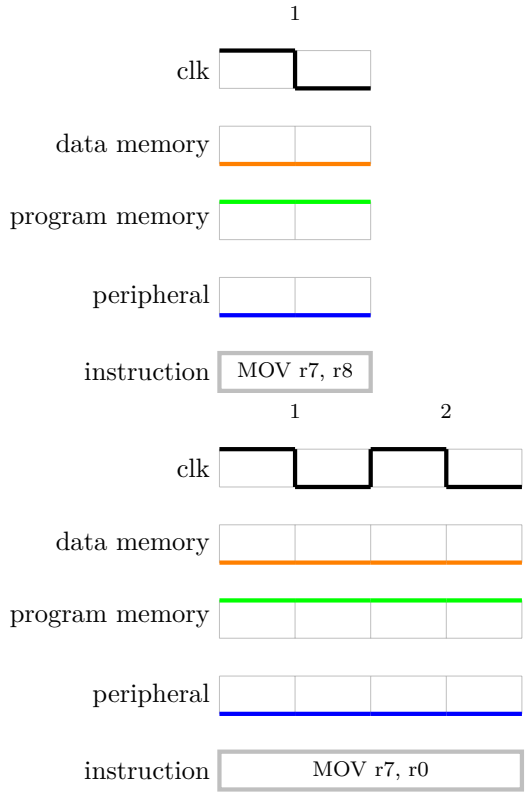


FIGURE 3.9: Memory traces for `mov` with different registers as its target operand

3.6.5 The influence of the previous instruction

A final angle to be considered is that the memory trace of a single instruction might not be entirely self-contained. As explained earlier, the instruction fetching of an instruction happens during the last execution cycle of the previous instruction. If the first instruction tries to access the program memory in its last cycle, that also creates

a resource contention for the program memory between the frontend that handles the instruction fetching and the execution unit that tries to access program memory for the execution of the current instruction.

This is solved in some cases by adding an extra cycle to instructions that modify program memory as shown in the previous section, but this is not always the case. Figure 3.10 compares two traces for a sequence of two instructions. The only difference between the two traces is that in the first case, the first instruction (swap bytes) accesses data memory in its last cycle, while in the second case it accesses the program memory. Even though the second instruction and its operands are the same in both executions, in the first case it issues an extra program memory request in the 5th cycle.

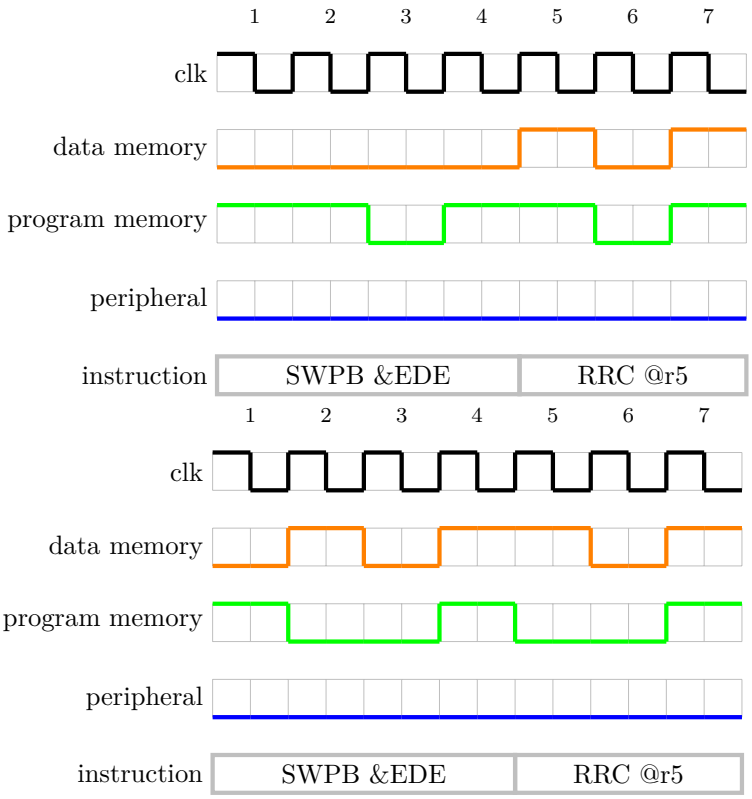


Figure 3.10: Memory traces with different addresses as the parameter of SWPB

3.7 Comparison with interrupt-based attacks

Section 2.5 introduced Nemesis, the state-of-the-art side-channel attack on Sancus.

Nemesis can measure the execution length of individual instructions in a secure enclave. This means that a Nemesis attacker not only knows how many instructions were executed in a given timeframe (interrupt counting), but can also narrow down the executed instructions based on their execution length (interrupt latency).

To see which attack provides more severe leakage, we will compare different enclaves, and see whether a Nemesis attacker and a DMA attacker are capable of reconstructing secret-dependent control flows.

When comparing the two attacks, we only consider detecting secret-dependent control flows: Nemesis can only differentiate between secret-dependent data flows if the parameter values change the execution length of the instruction. On Sancus, this can happen when writing to a program memory address as shown in section 3.6, on Intel SGX the execution length can depend for example on whether or not the target address is cached.

Attacker models The attacker models of the two attacks are also different. Nemesis requires an attacker that is capable of running code on the system and issuing interrupt requests with their own defined interrupt handler routine.

The DMA attacker requires physical access to the module to be able to connect the malicious peripheral. In our proof-of-concept, the attacker then also needs to run code on the system to capture and store memory traces, but the attacker peripheral could be modified in a way where it requires no software running on the code and captures memory traces in its internal memory. For further discussion on the attacker requirements, see section 3.8.

3.7.1 Stronger DMA leakage

Taking the running example from section 3.2 we see that the differentiating instructions were the `add` and the `mov`. We can see in both figure 3.2 and figure 3.3 that both of these instructions take 4 clock cycles to execute. This means that for a Nemesis attacker, both branches of this secret-dependent control flow have the same interrupt counts and interrupt latencies. Two instances of this enclave with different password values are thus contextually equivalent for a Nemesis attacker.

On the other hand, we have seen from the previous sections that our DMA attacker can reconstruct the secret-dependent control flow in the enclave.

3.7.2 Stronger interrupt leakage

It is not always the case that the DMA-based attack provides a stronger leakage than Nemesis. Listings 3.3 and 3.4 contain two elementary enclaves. It is clear by looking at the source code that the two enclaves are different, and a Nemesis attacker can also easily differentiate them by counting the number of interrupts during the execution of the enclave. The first enclave can be interrupted twice, while the second enclave can be interrupted three times, after each instruction.

For our DMA attacker, however, these modules are contextually equivalent, meaning they produce the same output (including the microarchitectural leakage) when given the same input.

Looking at the collected timing diagrams in figure 3.11, we can see that both enclaves access the program memory for the entire duration of the execution, but no other partitions.

```
;
mov #0x42 , r6
mov #0x42 , r6
```

Listing 3.3: Enclave 1

```
mov #0x42 , r6
mov r6 , r5
mov r5 , r6
```

Listing 3.4: Enclave 2

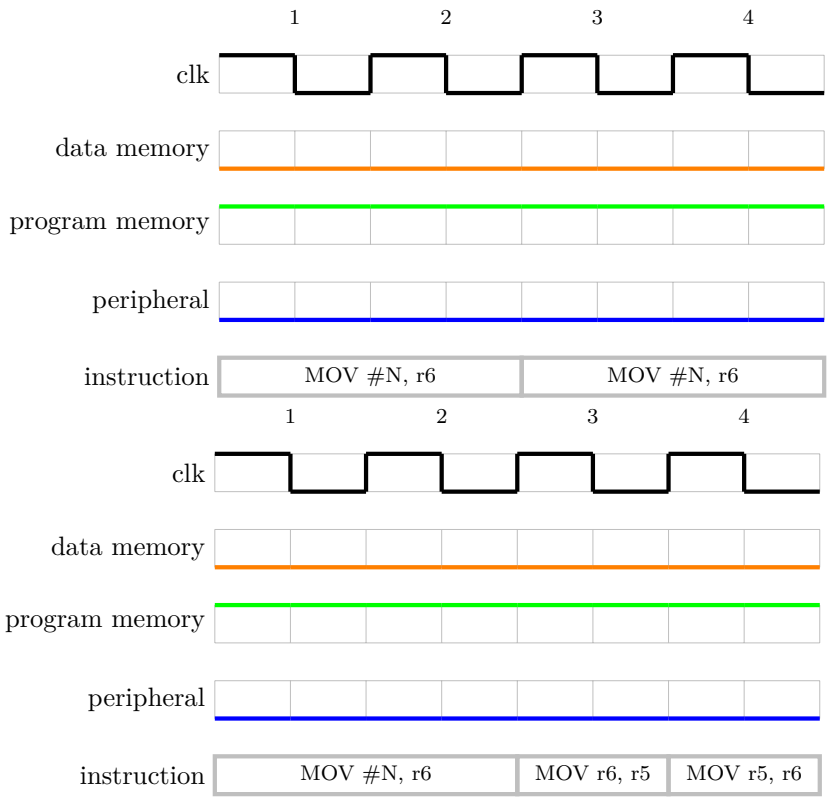


Figure 3.11: Timing diagram for the two enclaves

In conclusion, Nemesis and the DMA attack have different leakages when it comes to detecting secret-dependent control flows. Nemesis works at a granularity of instructions, and captures cycle-accurate measurements on the execution length of individual instructions, while the DMA attacker captures cycle-accurate memory access traces, but has no direct knowledge of the granularity of instructions. This means that the instruction boundaries can not always be reconstructed unambiguously from the captured memory access traces.

3.7.3 A combined attacker framework

Based on the previous examples, we can conclude that Nemesis can always identify the number of instructions and their execution lengths in a given execution of an enclave. The DMA attack can further differentiate between instructions of the same execution length, and can even identify which memory partition their parameters belong to.

It is clear that by combining these attacks, an attacker framework can be created that is capable of identifying secret-dependent control and data flows in enclaves vulnerable to a Nemesis attacker in addition to the enclaves vulnerable to a DMA attacker.

Extending Sancus-Step On Sancus, the Nemesis attack has been implemented in the Sancus-Step framework, which enables an easy-to-use software-only attack against enclaves. Sancus-Step uses the principle of Nemesis, it interrupts a victim enclave in the first cycle of each instruction's execution and reports the number of clock cycles the interrupt handler was delayed by, which corresponds to the execution length of the interrupted instruction.

We have extended Sancus-Step with our DMA attack. Provided that our attacker peripheral is connected to the system, we allow Sancus-Step to be configured to also capture a memory trace while the enclave's instruction is executing. This means that the attacker not only receives the execution length of the instruction but also a memory trace of a given memory partition captured during the execution of that instruction.

Convenience functions are provided in the Sancus-Step framework to configure the `address` and `countdown` registers of the peripheral and to retrieve the value of the `trace` register. In addition, the `countdown` value is configured to account for the delays caused by the Sancus-Step framework and the secure interrupt handling code.

All of these changes add up to only about 30 lines of extra C and assembly code in the Sancus-Step implementation.

A practical example For this demonstration, we will use the enclave defined in listing 3.5. This code is based on TI's bootstrap loader code presented in the Nemesis paper [49]. This enclave is defined in C code using inline assembly to simplify the attack. An enclave defined in high-level C code could also be attacked if the generated assembly code contained a vulnerable secret-dependent control flow.

Our enclave has a single defined entry point. The attacker will call this entry point with their guess for the secret key passed as a parameter. Before calling the entry point, the attacker sets up the Sancus-Step framework by calling the `ss_start()` function. The attacker also defines the interrupt handler routine that will be executed after the execution of every instruction in the victim enclave. The interrupt handler is shown in listing 3.6.

In the interrupt handler, the attacker performs interrupt counting. The DMA peripheral is also activated by setting its countdown to the value 0 – which means that the memory access pattern capturing should start immediately after the victim enclave starts executing.

For this attack, we will use a program memory address to initialize the `address` register of the peripheral. This program memory address needs to lie in an unprotected region of the program memory, we will use the pointer to the interrupt handler routine as the address.

Since we have access to the source code, we can select an instruction pair in the conditional branch that produces different memory traces, in this case these instructions are the `jmp` and `nop` in the other branch. The index of this instruction

pair (which is the number of instructions that execute before it in the enclave) is stored in the variable `INSTRUCTION_NUMBER_JMP`.

When this index is reached, the captured DMA trace is examined: a specific bit is selected with the help of a mask, which corresponds to examining the memory access in a single cycle. Based on this value, the attacker can differentiate between the `jmp` and the `nop` instructions in the two branches, which will tell the attacker whether the key was guessed or not.

```
void SM_ENTRY(foo) test(char key) {
    char * p = &key;
    __asm__ __volatile__(
        "mov %0, r6\n\t"
        "mov #0x42, r7\n\t"
        "cmp.b @r6+, r7\n\t"
        "jz 1f\n\t"
        "bis #0x1, r7\n\t"
        "jmp 2f\n\t"
        "1: nop\n\t"
        "nop\n\t"
        "nop\n\t"
        "2: nop\n\t"
        :
        :"m"(p)
        :"r6", "r7"
    );
}
```

Listing 3.5: The target enclave

```
void irqHandler(void) {
    if (instruction_counter == INSTRUCTION_NUMBER_JMP) {
        if (DMA_TRACE & TRACE_MASK) {
            pr_info("Key was not guessed!");
        } else {
            pr_info("Key was guessed!");
        }
    }
    ++instruction_counter;
    __ss_set_dma_attacker_delay(0);
    uint16_t *pmem_addr = (uint16_t*) irqHandler;
    DMA_ADDR = pmem_addr;
}
```

Listing 3.6: The interrupt handler routine

The captured traces by the peripheral are `1111000000000000` for the `jmp` instruction and `1110000000000000` for the single-cycle `nop`. We can see that these traces differ in the 4th bit, this allows an attacker to infer which branch was executed. This is the bit that is selected by the `TRACE_MASK`.

A possible new class of attacks We have shown that some enclaves are vulnerable against Nemesis attackers, and some against DMA attackers. It is possible that two enclaves can be constructed that cannot be differentiated by either a Nemesis nor a DMA attacker, but are vulnerable against a combined Nemesis-DMA attack. These enclaves could also be attacked using our extended Sancus-Step framework.

The requirements are the following: the conditional branches feature instructions of the same length, and when uninterrupted, produce the same memory trace.

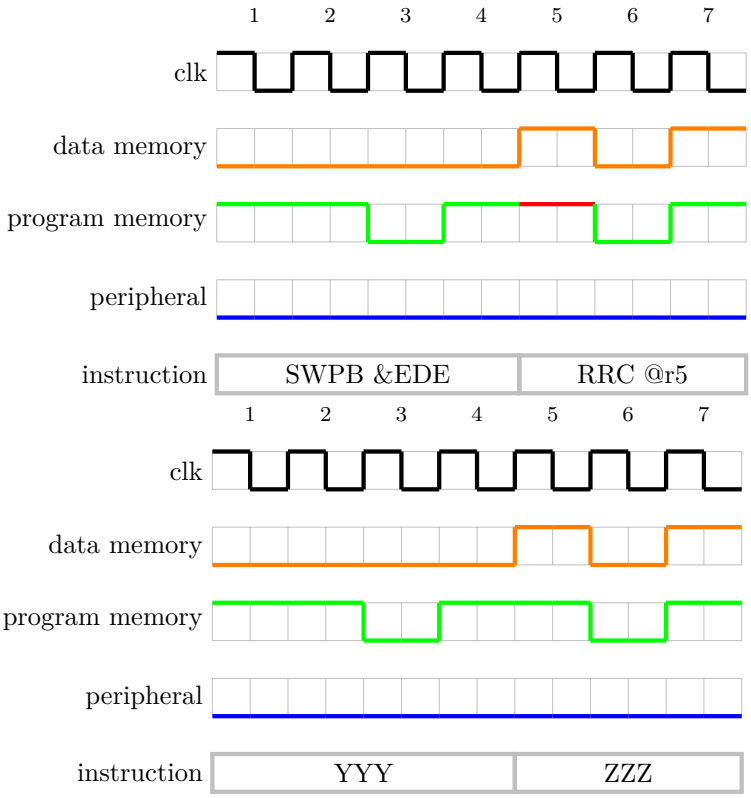


FIGURE 3.12: Hypothetical enclave example

In section 3.6 we have shown instructions that can influence the trace of the subsequent instructions. Figure 3.12 shows a theoretical example for two enclaves that can only be differentiated by the combined attack. In the first enclave, the program memory access in the 5th clock cycle (highlighted in red) is caused by the `swpb` instruction as shown in the previous section. In the second enclave, we have two hypothetical instructions executing that produce the same memory access pattern, but the program memory access in the 5th cycle is caused by the `zzz` instruction itself.

The only property of the Nemesis attack we use is the ability to precisely interrupt the enclave to separate the first instruction's effect from the second instruction. If the interrupt handler code runs between the two instructions, the first intrusction will not have the same influence on the second one as it would have with no interruption. This means that after resuming the enclave following the execution of the first instruction

and the interrupt handler routine, the program memory access in the first cycle of the second instruction will only happen if we run the second enclave, thus allowing an attacker to differentiate between the two enclaves.

Further instruction analysis needs to be carried out to determine whether such a set of instructions exists on Sancus.

3.8 Attack limitations

Our attack and the experimental setup have a number of limitations. We will discuss these, separating the limitations which are only a result of our elementary attack setup, and those that are fundamental to this side-channel.

3.8.1 Limitations of the experimental setup

The peripheral has been designed to be powerful enough to demonstrate a real-life attack, but it has some shortcomings.

Register sizes Most notably the length and timing of the capture window are limited. The countdown register is 16 bits wide, which means that the capture cannot start more than 65534 cycles after the timer has been set (the highest possible valid value is `0xFFFE` because `0xFFFF` is reserved as a "no capture" constant).

The register that stores the captured trace is also 16 bits wide, so only 16 consecutive cycles of memory activity can be recorded before the timer is reset and the trace needs to be copied before starting a new capture.

These limitations would be fairly easy to overcome by increasing the size of the registers or by modifying the peripheral to handle its arguments in the `countdown` and `trace` registers as pointers to unprotected data memory, read the countdown value from the pointed address and write the captured trace into an array in memory (that is potentially much longer than 16 bits wide). In this case, the number of cycles to record would also need to be passed to the peripheral as a parameter.

The register size is less of an issue when using the attack with the modified Sancus-Step framework, because here only the trace of a single instruction needs to be captured, which is never longer than 7 cycles long. In this scenario, the peripheral can be set up right before executing the instruction, so a longer countdown value is also unnecessary.

Control from software Currently, the attacker needs to both connect the peripheral to the system and then control it from software running on the system. In a specialized attack, the execution rights could be omitted from the attacker model by a peripheral that is capable of capturing and storing a large amount of memory traces in its internal memory, and that starts capturing as soon as it is connected or at a press of a button.

Code knowledge In our attacks so far we used knowledge of the source code to quickly identify the weak points of enclaves that make them vulnerable against our attack. These weak points also define which memory partition's accesses we need to capture and in which cycles.

If the attacker has no knowledge of the source code, they can still capture traces (in multiple runs) for the entire execution of the enclave, for all three memory partitions. In a later analysis stage when these traces are compared with each other and the inputs provided to the enclave, the enclaves that are vulnerable against the attack can still be identified.

3.8.2 Fundamental limitations

Physical access Our attack requires the ability to issue DMA requests in the system. This is most easily accomplished by connecting a malicious peripheral as we have demonstrated above. This is in scope of the attacker model of the version of Sancus that we are working with [40], but it still requires physical access to the device, so it is arguably harder to execute than a software-only attack.

A possible alternative to plugging in our peripheral is taking over an already connected DMA-capable peripheral. If the peripheral can be freely controlled by software or its firmware contains a vulnerability that allows modifying its behavior, the attacker can use it to issue their DMA requests [15]. For our attack we also need to be able to measure the timing of the DMA request, which might pose an additional challenge with this approach.

An emerging trend in computing is combining traditional CPUs with FPGAs to create homogeneous platforms where the FPGA can be used as a reprogrammable accelerator to the CPU. These arrangements are also appearing in cloud providers' offerings. [5, 4]

These homogeneous platforms often offer DMA capabilities to the FPGA for the shared memory module. In this case, it would be sufficient for an attacker to reflash the FPGA with the malicious peripheral to execute the attack.

Multiple runs The openMSP430 platform has a single DMA interface to the memory. This means that in a single cycle only one DMA request can be served. We have shown that there are three different partitions of memory that can be accessed by the CPU on Sancus. Because we only have a single DMA request per cycle, we can only check the access of one of these partitions. In other words, to capture the complete memory traces, we need to execute the enclave three times with the same inputs, each time targeting a different memory partition in our attacker peripheral to be able to compile the complete memory access trace.

This might seem like a big limitation, but it is also important to point out that in many cases (in fact, in all of the examples presented so far) tracing accesses to one memory region is enough to tell the difference between two executions. This of course requires prior knowledge about the possible memory access patterns of the victim enclave.

Multiple DMA devices We also need to consider noise in the system. The CPU's memory accesses are completely deterministic in Sancus, but if there are multiple peripherals with DMA capabilities, those could interfere with collecting the traces by causing additional resource contention. Usually, if a system supports multiple DMA-capable peripherals, it supplies a DMA Controller that is responsible for arbitration of DMA requests and possibly caching the results.

Although the current Sancus architecture does not feature a DMA Controller, from our attack's scope, it is important that the DMA Controller also lies outside of the TCB. This means that attackers are allowed to tamper with it, and as a result, we could modify it in a way where it acts in the same way as our attacker peripheral now, and simply ignores any incoming DMA requests from other peripherals.

Ambiguity While the number of theoretically possible memory traces generated by a single instruction is in the hundreds due to all the possible variables outlined in the previous section, there are still many overlaps. This means that a particular collected memory trace can be the result of different sequences of instructions. Reconstructing the instruction sequence from a captured memory trace is thus not always possible.

In the other direction, on Sancus the memory trace resulting from executing a sequence of instructions with the same parameters is always deterministic, meaning that executing the same program with the same inputs will always result in the same memory trace. This is due to the elementary design of the architecture with no optimization features such as caching or instruction reordering.

Another limitation is that memory reads and writes cannot be differentiated from each other in isolation – the memory bus can be constrained or not, but it provides no information about which direction it was transmitting data.

However, based on our instruction analysis, if the instruction boundaries can be determined, memory reads generally occur in the first half of the execution of a given instruction, while memory writes happen in the last few cycles of instructions. This also conforms to logic, since usually instructions first load a value from memory, then possibly transform it, and finally write the result back to memory.

3.9 Conclusions

This chapter introduced a novel side-channel attack that exploits the timing of DMA requests targeting unprotected memory to infer the memory access patterns of an executing enclave.

This information leakage was shown to enable extraction of secrets from an enclave and a malicious peripheral was presented on Sancus and used to carry out practical attacks.

An instruction analysis was carried out to identify the factors that influence the memory access pattern of a given instruction. The leakage of the attack was compared with the leakage of Nemesis, the only other side-channel attack on the Sancus platform. While Nemesis enables an attacker to collect the execution latencies of all instructions during the execution of an enclave, the DMA attack leaks an enclave's cycle-accurate memory accesses targeting all three memory partitions of openMSP430 (the underlying architecture of Sancus) separately.

Finally, some limitations have been identified regarding the attack, most significantly requiring physical access to connect the malicious peripheral to the system, or the ability to take over an existing peripheral with very specific requirements.

Chapter 4

Defenses

After having discussed the details of the side-channel leakage and the resulting attack, we turn our attention to possible defenses.

Because this attack uses microarchitectural implementation details to leak information about features in the software, defenses at both the hardware and the software level can be considered. In this chapter, we introduce different ideas and briefly outline their upsides and downsides, but this should not be considered a comprehensive list of possible defenses. For example, we do not consider probabilistic defenses here, which could work by introducing a random delay before serving DMA requests. With this approach, the implementation needs to be resilient against attacks that execute the enclave multiple times and perform statistical analysis on the results.

First, we note that in contrast to some other DMA-based attacks [25, 30], our attack does not bypass any protection mechanisms such as an IOMMU, and it does not necessarily generate a considerable overhead in memory traffic such as Rowhammer-style attacks [21, 43]. Indeed, we have shown examples where a single cycle-precise DMA request to unprotected memory is enough to differentiate between different executions of an enclave.

Consequently, simply patching an existing defense or employing existing monitoring approaches that are effective against attacks such as Rowhammer do not work out of the box against our attack. The DMA request(s) used in our attack are indistinguishable from legitimate memory usage.

The root cause of the attack is the resource contention on the memory bus. If DMA requests would be served through a dedicated interface of the main memory that is capable of serving requests in parallel to the conventional memory bus, DMA attackers would not be able to gather any information about the CPU's execution. The downside of this attack is that it requires fundamental changes to the design of the platform and the memory unit, and requires solving the challenge of handling parallel accesses to the same address without introducing a different side-channel.

Disabling DMA Disallowing attackers from connecting peripherals is a viable defense, but it reduces the functionality offered by the system, and previous chapters have shown that remote attacks using RDMA or a heterogeneous CPU-FPGA platform remain a concern with this configuration.

An obvious defense is disallowing DMA requests, but this would severely degrade the performance of the system as the processor would need to handle data transfers between peripherals and the memory (and all of the overhead this entails, such as polling or interrupt handling). In general, all of the defenses need to strike a balance between security, functionality, and performance.

A more reasonable approach is to only disable DMA while an enclave is executing on the processor. This allows the performance increase of DMA to still apply during unprotected execution.

A refinement to this solution is implementing new instructions to enable and disable DMA from code. This instruction pair could be used to protect the enclave when executing secret-dependent branches, but allow DMA during the remainder of the execution. This instruction could even be inserted automatically by the compiler if it is capable of detecting secret-dependent flows.

Other defenses At its core, our attack works on enclaves that contain a secret-dependent control or data flow, where executing the two branches produces a different memory access pattern. If the enclave does not contain any secret-dependent conditionals, our attack also cannot leak secrets. So-called constant-time programs feature no secret-dependent control or data flows, these programs are not vulnerable to our attack. Intel SGX recommends developers to implement their code as constant-time, as the TEE does not protect against side-channel attacks [20].

To prevent leaking information through secret-dependent conditionals, we either need to eliminate the difference between the memory access patterns of branches or prevent the attacker from collecting traces of the patterns. The defenses in 4.1 and 4.2 aim to balance the branches of conditionals in an enclave to eliminate leaking information with memory accesses, while 4.3 works by preventing using DMA requests to collect the memory traces.

4.1 Code balancing

Application developers are already responsible for keeping the secrets of their applications hidden by not exposing them to untrusted code intentionally – or unintentionally by introducing a bug in their code that allows the leakage of private information.

With this approach, the programmer for our attack would be responsible for either eliminating secret-dependent control and data flows entirely, or balancing the branches in a way that results in the same memory access patterns for all possible execution paths. Listing 4.1 contains a naive way of padding the secret-dependent control flow in our running example. In this case, we chose to duplicate the instructions in the two branches with a dummy target that resides in the same memory partition as the password variable. This way, the enclave always produces the same memory access patterns regardless of which branch is taken, but this comes with increased execution time, larger code and data sections.

However, in contrast to defending against unintentional leakage or buffer overflow attacks which require attention only at well-defined points of the software (at the defined entry points and when handling untrusted data), balancing every secret-dependent flow in the application by hand is an enormous effort, it requires testing, analyzing and modifying compiled assembly code.

```
    mov &password , r7
    cmp r6 , r7
    jnz 1f
    mov #0x00 , &guesses
    add #0x01 , &dummy
    jmp 2f
1:  mov #0x00 , &dummy
    add #0x01 , &guesses
    jmp 2f
2:  mov #0x0 , r7
```

Listing 4.1: Naive padding of the example enclave

To not break the isolation properties, every piece of data that directly or indirectly depends on the value of an address in the enclave's protected memory region needs to be considered a secret.

Comparison to padding against Nemesis Defending against Nemesis attacks (with no hardware changes) also requires eliminating or balancing secret-dependent conditionals, but in that case, the balancing is considerably easier. The source of leakage in that case is the execution length of instructions, this is what needs to be equivalent for the two branches.

Execution lengths on the openMSP430 platform vary from 1 to 7 cycles and only depend on the type of the instruction and the types of the parameters (e.g. register, absolute memory address). This means that the execution length can easily be determined by examining any single instruction in isolation. The only exceptions that need to be considered as shown in our analysis in section 3.6, are writing to program memory and the program counter register.

To determine the memory access pattern produced by executing an instruction, we refer back to the instruction analysis we conducted and presented in section 3.6. This shows that the memory accesses depend not only on the instruction type and the parameter types, but (1) the values of the parameters and (2) the previous instruction(s). Dependency (1) means that the memory access pattern cannot necessarily be determined at compile time because the value of a parameter might be determined at runtime (for example derived from user input). And because of (2), we cannot examine an instruction in isolation, we also need to consider the instructions that are executed before it.

In theory, an instruction with execution length n can have 2^{3n} different memory access patterns: in each cycle of the execution, the three different memory partitions can be accessed. The possible values for n – as mentioned above – are 1 to 7. More research needs to be carried out based on our instruction analysis to identify whether, for a given instruction, there always exists a different instruction with the same memory access pattern.

In case an instruction exists that cannot be balanced using other instructions, the defender has two options: either blacklist this instruction from being used in secret-dependent conditional branches, or balance the other branch with the same instruction, using dummy parameters for it that do not represent meaningful data, but reside in the same memory partition as the parameter in the first branch.

In contrast to Nemesis defenses, it is possible to have balanced branches with a different number of instructions in the two branches. In other words, one instruction's memory access pattern can be balanced using multiple instructions or only one instruction's partial memory access pattern, as in the example shown in section 3.7. In this case, however, the balancing cannot defend against Nemesis attackers.

4.2 Hardened compiler

To make the developer's task easier and to provide more robust and verifiable security, the Sancus compiler could be modified to automatically perform the aforementioned branch padding.

Using our instruction analysis as a basis, the compiler could be extended to perform code analysis and automatic branch padding to produce programs that are not vulnerable against our attack.

Importantly, only secret-dependent flows need to be balanced, the rest of the code cannot leak sensitive data through this side-channel. In an automated approach this requires the developer to manually tag sensitive values in the code, or the compiler can automatically tag every value originating from a protected memory region or the network as sensitive. Automatically detecting every secret-dependent flow is challenging, for example secret-dependent data flows do not necessarily feature conditional jumps, accessing an address that can point to different locations can be sufficient to leak information from an enclave.

The downside of both this and the manual approach is that existing programs remain vulnerable until they are recompiled and redistributed. Extending the compiler also means more opportunity for bugs to appear in the code that could undermine the security guarantee and result in unbalanced code being produced.

In some cases, the padding could also introduce a significant slowdown of the programs: if the two branches effectively need to be unified into one branch, executing both sequences of instructions regardless of the condition, but using dummy addresses in one case (while paying attention that these dummy addresses do not result in different memory traces, as it could be the case with the different memory partitions on Sancus). Choosing these dummy addresses at compilation time is a challenge in itself.

The extra instructions and memory locations for the dummy addresses also increase the memory footprint of the program.

4.3 Giving the DMA priority

As highlighted in the previous chapter, the root cause of the current leakage is the priority of the memory requests by the CPU over the DMA requests of the peripheral. This priority causes a measurable delay in the DMA requests that can be used to infer the memory accesses of the application running on the processor.

If the priority was given to the peripheral instead of the processor, the DMA requests would all complete in a deterministic amount of time that is independent of the memory accesses issued by the executing enclave on the processor. This way, no information could be inferred about the state of the enclave from the timing of the DMA requests.

The implementation of this defense needs to be considered very carefully. When a DMA request arrives while an instruction is currently executing, the execution of the instruction needs to be halted in a way that does not leak any information to the attacker. For example, if the instruction's execution is first finished, after which the processor is halted and the DMA request is served, the delay depends on the execution length of the instruction. This is the same leakage as that of the Nemesis attack, but in this case, the information is leaked through the timing of DMA requests instead of using interrupt requests.

Another approach is that the processor is not halted completely, only if there is a case of resource contention: the DMA request needs to access the same memory partition as the processor. But because the resource contention is not eliminated in this case, the leakage is still present in the system, it only requires a different approach to extract. With a combined attack using well-timed DMA requests and interrupts, an attacker could still gather information about the memory access pattern of the enclave in a given cycle using the execution latencies as in the Nemesis attack.

As an example, take an attacker that can execute an enclave twice that contains a single instruction. For both executions, they set up a Nemesis attacker framework to interrupt the instruction of the enclave and measure the execution latency. For the second execution only, they issue a DMA request in the first cycle of the enclave's execution. If in the second case the execution of the instruction took a cycle longer than in the first case, the attacker knows that the processor tried to access the memory in the same cycle as the DMA request. If the interrupt timings did not differ, that means that the processor was not using the memory in the given cycle.

Giving the priority to the peripheral also opens up the possibility of a Denial-of-Service attack, as pointed out by [40]. Because the processor is halted until the DMA request is being served, by issuing continuous DMA requests, the enclave's execution can be halted indefinitely. Practically, this is less of an issue, because most TEEs – including Sancus – provide no availability guarantees, as described in 2.1.

Since all of these suggestions involve changes to the hardware, these defenses are also difficult to apply to existing systems. TEEs flashed to an FPGA can be reflashed, but ASIC implementations need to be physically modified or replaced to implement the defense.

4.4 Vulnerability in Sancus with DMA priority

On the original openMSP430 platform, the signal that controls the priority is controlled by the DMA-enabled peripheral. In the current Sancus implementation, this signal has been hardwired to offer the CPU the priority to avoid the aforementioned Denial-of-Service attack after the security analysis of [40].

To implement the defense strategy outlined in the previous chapter, this signal would need to either be hardwired to always offer the DMA requests priority, or use the internal signal that is activated whenever an enclave is executing as the input to the priority signal. This would mean the priority is given to the CPU when unprotected code is executing, but to peripheral devices when an enclave is executing. This configuration might be desirable because it increases the performance and the availability guarantees offered by the system when executing unprotected code (for which the isolation guarantees do not need to hold) but still prevent information

leakage from protected enclaves by completely halting the CPU while serving DMA requests.

Vulnerability description Unfortunately, the current implementation of the openMSP430 core does not offer protection against our attack with the priority given to the DMA device. If a DMA request is detected while an instruction is in its execution stage, the execution is first finished, only thereafter is the CPU halted while the DMA request is served. While the execution is finishing, the system works as if the priority was given to the CPU.

This means that for a couple of cycles – a single instruction's execution at a time – the system works in exactly the same way as if the priority was given to the CPU and DMA requests are also allowed to be served. An attacker with precisely timed DMA requests can still extract the same information in that timing window as prior to changing the priority.

Proof-of-concept As a proof-of-concept, we modified our malicious peripheral to only issue DMA requests in every third (and the very first) clock cycle. This minimizes the time the CPU is halted during the execution, during which no information could be gathered about the enclave. Using this peripheral and knowledge of the source code, we were able to still identify the secret-dependent control flow from the original example of listing 3.1. Figure 4.1 shows the timing diagram of the attack.

This time, instead of showing the memory accesses of the enclave, we show the relationship between the DMA enable signal, which is high when issuing a DMA request, and the DMA ready signal, which is high if the request has completed successfully. In case of no contention, the ready signal is raised in the same cycle as the enable, which means that the output data will be valid in the next cycle.

By arming the peripheral with a data memory address, the different contention between the `add` and `mov` instructions is still visible in the 8th clock cycle of the DMA ready signal, even with this the priority given to the peripheral.

Mitigation Halting the execution of instructions immediately when a DMA request arrives would result in every DMA request being served immediately with no information leakage. The downsides of this approach are that it requires a significant microarchitectural modification to the processor and it offers no significant performance improvement over a system with no DMA access since the processor is halted during the DMA request's completion.

If the current instruction finishes executing, but DMA requests are not served until the CPU is halted, this introduces the Nemesis-type leakage as explained in the previous section. Leaking the execution delay of instructions – although undesirable – still reduces the complexity of defending against the attack. Software defenses such as compiler modifications or static analysis tools that have been developed to mitigate the original Nemesis are also effective against this modified DMA leakage.

A secure padding scheme has also been proposed as a hardware defense against Nemesis [9]. Using the insight from this research, a similar scheme could possibly be used to modify the serving of DMA requests on the microarchitectural level to eliminate the leakage.

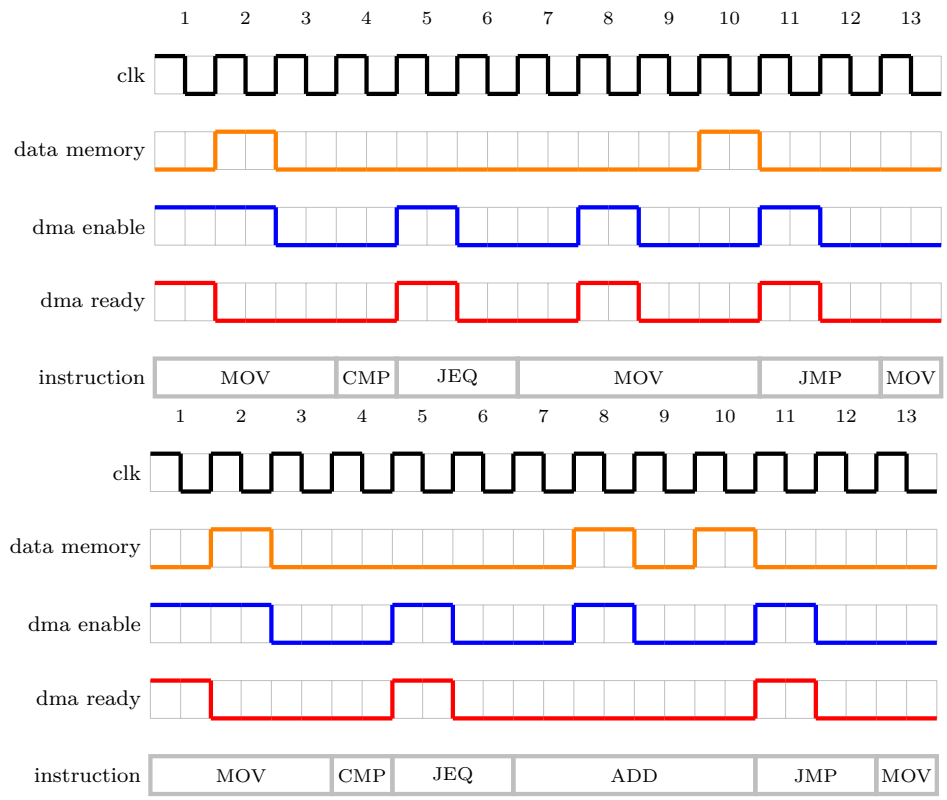


FIGURE 4.1: Timing diagram for a correct and an incorrect password guess with the DMA priority setup

The complete halting of the CPU still remains as an optimization problem, but the effect can be reduced by only reversing the priority when an enclave is executing or using a dedicated instruction as mentioned earlier to protect secret-dependent conditionals.

4.5 Conclusions

In this chapter, multiple defenses have been proposed for our attack. Aside from changing the memory unit's design, disabling DMA on the system, and completely eliminating secret-dependent flows from the enclave code, we consider two defenses: code balancing and reversing the memory bus priority.

For code balancing, a number of challenges are highlighted; based on the instruction analysis conducted in the previous chapter, many combinations are possible for the memory access pattern of an instruction, and it cannot be predicted at compilation time in every case. Detecting which instruction sequences' memory access pattern depends on a secret is another challenge. This solution would also result in a decrease in performance and an increase in code and data sizes for certain enclaves. Importantly,

this defense requires existing programs to be recompiled and redistributed to be protected against the attack.

Another approach is assigning the priority to the DMA requests over the CPU on the shared memory bus. Depending on the implementation, the leakage can either be completely eliminated, or reduced to that of Nemesis – instruction latencies – which is arguably easier to defend against, and already has proposed defense methods which can aid in mitigating this leakage.

Applying this defense is also challenging because it involves changing the hardware design and redistributing the modified hardware units.

On top of the discussion, we have also shown experimentally that the current implementation of reversing the priority on Sancus is still vulnerable against our attack with slight modifications.

Chapter 5

Conclusions and future work

Trusted Execution Environments (TEEs) [29, 41] are an emerging class of architectures that offer strong security guarantees to processes executing on them.

Recent research has presented many side-channel attacks [35, 52, 53] that exploit microarchitectural details of a system, some of which are effective even against TEEs.

This thesis presented a novel side-channel attack that targets the shared memory bus and leaks the memory access pattern of protected enclaves using the timing information of Direct Memory Access requests issued to unprotected memory regions. Using this information, an attacker can potentially extract secrets from a protected enclave.

Potential directions for future research are discussed in section 5.1 and the thesis is concluded by the summary of the contributions in section 5.2

5.1 Future work

This thesis has demonstrated the basic attack, performed an initial analysis to classify the leakage and to showcase the advantages and limitations of the attack, and proposed some defense ideas. Many aspects of these research directions have been left unexplored and could benefit from continued research.

Complete instruction analysis In this thesis we provided a description of factors that influence the memory access trace of instructions, and an incomplete pattern mapping for instruction is provided in appendix A.

Completing this analysis would benefit the implementation of defenses or a formal model of the attack. It can also reveal whether instructions exist in the system that enable constructing an enclave described in section 3.7, which is not vulnerable to Nemesis or the DMA attack individually, only a combined attack.

Defenses In the defenses chapter, we have only provided an informal description of a few possible defense methods and a vulnerability in one of them, but a more in-depth survey of possible defense methods and experimental implementations could prove to be useful for mitigating the effects of this attack.

Formal analysis A formal model of a slightly modified version of Sancus has already been created to develop and prove the effectiveness of a defense method against interrupt-based leakage [9].

A similar effort could be carried out to model the leakage provided by our DMA attack, and formally verify the correctness of proposed defense methods.

Commercial target platforms This thesis has focused on implementing the attack on the Sancus platform due to its uncomplicated design and deterministic nature. And interesting next step would be investigating the leakage of this attack on other, more widely used platforms.

Both Intel SGX [31] and ARM TrustZone [7] support DMA requests in the system, so the attack has potential, but due to their complexity and optimization features the implementation is likely more challenging.

Remote attacks In this thesis, we considered an attacker model where the adversary has physical access to the TEE and can plug in a malicious peripheral. Although this approach is in line with the current attacker model of Sancus [34] and mirrors other research [30], it does limit the applicability of the attack.

As described in section 2.6, both remote attacks using Remote DMA [26, 43] and attacks on heterogeneous CPU-FPGA platforms [18, 51] have been discovered. Using the insight from this research, our DMA attack could be attempted to be executed remotely using one of these setups.

Covert channel Side-channel leakage has been used to build covert channels to communicate between isolated processes, for example by using the row buffer's state on a DRAM chip as part of the DRAMA attack [35]. The Flush+Reload [53] cache attack has been used in the Meltdown [28] and Spectre [23] attacks as a covert channel.

A line of future research could seek to use our discovered leakage to establish a covert channel between processes, and compare its performance to other solutions.

In the context of Sancus, this would be especially interesting, because to the best of our knowledge no covert channel has been established yet on the platform.

A different source of leakage In our current attack, we use the resource contention of the memory bus between a memory request by the CPU and a DMA request. Future research could evaluate the possibility of modifying the attack to use a contention between requests from other origins, for example two cores of the same CPU.

5.2 Contributions

In this thesis, we presented a novel DMA-based side-channel attack that is effective even against processes running in a TEE enclave. Our attack is the second discovered side-channel attack on the Sancus platform. The attacker uses timing measurements of DMA requests targeting unprotected memory regions to infer the memory access patterns of the CPU. These measurements can be used to reconstruct the control and data flow of enclaves and can lead to the leakage of security keys or other sensitive information.

The detailed contributions of this thesis were the following:

- We presented our attack that allows an unprivileged attacker with the ability to connect a malicious peripheral to infer the memory access pattern of an executing enclave. The attack measures the latency of DMA requests targeting unprotected memory, which get delayed if the CPU accesses the memory in the same timeframe.
- Using practical examples, we showed that using the collected memory traces and some knowledge of the source code or the inputs to the enclave, an attacker can reconstruct control and data flows in some contexts, thus breaking the isolation guarantees of the enclave. Our examples also showed how this can lead to the extraction of sensitive information from the enclaves.
- We highlighted how certain features of Sancus' [34] underlying openMSP430 platform [17] contribute to even more severe leakage. Certain features of the architecture make the attack completely deterministic and accurate to a single clock cycle. The partitioning of the memory into three separate interfaces also makes the attack more fine-grained and provides more information about the memory accesses than a system with a single memory interface.
- On top of the theoretical explanation and the analysis of the leakage, we presented a minimal peripheral device implemented on the current, unmodified Sancus platform that conforms to the attacker model of the architecture and is capable of capturing these cycle-accurate traces.
- We compared the leakage with that of Nemesis, the only other side-channel attack against Sancus, and concluded that the two attacks provide a different leakage with neither being more powerful than the other. We extended the Sancus-Step attack framework to allow easy exploitation of both of these vulnerabilities from software.
- We acknowledged certain limitations of both the implementation of the experimental setup and the attack in general. Most notable is the need for physical access to connect the peripheral to the system or the ability to compromise an existing, connected peripheral with specific requirements.
- Lastly, we discussed potential defenses against the attack and highlighted some challenges. We explained how balancing the instructions of conditional branches could eliminate the leakage, but also showed some of the challenges with this approach. The possible hardware defense of reversing the priority over the memory bus was also considered, and a few different approaches of its implementation were discussed, one of which would result in reducing the leakage to that of execution latencies, as in the Nemesis attack.
- We experimentally showed using a modified attacker peripheral that the current implementation of reversing the priority on Sancus is flawed. Using only a slight modification in the malicious peripheral and interrupt capabilities, the attacker can capture the same information leakage as with the default priority setup.

The main code contributions of this thesis are in the process of being upstreamed to the Sancus repositories and are available on the following pages:

- https://github.com/sancus-pma/sancus-examples/pull/13
- https://github.com/sancus-pma/sancus-core/pull/19
- https://github.com/sancus-pma/sancus-support/pull/10

Appendices

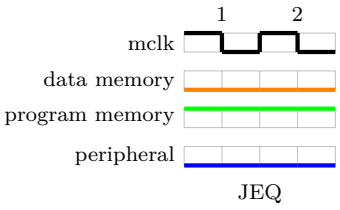
Appendix A

Memory traces of different instruction formats

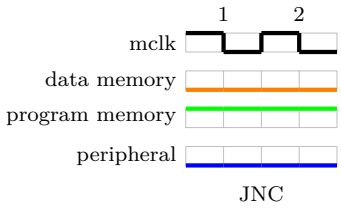
This section only aims to highlight the diversity of the possible memory access traces of a single instruction on openMSP430.

In the operands of the instructions, `r5` contains a data memory address, `r6` a program memory address, and `r7` a memory-mapped I/O address. The address `EDE` can refer to all three of the memory partitions.

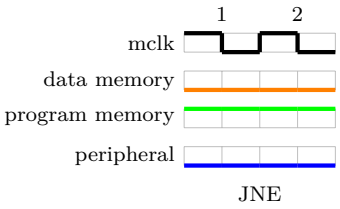
A.1 Jump instructions



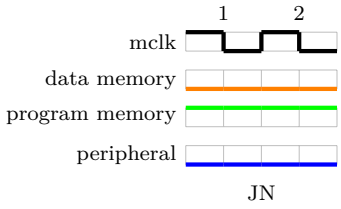
JEQ



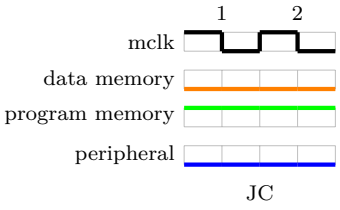
JNC



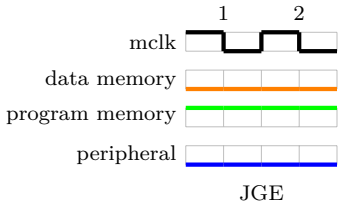
JNE



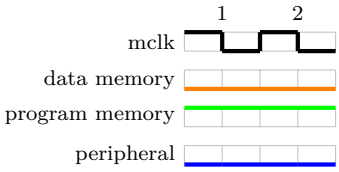
JN



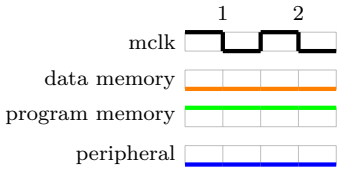
JC



JGE



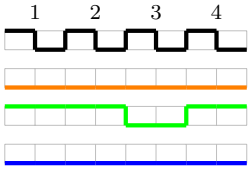
JL



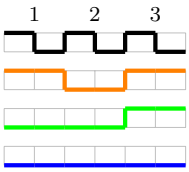
JMP

A.2 Single operand instructions

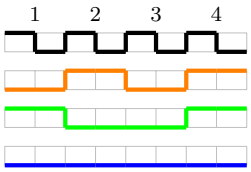
RRC r4



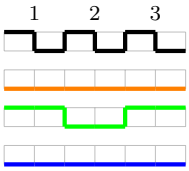
RRC &EDE



RRC @r5

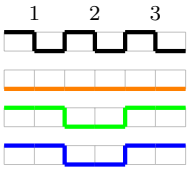


RRC &EDE

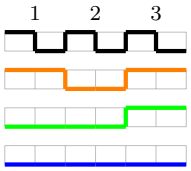


RRC @r6

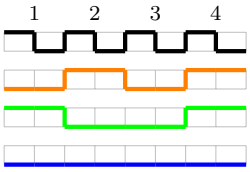
RRA r4



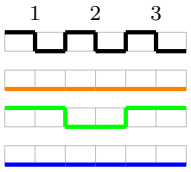
RRC @r7



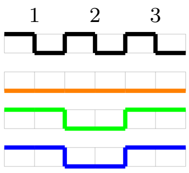
RRA @r5



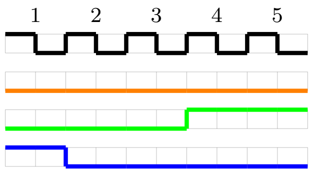
RRC &EDE



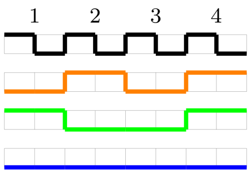
RRA @r6



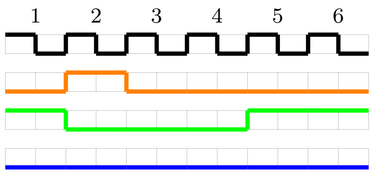
RRA @r7



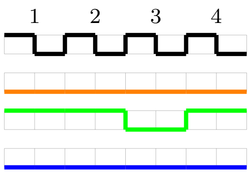
PUSH @r7



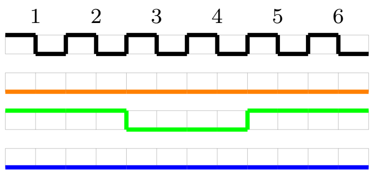
RRA &EDE



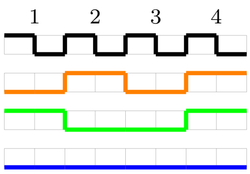
PUSH &EDE



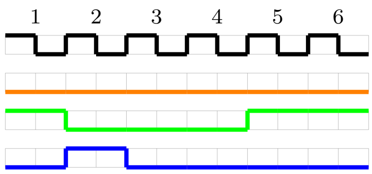
RRA &EDE



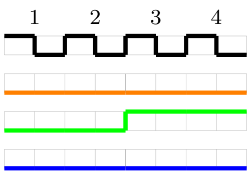
PUSH &EDE



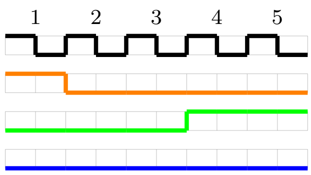
RRA &EDE



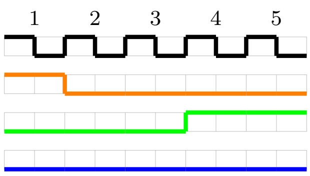
PUSH &EDE



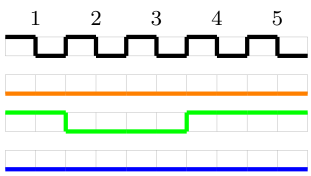
PUSH r4



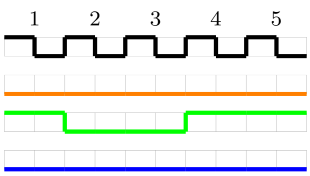
PUSH @r5



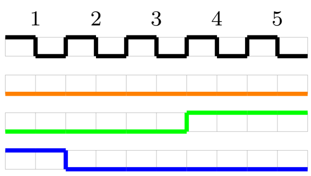
PUSH @r5



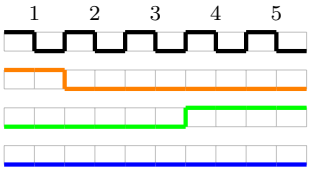
PUSH @r6



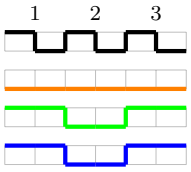
PUSH @r6



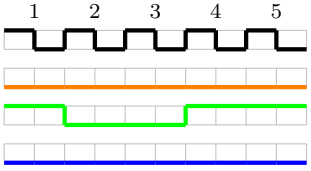
PUSH @r7



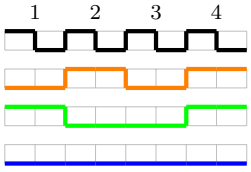
PUSH @r5+



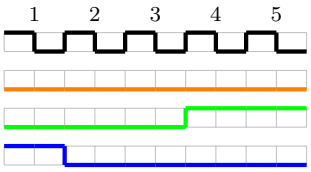
SWPB @r7



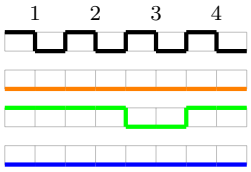
PUSH @r6+



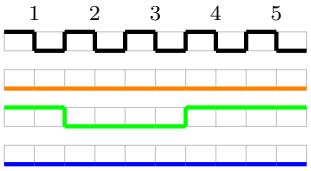
SWPB &EDE



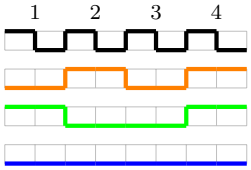
PUSH @r7+



SWPB &EDE

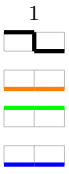


PUSH #N

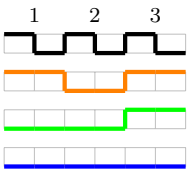


SWPB &EDE

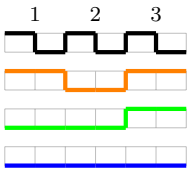
SWPB r4



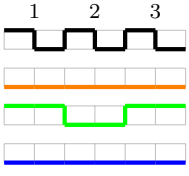
SXT r4



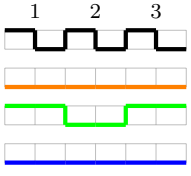
SWPB @r5



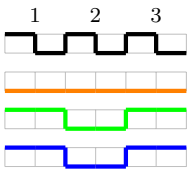
SXT @r5



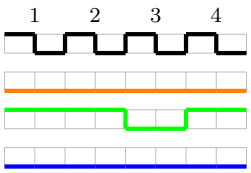
SWPB @r6



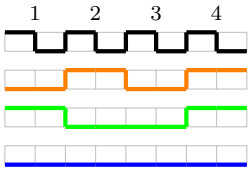
SXT @r6



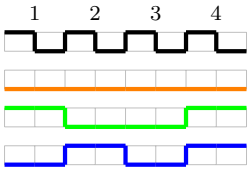
SXT @r7



SXT &EDE



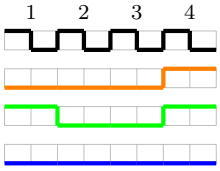
SXT &EDE



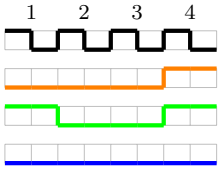
SXT &EDE

A.3 Double operand instructions

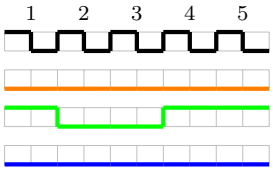
MOV r4, r4



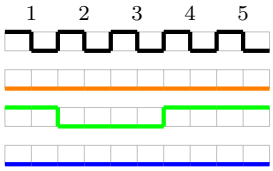
MOV r4, &EDE



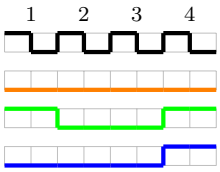
MOV r4, x(r5)



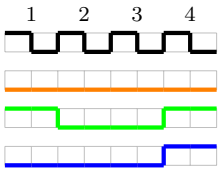
MOV r4, &EDE



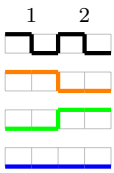
MOV r4, x(r6)



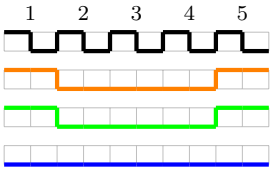
MOV r4, &EDE



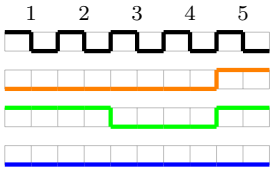
MOV r4, x(r7)



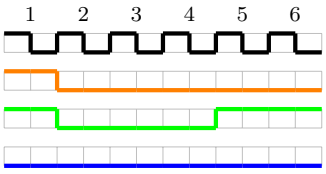
MOV @r5, r4



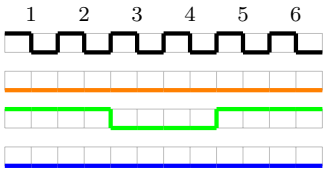
MOV @r5, x(r5)



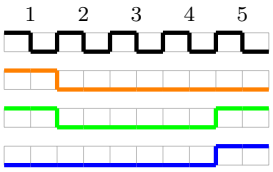
MOV @r6, x(r5)



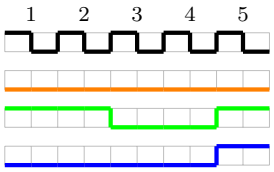
MOV @r5, x(r6)



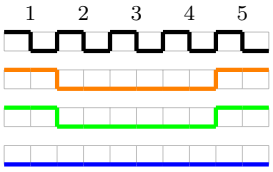
MOV @r6, x(r6)



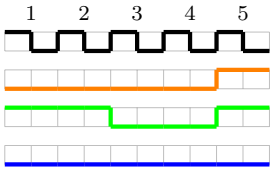
MOV @r5, x(r7)



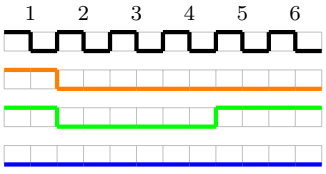
MOV @r6, x(r7)



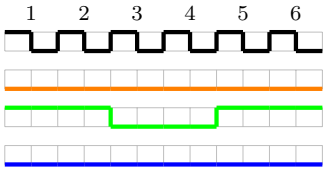
MOV @r5, &EDE



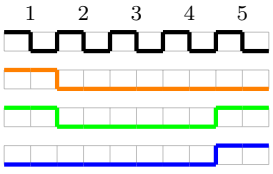
MOV @r6, &EDE



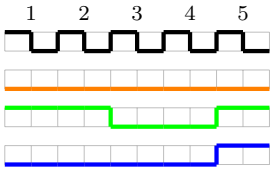
MOV @r5, &EDE



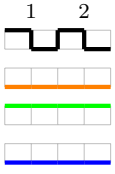
MOV @r6, &EDE



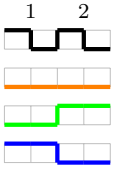
MOV @r5, &EDE



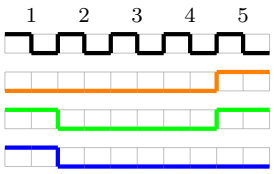
MOV @r6, &EDE



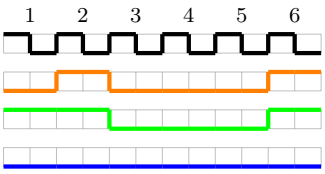
MOV @r6, r4



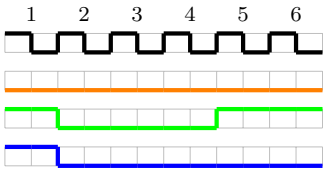
MOV @r7, r4



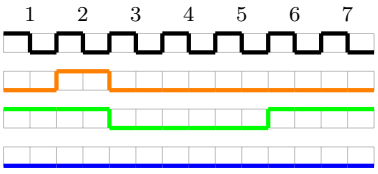
MOV @r7, x(r5)



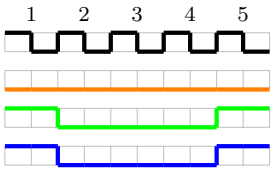
MOV &EDE, x(r5)



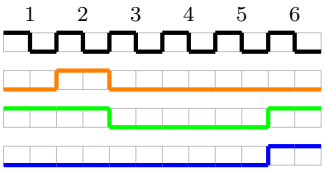
MOV @r7, x(r6)



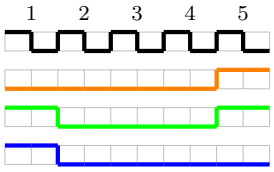
MOV &EDE, x(r6)



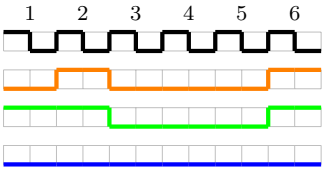
MOV @r7, x(r7)



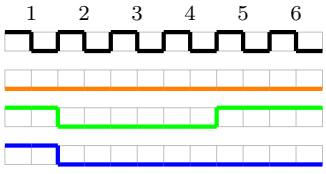
MOV &EDE, x(r7)



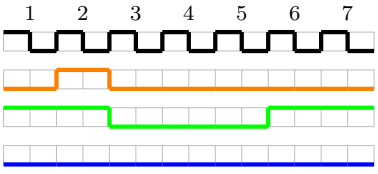
MOV @r7, &EDE



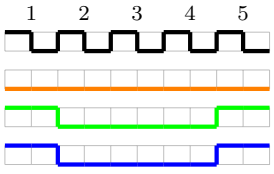
MOV &EDE, &EDE



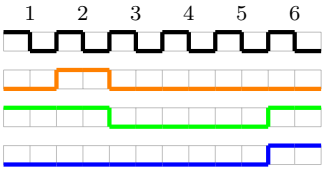
MOV @r7, &EDE



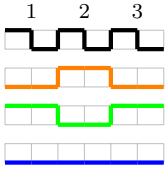
MOV &EDE, &EDE



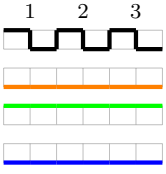
MOV @r7, &EDE



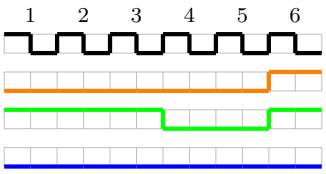
MOV &EDE, &EDE



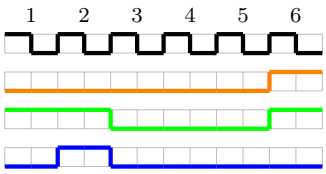
MOV &EDE, r4



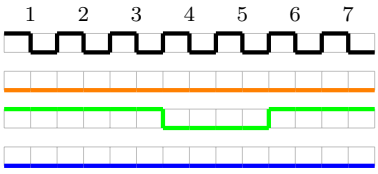
MOV &EDE, r4



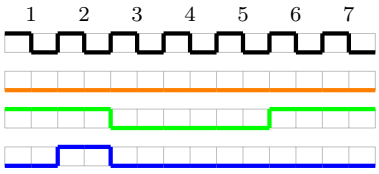
MOV &EDE, x(r5)



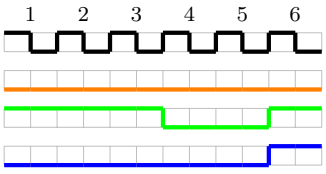
MOV &EDE, x(r5)



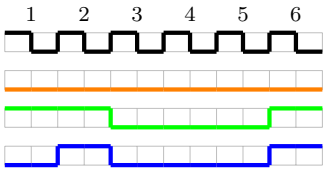
MOV &EDE, x(r6)



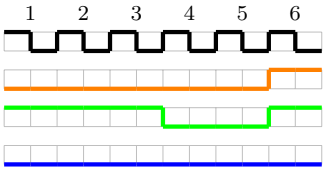
MOV &EDE, x(r6)



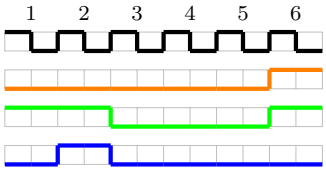
MOV &EDE, x(r7)



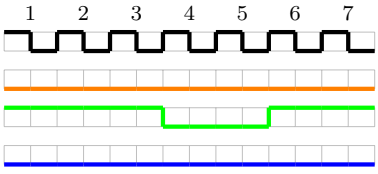
MOV &EDE, x(r7)



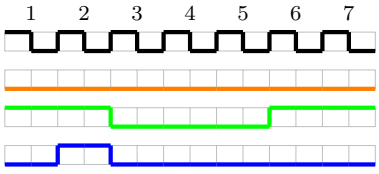
MOV &EDE, &EDE



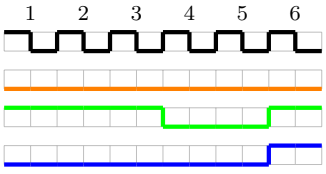
MOV &EDE, &EDE



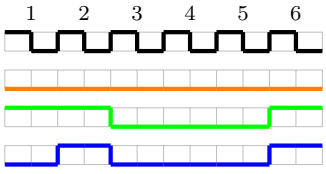
MOV &EDE, &EDE



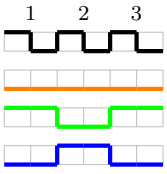
MOV &EDE, &EDE



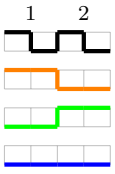
MOV &EDE, &EDE



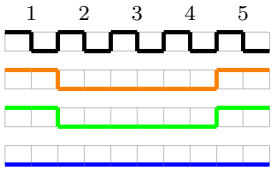
MOV &EDE, &EDE



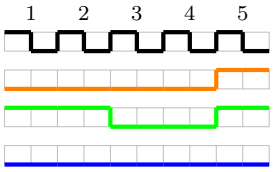
MOV &EDE, r4



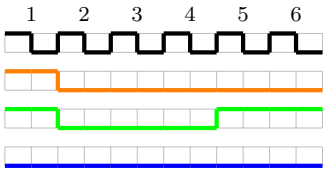
MOV @r5, r4



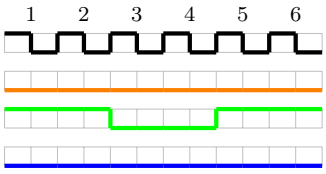
MOV @r5, x(r5)



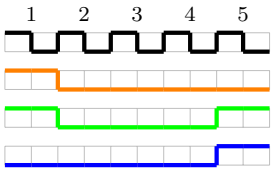
MOV @r6, x(r5)



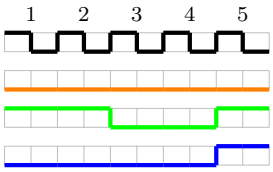
MOV @r5, x(r6)



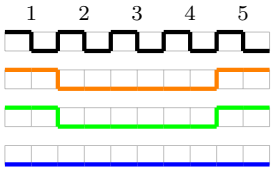
MOV @r6, x(r6)



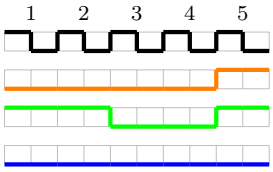
MOV @r5, x(r7)



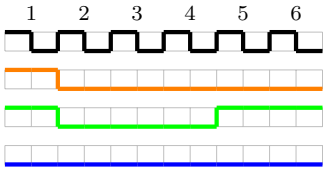
MOV @r6, x(r7)



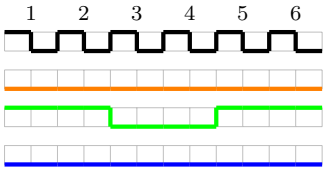
MOV @r5, &EDE



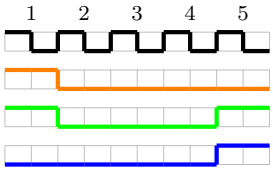
MOV @r6, &EDE



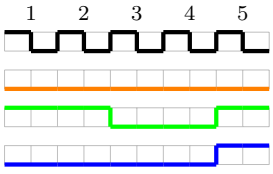
MOV @r5, &EDE



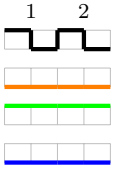
MOV @r6, &EDE



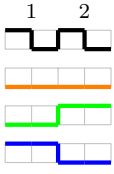
MOV @r5, &EDE



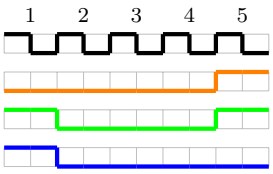
MOV @r6, &EDE



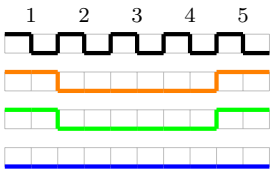
MOV @r6, r4



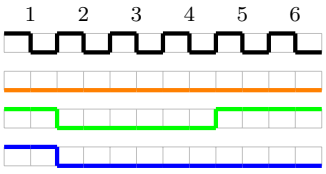
MOV @r7, r4



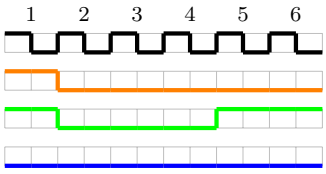
MOV @r7, x(r5)



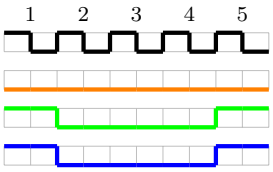
MOV @r5+, x(r5)



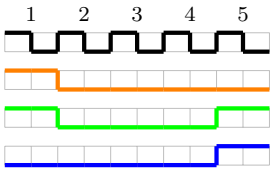
MOV @r7, x(r6)



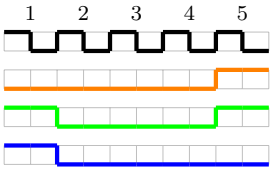
MOV @r5+, x(r6)



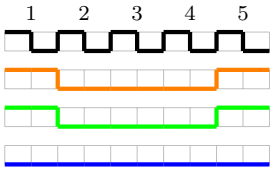
MOV @r7, x(r7)



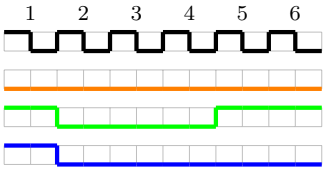
MOV @r5+, x(r7)



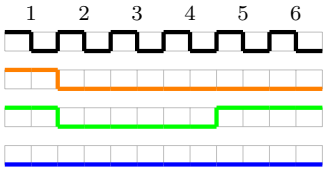
MOV @r7, &EDE



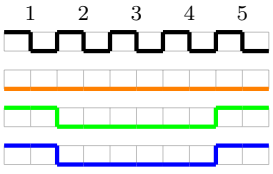
MOV @r5+, &EDE



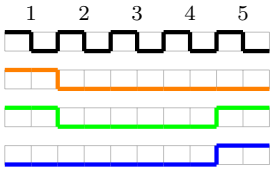
MOV @r7, &EDE



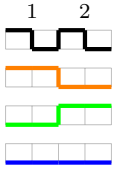
MOV @r5+, &EDE



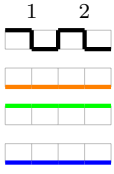
MOV @r7, &EDE



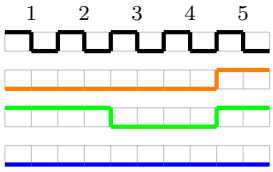
MOV @r5+, &EDE



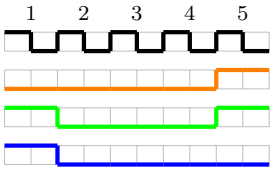
MOV @r5+, r4



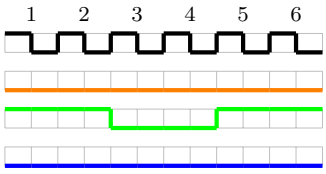
MOV @r6+, r4



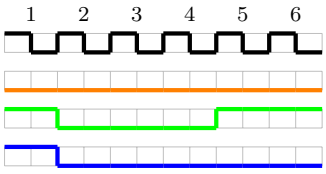
MOV @r6+, x(r5)



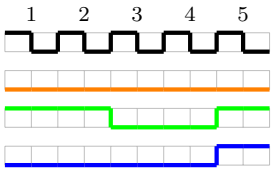
MOV @r7+, x(r5)



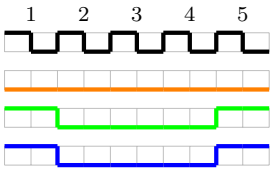
MOV @r6+, x(r6)



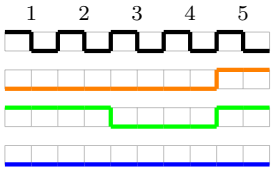
MOV @r7+, x(r6)



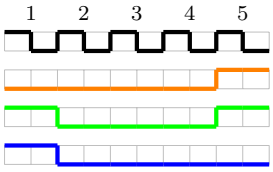
MOV @r6+, x(r7)



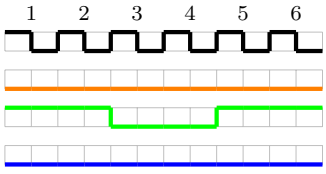
MOV @r7+, x(r7)



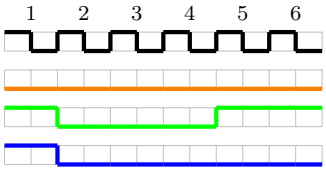
MOV @r6+, &EDE



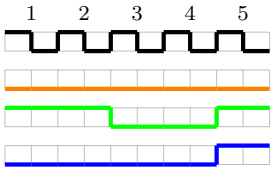
MOV @r7+, &EDE



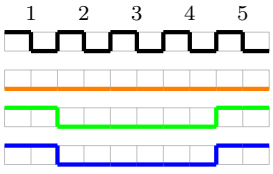
MOV @r6+, &EDE



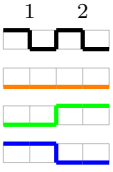
MOV @r7+, &EDE



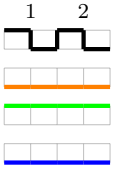
MOV @r6+, &EDE



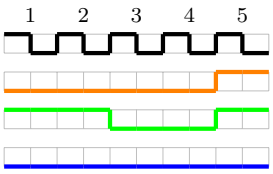
MOV @r7+, &EDE



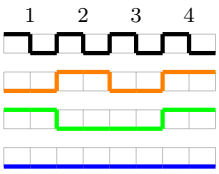
MOV @r7+, r4



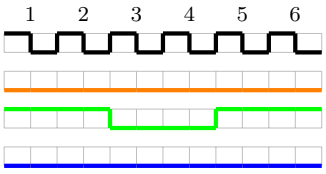
MOV #N, r4



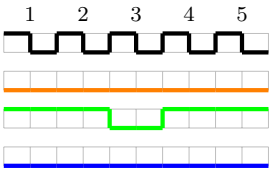
MOV #N, x(r5)



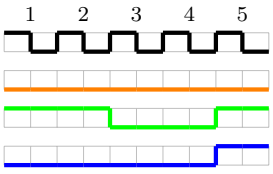
ADD r4, x(r5)



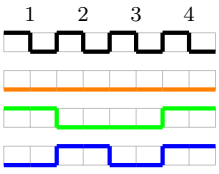
MOV #N, x(r6)



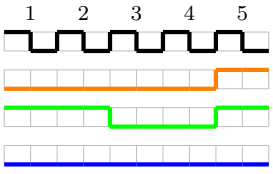
ADD r4, x(r6)



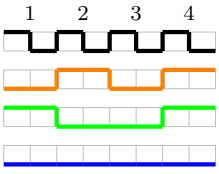
MOV #N, x(r7)



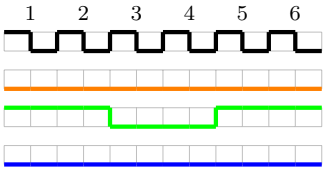
ADD r4, x(r7)



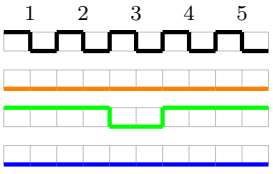
MOV #N, &EDE



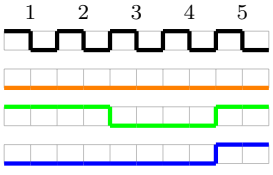
ADD r4, &EDE



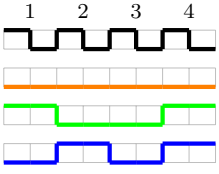
MOV #N, &EDE



ADD r4, &EDE

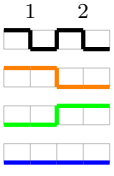


MOV #N, &EDE

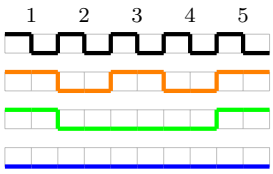


ADD r4, &EDE

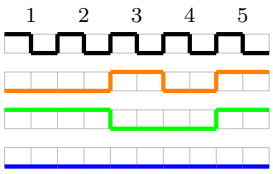
ADD r4, r4



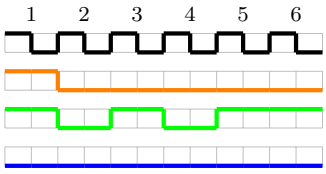
ADD @r5, r4



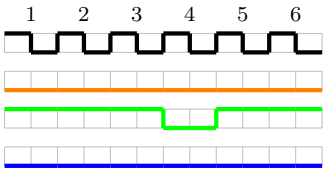
ADD @r5, x(r5)



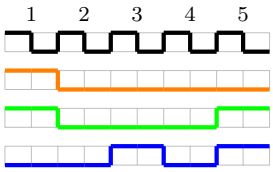
ADD @r6, x(r5)



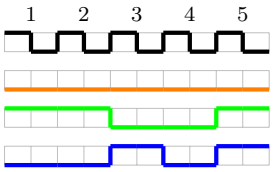
ADD @r5, x(r6)



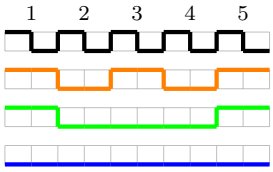
ADD @r6, x(r6)



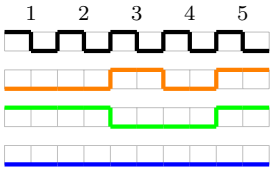
ADD @r5, x(r7)



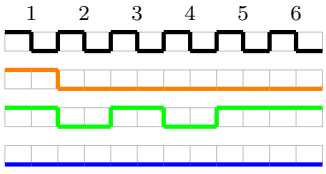
ADD @r6, x(r7)



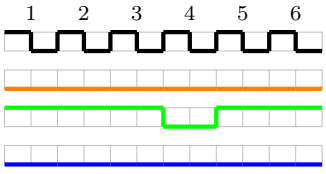
ADD @r5, &EDE



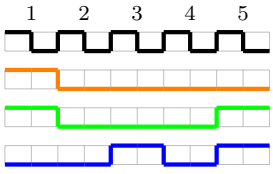
ADD @r6, &EDE



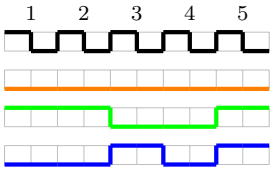
ADD @r5, &EDE



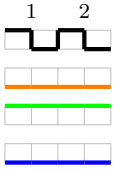
ADD @r6, &EDE



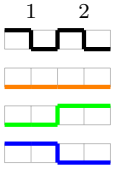
ADD @r5, &EDE



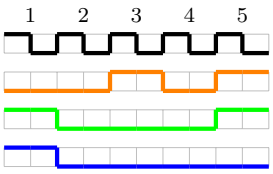
ADD @r6, &EDE



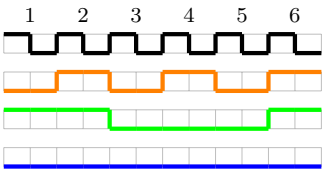
ADD @r6, r4



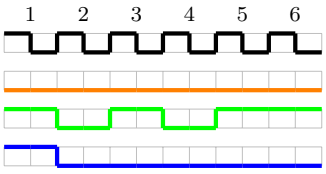
ADD @r7, r4



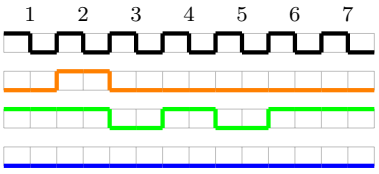
ADD @r7, x(r5)



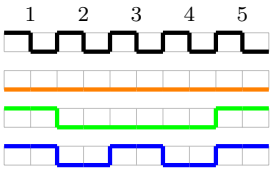
ADD &EDE, x(r5)



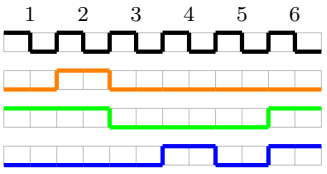
ADD @r7, x(r6)



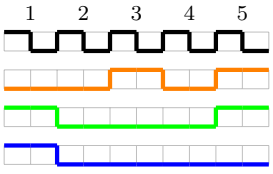
ADD &EDE, x(r6)



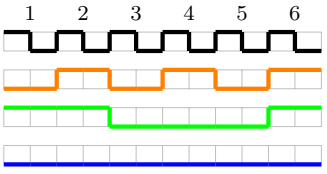
ADD @r7, x(r7)



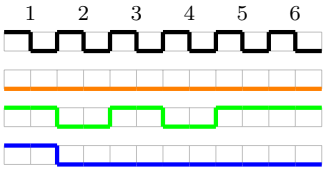
ADD &EDE, x(r7)



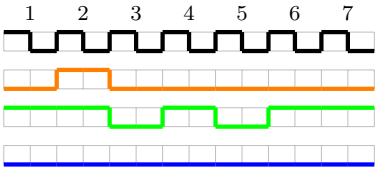
ADD @r7, &EDE



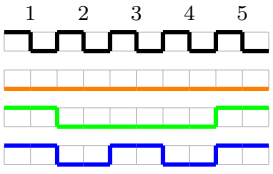
ADD &EDE, &EDE



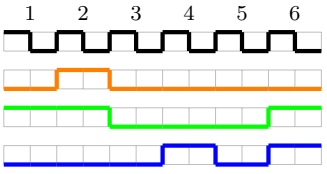
ADD @r7, &EDE



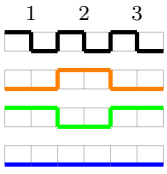
ADD &EDE, &EDE



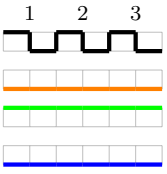
ADD @r7, &EDE



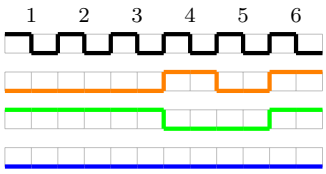
ADD &EDE, &EDE



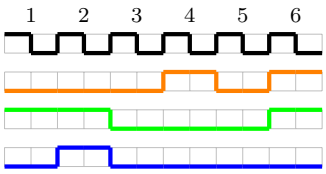
ADD &EDE, r4



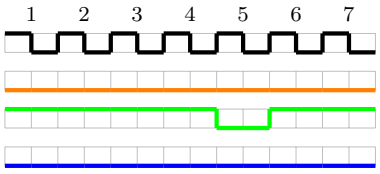
ADD &EDE, r4



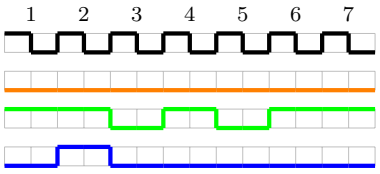
ADD &EDE, x(r5)



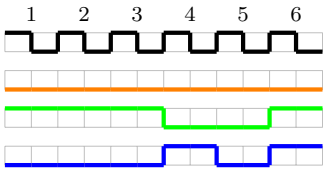
ADD &EDE, x(r5)



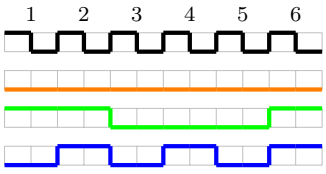
ADD &EDE, x(r6)



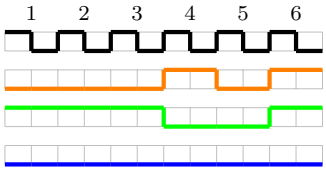
ADD &EDE, x(r6)



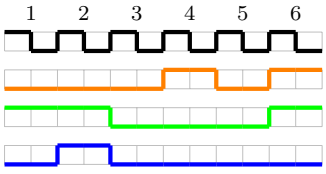
ADD &EDE, x(r7)



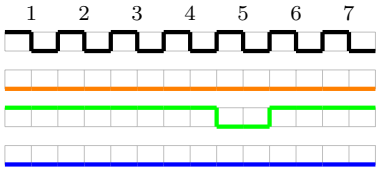
ADD &EDE, x(r7)



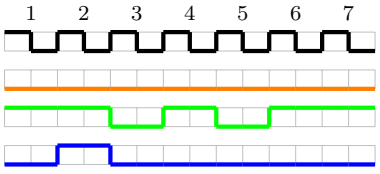
ADD &EDE, &EDE



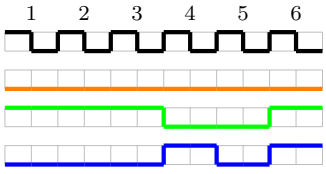
ADD &EDE, &EDE



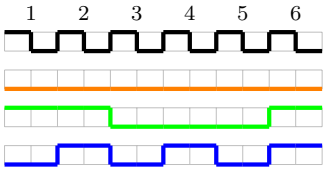
ADD &EDE, &EDE



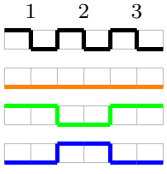
ADD &EDE, &EDE



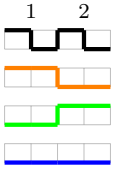
ADD &EDE, &EDE



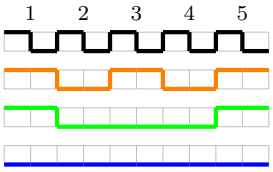
ADD &EDE, &EDE



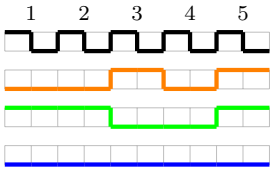
ADD &EDE, r4



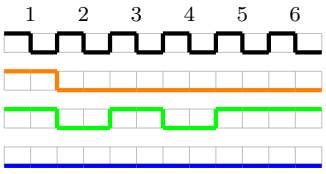
ADD @r5, r4



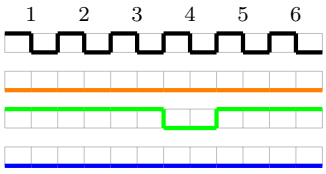
ADD @r5, x(r5)



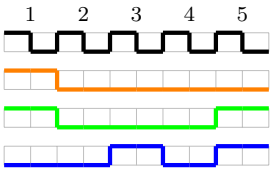
ADD @r6, x(r5)



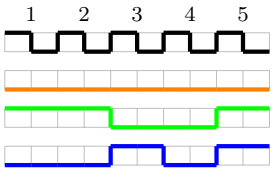
ADD @r5, x(r6)



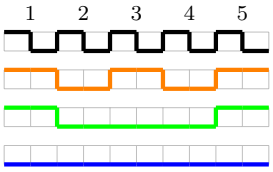
ADD @r6, x(r6)



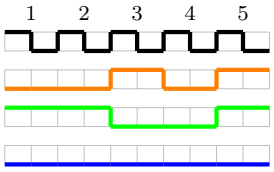
ADD @r5, x(r7)



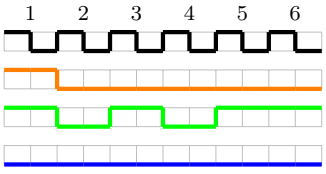
ADD @r6, x(r7)



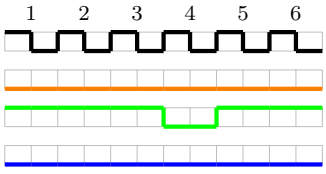
ADD @r5, &EDE



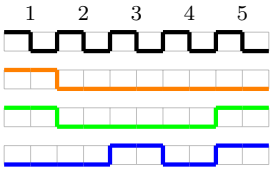
ADD @r6, &EDE



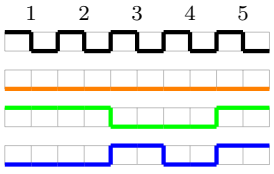
ADD @r5, &EDE



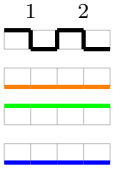
ADD @r6, &EDE



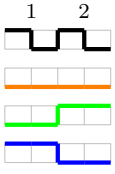
ADD @r5, &EDE



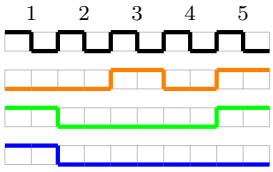
ADD @r6, &EDE



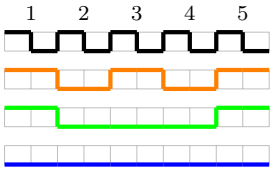
ADD @r6, r4



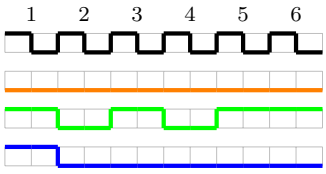
ADD @r7, r4



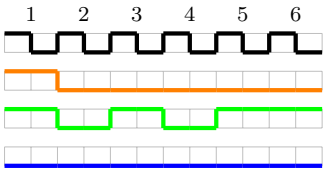
ADD @r7, x(r5)



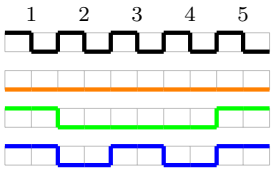
ADD @r5+, x(r5)



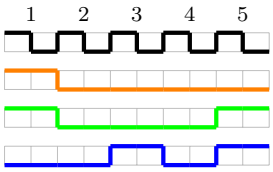
ADD @r7, x(r6)



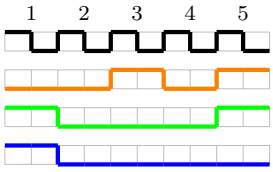
ADD @r5+, x(r6)



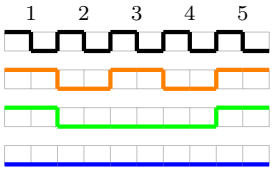
ADD @r7, x(r7)



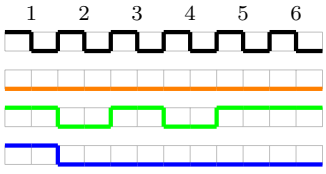
ADD @r5+, x(r7)



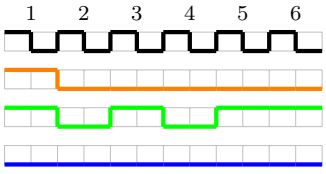
ADD @r7, &EDE



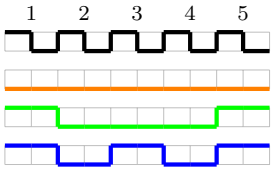
ADD @r5+, &EDE



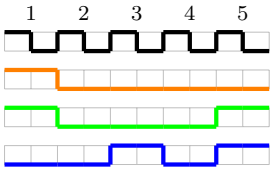
ADD @r7, &EDE



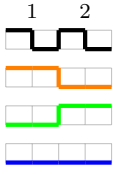
ADD @r5+, &EDE



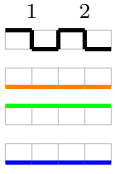
ADD @r7, &EDE



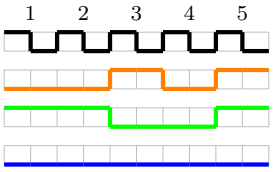
ADD @r5+, &EDE



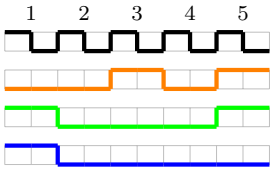
ADD @r5+, r4



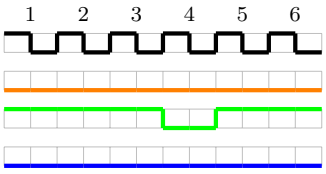
ADD @r6+, r4



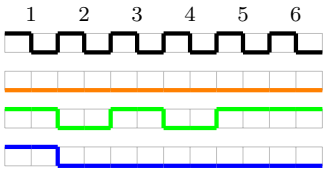
ADD @r6+, x(r5)



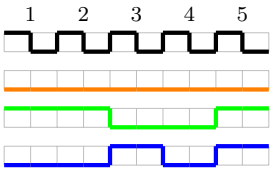
ADD @r7+, x(r5)



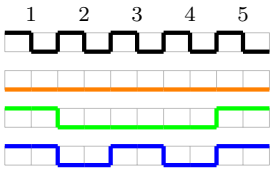
ADD @r6+, x(r6)



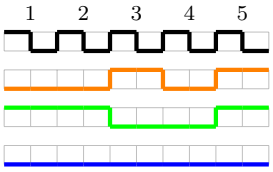
ADD @r7+, x(r6)



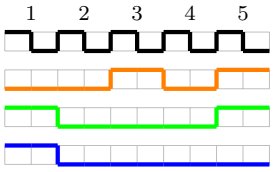
ADD @r6+, x(r7)



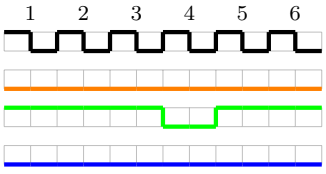
ADD @r7+, x(r7)



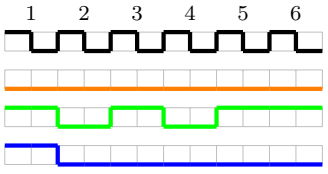
ADD @r6+, &EDE



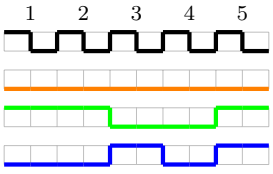
ADD @r7+, &EDE



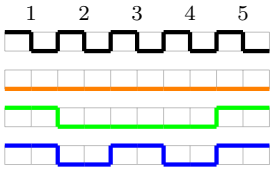
ADD @r6+, &EDE



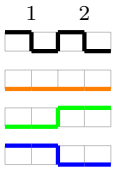
ADD @r7+, &EDE



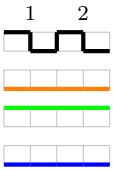
ADD @r6+, &EDE



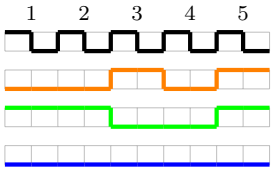
ADD @r7+, &EDE



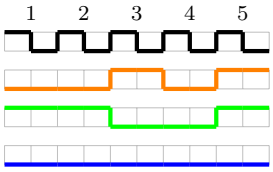
ADD @r7+, r4



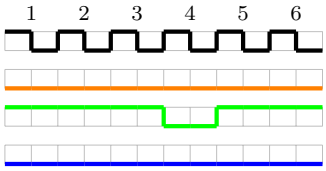
ADD #N, r4



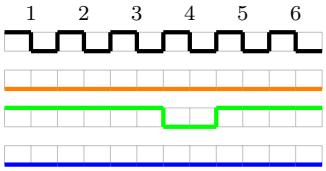
ADD #N, x(r5)



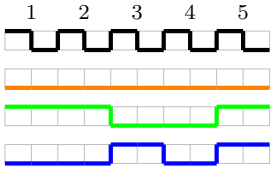
ADD #N, &EDE



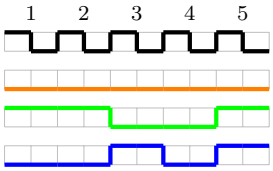
ADD #N, x(r6)



ADD #N, &EDE



ADD #N, x(r7)



ADD #N, &EDE

Bibliography

[1] M. Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer, 1999.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 171–185. IEEE, 2012.

[3] Aleph One. Smashing the stack for fun and profit. https://web.archive.org/web/20000817054314/http://www.shmoo.com/phrack/Phrack49/p49-14, 1996. Accessed 2020-06-04.

[4] Alibaba Cloud. Compute optimized instance families with FPGAs. https://www.alibabacloud.com/help/doc-detail/108504.html. Accessed 2020-06-04.

[5] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed 2020-06-04.

[6] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[7] ARM. Arm security technology: Building a secure system using trustzone technology, 2009.

[8] A. Boileau. Hit by a bus: Physical access attacks with firewire. *Presentation, Ruxcon*, 3, 2006.

[9] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors: Extended version. *arXiv preprint arXiv:2001.10881*, 2020.

[10] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv.org*, 2018.

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

[12] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, pages 857–874. USENIX Association, 2016.

[13] S. Cuyt. A security analysis of interrupts in embedded enclaved execution, 2019.

[14] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 30(2):198–208, 1983.

[15] L. Duflot, Y.-A. Perez, and B. Morin. What if you can't trust your network card? In *International Workshop on Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

[16] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

[17] O. Girard. openmsp430. https://opencores.org/websvn/filedetails?repname=openmsp430&path=%2Fopenmsp430%2Ftrunk%2Fdoc%2FopenMSP430.pdf, 2017. Accessed 2020-06-04.

[18] M. Gross, N. Jacob, A. Zankl, and G. Sigl. Breaking trustzone memory isolation through malicious hardware on a modern fpga-soc. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, pages 3–12, 2019.

[19] T. Instruments. Msp430x1xx family user's guide. http://www.ti.com/lit/ug/slau049f/slau049f.pdf, 2006. Accessed 2020-06-04.

[20] Intel. Intel® Software Guard Extensions Developer Guide: Protection from Side-Channel Attacks. https://software.intel.com/en-us/node/703016. Accessed 2020-06-04.

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[22] I. Kizhvatov. Side channel analysis of avr xmega crypto engine. In *Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–7, 2009.

[23] 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.

[24] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[25] G. Kupfer. Iommu-resistant dma attacks, 2018.

[26] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi. Netcat: Practical cache attacks from the network, 2020.

[27] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. *arXiv preprint arXiv:1912.01701*, 2019.

[28] 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.

[29] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2017.

[30] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[31] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[32] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. Copycat: Controlled instruction-level attacks on enclaves for maximal key extraction. *arXiv preprint arXiv:2002.08437*, 2020.

[33] J. T. Mühlberg, S. Cleemput, M. A. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. An implementation of a high assurance smart meter using protected module architectures. In *IFIP International Conference on Information Security Theory and Practice*, pages 53–69. Springer, 2016.

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

[35] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, 2016.

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

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[38] B. Ruytenberg. Breaking Thunderbolt Protocol Security: Vulnerability Report, 2020. Public version.

[39] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[40] S. Seminara. Dma support for the sancus architecture, 2019.

[41] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[42] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 344–361. Springer, 2010.

[43] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, pages 213–226, 2018.

[44] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[45] J. Van Bulck, J. T. Mühlberg, and F. Piessens. Vulcan: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 225–237, 2017.

[46] 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*, MODULARITY Companion 2016, pages 146–151. ACM, 2016.

[47] J. Van Bulck and F. Piessens. Tutorial: Uncovering and mitigating side-channel leakage in intel sgx enclaves. In *Proceedings of the 8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE'18)*. Springer, 2018.

[48] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[49] 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*, pages 178–195, 2018.

[50] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy*, 10(5):68–70, 2012.

[51] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *arXiv preprint arXiv:1912.11523*, 2019.

[52] 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. IEEE, 2015.

[53] 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, 2014.

Master's thesis filing card

Student: Márton Bognár

Title: Analyzing side-channel leakage in secure DMA solutions

UDC:

Abstract:
With the advent of the Internet of Things, industrial control systems, and increasing automation in all areas of life, networked embedded devices have become ubiquitous. These systems are often running safety-critical software or handle sensitive data, but attacks against them are discovered regularly.
Trusted Execution Environments (TEEs) have been introduced to provide strong security guarantees for executing programs, both on high-end systems and on small embedded platforms. These TEEs execute programs in so-called enclaves, which provide isolation from unprotected code and other enclaves.
One important class of attacks against TEEs are the so-called side-channel attacks. While the enclaves isolate their contents on the architectural level, side-channel attacks exploit microarchitectural implementation details of the target platform to extract information from the enclaves. These attacks often target optimization features or shared resources on the system, such as caches or the shared memory. Side-channel attacks are well-researched in the context of high-end processors, but attacks against small embedded systems have not received much attention.
In this thesis, a new side-channel attack is presented that is effective against enclaves running on a TEE. The attack targets the shared memory bus and uses the timing of Direct Memory Access requests issued to unprotected addresses in the memory to reconstruct the memory accesses of the processor running the enclave.
The attack is demonstrated on the Sancus platform, which is based on the 16-bit TI MSP430 microcontroller. An experimental peripheral is presented that is capable of capturing the memory access patterns of the processor. Examples in the thesis show how this leakage can lead to the extraction of secrets from an enclave.
An analysis of the leakage is also provided; the factors that influence the memory access pattern of an instruction are described, and the leakage of the attack is compared to Nemesis, the current state-of-the-art attack on the Sancus platform that utilizes interrupt latency measurements to leak secrets.
A number of defense methods are also proposed both on the software and the hardware level. One of the possible defense methods on Sancus is experimentally shown to contain the same leakage as the original attack.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Secure Software

Thesis supervisor: Prof. dr. ir. Frank Piessens

Assessors: Dr. Pieter Philippaerts
Ir. Jo Van Bulck

Mentor: Ir. Jo Van Bulck