KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Hardening enclave programs against side-channel vulnerabilities at compile-time

Steffie Joosen

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

Thesis supervisor:
Prof. dr. ir. Frank Piessens

Assessors:
Hans Winderix
Dr. ir. Koen Yskout

Mentors:
Dr. ir. Jan Tobias Mülhberg
Hans Winderix

Academic year 2020 – 2021

Preface

I would like to thank the reader for taking the time and effort to dive into this thesis.

A special word of gratitude goes out to my mentors, Hans and Jan Tobias. Thanks to our numerous discussions and question sessions, I succeeded at completing this thesis. Without their support, I would not have been able to accomplish the result that lies in front of you. I thank professor Frank Piessens for being so approachable when I started this investigation.

I am grateful to my parents for supporting me in my studies and never doubting my abilities to succeed. Thanks to Matthias, who was always ready to listen to my complaints about this thesis or share in my joy. Shoutout to Márton for offering his help with peculiarities during my work.

Steffie Joosen

Contents

Abstract

The environment of computing services, networking technologies, embedded devices and the like is evolving as never before. High-end processors for laptops as well as microncontrollers for embedded systems are continuously extended with new functionality. The purpose of these extensions is primarily to improve performance, reduce power usage or enhance security. To improve the security of embedded and high-end devices, Trusted Execution Environments (TEEs) have emerged. TEEs provide protection against other programs running in an environment by executing programs in an enclave. An extension improving performance is the functionality for Direct Memory Access (DMA) in a (micro)processor in order to allow for a peripheral to access main memory without involvement of the CPU. With this extension, CPU has no burden of handling the continuous memory access requests coming from the peripheral.

Alongside the development of computing systems and their security, attacks on these systems have expanded. A remarkable class of attacks is the set of the so-called side-channel attacks. Not the architecture of the system or a bug in it is exploited in this kind of attack, but rather the way in which this architecture executes instructions on a micro-architectural level. By means of recording micro-architectural execution, the isolation guarantees made by TEEs may be broken. In general, extending an architecture increases the attack vector, the set of channels through which an attacker may compromise the security of a system.

In this thesis, a defense against a side channel attack on a microcontroller based on Direct Memory Access (DMA) extension is investigated. A DMA attacker is able to record memory accesses by the CPU, leaking information from an enclave. Our defense aims at inserting instructions into the source code of the program so that no information from the memory accesses performed by the CPU may leak from an enclave. We therefore provide a toolchain to classify all possible instructions according to their memory access pattern and find dummy instructions that exhibit the same pattern. We implement a procedure that hardens programs (addressing a certain portion of memory) against the DMA attack using these dummy instructions. We evaluate the result of the defense mechanism in terms of performance and code size. Suggestions on how to extend our defense to cover the whole memory address space are given and we provide several alternatives to the static methods used in our defense.

Listings

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

As computer systems become ever-present, attacks on them have developed massively as well. All kinds of attacks have emerged, ranging from Denial-of-Service attacks on network infrastructures to buffer overflow attacks exploiting memory layout. One important environment composed of the omnipresent computer systems is a multi-tenant environment where different users use the same infrastructure. With this advance of multi-user systems, a way of protecting the data and code of the different users is required. We will explain in this chapter why the existing protection techniques, like virtual memory, do not suffice. Deploying Trusted Execution Environments (TEEs) is a valid alternative to ensure highly needed security. TEEs provide a hardware-based isolation to programs executing next to others in a multi-tenant environment or to programs operating on or with secret keys, which must not be accessed by any other program.
Nonetheless, as predicted by the well-known *attacker-defender race*, stating that every new technology will eventually be attacked and that defenses will always be one step behind newly invented attacks, TEEs are vulnerable to side-channel attacks. Side-channel attacks mostly involve a passive attacker who observes execution of a program without being able to modify it. Side-channel attacks exploit the way the processor executes instructions, looking for example at power consumption or execution timing.

1.1 Trusted Execution Environments

Trusted Execution Environments were studied to address the shortcomings of existing isolation techniques. In a multi-tenancy setting, separation between the executing processes and between the processes and operating system belonging to the platform is of key importance. This last distinction is denoted as the separation of privileged code (belong to the OS) and user-level code for applications of the tenants.

Previously deployed isolation techniques

Virtual memory and security rings are technologies that separate resources and applications. Virtual memory for an application provides it with an address space

that is perceived to be contiguous and private to the process. Virtual addresses are translated to physical addresses by the Memory Management Unit (MMU) and these are used to address physical memory, isolating the process from the actual layout of memory. Security rings are implemented by processors in cooperation with operating systems where the operating system kernel operates in ring 0, giving it the highest privilege, followed by device drivers with slightly less privileges. User applications operate in ring 3, giving them the least privilege. Communication between rings to access resources for which higher privileges are needed, is strongly regulated in order to protect those resources to enforce the best possible security guarantees.

The above techniques successfully protect applications and the kernel in these multi-tenant system, but come with two main problems. First of all, they rely on a large Trusted Computing Base. The Trusted Computing Base (TCB) for a protection technique is the set of hardware and software components that it assumes to work correctly, guaranteeing they cannot be compromised. Security rings and virtual memory rely on a TCB that the operating system is part of. But the operating system is large, complex and its correctness and safety is very hard to verify, resulting in a weak guarantee for isolation technologies that rely on this security.
A second problem with security rings and virtual memory as protection mechanisms is the overhead they generate. Security rings require an extension to the processor together with software being compatible with it to ensure the correct isolation. Virtual memory requires an extra pass through the MMU which causes extra computational overhead. Trusted Execution Environments serve as an alternative to provide process isolation.

Trusted Execution Environments

A Trusted Execution Environment [43] serves as a protection mechanism for a program executing in a multi-tenant environment and the data it operates on. It executes the program in an *enclave* that only has well-defined entry and exit points so access to it can be strongly controlled. TEEs enforce that no program outside of it can access the enclave provided by the TEE. The Sancus TEE [25] for example uses program counter-based protection, checking the value of the program counter to make sure it respects the isolation of the enclave.

Attack surfaces

With every new technique being developed, either to protect against an existing attack or to make computing, networking, storage and the like more efficient, possibilities for attackers to compromise a system become more various. An attacker that successfully undermines a system's security could for example steal confidential data, bribe the system owners or make the system's availability drop acutely.

Now that devices are connected to the Internet almost by default, attackers have the possibility to attack remotely, as in a Distributed Denial of Service (DDoS) attack

[46], where the system's availability is diminished due to service requests that the attacker generates in bulk.
The term *attacker-defender race* describes the fact that developments in security give rise to new attacker surfaces, causing new defense techniques to be developed in their turn. The process of this development is fueled by attackers inventing new technologies to attack existing environments with their protection mechanisms, exploiting the attack surface. To mitigate against these attacks, new technologies come to the surface and these are again subject to new attacks, invented as a response to these technologies. TEEs are also subject to this attacker-defender race.

1.2 Side-Channel Attacks

Side-channel attacks have been studied to extract information from TEEs, circumventing all the protection mechanisms provided by the latter. Side-channel attacks exploit some property of the execution of a program on a certain architecture. Probably the best-known example of such a property is the power consumption that comes with the execution of a program and, more importantly, how it differs when the data the program operates on is different. Side-channel attacks that log power consumption are called power-analysis attacks [28][38]. Other side-channel attacks exploit the way in which some features of the architecture work, like the cache for example. The cache is an attribute serving to speed up memory accesses the processor needs to do for certain instructions. Cache-based side-channel attacks [37] exploit this feature, filling the cache for example and timing the execution of the program under attack to find out whether it addresses the cache or not. Examples of cache-based side-channel attacks are Flush+Flush [21], Flush+Reload [56] and Prime+Probe [32]. Other side-channels, like execution timing [29] and electromagnetic radiation [12] have also been exploited in order to attack enclaves. I this thesis, the Nemesis side-channel attack is referred to for parts of our study. A Nemesis attacker exploits the interrupt-handling mechanism of the CPU in order to leak latencies of individual instructions.

Recently, a new side-channel has been explored: functionality for Direct Memory Access (DMA) may be exploited to leak information on the execution of instructions in the processor [8]. With DMA, a peripheral may address memory without interrupting the CPU. Doing so in a clock cycle where the CPU addresses memory too leads to a latency incurred by that peripheral. Whether the peripheral incurs a latency or not, leaks control flow and, consequently, sensitive information if the control-flow depends on a secret.

Execution flow and secret-dependent branches

When a program executes, the *control flow* defines which instructions are executed, in which order and whether they are executed multiple times in a loop. Control flow of a program may consist of branches, emerging from statements in a program that make the program execute a piece of a code only when a certain condition is true or

false. Consider the program in listing 1.1. Whether the instruction that adds one to `x` is executed, depends on the condition in the if-statement. This program is said to *branch on the if-condition*, where one branch contains the increment-one instruction and the other branch the increment-two instruction. Both branches come together where further computations are performed on line 8. One of the branches in the program can also be empty, when no else-block is provided.

```
1 int x = 2;
2 if (some_condition) {
3   x+=1;
4 }
5 else  {
6   x+=2;
7 }
8 // further computations
```

Listing 1.1: A program with branches

A program can also use conditions in if-statements that operate on some secret, a password for example. If the condition on line 2 is a check whether the secret password contains some character, this condition is said to be *secret-dependent*. The branches that exist in the control flow of the program are then called *secret-dependent branches*.

The DMA attack may leak information on the control flow of a program, causing the attacker to possibly find out which branch in the program is executed. Moreover, if these branches are secret-dependent, leaking information on execution of one or the other branch leaks information on the secret.
The DMA has an impact on systems being remotely addressable over the Internet. DMA may be used when sending network packets to a system [13] and a multi-tenant setting may be vulnerable to this DMA attack. If a DMA attacker generates network packets to find out which instructions (and therefore which branches) are being executed by the CPU on the platform, secrets may be leaked from the environment. Not all environments are vulnerable to this attack. Running programs that do not contain secret-dependent branches [11] makes an environment resilient against this kind of attack. Also, this attack is mostly applicable to resource-constrained devices, where several accesses to memory can not be handled simultaneously. For example, Intel processors support concurrent memory accesses, so they are no victim to this attack and are out of the scope of this thesis.

1.3 Contributions

The research hypothesis of this thesis is that we can defend against the DMA attack altering the source code of the program at compile-time. Alteration refers to insertion of instructions into the source code, without modifying control flow, so that secret-dependent branches in the program cannot be distinguished using the DMA attack. The secret on which execution of one or the other branch depends is therefore protected: execution of both branches is the same to a DMA attacker. Concretely, the contributions of this thesis are:

- **Instruction classification** We classify instructions according to their memory access trace. An instruction's memory access trace is a time series over the length of its execution, indicating for every clock cycle of the instruction whether it accesses some of the three memory regions (data, program and peripheral memory).

- **A simulation toolchain** We describe a toolchain that simulates instructions on an open-source TEE for the purpose of classification. A script that outputs the memory trace class of an instruction and one that classifies several instructions is implemented for this thesis.

- **Dummy instructions** For each memory trace class found in this thesis, we provide a dummy instruction that may be used to insert into a program's source code in order to make secret-dependent branches exhibit the same memory access trace. Most classes are assigned one single dummy instruction. A few other classes have set of dummy instructions that may be used for aligning secret-dependent branches.

- **A new compiler pass** We describe and implement a new compiler pass (a transformation part of the compilation process) based on a previously implemented compiler pass. It inserts the found dummy instructions into the program so that secret-dependent branches are aligned and the program is hardened against the DMA attack. We report possible weaknesses of the described approach. We perform an evaluation in terms of code size and performance of the compiled program. Further, we show with a rough statistical analysis that a combination of this compiler pass and one that hardens a program against another attack is not appropriate to harden a program against both attacks.

- **Alternatives** We provide ideas for alternatives to the compiler pass investigated in this thesis. Other than the software-based approach in the form of a compiler pass, one could extend the hardware of the platform executing the program to mitigate against the DMA attack. On the one hand, a random signal in the CPU could be an alternative to our approach. On the other hand, a time-padding implemented in hardware might cover up the side-channel that was opened by adding DMA functionality to the platform.

Contributions in the form of implementation for this thesis can be found here:

- https://github.com/SteffieJoosen/llvm-project/tree/master/llvm/utils/TableGen/MemoryTraceGeneration
- https://github.com/SteffieJoosen/llvm-project/blob/master/llvm/lib/Target/MSP430/MSP430DMADefender.cpp
- https://github.com/SteffieJoosen/sllvm/tree/DMADefender/test/sancus (benchmark programs)

1.4 Outline

This thesis is structured as followes:

1. **Chapter 2** We provide background information in this chapter, including information on Trusted Execution Environments, side-channel attacks on them and the Sancus platform based on the openMSP430 architecture. We explain the Nemesis attack and the countermeasure that was taken against it, touching upon the SLLVM compiler framework. Lastly, the DMA attack is outlined.
2. **Chapter 3** We explain the main work of this thesis. To defend against the DMA attack, we use an approach similar to the one that was taken to defend against the Nemesis attack. To that end, we classify instructions, find dummies for the found classes and use these dummies to extend the SLLVM compiler framework in order to harden a program against the DMA attack.
3. **Chapter 4** We evaluate the defense explained in chapter 3. We discuss the amount of memory trace classes found, whereafter we use an example program to evaluate the defense based on performance and code size. We discuss some alternatives to our approach; like using runtime checks or giving non-deterministic priority to the CPU or peripheral to access memory. As a final note in this chapter, we show why a combination of the defense against Nemesis and the defense against the DMA attack is not appropriate to harden a program against both attacks.
4. **Chapter 5** Related work is outlined. Although a combination of compiler passes to defend against both Nemesis and the DMA attack does not work, one could combine the software-based defense from this thesis with the hardware-based defense against the Nemesis attack. Related research that could be applied to undermine our defense is also discussed.
5. **Chapter 6** A conclusion and suggestions to future work are given.

Chapter 2

Background

In the past decade, Trusted Execution Environments (TEEs) have been of increasing interest in the field of secure computing. Trusted Execution Environments are parts of the processor and related components that isolate programs to protect them from other possibly malicious programs running in the environment. On these TEEs also attacks have been discovered, the so-called side-channel attacks and we will take a closer look at some examples of these attacks in this chapter.

2.1 Trusted Execution Environments

Trusted Execution Environments are a mechanism to provide executing processes with a certain level of trust on the protection of their execution and the data they operate on. In their attempt to define a TEE, Sabt et al. [43] state that untrusted code should never be able to influence events in a TEE. They aggregate previous definitions from amongst others [3] and [16] that mention isolated execution and secure storage. Next to these two, attestation is a next property of trusted computing which comprises the verification of integrity and identity of the state of software running on a certain device.

Trusted Execution Environments come with their definition of a Trusted Computing Base (TCB). The TCB is the set of system components that can be trusted by a process executing in an enclave provided by the TEE. Implementations of trusted execution environments aim to keep their TCB small, even excluding the operating system from it. This means that compromising the OS should not affect a program running in a TEE, because the TEE is protected from it.

Cryptographic algorithms are a very common target to attacks, because they operate on secret keys that can be used to decrypt secret data. Therefore, evaluation of a TEE implementation generally involves cryptographic algorithms, as in [39], [43], [14], [31].

From industry, the probably best known implementations of trusted execution environments are ARM Trustzone [6] and Intel Software Guard Extensions (Intel

Addressing mode	Example operand
Register mode	r4
Indexed mode	2(r4)
Symbolic mode	EDE
Absolute mode	&EDE
Indirect register mode	@r4
Indirect auto-increment mode	@r4+
Immediate mode	#0x42

Table 2.1: MSP430 addressing modes

SGX)[51]. ARM TrustZone divides resources into the *Secure world* and the *Normal world* which have their own operating system with the responsibility to manage the resources in it. Intel SGX defines a new set of instructions for processes to build their enclave in which they can execute securely.

A TEE platform developed in the academic context rather than in industry is Sancus, which is elaborated on in the next section.

2.2 Sancus

The TEE that serves for experimentation in this thesis is the Sancus TEE [25], a program-counter based TEE with a Trusted Computing Base that only involves the platform's hardware. The architecture for Sancus is based on the openMSP430 microcontroller [20], the open-source version of the MSP430 architecture [48]. Sancus has facility for interrupt handling, an important feature that is exploited by the Nemesis attack (section 2.4). Also DMA capabilities were added to Sancus, which gave rise to the DMA attack, discussed in section 2.6.3.

2.2.1 MSP430

The openMSP430 architecture [20] for the Sancus platform is based on Texas Instruments' MSP430 architecture [48]. MSP430 defines 27 core instructions (appendix A) divided in three groups: single-operand instructions, double-operand instructions and jump instructions. For the operands, seven addressing modes are available, as in table 2.1. The first four modes in this table are available for the destination operand, while all of them can be used in the source operand. In this table, the EDE label stands for any memory address in the 64KB memory address space of openMSP430.

The MSP430 User Guide [48] explains the different addressing modes. Register mode works as expected and similarly to other architectures. An operand in indexed mode means the operand is to be fetched from memory, namely at the memory address that is in register r4 (in the example from table 2.1) added with the value two. Symbolic addressing mode an operand is to be fetched from memory address

EDE, but the address is calculated using an offset relative to the program counter. The memory address after the instruction using the addressing mode contain this offset, namely X = EDE - PC. When fetching the operand, the memory address is calculated as X + PC. When absolute mode is used, the word after the instruction contains just the value EDE. Indirect register mode results in the same semantics as when using an operand 0(r4). Indirect auto-increment mode works similarly, but the value in the register is incremented by one or two, depending on whether a byte operation (8 bits) or a word operation (16 bits) is used.

MSP430 instructions have latencies between one and six clock cycles and these fully depend on the combination of addressing modes used in source and destination operand. We say instruction latencies in MSP430 are deterministic, unlike latencies in for example Intel processors that make use of a wide range a performance-increasing techniques like speculative execution. This is out of the scope of this thesis, since we study the Sancus platform based on openMSP430.

2.3 Side-channel attacks

Trusted Execution Environments isolate running programs to protect their code and data to guarantee confidentiality and integrity [22], resulting in an enclave in which a program can securely run. Alongside TEEs, attacks on them have been studied, and these have gained interest over the past years, as well as defenses against them.

Side-channel attacks undermine the isolation guarantees provided by Trusted Execution Environments by exploiting microarchitectural properties of processors executing some program in an enclave. Microarchitectural properties of a processor are different from its architectural elements. Architectural state holds the state of a program and is defined by memory, registers used for the program and status registers in the processor [24]. Ronen et al. [42] spell out that microarchitecture on the other hand is the logic behind execution and has as responsibility that instructions are carried out in the best possible way conforming the needs of the users of the system. The CPU's cache for example is part of the microarchitecture, with which designers aim to increase performance when the CPU addresses memory.
Micro-architectures and the Instruction Set Architecture they implement (for example, the x86 ISA on Intel processors) together form the computer architecture as a whole and several microarchitectures may implement the same ISA [47].

Side-channel attacks can use the microarchitecture to exploit some side-channel coming forth from it to gain information out of a program executing in an enclave. A side-channel can be execution time of some operation, power-usage, the cache and many others. One of the most commonly known side-channel attack is for example the Flush+Flush attack on Intel SGX enclaves. It uses the fact that the `clflush` instruction on Intel processors takes more time to execute when some memory line

resides in the cache compared to when it does not [21]. Other side-channel attacks exploit the fact that different instructions differ in hardware power consumption and therefore are called power analysis attacks [38], which are commonly used on smart cards [28] [38].
Recent investigation [52] has established a side-channel attack that uses the interrupt mechanism of a CPU as a side-channel to reveal information on some secret in an enclave. The so-called Nemesis attack discovers timings of individual instructions by exploiting the interrupt-mechanism of the CPU. The attacker differentiates secret-dependent branches based on the instruction latencies discovered. More detail is given in the next section.
Next to Nemesis, another side-channel attack has recently been studied that uses functionality for Direct Memory Access (DMA) as a side-channel [8]. This attack learns the memory accesses a CPU performs and distinguishes secret-dependent branches based on that. In aforementioned attacks, differentiating between secret-dependent branches leads to information being leaked from the enclave, which of course should be avoided.

2.4 The Nemesis side-channel attack

Investigation has established that when a program contains secret-dependent branches, these branches should not differ in execution timing. Time-based side channel attacks [17] for example exploit different end-to-end execution timings of secret-dependent branches to gain information on the secret. To illustrate, consider the following program:

```
1  cmp #4, r13
2  jge 5f
3  mov r13, &pin     4 clock cycles
4  jmp 11f           2 clock cycles
5  nop               1 clock cycle
6  nop               1 clock cycle
7  nop               1 clock cycle
8  nop               1 clock cycle
9  nop               1 clock cycle
10 nop               1 clock cycle
11 ret
```

Figure 2.1: A program hardened against an end-to-end timing attack

It compares the value four with a register containing some secret value. It executes the instruction on line 2 and depending on the comparison from before, it either takes the conditional jump, executes the MOV and JMP instruction on lines 3 and 4 and returns or it executes the six NOPs before it returns. Both branches take 6

clock cycles and therefore, this program is hardened against an end-to-end timing attack, where the total execution time of secret-dependent branches is measured. The branches are secret-dependent, since whether they are executed depends on the secret value in r13, but they are indistinguishable to an end-to-end timing attacker. This program can nevertheless be successfully attacked by a Nemesis attacker using interrupt requests. MSP430 is a Von Neumann architecture [48], in other words: it inhabits a *fetch-decode-execute* operation. The instruction is fetched from memory, decoded by the CPU and then executed. Importantly, interrupt requests are only handled upon instruction retirement, meaning that when an interrupt is raised at some moment in time during this *fetch-decode-execute* process, the instruction that is being executed first completely finishes before the CPU handles the pending interrupt.
A Nemesis [52] attacker times execution times of individual instructions by exploiting this interrupt-handling mechanism and can therefore leak information on some secret if secret-dependent branches use instructions with different instruction latencies. Before we go deeper into how this is done, we outline the attacker model used when studying the Nemesis side-channel attack.

2.4.1 Attacker model

Examining the Nemesis attack and defenses against it, Van Buclk et al. [52] assume an attacker with access to the source code of the victim program, who is able identify where the program branches out based on some secret. Moreover, this attacker has full control over the operating system. This means that the attacker can configure timers and schedule interrupts. This is in line with for example the threat model used by Intel SGX [51], where the operating system is untrusted.

2.4.2 The attack

A Nemesis [52] attacker exploits the interrupt mechanism of the MPS430 architecture to attack the program in figure 2.1 as follows: if they generate an interrupt right after the conditional jump on line 2, they see a timing difference in how long it takes for his interrupt to be handled. If the jump was taken, his interrupt came at the very beginning of the first NOP instruction on line 5 and the interrupt is handled after one clock cycle, which is exactly the latency of the NOP instruction. Contrarily, if the jump was not taken, the interrupt came at the very beginning of the MOV instruction and the interrupt is handled after four clock cycles. Similar to what we saw in the case of a NOP, this waiting time is exactly the latency of the MOV instruction, the instruction being interrupted. Both branches are different to a Nemesis attacker due to the use of either a MOV or a NOP. Consequently, since execution of one or the other branch depends on a secret (the value in r13), information on the secret is leaked.
A Nemesis attacker abuses timing differences at instruction-level granularity in contrast to an end-to-end timing attacker, who times execution of whole branches, which is a much more coarse granularity.

2.5 Defending against Nemesis

Because the Nemesis attack exploits more granular timings than an end-to-end timing attack, the defense against it hardens programs at the same instruction-level granularity. SLLVM is a security-enhanced version of LLVM, an open-source compiler framework [33]. Part of SLLVM is functionality to align branches according to the timing of individual instructions, which is an effective defense against Nemesis on embedded systems with deterministic execution timings [54]. We first elaborate on the origin of SLLVM, the LLVM compiler infrastructure [33]. Then we explain the way in which SLLVM mitigates against the Nemesis attack with a compiler pass specifically written for that purpose.

2.5.1 LLVM

LLVM is a project providing compiler functionality amongst other toolchains (like TableGen, see section 3.2.3) [33]. LLVM defines *Basic Blocks*, which are sets of instructions that always execute sequentially [18]: there are no jumps into the middle of this block, no jumps out of the middle, no loops or conditional branches. We distinguish between entry basic blocks and exit basic blocks. The former are basic blocks that end in an instruction giving rise to a loop or conditional branch. The latter are basic blocks that start with an instruction that is executed by all control flows previously split in branches. LLVM builds a control-flow graph (CFG) of the program. Basic block form the nodes of this graph.
LLVM defines a number of compiler passes to analyze or transform code submitted for compilation. An LLVM Analysis Pass is a pass that does not change source code, but simply reads it and distracts some properties. An analysis pass for example that is predefined in the LLVM framework is for example the analysis is the `Find Used Types` pass, which prints out all types that the program uses [34], but does not change anything to the source code. Unlike analysis passes, transform passes alter the program like for example a pass that removes dead code. The third kind of LLVM passes are utility passes, which are neither analysis passes nor not transform passes. The `View CFG` pass for example does not analyze source code (the CFG has already been built) and it certainly does not change the source code.

Other than the examples of LLVM passes above that were all passes predefined in the LLVM Project [33], one can write a new LLVM pass as required [36]. This was done to define the defense against the Nemesis attack.

2.5.2 Hardening against Nemesis

SLLVM[53] is an extension of LLVM. It comprises a set of additions to the existing LLVM compiler framework, resulting in *Security Enhanced LLVM* [53], which was later renamed to just SLLVM. Amongst others, SLLVM defines a new operation on programs submitted for compilation, namely hardening them against the Nemesis attack. The Nemesis defense aligns instructions in secret-dependent branches accord-

ing to their timing and appropriately belongs to the class of alignment algorithms. Winderix et al. [54] define alignment as the process of transforming a program such that instructions of alternative execution paths, originating from the same branch and residing at the same distance from that branch have equal execution times or latencies. Alignment is done in three steps:

Equalising path lengths The program's CFG is converted to a functionally equivalent one so that all secret-dependent branches starting from the same entry block consist of the same number of basic blocks.

Computing level structure Assign to every basic block that is in a secret-dependent branch the distance to the entry block of the CFG region of this branch.

Aligning basic blocks For blocks at the same level in the level structure, make the blocks have the same *latency trace*.

Of interest in this thesis is the last step: aligning basic blocks. Here, a *latency trace* is the number of instructions a block has and the latency each individual instruction exhibits, in which the order of instructions is important. As an example, three basic blocks are shown figure 2.2. This picture was taken from the paper that introduces the Nemesis defense in SLLVM [54]. Here, instructions are denoted by a number that indicates their latency in clock cycles.

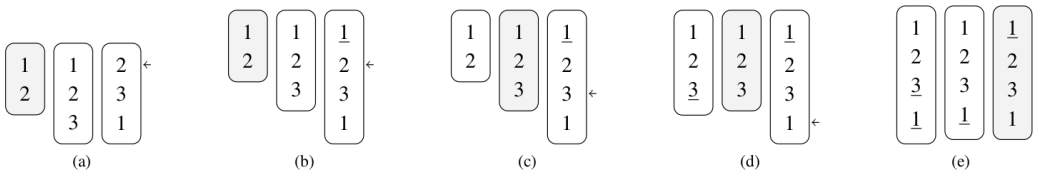

FIGURE 2.2: Alignment of basic blocks in the Nemesis defense of SLLVM

The process of alignment is an iteration over basic blocks. Per basic block, it iterates over the contained instructions and inserts dummy instructions (underlined) in the other basic blocks if for the current instruction (indicated by the arrow in figure 2.2) there is no instruction in the other basic blocks at the same height having the same latency. In figure 2.2a, the instructions in the shaded basic block are compensated for using a single dummy in the rightmost basic block, resulting in the set of basic blocks in figure 2.2b, where all first instruction have the same latency of one clock cycle. In this figure, all second instructions already have an equal amount of clock cycles, and since all instructions from the shaded block are covered, we move on to the next basic block, shaded in figure 2.2c. The process continues and the result (2.2e) is a set of aligned basic blocks.

Going back to the program in figure 2.1, we now see the program needs patching not just with NOPs, but with instructions resulting in the same latency trace for both branches. The branch starting with the MOV instruction and the branch

starting with a NOP both need an equal amount of instructions, where instructions at the same height from the start of the branch need equal latencies. SLLVM has a newly defined compiler pass to harden programs against Nemesis, which patches the program from figure 2.1 so that it results in the program in figure 2.3. At the top of the figure stands the assembly code resulting from the new compiler pass. Underneath it, the corresponding CFG can be seen.

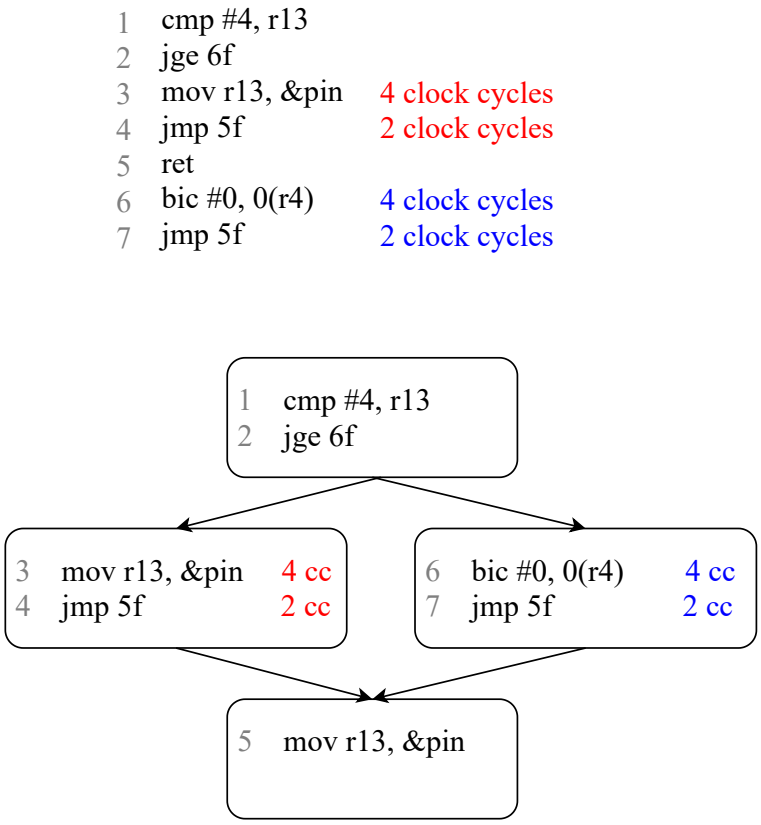

Figure 2.3: Program hardened against the Nemesis attack. 'cc' stands for 'clock cycles'.

The MOV and JMP instruction are always executed sequentially, so they reside in one basic block. The same holds for the block representing the BIC-branch. Both basic blocks are aligned: the first instructions in the blocks (MOV and BIC) both take 4 clock cycles and both the JMP instructions of course show the same latency too. We see that the alignment of basic basic blocks under influence of secret-dependent branches, results in branches that are indistinguishable to a Nemesis attacker.

2.6 The DMA side-channel attack

The Nemesis attack makes use of the interrupt capability of a processor and is applicable to the Sancus platform based on the openMSP430 microcontroller. The Sancus platform was recently extended with Direct Memory Access (DMA) capability [44] and that gave rise to a new side-channel based on DMA requests.

2.6.1 DMA functionality in openMSP430 and Sancus

OpenMSP430 is a microcontroller, meaning it is well-suited for IoT devices and in its design, a lot of attention is spent on keeping power consumption as small as possible. Direct Memory Access (DMA) functionality enables data handling without intervention of the CPU [49]. A peripheral can read from or write to memory and only notify the CPU when it is done transferring data. In this way, the CPU can stay asleep longer which saves power.

Memory regions MSP430 defines three memory regions: program memory, data memory and peripheral space. The CPU and the peripheral can access the full memory space. RAM and peripheral registers are mapped to address values, which means that when the CPU accesses an address, it may refer to a portion of physical RAM, or it can instead refer to memory of a peripheral. Program memory is mapped onto RAM or ROM (Read-Only Memory) [20].

Priority for accessing memory In openMSP430, the DMA master that serves as the controller connected with the DMA interface. The DMA master controls a signal that indicates whether DMA reads or writes to memory should have priority over the memory accesses of the CPU. However, the TCB of Sancus does not include this DMA master, so it cannot be trusted. To safeguard Sancus from Denial of Service attacks[1], Sergio Seminara decided to let the CPU always have priority over a peripheral on the Sancus platform[44].

Accesses to different memory regions can be handled in parallel. The priority decision is needed when a peripheral and the CPU want to access the same memory region in the same clock cycle. Whenever such a situation arises, the peripheral has to wait. Accessing memory takes one clock cycle, so that will be exactly the delay seen by the peripheral issuing the DMA request.

2.6.2 Attacker model

The DMA side-channel attack [8] assumes an attacker who controls peripherals and is capable of scheduling DMA requests issuing them with these devices. Peripherals therefore lie outside of the trusted computing base (TCB). With a cycle-accurate clock and knowledge of the source code also part of the attacker's resources, an attacker can schedule DMA requests precisely and time whether or not it incurs

[1] If the peripheral were to access memory in bulk with a high priority, the CPU would be denied access to memory for an indefinite amount of time.

a delay in the DMA request being served. Since the CPU is given priority over a peripheral when both parties request a memory access at the same time, the peripheral has to wait one clock cycle longer than normally for its request to be served. The attacker does not have physical access to any component in the TCB.

2.6.3 The attack

Looking back at the program that is compiled by SLLVM (figure 2.3), we see that two different instructions are used: a MOV in one branch and a BIC in the other. Depending on the secret value in r13, the CPU executes one or the other instruction. A DMA attacker can distinguish between the two branches. Before we explain why, the notion of a *memory access trace* is required.

Memory access traces As said in section 2.6.1, three different memory regions are accessible and from section 2.2.1, we know that instructions can take one to six clock cycles. A memory access trace takes into account these two aspects. Consider for example the MOV instruction from our program in figure 2.3. Its memory access trace (or memory trace) is shown in figure 2.4. The numbers at the top of the figures represent the clock cycle and the figure represents a time series over the execution of the `mov r13, &pin` instruction. The `DATA` label stands for any addressing pointing to data memory. Now it can be seen that this instruction executes in four clock cycles, never accesses peripheral memory, accesses data memory in the last clock cycle and program memory in the first and last clock cycle.

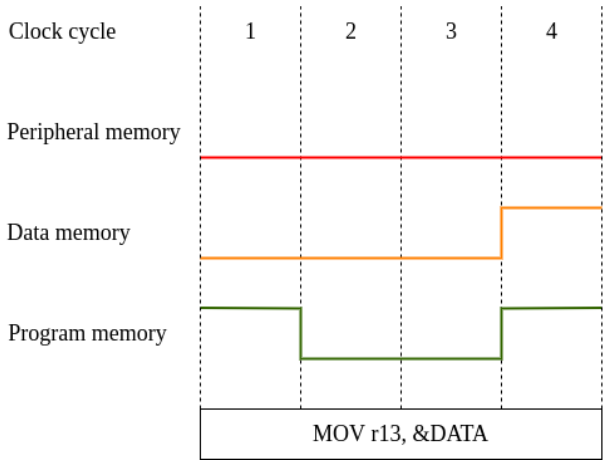

Figure 2.4: Memory trace for the MOV instruction

The example program uses the MOV instruction in one branch. The other branch contains the BIC instruction. The memory access trace for the BIC instruction is given in figure 2.5

From both traces, we can see that the two instructions differ in their second clock cycle, since the BIC instruction accesses data memory in that cycle and the MOV

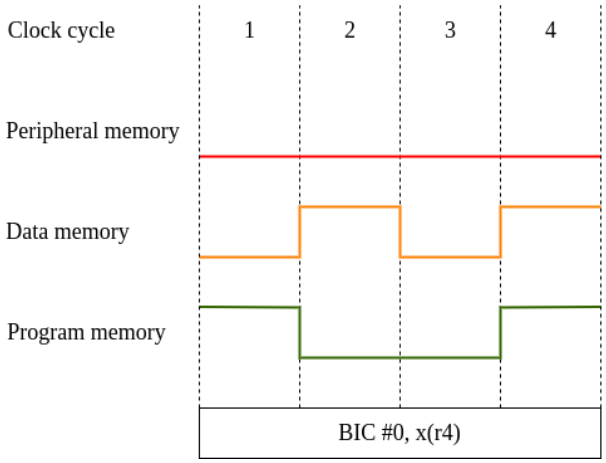

FIGURE 2.5: Memory trace for the BIC instruction

instruction does not. As explained in section 2.5.2, figure 2.3, these instructions are indistinguishable to a Nemesis attacker. However, to a DMA attacker, they are different. If the CPU executes the first secret-dependent branch (with the MOV instruction), it accesses data memory in the second clock cycle of the MOV. If the other branch was taken, this access to data memory is not carried out. If a DMA peripheral were to address the data memory region in this exact clock cycle, it might have to wait, depending on whether the CPU wants to access data memory. If it does, it knows the BIC instruction was executed. A DMA request then takes one extra clock cycle for the peripheral. If the peripheral is granted access to memory right away, the CPU did not access data memory in this clock cycle and the MOV instruction was executed. The secret-dependent branches therefore are distinguishable in the case of a DMA attacker.

2.7 Timing DMA requests

The Nemesis attack employs the ability of the adversary to time interrupt requests at a well-defined moment. The openMSP430 architecture comes with a timer that generates interrupts when it expires. An attacker knows what value to give the timer, because as we know from section 2.2.1, MSP430 instruction latencies are deterministic. Since the attacker model spells out that the attacker has access to source code, setting the timer comes down to looking at the source code and adding up the latencies of executed instructions. The same holds for a DMA attacker. Since the source code is known, the attacker knows exactly after how many clock cycles one of the instructions of interest (which have different memory traces) is executed.

2.8 Conclusion

Side channel attacks leak microarchitectural information on execution of a program. Since a few years, the Nemesis side-channel attack on the openMSP430-based Sancus TEE was studied. It exploits the timing difference in secret-dependent branches at instruction-level granularity. The latency of individual instructions is revealed by interrupting the program at well-chosen times, and therefore the interrupt mechanism of the CPU serves as a side channel.
SLLVM is a compiler infrastructure that mitigates against this attack. It aligns instructions and makes sure instructions on the same level produce the same latency, so a Nemesis attacker cannot distinguish them. This is done for all secret-dependent regions in a program (loops, if conditions ...) and results in a program where the secret-dependent branches are identical to a Nemesis attacker, making it resilient against the Nemesis attack.
Adding functionality for Direct Memory Access (DMA) to a microcontroller may result in reduction of its poweer consumption. However, adding DMA functionality of the openMSP430 architecture used for Sancus gave rise to a new side-channel attack. The DMA attack abuses the difference in memory access traces of instructions. These traces represent how different instructions access memory. Issuing a DMA request at a well-timed moment, an attacker can recognize which instruction was executed and can therefore differentiate secret-dependent branches using different instructions. Although these branches might exhibit the same latency trace, the fact that in this case, their memory trace is different, gives rise to the described vulnerability.

Chapter 3

The extended SLLVM

We have introduced the Nemesis attack and how the SLLVM compiler could be used to defend against it. SLLVM was used to harden programs against interrupt side-channels by balancing out secret-dependent execution paths with dummy instructions so that these paths become indistinguishable to a Nemesis attacker, who observes interrupt latencies. After SLLVM, the DMA side channel attack was presented, in which a DMA attacker can observe the pattern of memory accesses that the CPU performs. To harden against this attack, we need to determine these memory access patterns for instructions whereafter dummy instructions can be used to balance out secret-dependent execution paths.

3.1 Defending against the DMA attack

By extending SLLVM, we want to be able to mitigate against the DMA side-channel attack as described in section 2.6. It exploits the fact that different instructions access memory in different clock cycles. In this section, we outline the attacker model and afterwards explain on a high level how the above mentioned differences can be exploited.

3.1.1 Attacker model

Of course, when studying a defense against the DMA side-channel attack, we use the attacker model as described in the investigation of the attack itself [8]. We assume an attacker who is capable of issuing DMA requests using a peripheral which he connects himself or that is compromised to issue the requests on behalf of the attacker. On the Sancus platform, requests for memory that come from such a peripheral are given a lower priority than the requests issued by the CPU. The memory controller is responsible for this and whenever it is queried by the CPU and a peripheral at the same time, it serves the CPU first. Next to DMA capabilities, the attacker has a cycle-accurate clock. Memory accesses take one clock cycle and issuing a request for memory in the same cycle as the CPU results in a latency of one clock cycle which can be measured with a cycle-accurate clock. Knowledge of the source code executing

in the enclave is a third assumption in the attacker model in [8]. This allows an attacker to point out the interesting clock cycles where to issue DMA requests in order to follow secret-dependent execution, but alternatively a lack of knowledge of source code does not hinder the attacker in breaking isolation provided by the enclave [8]. Generating for example a DMA request in every clock cycle during execution of the program allows the attacker to log the memory accesses performed by the CPU over the execution of the program. Running the program several times and comparing differences between the memory accesses performed might leak sensitive information on execution flows.
Conforming to the attacker model used for the Sancus enclaves, the attacker has no physical access to any component in the Trusted Computing Base (TCB). The TCB is the set of components we assume can be trusted (are not tampered with) and the attacker therefore has no physical access to the memory controller or cannot access the memory region of the enclave program.

3.1.2 The attack

As mentioned earlier, instructions differ on their memory accesses in different clock cycles. We say that there is a difference in *memory access traces* of instructions. A memory access trace indicates whether an instruction accesses memory and when exactly it does so during execution of the instruction. When secret-dependent branches use instructions with a different memory access trace, they can be distinguished and information on the secret is leaked.

To illustrate, consider the program in figure 3.1. The example is based on the keypad program from the benchmark programs for SLLVM [23] and was used earlier in this thesis to explain the defense against Nemesis (section 2.5). It compares the value four with whatever value is in register r13. Assume the value in r13 is sensitive information. If four is greater than or equal to the value in r13, the program jumps to line 6 and continues with the BIC and JMP instruction, whereafter it returns (line 5). Otherwise, the program does not take the jump and executes the MOV and JMP instruction before returning. This program is resistant against a Nemesis attack, since the branch consisting of the MOV and JMP instruction has the same latency trace as the branch consisting of the BIC and JMP instruction. However, figures 2.4 and 2.5 show different memory traces for the MOV and BIC instruction. Whether the BIC or the MOV instruction is executed depends on a secret, namely the value in r13.
Since the two instructions differ in their memory trace classes, a DMA attacker who is capable of measuring memory traces of instructions can discern information on this secret value in r13, namely whether or not it was greater than four.

3.1.3 The Defense

To harden this program against the DMA attack, we need to make the branches identical according to their memory traces by inserting dummy instructions that exhibit the desired memory trace. SLLVM is a valid solution for this. SLLVM is

```
1  cmp #4, r13
2  jge 6f
3  mov r13, &pin     4 clock cycles
4  jmp 5f            2 clock cycles
5  ret
6  bic #0, 0(r4)     4 clock cycles
7  jmp 5f            2 clock cycles
```

FIGURE 3.1: A program hardened against Nemesis together with memory access traces for each instruction

a compiler framework that implements one compiler pass that hardens programs against the Nemesis attack. It inserts dummy instructions to align secret-dependent branches. If we let a similar compiler pass use dummies with a certain memory trace, it serves as a valid defense against the DMA attack at compile-time. The dummy instructions should be chosen so that they generate memory traces that make the secret-dependent paths identical according to their memory access traces.

To find the memory traces that are needed for these dummy instructions, we search for the memory access traces of all instructions. Some instructions have the same memory trace and we can group them in classes, resulting in *memory trace classes*. Having found these memory trace classes, a dummy instruction is needed that exhibits the same memory trace as all the instructions in the class while not affecting a program's logic in any way other then the memory access pattern and execution timing of the program when inserted into the source code.

In the next section, we explain how the memory trace classes can be found using simulation of instructions. Later, we move on to finding dummy instructions for the defined memory trace classes.

3.2 Classification

Instructions exhibit a certain memory trace and some instructions produce equal traces. According to these memory traces, we want dummy instructions with appropriate memory traces. When inserted into the source code, these dummies then cause for secret-dependent branches to have identical memory traces. Instructions with identical memory traces can be coupled with the same dummy and that is why we group such instructions into memory trace classes. To find all such classes, we need to discover the memory access traces of all instruction instances that can come forth on the MSP430 architecture.

3.2.1 Instruction instances

The MSP430 architecture defines 27 opcodes. In our approach, it does not suffice to generate one instruction for some opcode. There are seven addressing modes available in MSP430, all of which can be used in the source operand and four of which are possible for the destination operand. With these, the complete address space is addressable, which consists of the three regions discussed: program memory, data memory and peripheral memory. Given all the above, there are a lot of combinations to generate an instruction, which result in a lot of possible instruction *instances*, being unique by their

- opcode;
- addressing mode for source operand;
- addressing mode for destination operand;
- region of memory addressed by source operand;
- region of memory addressed by destination operand.

An `ADD` instruction with both operands in register mode, like

```
ADD r4, r5
```

for example is different from

```
ADD r4, &EDE
```

which is again distinct from

```
MOV r4, &EDE
```

As denoted in the list above, the value of the operands also makes a difference in their memory traces. In the last MOV instruction above, the operand `&EDE` means that the operand is to be fetched from memory, more specifically from address EDE. The memory access trace of this instruction when the destination operand `&EDE` points to data memory might be different from a trace when the destination operand points to program memory.

The instruction-trace pairs in table 3.1 can be found with our simulation and they show different memory access traces for different values of one of the operands. The first instruction was simulated with as destination operand a pointer to peripheral memory, the second with an operand pointing to data memory and the third with one referencing program memory. The traces that were found differ from each other just by changing the value of the `&EDE` operand. Consequently, we need to simulate every instruction with operands pointing to all different memory regions.

Having established that memory traces can differ depending on the opcode of the instruction, addressing modes and values of the source and destination operand, we need to find memory traces for all instruction instances, where an instance is defined as a combination of a specific opcode, addressing mode combination and combination of operand values, resulting in a certain combination of memory regions (program, data and peripheral memory) addressed.

Instruction	Memory access trace
mov r4, &EDE	4 | 0001 | 0000 | 1001
mov r4, &EDE	4 | 0000 | 0001 | 1001
mov r4, &EDE	5 | 00000 | 00000 | 10011

TABLE 3.1: MOV with different values for destination operand

3.2.2 Memory trace classes

In Chapter 2, we already touched upon memory access traces. Memory access traces or memory traces visualize the way in which instructions are executed. An instruction takes a certain amount of clock cycles to execute, and in each clock cycle, this instruction may or may not address memory. Remember the MSP430 architecture defines three memory regions: peripheral memory, data memory and program memory [20]. We must therefore define a memory access trace in terms of each of these regions. For example, the memory trace for the MOV instruction in figure 2.4 (page 16) states that this MOV takes four clock cycles, never accesses peripheral memory, accesses data memory in the last clock cycle and addresses program memory in the first and last clock cycle.

Some instructions generate the same memory access trace. An ADD instruction like the one in figure 3.2 has as memory access trace that is the same as the memory access trace for the BIC instruction in figure 2.5 (page 17).

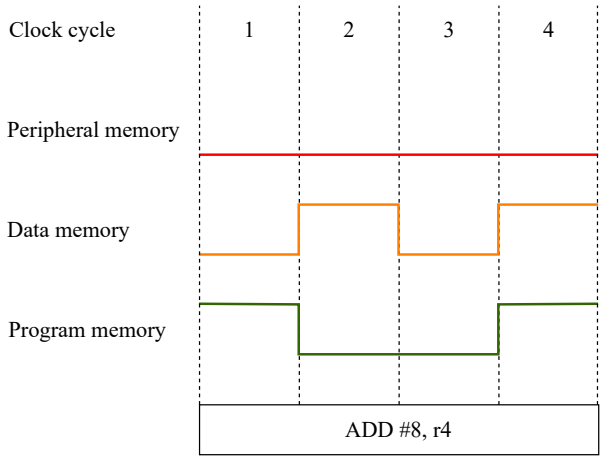

FIGURE 3.2: Memory trace for an ADD instruction that reads a constant from program memory and adds it to a register

We will call a set of instructions exhibiting the same latency and memory trace a *memory trace class*. We denote a memory trace class as a set of three tuples of bits, prepended by a decimal number. The leading decimal digit indicates the number of clock cycles an instruction takes and the following three sets of bits each consist

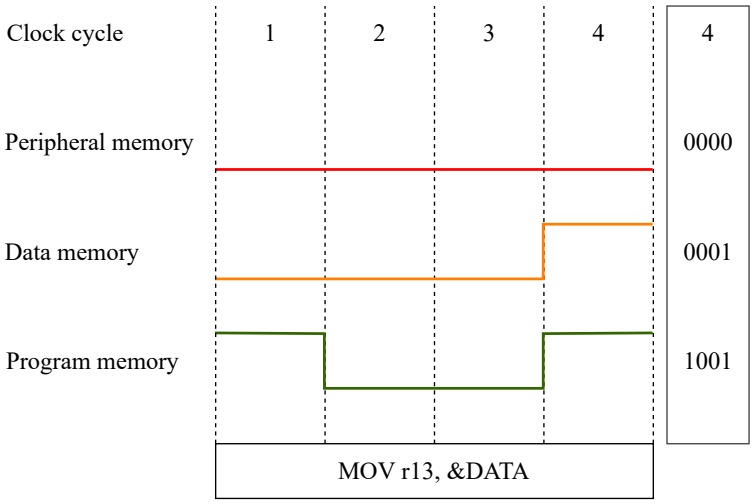

Figure 3.3: Memory trace for a MOV instruction that reads from a register and stores to data memory. On the right, bit strings were added that represent each of the three signals for the accesses to the memory regions

of exactly the amount of bits that this first number specifies. The three bit strings indicate whether the instruction accesses the corresponding memory region (1) or not (0), the corresponding memory regions being peripheral memory, data memory and program memory, in that order.

For example, the MOV instruction in figure 3.3 has the following class:

$$4 \mid 0000 \mid 0001 \mid 1001$$

We include the instruction's latency as a decimal number in the class because we will need it when discussing this defense in terms of resilience against Nemesis in section 4.5.1. We extended figure 2.4 with the components that make up the memory trace class of this MOV instruction, resulting in figure 3.3. The instruction takes four clock cycles to execute, so that is the first number making up the class. Thereafter, bit strings are used to indicate the three following signals. When an instruction accesses some memory region, the signal corresponding to that region is high, which results in a '1' in the bit string. This MOV instruction never accesses peripheral memory, the signal is low for all four clock cycles, and therefore the bit string for this memory region contains only zeros. The same is done for data and peripheral memory, using a '1' in the bit string when the signal for the memory region is high.
The described approach leads to the strings on the right of figure 3.3, and we conclude this instruction has class 4 | 0000 | 0001 | 1001 which is the same class as the one for the BIC instruction in figure 2.5.
Now that we have defined memory trace classes and given the memory traces for the MOV and BIC instructions from section 2.6.3, the program in listing 3.1 can now be denoted as:

```
1  cmp #4, r13        1 | 0    | 0    | 1
2  jge 6f             2 | 00   | 00   | 11
3  mov r13, &pin      4 | 0000 | 0001 | 1001
4  jmp 5f             2 | 00   | 00   | 11
5  ret                3 | 000  | 100  | 000
6  bic #0, 0(r4)      4 | 0000 | 0101 | 1001
7  jmp 5f             2 | 00   | 00   | 11
```

Listing 3.1: The program from figure 3.1, annotated with the memory trace of the instructions

This is the notation we will use throughout this chapter.

Instruction versus combined memory traces

In the above program, two different kinds of memory traces can be distinguished: instruction memory traces and combined memory traces. Instruction memory traces are denoted on the right, next to the corresponding instructions. Combined memory traces are memory access traces of several instructions that are executed sequentially. The memory trace of a sequence of instructions can now be denoted as the sum of all the memory traces of its component instructions, where the sum of memory access traces is defined as a memory access trace having as first digit the sum of the digits of the operand traces and as bit strings the concatenation of the bit strings of the operand traces. For example, the sum of the traces of the instructions on line 6 and 7 of the above program is:

$$4 \mid 0000 \mid 0101 \mid 1001 \; + \; 2 \mid 00 \mid 00 \mid 11 \; = \; 6 \mid 000000 \mid 010100 \mid 100111$$

It is clear from this calculation that the leading number indicating the total latency of the memory trace is redundant. However, this definition of a memory access trace might be a good starting point when a different definition of a trace is necessary, one that indicates the individual instructions making up this memory trace. Addition of the two above traces could in that case be defined as

$$4 \mid 0000 \mid 0101 \mid 1001 \; + \; 2 \mid 00 \mid 00 \mid 11 \; = \; 4.2 \mid 000000 \mid 010100 \mid 100111$$

A trace like this might be needed to define traces that are indistinguishable to both a Nemesis attacker, who times individual instruction latencies, and a DMA attacker, who sees memory accesses without discerning between individual instructions. Why the first calculation in this paragraph is insufficient to that end, is explained next.

Vulnerability to a Nemesis attack

For the bit strings in the traces, the order is important because concatenation is not commutative. Notice however that the latencies at the front of the traces are combined with integer summation, which, unlike concatenation, is commutative.

Since we have defined a combined memory trace as above, we do not consider the fact that different instruction latencies can be seen by a Nemesis attacker. Indeed, the attacker model used when studying the DMA attack and a defense against it as described in section 3.1.1 portrays an attacker with his most important capability being able to issue DMA requests and consequently mapping memory accesses of the CPU. A DMA attacker does not know which memory accesses belong to which instruction. A Nemesis attacker has interrupt capability leading to the power of timing individual instructions. Using commutative summation for the instruction latencies in the memory access traces makes explicit the difference between the attacker model from the Nemesis attack and the attacker model used in the DMA attack, which will be discussed more deeply in section 4.3.

Instructions show memory traces, as do branches that consist of these instructions. Instructions belong to a certain memory trace class according to their memory traces. In other words, the given that some instruction belongs to a certain class means that it has a memory trace that is exactly its memory trace class. The next thing illustrated is then how to find the memory access trace for any instruction.

3.2.3 Simulation

By executing instructions in a simulator of the Sancus processor, we can generate a memory trace for them. To exhaustively generate all instructions under consideration in this step (for this thesis, all MSP430 instructions), we use TableGen (section 3.2.3) to generate all MSP430 instructions with their addressing modes after which a simulation on the Sancus platform results in memory access traces. Then we can advance to classification of the captured instructions. With the results, we can proceed to finding dummy instructions for the classes found during simulation.

The Sancus simulator

The simulator provided by Sancus [1] is an interpreter for verilog code. Sancus comes with verilog code for its hardware design, defining all signals in the processor. Sancus-sim simulates this design, and outputs time series recording the states of all these signals in a VCD (Value Change Dump) file that can be interpreted in a wave viewer like GTKWave [4]. To simulate the `MOV` instruction in our running example from figure 3.1, we generate a small C program containing it. Using a Makefile as provided by the Sancus simulation framework [26]. With the described simulation, we can generate the memory trace of one simulated instruction. Using TableGen, we can find such a trace for all possible instructions. The process of simulating and classifying instructions is denoted in figure 3.4.

TableGen

TableGen [35], a tool that is part of the LLVM compiler framework [33], allows to create records of domain-specific information and operate on them. In this way, TableGen provides a description of the MSP430 platform: it provides abstract data types for 8 bit and 16 bit MSP430 instructions, taking the source and destination

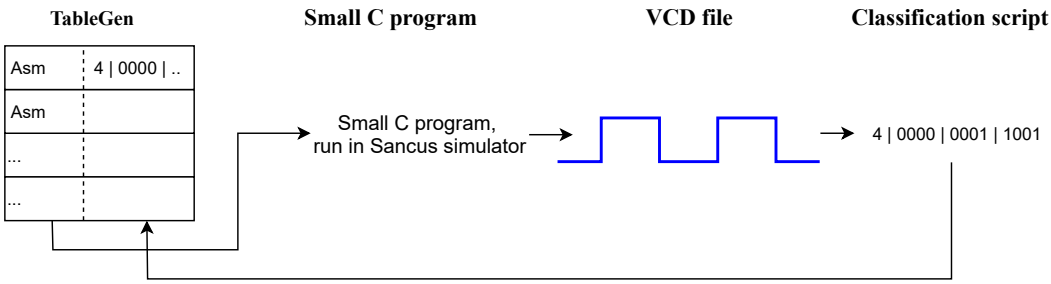

FIGURE 3.4: The simulation-classification toolchain used to classify instruction according to their memory access traces. It is explained in sections 3.2.3 and 3.2.4

Instruction	Memory access trace
mov r4, r5	1 | 0 | 0 | 1
mov @r4, r5	2 | 00 | 10 | 01
mov 2(r4), r5	3 | 000 | 010 | 101
mov r4, &EDE	4 | 0000 | 0001 | 1001
mov #N, &EDE	5 | 00000 | 00001 | 11001

TABLE 3.2: Traces after simulating MOV instructions with different addressing modes

addressing modes into account. Writing a backend for TableGen allows to generate a table of all kinds of assembly instructions that can come forth on an MSP430 platform. After classifying these instructions (as described in section 3.2.4), we extend the TableGen backend to generate not only the instructions, but also the memory trace class this instruction belongs to, making it generate a table with both assembly instructions and their memory trace classes. Using the resulting generated table, SLLVM can then find the class of any instruction encountered in a program that needs patching. TableGen is open-source and part of the LLVM implementation.

3.2.4 Classification

Simulation on the Sancus platform results in a VCD file with the states of all signals in its processor. To classify our MOV instruction from program 3.1, this generated VCD file is analyzed. This file is parsed with `vcdcat` [7], a tool that selects certain signals from a VCD file and outputs them in a easy-view format, and from this, we discover the memory access trace for an instruction, which is its memory trace class as explained previously. Simulating several MOV instructions with different addressing modes, we find the memory access traces as in table 3.2. All memory operands in this simulation were made to point to data memory.

From the table, consider instruction instance `mov r4, &EDE`. The `&EDE` operand stands in absolute addressing mode and as explained in section 2.2.1, the operand is in that case fetched from memory address EDE. In table 3.2, EDE is a generic

label that stands for any address pointing into data memory. When simulating an instruction that addresses a memory location in data memory, we find that this particular MOV instruction instance takes 4 clock cycles to execute, never addresses peripheral memory, accesses data memory in the last clock cycle and accesses program memory in the first and last clock cycle. It therefore has memory access trace

$$4 \mid 0000 \mid 0001 \mid 1001$$

which is also the memory trace class of this instruction. The addressing modes of source and destination operand are exactly the modes that are used in the MOV instruction of program 3.1, and the value of the `&pin` operand is an address pointing to data memory. `pin` is a global variable in the program that this code piece is part of and global variables reside in the data section of a program, which corresponds to the data memory region.

3.2.5 Evaluation of the classification

Using the procedure as described above, the classes listed in appendix A were found. We included the classes for the ADD, BIC and MOV instruction for the double-operand instruction. Instances of the ADD, BIC and MOV instruction taking up to three clock cycles end up the same class. This is valid for all opcodes, so double-operand instructions of maximum three clock cycles all end up in the same memory trace class. They all have operands in register mode (e.g. `r4`), indirect register (e.g. `@r4`) or indirect auto-increment mode (e.g. `@r4+`. Next, with only the source operand in either indexed (e.g. `2(r4)`), symbolic (e.g. EDE) or absolute mode (e.g. `&EDE`), only one class is found for all double-operand instructions. However, whenever the source operand is in one of these modes, the MOV instruction falls out of the class that ADD and BIC are in. In general, the MOV instruction accesses data memory one time fewer than other double-operand instructions. All other opcodes requiring two operandS always fall in the same class as the ADD and the BIC instruction.
Using a constant that is in {-8, -4, -2, -1, 0, 1, 2, 4, 8} in the source operand generates no extra access to program memory, because these constants are generated by the constant generator from the MSP430 architecture. Instructions with constants from this set therefore fall in the same class as instructions using a register in the source operand. Using constants outside the above set in the source operand generates an extra clock cycle in which program memory is accessed, because the compiler typically puts constants in program memory. For example, the `mov #0x0008, 2(r5)` is in the same class as the `mov r4, 2(r5)` instruction, namely 4 | 0000 | 0001 | 1001, but the `mov #0x0045, 2(r5)` takes one clock cycle more and accesses program memory once more. Its class is 5 | 00000 | 00001 | 11001. Single-operand instruction also tend to fall in the same memory trace class, except for the PUSH and POP instructions. A more extensive evaluation of the instruction classification is given in chapter 4.

3.2.6 Exhaustiveness of classification

TableGen provides us with an exhaustive enumeration of the instructions that can come forth on an MSP430 microcontroller. It generates instructions having every possible opcode and combination of addressing modes for source and destination operand. When an operand addresses memory, we simulate and classify the instruction for each of the three memory regions that could be pointed at by the operand (program, data and peripheral space).

Given the above and the list of properties by which an instruction instance is defined (described in section 3.2.1), we know that we can find all possible classes that can possibly be elaborated by MSP430 instructions.

3.3 Dummy instructions

In the previous section, we explained why we can find all possible classes with our simulation framework. We now need a dummy instruction for each class to extend SLLVM to insert them at the correct places to harden a program against the DMA attack.

3.3.1 One dummy per class

A dummy instruction is an instruction that does not affect the intended semantics of the program, takint into account the constraints upheld by LLVM. An example is an instruction that moves the value in a register to the same register. The program in figure 3.1 can be visualized as in figure 3.5. The possible execution paths can be easily followed in this representation. The program has two secret-dependent branches: depending on the outcome of the CMP instruction in the first block, the branch consisting of the MOV and JMP instruction is executed or the branch containing the BIC and JMP instruction is executed.

At present, the branches exhibit different combined memory access traces, because the MOV and BIC instruction in the opposing blocks have different memory traces. But, for a program to be resilient against the DMA attack we need the branches to be identical in this respect, meaning that they present the same combined memory trace over the whole secret-dependent branch. The defense against the Nemesis attack (sections 2.4 and 2.5) uses the BIC instruction to defend against Nemesis, but this instruction does not suffice to make the branches identical according to their combined memory trace. For the DMA defense, we therefore need dummies with a very specific memory trace, for which we give an example in section 3.3.2.

In the above program, we also see that some instructions have the same memory access trace, for example the JGE and the JMP instructions. These instructions therefore belong to the same memory trace class. If we ever seek to find a dummy for one of these instructions, we could use the same dummy instruction that is used

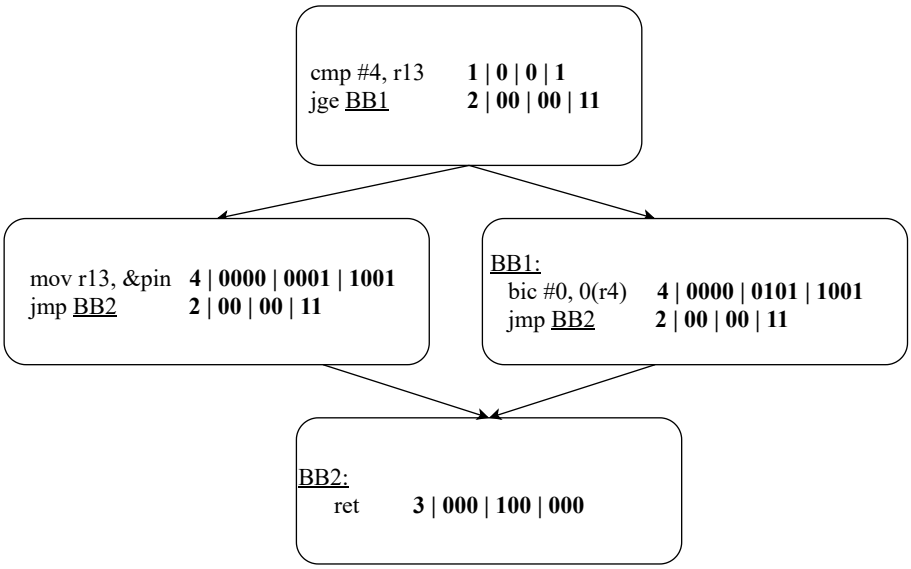

Figure 3.5: The program from figure 3.1, visualized using basic blocks that make up each execution path

for each of the JMP or JGE instructions. As a consequence we see that we only need one dummy per memory trace class.

3.3.2 Finding dummies

Coming up with a dummy instruction for a certain memory trace class requires some creativity. Starting from the addressing modes and the opcode, we need an instruction that has no effect on the program, but generates the memory trace for the correct class. As can be seen in appendix A.2, several different opcodes can be part of the same class, which means that an opcode for which not necessarily a dummy can be found, can be perfectly patched with a dummy having some other opcode for which a dummy is more straightforward, as long as it belongs to the same memory trace class.
What can also be seen in the found classes in the appendix, is that belonging to a certain class highly depends on the addressing modes of source and destination. This seems logical, as these classes influence the accesses to memory.

The `MOV r4, &pin` instruction from our example belongs to class 4 | 0000 | 0001 | 1001. A possible dummy instruction for this class could be `MOV #8, 2(rDummy)`, where `rDummy` is a register that we allocate to use it for dummy instructions with an operand in indexed mode. How and when we allocate this register, is explained in section 3.4.4. In this thesis, a dummy instruction for every discovered memory trace class was found and a full mapping of instruction classes to dummy instructions is in appendix A.

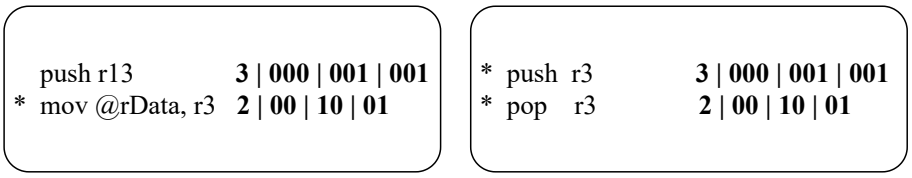

FIGURE 3.6: A PUSH instruction is patched using three dummy instructions: a PUSH, a POP and a MOV instruction

3.3.3 Memory trace classes without a single dummy

If we can find a dummy for every found class, we can compensate any MSP430 core instruction and the extended SLLVM will give a complete hardening. Emulated instructions are replaced by the appropriate core instructions at compile time [48]. It therefore suffices to find dummies for core instructions. However, not for every class a single dummy instruction can be found. The PUSH instruction falls in a memory trace class on its own, namely

$$3 \mid 000 \mid 001 \mid 001$$

when the operand is in register mode (e.g. `push r4`). No other instruction falls under this memory trace class, so the only possible dummy for a PUSH instruction is another PUSH instruction, which changes the stack. To undo this change (making sure that the stack is as it would have been without this dummy PUSH instruction), we insert a POP instruction right after it into the program. After that, we patch against the POP instruction in the original basic block. For example, if some basic block in a secret-dependent branch looks like the leftmost block in figure 3.6, it is patched with the PUSH and POP instruction in the rightmost block in the figure. Then, to compensate for the POP instruction, a dummy MOV instruction is inserted in the leftmost basic block. Dummy instructions are annotated with an asterisk. For most classes, we can find a single dummy (appendix A). If we need two dummies, like the PUSH and POP combination, we follow the approach as described above and therefore use three dummies instead of one.

For classes that contain instructions with an operand in indexed mode (e.g. `2(r4)`), we need a dummy that also uses an indexed operand. But to make it access the correct memory region, we need to reserve the register for the indexed operand of the dummy instruction. We explain more in the next section.

3.3.4 Allocating registers

For some dummy instructions, one of the operands is in indexed mode (e.g. `2(r4)`). Because a memory trace class depends on the memory region that is accessed, we need this operand to point to a well-chosen memory address, namely one in the same memory region as the region that is accessed by the instruction we want to patch with

this dummy. Therefore, we need to allocate a register to make it contain a value that, when the register is used in indexed mode, the operand using it points to the desired memory region. We therefore check for secret-dependent branches which register they do not use in the component instructions and use that register in dummies when an operand in indexed mode is needed. If we cannot find such a register, the compiler could use the absolute addressing mode in the dummy instruction, but using dummies with an operand in absolute mode has not been investigated in this thesis. More on how registers are allocated is explained in section 3.4.4.

3.4 Extending SLLVM

SLLVM [53] is the LLVM compiler framework [33] extended with security functionality and has also been used to defend against the Nemesis attack [52], as explained in section 2.5.2. An important aspect of providing Nemesis resistance is inserting dummy instructions to make secret-dependent branches indistinguishable to an attacker who is capable of interrupting a program and determining instruction-granular latencies. When a program's execution flow branches based on some secret, differing instructions latencies can leak information on the secret. The defense against the DMA attack investigated in this thesis uses the dummy-inserting method in a similar way. We have found memory access traces of instructions and of branches (section 3.2.2). We have also found dummy instructions to insert into programs in order to make their secret-dependent branches generate the same memory access trace (section 3.3.2). In this section, investigation is continued on how the approach of SLLVM to defend against Nemesis can be used to defend against the DMA side-channel attack, inserting the dummy instructions as found in section 3.3.2.

3.4.1 The Nemesis Defender Pass

LLVM is a compiler framework that works with compiler passes. An LLVM pass either traverses the source code of a program under compilation to gain information or transforms the program[34].
For the Nemesis defense, a new compiler pass was developed. The *Nemesis Defender Pass* falls under the class of alignment algorithms[54], which build a Control Flow Graph (CFG) of the program under consideration in order to perform alignment. The CFG has as nodes Basic Blocks, which are sequences of instructions that always execute sequentially, without jumps or conditionals.

Alignment requires three operations: equalising path lengths, computing level structure and aligning basic blocks. The first two steps operate on the CFG, where equalising path lengths is described in [54] as arranging the CFG so that every path in a CFG region starting at a secret-dependent branch goes through an equal number of basic blocks. Computing level structure comes down to finding out the distance from each basic block in a CFG region to the entry block of that region.
The third step operates on basic blocks themselves and is of most interest in this thesis. Here, the goal is to make basic blocks at the same level exhibit the same latency trace

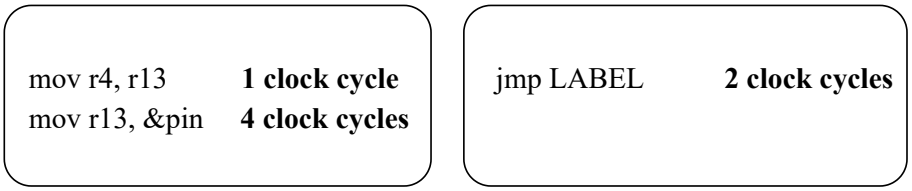

FIGURE 3.7: Non-aligned basic blocks

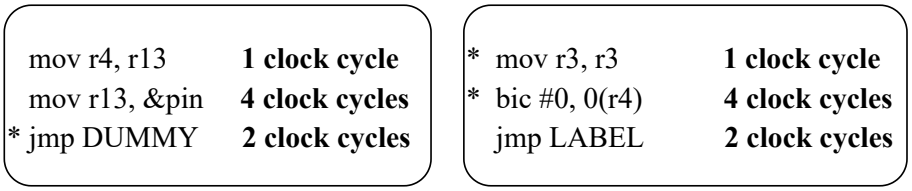

FIGURE 3.8: Aligned basic blocks

and in this regard, dummy instructions are inserted into the basic blocks. The result is a set of basic blocks with the same number of instructions and with each tuple of instructions at the same position in the instruction sequences having equal timings.

For example, the basic blocks in figure 3.7 are aligned by inserting the dummy instructions in figure 3.8, annotated with an asterisk. Here it can be seen that the two blocks have an equal amount of instructions and that the first two, the second two and the third two instructions have an equal amount of clock cycles. These blocks are now indistinguishable to a Nemesis attacker. For each tuple of secret-dependent branches, their corresponding basic blocks (blocks at the same level in the level structure of the CFG) are aligned using the described technique, resulting in the branches being aligned and indistinguishable to a Nemesis attacker.

3.4.2 The DMA Defender Pass

The *DMA Defender Pass* works in almost the same way as the Nemesis Defender Pass, but differs in the last step when aligning basic blocks. While the Nemesis Defender Pass inserted dummy instructions that have the same latency as their opposing instructions, dummies for the DMA Defender Pass should have an appropriate memory access trace. A trace is appropriate when it has the exact same instruction memory trace as the opposing instruction. Alternatively, one could insert instructions that have the same memory trace as the combined memory trace as some consecutive instructions in the corresponding basic block, an approach that is explained in section 4.5.1. Just like in the Nemesis Defender Pass, aligning basic blocks is the necessary step to take when making secret-dependent branches indistinguishable, this time to a DMA attacker. Therefore, we will again consider opposing basic blocks when giving examples about the DMA Defender Pass.

Appropriate dummies address the same memory regions as the instruction they compensate for. We therefore need to find out which memory regions are addressed by some instruction. An instruction like `mov &EDE, &EDE` has a different memory trace when the operands point to data memory than when they to program memory and therefore, we need to check whether an instruction has an operand that points to memory and if so, which memory region it points to.

3.4.3 Check accessed memory regions

In the current implementation, we only harden programs against the DMA attack that address only data memory. When we encounter a sensitive region in the CFG, we check for all basic blocks in it whether the instructions in it, if they address memory, address the data memory region. Checking whether an instruction has one of its operands pointing to memory can be done statically using the `memoperands` method from LLVM [19], which gives all the operands of an instruction that point to memory. It can be invoked on a `MachneInstr` class, used for the instructions in the basic blocks of the program. We use the LLVM `Value` of a memory operand to check whether it is a global variable, which is mostly stored in RAM memory. This kind of memory roughly corresponds to data memory. So checking whether some memory operand (operand pointing to memory) points to a global variable is a way of finding the accessed memory region by some instruction in a static way, without need for running the program.

If it is not a global variable that the memory operand points to, we check whether or not it uses the stack pointer in indexed addressing mode. Encountering an operand like `x(SP)` probably indicates that a parameter that is passed on the stack was used.

This approach was sufficient for the benchmark programs used to evaluate the DMA Defender Pass, but it might be an insufficient approach when another memory region is addressed in a way not covered yet by this static check. We discuss some alternatives in the next chapter, but checking for accesses to the other memory regions would be a valid subject for future work. Another limitation of the described implementation is that the `memoperands` method does not give a guarantee about whether the instruction accesses memory if no memory operand is found [19]. We see this claim at work with for example an instruction like `mov #42, r6`. Here, no memory operands are found by the `memoperands` method from LLVM, but this instructions uses a constant stored in the program memory region. This indicates that the `memoperands` method will need to come along with other checks in the future in order to make checking for the memory regions an instruction accesses as exhaustive as possible.

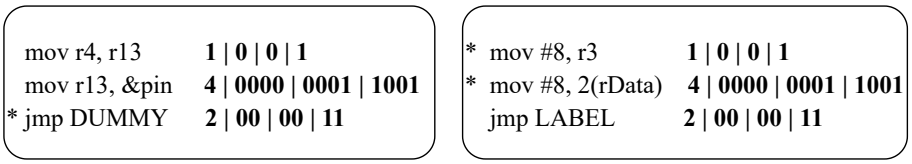

FIGURE 3.9: Instruction-level aligned basic blocks

3.4.4 Find unused register

To defend against the DMA attack, we need dummies to have the correct memory trace. Using a dummy with the exact same memory trace as the corresponding instruction instruction in the opposing branch results in branches indistinguishable to a DMA attacker. Take for example the basic blocks from 3.8 (without the dummy instructions). The DMA Defender Pass aligns these basic blocks as in figure 3.9. Here, each instruction is compensated with a single dummy (indicated by an asterisk) and as a consequence, the total memory access trace of the blocks is the same. These blocks are the same according to a DMA attacker and a secret could not be extracted using the traces of the secret-dependent branches that these basic blocks are part of. In the second dummy instruction, the `rData` register is a register that holds a certain value so that the `2(rData)` operand points to a location in data memory.

It is important that a dummy instruction addresses the correct memory regions in order to exhibit the wanted memory access trace. When we need a dummy with an operand in indexed addressing mode (e.g. `2(r4)`), we need to make sure the used register holds a value that makes the operand point to the wanted memory region (data, program or peripheral memory). To discover which register may be used when generating a dummy, we check which registers are unused when entering a sensitive region in the Control Flow Graph built by the DMA Defender Pass. When we find three such registers, we push them onto the stack and pop them again when exiting the sensitive region. After pushing the register values, we load values into the registers that, when the registers are used in indexed mode, the operand points to some reserved address in one of the three memory regions. For example, if we find that register r4 is not used in some set of basic blocks, we push r4 onto the stack and load value `0x0402` in it. If we then make a dummy using the operand `2(r4)`, we use the address `0x0402 + 0x0002 = 0x404` which points to data memory. Loading other values in registers and using them in the described way, we can make a pointer to any memory region (peripheral, data or program memory).
In the current implementation, we assume a configuration of memory used as in openMSP430 [20], so we load values between the appropriate ranges into the registers. For program and data memory, we use a value close enough to the fixed boundaries (`0xFFDF` and `0x0200`), assuming these are not used by the compiled program already. The above approach has thus the following requirements: three unused registers in some set of basic blocks (the basic blocks that make up a set of secret-dependent branches) and three memory lines not used by the program (namely, the memory lines

referenced by an operand like `2(r4)`). As mentioned in section 3.3.4, the compiler could use the absolute addressing mode in the dummy instruction if we cannot find three unused registers, but using dummies with an operand in absolute mode has not been investigated in this thesis.

The three registers that we use to point to peripheral, data and program memory are referred to as `rPeriph`, `rData`, `rProgram` when specifying dummy instructions (as in appendix A) and we refer to operand pointing to the corresponding memory lines as `x(rPeriph)`, `x(rData)`, `x(rProgram)`.

3.4.5 Implementation

In the previous sections, we touched upon the elements that make up the implementation of the DMA Defender Pass: checking accessed memory regions, finding unused registers and inserting dummy instructions. Here, we bring them together. Pseudocode for the most important methods in the DMA Defender Pass is given in listing 3.2.

The implementation is mostly based on the implementation of the Nemesis Defender Pass [54]. The compiler pass builds the CFG of the program, equalises path lengths by cloning basic blocks so that secret-dependent paths contain an equal amount of blocks, whereafter it performs a sensitivity analysis in order to identify sensitive regions. Code for this may be found in the implementation in the repository mentioned in chapter 1[1].

Before diving into aligning a tuple of secret-dependent branches in the sensitive region (using `AlignSensitiveBranch`, the `AlignSensitiveBranches` method in the DMA Defender Pass performs the extra step of finding which registers are not being used in these branches. The method `FindUnusedRegs` returns a set of registers that are not used in the instructions in any of the basic blocks in the branches running from `BB` to `ExitBB`. If at least three registers are free, the registers are allocated as described in section 3.4.4. Otherwise, the compiler pass aborts with the `assert` statement.

In [54], the method `AlignSensitiveBranch` is used to either insert dummy loops if a loop in the CFG is detected or to align the basic blocks at the current level in the CFG. This method `AlignSensitiveBranch` has not changed compared to the implementation of the Nemesis Defender Pass, so it is not included in listing 3.2.

When the `AlignSensitiveBranch` method has to align basic blocks at the current level of the CFG, it calls another method, `AlignInstructions`, which aligns the individual instructions in the set of basic blocks under influence of some secret-dependent branch. It defines an instruction iterator for every basic block, so that one always looks at instructions at the same 'height' in the basic blocks, the corresponding

[1] https://github.com/SteffieJoosen/llvm-project/blob/master/llvm/lib/Target/MSP430/MSP430DMADefender.cpp

instructions. Per basic block, this method iterates over the instructions in it, and for every other basic block where it finds an instruction at the same height with a non-matching memory trace class, it inserts a dummy. When compensation for a PUSH instruction is done, we insert another MOV instruction into this basic block, as explained in section 3.3.3.

Other implementation work, like that for the FindUnusedRegisters or `CheckAccessedMemoryRegions` methods can be found on the provided links in chapter 1 and 6[2].

```
1  void AlignSensitiveBranches() {
2    for (BB : in BB starts a secret-dependent branch) {
3      ExitBB = ExitOfSensitiveBranch(BB);
4      UnusedRegs = FindUnusedRegisters(BB, ExitBB);
5      CheckAccessedMemoryRegions(BB, ExitBB);
6      if (UnusedRegs.length >= 3) {
7        BB.Insert("push UnusedRegs[0]");
8        BB.Insert("mov 0x0402, UnusedRegs[0]");
9        BB.Insert("push UnusedRegs[1]");
10       BB.Insert("mov 0xFFDC, UnusedRegs[1]");
11       BB.Insert("push UnusedRegs[2]");
12       BB.Insert("mov 0x0010, UnusedRegs[2]");
13     } else {
14       assert(false && "Not enough registers to allocate");
15     }
16     AlignSensitiveBranch(BB, UnusedRegs[0:3])
17   }
18 }
19
20 void AlignInstructions(set<BB> BBs, UnusedRegs) {
21   map<BB, InstructionIterator> II;
22   for (BB : BBs) {
23     for(Instruction : BB) {
24       MemTrClass = getMemoryTraceClass(Instruction);
25       for (BB' : BBs where BB' != BB) {
26         Instruction' = II[BB'];
27         MemTrClass' = getMemoryTraceClass(Instruction');
28         if (MemTrClass != MemTrClass') {
29           InsertDummy(BB', MemTrClass);
30           if (Instruction == PUSH) {
31             BB.insert("mov @r3, r3");
32             II[BB]++;
33           }
34         } else {
35           // Memory access traces match
36           II[BB']++;
37         }
38       }
39     }
40   }
```

[2] https://github.com/SteffieJoosen/llvm-project/blob/master/llvm/lib/Target/MSP430/MSP430DMADefender.cpp

```
41 }
```

Listing 3.2: Pseudocode for the DMA Defender Pass

3.5 Conclusion

The DMA attack relies on the fact that a peripheral incurs a latency when the CPU generates a memory request in the same clock cycle. Using this timing, a DMA attacker can outline the memory accesses a CPU does. If some program uses secret-dependent branches that contain instructions that cause the CPU to exhibit a different memory access pattern, a DMA attacker can discover this and leak information on the secret. For our defense against the DMA attack, we want to insert dummy instructions in a program under compilation in order to make secret-dependent branches indistinguishable to a DMA attacker. Therefore, we classify each instruction that can possibly come forth on the openMSP430 architecture into a memory trace class. For every memory trace class, we try to find a dummy that we can insert into a program in order to compensate for an instruction belonging to that class. Insertion of dummy instructions is done in a new compiler pass, the DMA Defender Pass. Compiling a program with this compiler pass resuts in a program that is hardened against the DMA attack.

In the next chapter, we evaluate the instruction classification and the compiler pass discussed in this chapter. We also provide alternatives to the approach taken here.

Chapter 4

Evaluation

For our defense against the DMA attack, the memory access traces of all MSP430 instructions are needed. In chapter 3, we made a classification according to these traces and we evaluate in this chapter. Compiling a program using the DMA Defender leads to increase in code size and execution timing, which we will discuss next. The DMA Defender uses static checks on the source of a program to find the memory region that is addressed by some instruction in order to compensate for it with a dummy addressing the same region. These static checks might not always suffice, and we discuss some alternatives to it in this chapter. In terms of code size, the DMA Defender does not use the optimal way to compensate for a PUSH instruction, while different approaches to it have other downsides. Although the currently used technique hardens a program against both Nemesis and the DMA defense, an alternative could be to use combined memory access trace to find dummy instructions for.

4.1 Instruction classification

As can be seen in appendix A, 40 classes were found for all instructions possible in the openMSP430 architecture, which is a beneficial result for the implementation of the DMA Defender Pass, since one could expect a lot more classes. TableGen (section 3.2.3) generates 20 of the 27 core instructions (appendix A.1.1), namely the single- and double-operand instructions, plus the JMP instruction. TableGen has one data type for a conditional jump, for which the backend implemented for this thesis generates all 7 assembly opcodes possible for conditional jumps. With these, we generated all 27 possible opcodes for the MSP430 architecture.

For source operands, seven addressing modes are available, of which five address memory. For the destination operand there are four addressing modes available, 3 of which address memory. When a source operand that addresses memory is used in the instruction, we multiply the possible memory trace class by three, because the operand may address each of the three memory regions, possibly resulting in another memory trace class. For a destination operand that addresses memory, we multiply

it by two, omitting destination operands that address program memory, since this region mostly is implemented in ROM and therefore unmodifiable. The worst-case scenario (where all instructions fall in a separate class) occurs when we find

$$12 \cdot \underset{\sim}{2} \cdot \underline{1} + 12 \cdot \underset{\sim}{5} \cdot \underset{\sim}{3} \cdot \underline{1} + 12 \cdot \underset{\sim}{2} \cdot \underline{3 \cdot 2} + 12 \cdot \underset{\sim}{5} \cdot \underset{\sim}{3} \cdot \underline{3 \cdot 2} + 6 \cdot \underline{1} + 6 \cdot \underline{3 \cdot 2} + 1 + 1 + 8 = 1480$$

classes using TableGen. In the above calculation, factors for source operands have been underlined with a squiggly line, factors for destination operands have a straight line. We have twelve double-operand instructions, hence the factor 12. Six opcodes use just a single operand. We add one class for the RETI instruction, one for the RET instruction. Eight more classes cover the possible classes for unconditional and conditional jumps. With this, we come to an estimation of how many classes there could possibly be.

Even though this calculation predicts an enormous amount of memory trace classes, only 40 were eventually found. The first reason is that when the same addressing modes are used, almost all opcodes fall in the same class, except for the MOV instruction. Most single-operand instructions fall in the same class as most double-operand instructions using the same addressing mode in source or destination operand. Therefore, the factors 12 and 6 in the above calculation may be canceled out. A consequence is that for the dummy instructions, we generally only need a MOV, an ADD and an RRA instruction (for single-operand instructions with operands that cannot be used as destination operand in double-operand instructions). Next to these three, we need the RET and RETI instructions that serve as dummy for those same instructions. The PUSH and POP instruction are used together to patch against a single PUSH (section 3.3.3).

A second cause for the discovery of so few classes is the fact that instructions using the indirect (e.g. `2(r4)`, absolute (e.g. `EDE`) or symbolic addressing mode (e.g. `&EDE)` in the same operand all fall in the same class, as long as these operands point to the same memory region.

Another explanation is about immediate operands. Instructions with a source operand in immediate mode generate the same memory access trace as instructions with the same opcode and a source operand pointing to program memory. The MSP430 architecture stores constants in program memory, so addressing program memory or using a constant in an instruction does not incur a different memory access trace and therefore these instructions have the same memory trace class.

Now that we have evaluated the classification of instructions, we can evaluate the implementation of the DMA Defender Pass, the new compiler pass in the SLLVM compiler framework that uses the instruction classes and the corresponding dummies to align secret-dependent branches in a program under compilation.

4.2 Evaluation based on an example program

For the purpose of an evaluation of the established defense against a DMA attack, consider the small C function in figure 4.2 based on the benchmark programs [23] previously used to asses the Nemesis defense [54]. It defines some value `result` which has the value three and depending on the if-condition on line 9, it increments some global variable and adds its value to the result. The `__attribute__` keyword belongs to the set of keywords recognized by the GNU GCC compiler [15], the use of `secret` as attribute specification is something specifically recognized by the SLLVM compiler framework [55]. Since the if-condition uses the secret argument `a`, we now this branch is a secret-dependent one and needs to be compensated for when defending against the DMA attack. This method of using the `secret` keyword was originally used in the benchmark programs for the research on the defense against the Nemesis attack [23] and can be reused in the defense outlined in this thesis.

```
1  # global variables
2  int v;
3  int u;
4
5  int some_function( __attribute__((secret)) int a, int b)
6  {
7    int result = 3;
8
9    if (a < b)
10   {
11     v += 10;
12     u += v;
13     result = v;
14   }
15     return result;
16 }
```

Listing 4.1: Example program to be hardened against the DMA attack

Compiled to the MSP430 architecture, the assembly code of this program is listed in listing 4.2, which serves as a point of comparison to evaluate the DMA Defense in terms of code size and efficiency. The program contains three basic blocks, two of which are always executed (basic block 0, `BB_0`, and basic block 2, `BB_2`) whereas basic block 1 is only executed depending on the result of the CMP-instruction on line 4. According to MSP430's calling convention [50], the arguments a and b are passed to the function in respectively registers r12 and r13, so we know that the secret argument a (according to the source code in listing 4.2) resides in r14 after the MOV on line 2, making the conditional jump on line 5 secret-dependent. Therefore, this program has secret-dependent branches: a branch where basic block 1 is executed and one where it is not.

```
1  ; BB_0:
2    mov r12, r14
3    mov #3, r12
4    cmp r13, r14
5    jge .BB_2
6
7  ; BB_1:
8    mov #10, r12
9    add &v, r12
10   mov r12, &v
11   add r12, &u
12
13 ; BB_2:
14   ret
```

Listing 4.2: Assembly code for program 4.1

Since we know that basic block 1 in figure 4.2 is executed depending on a secret, it needs to be compensated for with a block containing dummy instructions. The compensating block is `BB_3` in listing 4.3: depending on the secret value in r14, either basic block 1 or basic block 3 is executed. Instructions asking for attention here are the MOV and PUSH instructions in basic block 1: after the DMA Defender Pass has found that none of the registers r4, r5 or r6 is used in basic blocks 1 and 3, it inserts instructions that saves the values in them and stores new values in order to use them in indexed addressing odes used in a dummy instruction, as the one on line 31 in listing 4.3. This approach is explained in section 3.4.4. After the branches merge again in basic block 2, the registers are restored in their previous state with the POP instructions.

```
1  ; BB_0:
2    mov r12, r14
3    mov #3, r12
4    push  r4
5    mov #0x0402, r4
6    push  r5
7    mov #0xFFDC, r5
8    push  r6
9    mov #0x0010, r6
10   cmp r13, r14
11   jge .BB_3
12   jmp .BB_1
13
14 ; BB_1:
15   mov #10, r12
16   add &v, r12
17   mov r12, &v
18   add r12, &u
19   mov #42, r3
20   mov #42, r3
21
22 ; BB_2:
23   pop r6
24   pop r5
25   pop r4
26   ret
27
28 ; BB_3:
29   mov #42, r3
30   mov #42, r3
31   mov 2(r4), r3
32   mov #8, 2(r4)
33   rra 2(r4)
34   mov #42, r3
35   jmp .BB_2
```

Listing 4.3: Assembly code for program 4.1 patched against a DMA attack

The instructions in basic block 3 all compensate some instruction in basic block 1 and are therefore all dummy instructions. Table 4.1 aligns the instructions that compensate each other in both basic blocks together with their line numbers in the program. Instructions needing special attention are the MOV instruction form line 29 of the program and the instructions on line 19 and 20. The MOV instruction on line 29 compensates no instruction in BB_1, but is inserted because the JMP instruction on line 12 is executed only when the conditional jump on line 11 is not taken. Whether this JMP is executed also depends on a secret, and therefore needs to be compensated. The MOV instructions on line 19 compensates the one on line 34, which is added to basic block 3 in order to make it end in two terminating instructions. The LLVM Pass that hardens against Nemesis [54] uses this method to make the implementation more comprehensible. Since this extra MOV appears in basic block 3, it needs to be compensated for in basic block 1. This explains the dummy instruction on line 19. The instruction on line 20 compensates the JMP

Line	Instruction in BB_1	Line	Instruction in BB_3	Memory access trace
12	jmp .BB_1	29	mov #42, r3	2 | 00 | 00 | 11
15	mov #10, r12	30	mov #42, r3	2 | 00 | 00 | 11
16	add &v, r12	31	mov 2(r4), r3	3 | 000 | 010 | 101
17	mov r12, &v	32	mov #8, 2(r4)	4 | 0000 | 0001 | 1001
18	add r12, &u	33	rra 2(r4)	4 | 0000 | 0101 | 1001
19	mov #42, r3	34	mov #42, r3	2 | 00 | 00 | 11
20	mov #42, r3	35	jmp .BB_2	2 | 00 | 00 | 11

Table 4.1: Memory access traces of the instructions in the opposing basic blocks in program 4.3

instruction on line 35. No JMP is needed basic block 1, because basic block 2 is executed after it anyway. Basic block 2 is said to be the *fall-through basic block* for basic block 1 [54].

4.2.1 Code size

The MSP430 specification [48] list instruction sizes in bytes, which fully depend on the instruction format (double-operand, single-operand or jump instructions) and the addressing mode used. Instructions being compensated with dummies of the same format and using the same addressing modes cause an increase in code size with their own length. For example, the ADD instruction on line 16 and the compensating MOV on line 31 are both two bytes long. In general, dummy instructions have the same format and addressing mode as the instruction they compensate for, given that they exhibit the same latency in that case, which is a minimal requirement for the dummies exhibiting the same memory access traces as their opposing instructions. Having the same format and addressing modes also results in the same size for the dummy as for the opposing instructions.
Dummies do not have the same format as the instructions they compensate for in some special cases, but to the best of our knowledge, we can always compensate instructions with dummies being at most as large as the instructions themselves. For example, to compensate for JMP instructions in this example, MOV instructions are used, but these instructions are of equal size too.

For PUSH instructions, a more subtle analysis of the code size should be made. In the current implementation, we patch against a PUSH with three extra instructions, resulting in an increase in code size of three bytes for a single PUSH of only one byte. When we compensate for a PUSH instruction with three dummies, as in figure 3.6 we have an increase in code size depending on the addressing mode used by the PUSH instruction: 3 bytes extra when the register (r4), indirect register (@r4) or indirect autoincrement (@r4+) mode is used and 6 bytes when indexed (2(r4)), symbolic (EDE) or absolute (&EDE) mode is used. In this investigation, this was the only

case found in which compensation leads to more than the instruction's size increase in the total code size.
An alternative method to compensate for PUSH instructions, is inserting a dummy for *combined* memory access traces, as we will describe in section 4.4.1. With this approach, a MOV instruction of 3 bytes compensates for a PUSH and a MOV instruction of each 1 byte, which is an increase of $\frac{3}{2} = 1.5$ bytes per byte of the original instructions, a better result than the increase of three bytes for a single PUSH. The upside of the more expensive triple-dummy patching that we use now is that the resulting program is still hardened against the Nemesis attack [52] after using the DMA Defender. A trade-off is to be reflected on here and further investigation should decide whether large instructions may be combined to compensate for with a single dummy that is smaller in terms of code size.

In general one could say that the defense against the DMA attack always results in larger code size, where the worst case arises when every instruction of any block in a sensitive region is a PUSH, but this is an unlikely situation. The nearly worst-case is one where every instruction in any basic block of a sensitive region needs compensation with a dummy. Since we can always compensate an instruction with a dummy of the same size, the resulting size of a secret-dependent region in a patched program is then twice the sum of all basic blocks in that region in the original program.

4.2.2 Performance

In order for some branch having the same combined memory trace as the corresponding branch, it is a minimal requirement that both of them exhibit the same latency. Here the nearly worst case occurs when for every instruction in some sensitive region an extra dummy should be inserted, which are the same circumstance as for the nearly worst case in terms of code size (previous section). This results in a latency which is the sum of the latencies of both branches [54], leading to the same efficiency of a previously studies cross-copying method to align branches [5]. Performance that is even a bit worse on average is incurred, because for every PUSH instruction, there is an increase of 5 clock cycles per PUSH when one branch is taken (the one with the dummy PUSH and POP, the basic block on the right of figure 3.6) and an increase of two per PUSH in the other branch (the left of figure 3.6). The absolute worst case is when every instruction needing a dummy is a PUSH instruction. In the best case, the latency of corresponding basic blocks is the maximum of their original latencies. In this case, the set of memory traces of the instructions from the smallest basic block is a subset of the traces of the instructions of the largest basic block, where the intersection of both sets are the memory traces of the instructions that can compensate each other, because they are on the same 'height' in the basic block. For instance, the memory traces of the non-dummy instructions in the rightmost basic block in figure 4.1 form a subset of those of the leftmost basic block. Because the memory traces stand in the right order, they both can compensate for one of the instructions in the left most basic block. All that is left is then to insert a dummy

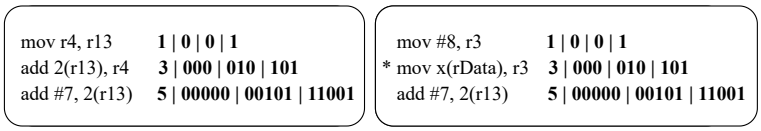

Figure 4.1: The best case latency of these basic blocks is 9 clock cycles, the maximum of the total latencies of both basic blocks

for the remaining instruction of three clock cycles, and the resulting latencies of both basic blocks are equal to the maximum of their original latencies, nine and six clock cycles.

4.2.3 Assumptions and reservations

In the development of the defense against the DMA attack, a number of assumptions and reservations were made. First of all, we assume there is possibility to reserve three registers in the scope of a sensitive region, meaning that all the blocks in this region do not use these registers, so the DMA Defender Pass can load values into them in order to use them in an indexed operand for a dummy instruction. The DMA Defender Pass also needs reservation of three memory locations, one in each memory region (data, peripheral and program memory) that dummy instructions can address (and possibly change) to generate the desired memory traces with these dummies.
Other than the above described reservations, we made some assumptions for the defense against the DMA attack. As a first assumption, we suppose a RET or RETI instruction in MSP430 should never be patched with a dummy other than these instructions themselves. Whenever a RET instruction occurs in some branch, it suffices to return from the function in the other branch, because only this solution results in equal memory access traces. If one branch executes longer than the other in some sensitive region, it is impossible to align them without pushing the RET instruction down to the very end of the branch. If we were to encounter a RET or RETI instruction to compensate for, we compensate for it with a RET or RETI respectively.

4.3 Alternatives to static checks

In the current implementation of the DMA Defender pass (section 3.4.2), we use static checks on operands of instructions that are pointers to see whether they point to constants (which reside in program memory) or to global variables (residing in data memory). If pointer operands do not point to any of these, we assume they point to some memory address in peripheral memory. The described examination of operands that point to memory is somewhat inaccurate. If some program uses a pointer to some memory region, but close to the border between this region and the adjacent memory region, and then indexes this pointer, the resulting operand might

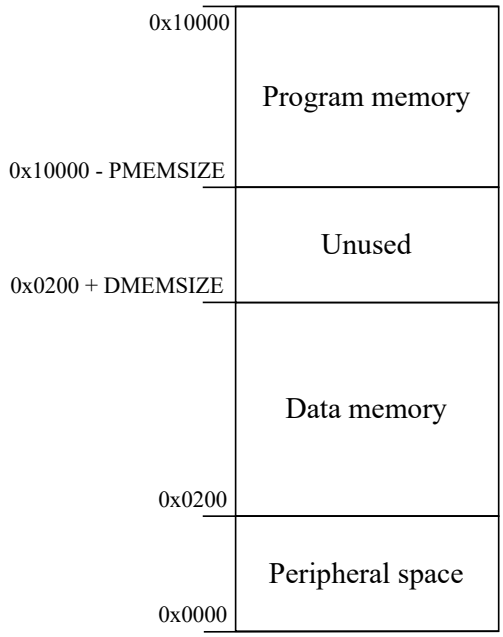

FIGURE 4.2: Memory configuration for the openMSP430 architecture

point to the other memory region. If the static check for example sees a pointer to a constant, an access to program memory is assumed. Indexing this pointer to make it point to data memory for example completely undermines the result of the static check. Alternatives exist and we discuss these next.

4.3.1 Runtime checks

Runtime checks could complement the static checks as described above, because they are more precise and could compensate for the limitations of the static checks. However, they come with the cost of increased code size and degraded performance. Using the technique of inserting runtime checks for checking which memory region is accessed, we would patch an instruction with three alternative instructions, depending on whether it accesses data, program or peripheral memory. With runtime checks, we should check the value of a pointer and depending on this check, we would execute a dummy instruction addressing one of the three memory regions. The MSP430 microcontroller has several configuration options with respect to memory organization. One such configuration is the small memory model for devices with a memory space of 64 KB or less [20], as shown in figure 4.2. This is the configuration used for the openMSP430 microcontroller, the core under investigation in this thesis.

Given this, we know which values to compare a pointer value with to find out which of the three memory regions it points to. The constants `DMEM_SIZE` and `PMEM_SIZE` can be found in configuration files as needed and therefore the bounds of all three memory regions are known.
Knowing how to find the accessed memory region from an instruction operand

mov r4, r13	**1 | 0 | 0 | 1**
mov r13, &pin	**4 | 0000 | 0001 | 1001**
jmp LABEL	**2 | 00 | 00 | 11**

Figure 4.3: Basic block to be compensated for using runtime checks on the accessed memory region

pointing to memory, we need to insert this check into the code of the program. In the following example, investigating the basic block in figure 4.3, we only check for accesses to peripheral memory to keep the example readable.

In figure 4.4, the instructions in the leftmost basic block are patched with the instructions annotated with an asterisk in the rightmost set of basic blocks: `mov r4, r13` is patched with `mov #8, r3`, just as in figure 3.9 and the last JMP instruction is patched as was done in figure 3.9. To compensate for `mov r13, &pin`, a runtime check was inserted in the form of the instructions highlighted in figure 4.4. The MOV instruction places the value of the pointer in register r4. With the AND instruction, we check if the value lies within the bounds of the peripheral memory region (***perStart*** and ***perEnd***). If so, the conditional jump (JC) is taken and the leftmost basic block is used, where a dummy that also addresses the peripheral memory region is executed. Otherwise, the jump to BB2 is taken and we execute another dummy instruction. All these instructions also reside in the original basic block on the left, to keep alignment of individual instructions. The described approach causes an extra conditional branch, where the condition is whether or not the memory access is to peripheral memory. The extra instructions needed for the runtime check (highlighted in figure 4.4 cause extra code size. The runtime check also makes the program a few clock cycles longer. Given that this runtime check, with its extra code size and total latency, has to be inserted for every instruction that accesses memory (here, for the `mov r13, &pin` instruction), the overhead of this approach can become severe and undesirable for practical implementation.

Given that the static checks are insufficient and that dynamic checks produce overhead in terms of latency and code size, concolic testing [45] might be a solution to check for the accessed memory regions. The word ‘concolic’ is a contraction of ‘concrete’ and ‘symbolic’, forming an appropriate name for a technique that combines concrete and symbolic execution of a program in order to build symbolic constraints for branches. This technique could be used to build the correct constraints in the compiler backend that checks the memory regions accessed.

4.3.2 Non-deterministic priority for peripheral and CPU

As an alternative to making branches identical according to their combined memory access traces, one could change the policy that is used to give priority to components

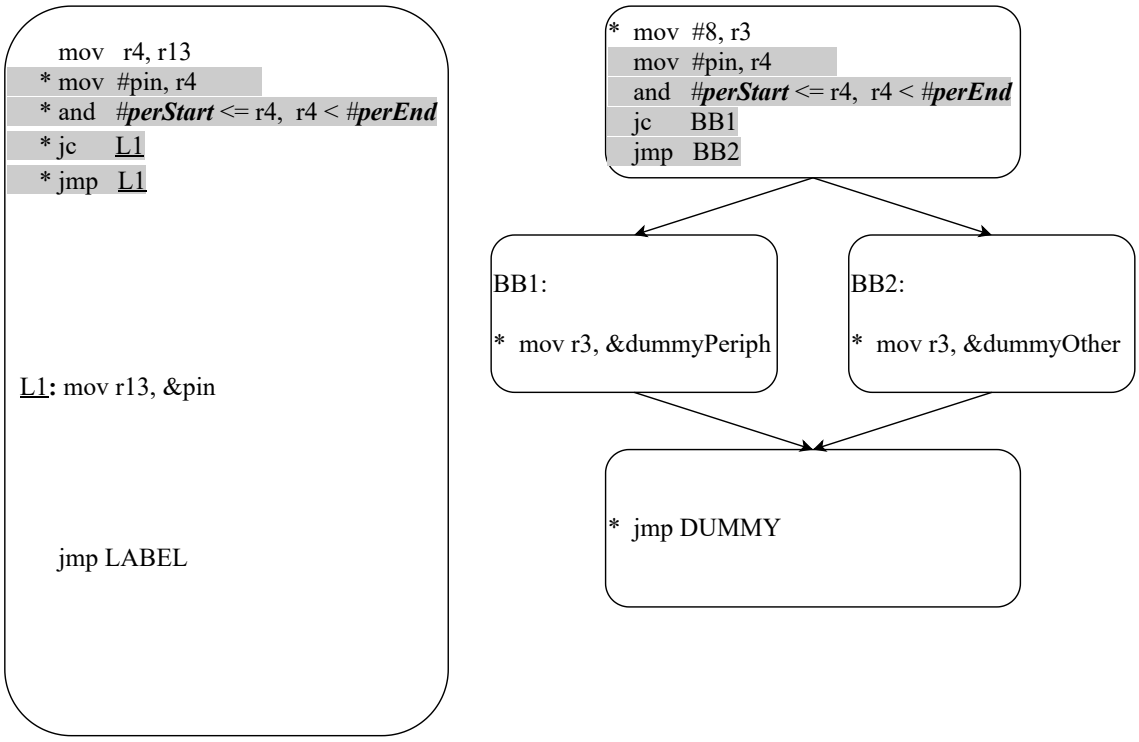

FIGURE 4.4: Compensation with runtime check

that request to access memory. In Sancus, the platform based on MSP430 which is subject to this thesis, priority on the memory bus is always given to the CPU when the memory bus suffers from contention and this is hardwired in the platform. Nevertheless, in the openMSP430 implementation the DMA interface defines a signal `dma_priorirty`, which indicates which component may access memory first in case of bus contention. Applications may use this signal as required. The bootloader for example should be executed with the `dma_priority` signal set (giving peripherals priority over the CPU), because the CPU may not execute instructions from uninitialized memory [20].
An alternative to giving the CPU priority by default in Sancus, one could set or clear the `dma_priority` signal non-deterministically. As a consequence, a peripheral generating a DMA request will unpredictably incur a delay, but this leaks no information on whether the CPU accessed memory or not in the same clock cycle.

The original reason for keeping the priority at the CPU's side for accessing memory is to mitigate against Denial-of-Service (DoS) attacks on the platform. A (possibly tampered with) peripheral generating successive DMA requests rapidly might cause the CPU to be left out from accessing memory and therefore in blocking state. When the priority changes non-determistically, the expected number of memory access requests from the CPU being served in case of a Denial-of-Service attack is around half of the total of requests for memory. Investigation is needed to determine whether

mov r4, r13	**1 | 0 | 0 | 1**
push r13	**3 | 000 | 001 | 001**
* mov @rData, r3	**2 | 00 | 10 | 01**
* jmp DUMMY	**2 | 00 | 00 | 11**

* mov #8, r3	**1 | 0 | 0 | 1**
* push r3	**3 | 000 | 001 | 001**
* pop r3	**2 | 00 | 10 | 01**
jmp LABEL	**2 | 00 | 00 | 11**

Figure 4.5: The DMA Defender compensates for a PUSH instruction with three dummies: a PUSH, a POP and a MOV instruction

this is feasible or still insufficient to resist against a DoS attack.

4.4 Alternatives to triple-dummy patching for a PUSH instruction

In section 3.3.3, we touched upon the fact that we cannot find a single dummy to compensate for the PUSH instruction because it falls in a memory trace class on its own. Therefore, we need two dummies in the opposite basic block, namely a PUSH and a POP, and a third one in the block containing the original PUSH resulting in three dummy instructions to patch only one. This is a significant overhead in code size and unfavourable in large programs possibly containing a lot of PUSH instructions.

4.4.1 Using combined memory access traces

In section 3.3.3, we saw that the DMA Defender currently compensates for a PUSH instruction with three dummy instructions: another PUSH, followed by a POP, and a MOV instruction in the original basic block to compensate for the POP. The result of the DMA Defender consists then of the basic blocks in figure 4.5. The instructions `push r3`, `pop r3` and `mov @rData, r3` serve as dummies for the `push r13` instruction in the leftmost basic block from figure 4.5.

This approach leads to a great increase in code size, since a single PUSH instruction is being compensated for with three dummy instructions, with a size summing up to a total of three bytes. A PUSH instruction is only 1 byte long, so that is an undesirable increase, especially when a lot of PUSH instructions are present in the program.

An alternative could be to use the notion of combined memory access traces, as is done in figure 4.6. A combined memory trace can be seen as the sum of several instruction memory traces. The PUSH instruction has memory trace class

$$3 \mid 000 \mid 001 \mid 001$$

meaning it takes three clock cycles to execute, never accesses peripheral memory, accesses data memory in the last clock cycle and as well as program memory. Following the definition of a combined memory trace, we see that the combined

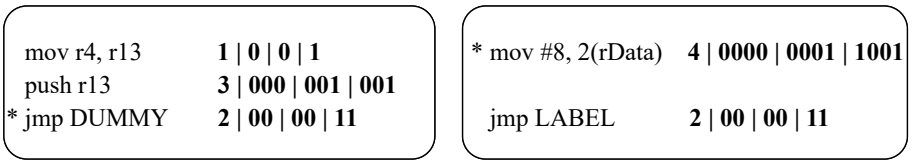

FIGURE 4.6: Branch-level aligned basic blocks

memory trace of the MOV and PUSH instruction in the leftmost block from figure 4.6 is

$$1 \mid 0 \mid 0 \mid 1 \ + \ 3 \mid 000 \mid 001 \mid 001 \ = \ 4 \mid 0000 \mid 0001 \mid 1001$$

and for the 4 | 0000 | 0001 | 1001 class, we do have a dummy instruction, namely the `mov 8, x(rData)` instruction (appendix A), where `rData` is a register that holds an address pointing into data memory. Now we can compensate for both the MOV and PUSH instruction together with a single dummy instruction, as can be seen in figure 4.6. Now both basic blocks exhibit the same memory access trace, namely

$$1 \mid 0 \mid 0 \mid 1 \ + \ 3 \mid 000 \mid 001 \mid 001 \ + \ 2 \mid 00 \mid 00 \mid 11$$

$$= \ 6 \mid 000000 \mid 000100 \mid 100111$$

$$= \ 4 \mid 0000 \mid 0001 \mid 1001 \ + \ 2 \mid 00 \mid 00 \mid 11$$

This approach highlights an important difference between the Nemesis and the DMA attacker, on which more detail is given in section 4.5.
The above described method to compensate for a PUSH instruction leads to no increase in latency in the original branch that contains the PUSH. As discussed in section 4.2.1, the increase in code size is only 1.5 bytes per byte of the original instruction pair (the MOV and PUSH), so this is a promising alternative to the triple-dummy patching technique used in the current implementation of the DMA Defender Pass.

Another alternative to compensate for a PUSH instruction exists, which is covered in the next section.

4.4.2 A single PUSH and a single POP

A possible alternative for patching with three dummies is to simply patch a PUSH with one PUSH and a POP with a POP instruction. For this to work, we need to make sure that the stack is not referred to in between the dummies. Imagine we patch the PUSH instruction in the leftmost basic block in figure 4.7 with the dummy instruction `push r3` in the rightmost basic block. If a POP appears in the same basic block, as is the case on the left (`pop r13` is still in the same basic block as `push r13`), we can patch this POP with a dummy POP instruction on the right (with `pop r3`) so that we do not influence the program since the register we pushed onto the stack (`r3`) is popped again with a dummy instruction. This only works assuming

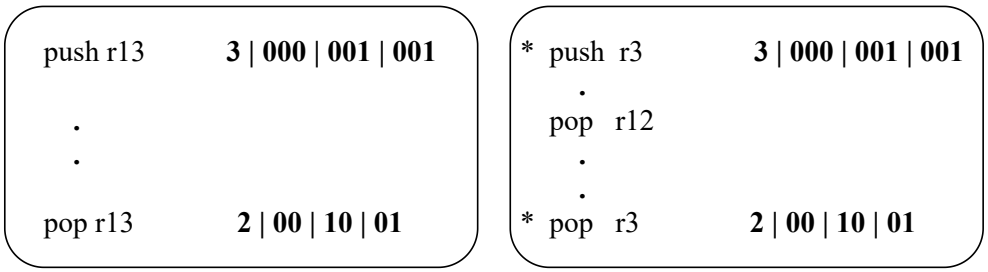

Figure 4.7: Compensation with a single PUSH and a single POP

the program does not expect the stack to be in some specific state in between these dummy instructions. In figure 4.7, the `pop r12` instruction was originally there, and it is probably assumed some value was pushed onto the stack beforehand. Now since the `push r3` instruction was inserted before this POP, another value is popped from the stack than is expected originally and semantics of the program are probably not preserved, which is the exact thing a dummy instruction may not do. Therefore, for this approach to work, we need basic blocks to not contain POP instructions (the right block in figure 4.7) appearing in between a PUSH-POP dummy pair (the left block in figure 4.7). The execution path in between the dummy PUSH and POP may not flow through another POP or any other instruction that assumes the stack to be in a predefined state, because this state cannot be guaranteed anymore after the dummy PUSH is inserted.

4.4.3 Replacing a PUSH instruction

Compensating for a PUSH instruction obviously involves overhead, whether we use three dummies for it, causing extra latency and code size, or one of the techniques described above, requiring careful analysis of the program by the compiler pass. Instead of trying to compensate for a PUSH instruction, one cloud replace the PUSH instruction with other instructions showing the same semantics. For example, we could replace a `push r5` instruction in the program with the instructions

```
sub #2, r1

mov r5, 0(r1)
```

where `r1` is the stack pointer in the msp430 architecture [48]. These instructions exhibit the same semantics as a single PUSH instruction and have dummies

```
mov #8, r3

mov #8, 2(rData)
```

as stated in appendix A.
This is an easy way to compensate for a PUSH instruction, especially compared to the approach described in section 4.4.2, which requires thorough analysis of the

source code. However, it scores no better than the triple-dummy patching used in the current implementation of the DMA Defender (section 3.3.3) in terms of code size. We replace the original PUSH instruction of one byte with four instructions of a total of six bytes, resulting in a code size increase of five bytes, which is less desirable than the code size increase of three bytes when using the triple-dummy patching technique. In terms of performance, this technique scores equally as good as the triple-dummy patching. We know from section 4.2.2 that compensating for a PUSH leads to an increase of two clock cycles when the branch with the original PUSH is taken whereas five extra clock cycles are added in the other branch. The same is true when using the technique described in this section. In one branch, the three-cycle PUSH instruction is replaced with two instruction taking five clock cycles to execute in total, leading to an increase in latency of two cycles. In the other branch, five extra clock cycles are added, as is the case for the triple-dummy patching technique.

The above discussion of alternatives to triple-dummy patching, as implemented now in the DMA Defender Pass, showed that a promising alternative is to use combined memory access traces to compensate for with dummy instructions. A trade-off has to be made here between code size and keeping a program hardened against the Nemesis attack. The defense developed in this thesis is based on the defense against Nemesis, and a possible consideration could be not to undermine the Nemesis defense, which our implementation complies to. We go into detail in the next section.

4.5 Comparison with Nemesis defense

The DMA attacker and the Nemesis attacker (section 2.4) have different capabilities. The Nemesis attacker controls interrupt, a capability that can be used to discover instruction latencies. When a Nemesis attacker generates an interrupt request in the very first clock cycle an instruction executes, a waiting time of exactly the instruction's latency is incurred before the interrupt is being served. The DMA attacker on the other hand is capable of generating DMA requests in arbitrary clock cycles and uses this power to discover memory access traces of some execution path. Both attacks have been defended against by inserting dummy instructions in order to make branches indistinguishable in terms of individual instruction's latencies (for a Nemesis attacker) or in terms of memory access traces (for a DMA attacker). In the current implementation, a program under compilation is hardened against both the Nemesis attack and the DMA attack, since we use the triple of dummies to compensate for a PUSH. However, as stated in the previous section, this approach leads to a large increase in code size and therefore using combined memory access traces might be more desirable. Whenever this technique is applied, patching a program against the DMA attack might render a program vulnerable to Nemesis again. Conversely, using the Nemesis defense might also undermine security against the DMA attack. An example of the first situation is discussed in section 4.5.1.

4.5.1 Compensate for combined memory traces

Some programs that are hardened against a Nemesis attacker are not yet resilient against the DMA side-channel attack, as was shown in section 2.5.2. The program used the following two instructions each in one secret-dependent branch:

```
mov r13, &pin
```

```
bic 0, 0(r4)
```

These instructions both take 4 clock cycles and to a Nemesis attacker they are identical. But since they have different memory traces, namely

$$4 \mid 0000 \mid 0001 \mid 1001$$

$$4 \mid 0000 \mid 0101 \mid 1001$$

they are different to a DMA attacker. A program using these instructions in opposing secret-dependent branches is vulnerable to a DMA attacker. In this case, the DMA attacker is stronger than the Nemesis attacker.
On the other hand, some program may be hardened against a DMA attack while still being vulnerable to the Nemesis attack. This is the case when we use a single dummy instruction to compensate for two instructions. In section 4.4.1, figure 4.6, we used the `mov r3, &dummy` instruction to compensate for a `mov r4, r13` and `push r13` instruction together. This is sufficient to make the two blocks indistinguishable to a DMA attacker. Consider the time series of both basic blocks in figure 4.8. At the bottom, the instructions from each branch are placed above one another. Both branches exhibit the same memory trace. If a DMA attacker were to issue a DMA request in some clock cycle, he would incur the same delay for handling that request in both branches. If the CPU accesses memory (at least one high signal) in some clock cycle, the request for the attacker is handled with one clock cycle delay, otherwise the attacker incurs no delay. While a DMA attacker sees no difference in these branches, a Nemesis attacker can differentiate between the above branches by issuing an interrupt request. Issuing such a request in clock cycle one (on the figure) result in this request to be handled whenever the currently executing instruction finishes. If the `mov r4, r13` instruction is executed, the interrupt request from the attacker is handled after two clock cycles. If, on the contrary, the `mov r3, &dummy` instruction is executed, the attacker sees a delay of four clock cycles. A Nemesis attacker can, unlike a DMA attacker, find a difference between these branches.
In the above example, we used the fact that a sequence of instructions exhibiting some memory trace is the same as one instruction exhibiting that same memory trace, which we established in section 4.4.1. To substantiate this theorem, we used the addition of memory traces:

$$1 \mid 0 \mid 0 \mid 1 \;+\; 3 \mid 000 \mid 001 \mid 001 \;=\; 4 \mid 0000 \mid 0001 \mid 1001$$

Adding bit string is not commutative, because we use string concatenation for their addition, but adding latencies (3 and 1 in this case) is simply done with the commutative integer addition. The fact that we use integer addition merges the two distinct

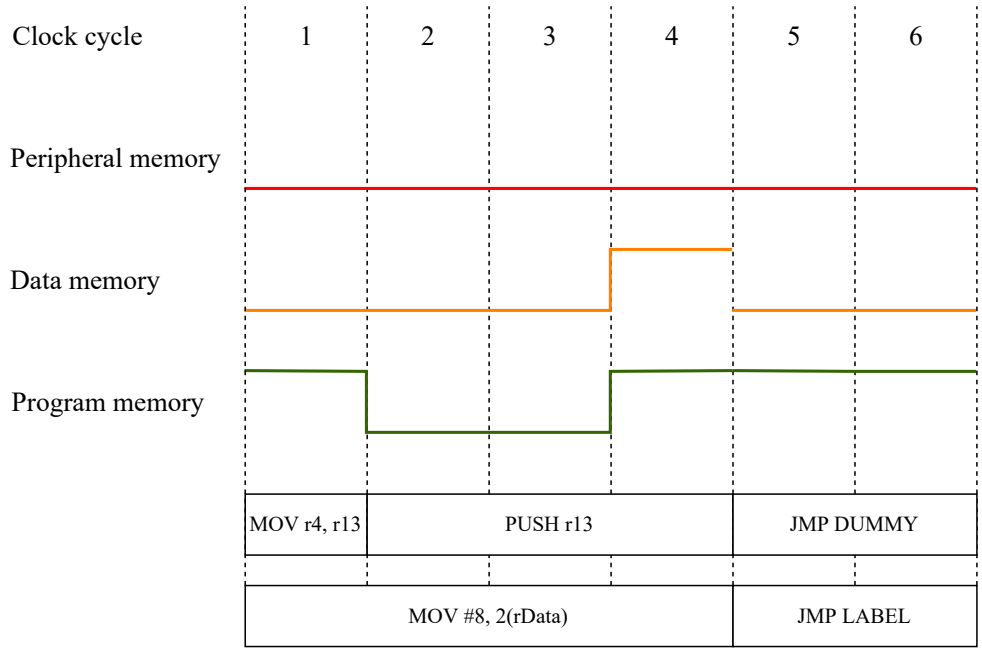

FIGURE 4.8: Alignment of branches with a single dummy for two original instructions

latencies into one, so that they cannot be distilled from the resulting trace (4 | 0000 | 0001 | 1001) anymore. It is in this operation that the vulnerability to a Nemesis attacker remains.

A Nemesis attacker and a DMA attacker are stronger or weaker than one another depending on the situation. Hardening a program against the DMA attack using combined memory access might result in a program that is still vulnerable to a Nemesis attacker, although it is a preferable method in terms of code size. In chapter 5, we discuss a hardware-based defense against the Nemesis attack, which could be combined with the defense studied in this thesis to make a program resistant against both attackers at once.

4.5.2 Combining Nemesis Defender and DMA Defender

In this thesis, we investigated whether a combination of the Nemesis Defender Pass and the DMA Defender Pass could render a program hardened against Nemesis and the DMA attack at once. Unfortunately, combining the Nemesis Defender Pass with the DMA Defender Pass completely undermines the first and vice versa. Using the DMA Defender on a Nemesis-resistant program mostly incites the worst case code size, since most instructions that Nemesis aligns with other instructions have a different memory trace although they have the same latency. In the worst case, every instruction that the Nemesis Defender Pass inserted has a mismatched memory trace and the DMA Defender needs to compensate for it, completely undermining

the original Nemesis defense. On top of that, the original instruction also still needs compensation with a dummy instruction. This worst case scenario is actually quite common.

The instructions for which the Nemesis Defender inserts dummy instructions with the wrong memory access trace are instructions with the same addressing mode(s) for the operand(s), but an inappropriate opcode. The Nemesis Defender almost always uses MOV instructions as dummies, except for when an instruction of four clock cycles is need (a BIC is used then). But as explained in section 4.1, almost all opcodes fall in the same class when using the same addressing modes in the instructions, except for the MOV instruction. Using a MOV instruction as dummy therefore generates the wrong memory access trace in order to make a program resilient against the DMA attack. Also when other memory regions are addressed with the operands, another memory access trace exists behind the latency.
Kankowski makes a statistical analysis of the frequency of opcodes and addressing modes used in x86 architectures [27]. Assuming his analysis is roughly applicable to the openMSP430 architecture, we could predict in how many of the cases the Nemesis Defender uses a dummy instruction with a memory access trace that is still different from the instruction it compensates for, resulting in the DMA Defender having to insert another dummy on top of the dummy inserted by the Nemesis Defender. We attempt to give such a chance of 'mispatching' per latency class. For example, 35 percent of the instructions is a MOV and there is chance of having an operand in the indirect addressing mode of 23 percent [27]. The instruction `mov @rData, 2(rData)` has all these properties and is in class 5 | 00000 | 10001 | 10001 (appendix A). It is the instruction used in the Nemesis Defender Pass as dummy for a latency of five clock cycles. We calculate the chance of encountering its memory trace. MOV is the only opcode in this class, so we may use only the 35 percent chance of this opcode in our calculation, without considering chances of any other opcode. Now the chance of finding this class in a program can be roughly estimated as

$$0.35 \cdot \frac{0.23}{2} \; \frac{1}{3} \cdot \frac{0.23}{2} \cdot \frac{1}{2} = 0.08\%$$

In the above calculation, the 0.23 factor is divided by two because in a MOV instruction, the indirect addressing mode may be used in both the source and destination operand. Here we only need one case. The chance for the source operand is divided by three because the memory operand points to data memory, and this factor assumes that all memory regions might be addressed with an equal chance. For the destination operand we only consider data en and peripheral memory being possibly addressed, and we divide the chance for the destination operand by two. Further investigation is needed to assess the assumption of memory regions being addressed with a uniformly distributed chance. With the chance of encountering a certain memory trace class like the one above, we can find the chance of finding an instruction with a latency of 5 clock cycles by adding all the frequencies of the classes with that latency, leading to a chance of 3.3%. Another way is to calculate

the chance of encountering an instruction with an immediate as source operand and an indirect address as a destination operand, added to the chance of an instruction with both operands in indirect addressing mode. This also gives a chance of about 3.3%. Given that we have a chance of 0.08% of having an instruction from the same memory trace class as the dummy that the Nemesis Defender uses for patching some instruction of latency 5 and since there is a 3.3% chance of having an instruction of latency 5, the chance that the Nemesis Defender patches an instruction with the wrong memory access trace class is

$$1 - \frac{0.08}{3.3} = 98\%$$

So in around 98% of the cases where a latency of five cycles is incurred for an instruction, the DMA Defender would have to insert new dummies to compensate for the instruction inserted by the Nemesis Defender, because the memory access traces do not match. The Nemesis Defender Pass was not implemented at all to use specific memory access traces, and the above analysis shows that we can also not rely on good luck, hoping that the Nemesis Defender uses the correct dummies.

Invoking the Nemesis Defender Pass on a program that already has gone through the DMA Defender Pass also does not render much advantage to the Nemesis Defender Pass. The case where the DMA Defender Pass could make a program vulnerable against the Nemesis attack is when it uses combined memory traces to find a dummy for, as shown in section 4.5.1. In this thesis, the DMA Defender was implemented to never even use this technique, so the following analysis is hypothetical, for when the DMA Defender would ever use combined memory access traces to compensate for.

The Nemesis Defender Pass would compensate for the instruction of 1 clock cycle and the instruction of 3 cycles (the PUSH instruction) separately and additionally for the instruction of 4 clock cycles inserted by the DMA Defender Pass, with again the risk of using a dummy with an incorrect memory trace. In this case the Nemesis Defender Pass also completely undermines the work done by the DMA Defender Pass.

As a conclusion, combining the Nemesis Defender Pass and the DMA Defender Pass does not help one patching a program against both Nemesis and the DMA attack.

4.6 Conclusion

Instruction classification has lead to far less classes than one could expect and even for every class, a dummy to compensate for it was found. The DMA Defender uses static checks to find out which memory regions are accessed by some instruction that needs to be compensated for. This approach might be to coarse-grained, because it can not check for runtime values of a pointer, for example when it is indexed. Inserting runtime checks into the source code is an alternative that unfortunately leads to

an enormous increase in code size. Concolic testing or a non-deterministic priority between the CPU and a peripheral to access memory might be good alternatives, but require investigation.
A second point that needs reconsideration is the way in which SLLVM now patches against PUSH instructions. A single PUSH instruction needs three dummy instructions to be patched, leading to a significant increase in code size. An alternative solution is to patch a PUSH with a PUSH and a POP with a POP instruction, where the dummies should be in an execution path that does not address the stack in between the dummies.
Using combined memory access traces is a more efficient way to compensate for a PUSH, but renders a program vulnerable to Nemesis again. A combination of defenses against Nemesis and the DMA attack does not help to address this problem, but maybe a hardware-based defense against the DMA attack opens new possibilities. The next chapter discusses how that approach was taken to defend against Nemesis.

Chapter 5

Related work

In this chapter, we discuss some research related to the investigation done in this thesis. In the previuos chapter, we showed that the compiler passes to defend against Nemesis and the DMA attack cannot be combined to render a program hardened against both attacks. Here, we look at a hardware-based defense against Nemesis, which could be combined with the software-based DMA defense.
An attack similar to the DMA attack is MEMBUSTER, where the attacker snoops in on adresses on the memory bus. It could undermine the defense against the DMA attack developed in this thesis.

5.1 Hardware-based defense against Nemesis

In chapter 4, we elaborated on the different capabilities of an attacker with DMA capabilities and one with interrupt capabilities. The DMA attacker can distinguish branches by their memory traces, whereas a Nemesis attacker who can interrupt enclave programs is able to discern branches based on the latencies of individual instructions in it. The most efficient way in terms of code size to mitigate against the DMA attack is to compensate for combined memory traces, which are traces of multiple instructions taken one after the other. For example, two instructions having memory traces 1 | 0 | 0 | 1 and 3 | 000 | 001 | 001 respectively together have memory trace

$$1 \mid 0 \mid 0 \mid 1 \ + \ 3 \mid 000 \mid 001 \mid 001 \ = \ 4 \mid 0000 \mid 0001 \mid 1001$$

if the first is executed after the second. To a DMA attacker, two instructions exhibiting this trace together are no different than one instruction exhibiting this whole trace. A Nemesis attacker, however, sees the difference. Using a single dummy instruction having 4 | 0000 | 0001 | 0001 as memory access trace to compensate for two instructions exhibiting the component traces, makes a program that is DMA attack resistant again vulnerable to a Nemesis attacker.

Busi et al. [9] propose a hardware-based mitigation against Nemesis and showing its security by formally proving the preservation of isolation properties. The first solution they propose is to make the interrupt latency an attacker incurs is constant

by padding with extra clock cycles right before the interrupt service routine, so the attacker cannot differentiate between instructions anymore based on how long it takes to serve an interrupt request. Secondly, the authors suggest a padding after the service routine for the interrupt has finished, to defend against resume-to-end timing. To thwart interrupt-counting attacks [52], an interrupt arriving during the padding time before the Interrupt Service Routine (ISR) is started, is ignored. An interrupt arriving during the second kind of padding (after the ISR had finished) is treated as a normal interruption of the enclave. Details on this scheme and a formal description of it is provided in the extended version [10] of [9].

Using the software-based defense against the DMA attack from this thesis, one might choose to pad tuples of instructions with one single dummy like we did for a PUSH instruction in section 4.5.1, as this approach leads to a smaller total code size. This may however render a program vulnerable to Nemesis again and therefore a suggestion to defend against both the DMA attack and the Nemesis attack would be to combine the hardware-based defense for Busi et al. [9] with the software-based defense from this thesis.

5.2 Undermining the DMA defense

The defense against a DMA side-channel attack outlined in this thesis uses three memory addresses that need to be reserved for dummy instructions to access, one in each memory region. These memory addresses should never be used by non-dummy instructions and therefore one knows that these memory lines are accessed if and only if a dummy instruction inserted by the DMA Defender Pass (section 3.4.2) is being executed. Dayeol et al. [30] describe how hardware enclaves on a platform that does not encrypt the memory bus can leak secrets via their memory accesses. Unfolding MEMBUSTER, their new off-chip attack on enclaves, they provide a way to learn on memory addresses being accessed by installing a signal analyzer to amplify the signals sent to the memory module and store them for later analysis [30]. To attack a program that was hardened against the DMA attack, one could observe the DRAM commands placed on the memory bus, and if translation to virtual addresses (to which an approach is also provided in [30]) results in an access to one of the three memory addresses reserved for the DMA Defender Pass, the attacker knows a dummy instruction accessing memory was executed. Assuming the attacker knows how the DMA Defender Pass patches secret-dependent branches, he will be able to find out which secret-dependent branch was executed, which leaks information. A program that is not hardened against the DMA attack is also vulnerable to this, but this requires the attacker to find a mapping between variables used in the source code and the memory locations being accessed, in order to know from the decoded addresses on the memory bus which branch was executed. On the other hand, this is not necessary when performing the MEMBUSTER attack on a program that went through the DMA Defender Pass. Seeing an access to one of the reserved addresses in a clock cycle where the attacker expects a dummy in a certain branch, he knows

that exactly that branch was executed, of course. In this sense, the DMA Defender Pass makes the MEMBUSTER attack a lot easier.

5.3 Conclusion

The DMA Defender Pass is a compiler pass, a software-based solution to the end of hardening programs against the DMA attack. A prevalent question throughout this thesis is whether one could render a program hardened against both Nemesis and the DMA attack. A hardware-based defense against Nemesis was discussed in this chapter, which is combinable with our DMA Defender Pass. Other related work in the context of this thesis is the MEMBUSTER attack, which analyses addresses on the memory bus and could undermine our defense.

Chapter 6

Conclusions and future work

6.1 Conclusion

In this thesis, we investigated a defense against a DMA attacker who is capable of logging the memory access trace performed by the CPU. The research hypothesis of this thesis is that altering the source code of a program at compile-time is a valid technique for this purpose. To this end, we can insert dummy instructions into the program that do not affect it control flow, but make secret-dependent branches indistinguishable to a DMA attacker. In this thesis, we succeeded in using this approach to harden a program that only addresses one memory region against the DMA attack. We believe it is possible to extend to the other memory regions (peripheral space and program memory), but further research is required. Limitations of the approach taken in this thesis is that static checks at compile-time might not always suffice to find which memory region is accessed. Certain instructions may be compensated for in several ways, but a choice needs to be made whether only a defense against the DMA attack is required or that we also need a program to be resilient against Nemesis, a similar attack leaking individual instruction timings.

- **Instruction classification** We classified all instructions possible in the MSP430 architecture. The results of this classification can be summarized as follows: 40 classes were found, that group together instructions using the same addressing modes. Most opcodes fall under the same classes, except for a few opcodes discussed in the chapter.
- **A simulation toolchain** The toolchain used for simulating and classifying instructions was implemented. Using this, all opcodes possible were simulated and classified.
- **Dummy instructions** For every memory trace class found in this thesis, a dummy was found. Some classes need several dummies, but for most classes a single dummy suffices.
- **A new compiler pass** The DMA Defender Pass is the new compiler pass implemented in this thesis. It checks statically whether data memory is accessed

and uses dummy instructions to align branches. Compensating for instructions that access the other memory regions could be subject to future work.

- **Alternatives** We provided ideas for alternatives to the compiler pass investigated in this thesis. Other than the software-based approach in the form of a compiler pass, one could extend the hardware of the platform executing the program to mitigate against the DMA attack. One the one hand, a random signal in the CPU could be an alternative to our approach. On the other hand, a time padding implemented in hardware might cover up the side-channel that was opened by adding DMA functionality to the platform

Contributions in the form of implementation for this thesis can be found here:

- https://github.com/SteffieJoosen/llvm-project/tree/master/llvm/utils/TableGen/MemoryTraceGeneration
- https://github.com/SteffieJoosen/llvm-project/blob/master/llvm/lib/Target/MSP430/MSP430DMADefender.cpp
- https://github.com/SteffieJoosen/sllvm/tree/DMADefender/test/sancus (benchmark programs)

6.2 Future work

The work in this thesis needs extension and possible paths could be to find the memory access traces for the pseudo instructions, which were not yet classified or exploring the alternatives to defend against the DMA attack outlined in chapter 4. Another interesting path to future research is one where a hardware defense instead of a software-based one for the DMA attack could be found.

6.2.1 A hardware defense against the DMA attack

Next to a combination of the DMA defense and the hardware-based protection against Nemesis (section 5.1), another suggestion might be to use a similar approach for the DMA defense as the one used by Busi et al. [9]. The authors suggest a padding of the ISR of which the duration depends on the instruction being executed and the moment in time the interrupt request arrives. A padding might also be used for a memory request to be served when a peripheral does not have to wait one extra clock cycle due to the CPU being given priority (in Sancus) when both request for memory at the same time. One could ensure a constant waiting time for a peripheral of one clock cycle added by the DMA controller. Since Sancus gives priority to the CPU by default, an extra padding is only needed when the CPU does not address memory and the peripheral can access memory immediately. This can easily be distracted from the current decoded instruction in the execution unit. If the padding added to the time to serve a DMA request is one clock cycle, it should be zero when the CPU accessed memory, because the peripheral notices one extra

clock cycle there already. When the CPU accesses memory, a signal should be given to the DMA controller, which adds a delay by default, to omit the delay if a DMA request is generated in that clock cycle. Implementation should not introduce a new side-channel and important to note is that this signal should be inaccessible to the attacker. Investigating this solution could be subject to future work.

6.2.2 Checking accessed memory regions

Unlike suggested above, one could keep the DMA Defender Pass and stick with a software-based defense. In section 3.4.3, we explained that the static checks in the compiler backend do not suffice to harden a program addressing any of the three memory regions. For now, only accesses to data memory were checked and that suffices for the benchmark programs used to assess the defense. In the future, static checks to cover two or even three memory regions would be an interesting extension to the work done in this thesis. However, static checks might not suffice at all to completely defend against the DMA attack and therefore, the use of concolic testing (section 4.3.1) could be an alternative to static checks.

Appendices

Appendix A

Instruction classifications

A.1 MSP430

A.1.1 27 core instructions

In the following overview of the MSP430 core instuctions, PC = program counter and SR = status register.

Single operand instruction opcodes [2]	
Opcode	**Description**
RRC	Rotate right though carry
SWPB	Swap bytes
RRA	Rotate right arithmetic
SXT	Sign extend byte to word
PUSH	Push value onto stack
CALL	Subroutine call (push PC and move source operand to PC)
RETI	Return from interrupt; pop SR and then PC

Jump instruction opcodes [2]	
Opcode	**Description**
JNE/JNZ	Jump if not equal/zero
JEQ/JZ	Jump if equal/zero
JNC/JLO	Jump if no carry/lower
JC/JHS	Jump if carry/higher or same
JN	Jump if negative
JGE	Jump if greater or equal
JL	Jump if less
JMP	Jump unconditionally

Double operand instruction opcodes [2]		
Opcode	**Description**	**Semantics**
MOV	Move source to destination	dst=src
ADD	Add source to destination	dst+=src
ADDC	Add with carry	dst+=(src+C)
SUB	Subtract source from destination	dst-=src
SUBC	Subtract with carry	dst-=(src+C)
CMP	Compare; SR is affected	(dst-src); discard result
DADD	Decimal ADD (BCD) addition	dst += src
BIT	Test bits	(dst & src); discard result
BIC	Bit clear	dst & s̃rc
BIS	Bit set	dst |=src
XOR	Bitwise XOR	dst $^{=}src$
AND	Bitwise AND	dst &= src

A.2 Memory trace classes

This section contains the instructions that were classified with the simulation framework described in chapter 3 of this thesis. The labels EDEd/TONId, EDEp and EDEper/TONIper can stand for any address pointing to respectively data memory, program memory or peripheral space. Registers `rData`, `rProgram`, `rPeriph` are registers allocated to contain some value so that when used in an operand as `@rData` or `2(rData)`, the operand addresses the corresponding memory region.

A.2.1 Double operand instructions

1 | 0 | 0 | 1
add #0x0008, r5
add r4, r5
bic #0x0008, r5
bic r4, r5
mov #0x0008, r5
mov r4, r5

2 | 00 | 00 | 11
add #0x0045, r5
bic #0x0045, r5
mov #0x0045, r5

2 | 00 | 10 | 01
add @rData, r5
add @rData+, r5
bic @rData, r5
bic @rData+, r5
mov @rData, r5
mov @rData+, r5

2 | 10 | 00 | 01
add @rPeriph, r5
add @rPeriph+, r5
bic @rPeriph, r5
bic @rPeriph+, r5
mov @rPeriph, r5
mov @rPeriph+, r5

3 | 000 | 010 | 101
add 2(rData), r5
add EDEd, r5
add &EDEd, r5
bic 2(rData), r5
bic EDEd, r5
bic &EDEd, r5
mov 2(rData), r5
mov EDEd, r5
mov &EDEd, r5

3 | 000 | 000 | 111
add 2(rProgram), r5
add EDEp, r5
add &EDEp, r5
bic 2(rProgram), r5
bic EDEp, r5
bic &EDEp, r5
mov 2(rProgram), r5
mov EDEp, r5
mov &EDEp, r5

3 | 010 | 000 | 101
add 2(rPerph), r5
add EDEper, r5
add &EDEper, r5
bic 2(rPerph), r5
bic EDEper, r5
bic &EDEper, r5
mov 2(rPerph), r5
mov EDEper, r5
mov &EDEper, r5

4 | 0000 | 0001 | 1001
mov #0x0008, 2(rData)
mov #0x0008, TONId
mov #0x0008, &TONId
mov r4, 2(rData)
mov r4, TONId
mov r4, &TONId

4 | 0000 | 0101 | 1001
add #0x0008, 2(rPeriph)
add #0x0008, TONIper
add #0x0008, &TONIper
add r4, 2(rPeriph)
add r4, TONIper
add r4, &TONIper
bic #0x0008, 2(rPeriph)
bic #0x0008, TONIper
bic #0x0008, &TONIper
bic r4, 2(rPeriph)
bic r4, TONIper
bic r4, &TONIper

4 | 0000 | 0101 | 1001
add #0x0008, 2(rData)
add #0x0008, TONId
add #0x0008, &TONId
add r4, 2(rData)
add r4, TONId
add r4, &TONId
bic #0x0008, 2(rData)
bic #0x0008, TONId
bic #0x0008, &TONId
bic r4, 2(rData)
bic r4, TONId
bic r4, &TONId

4 | 00001 | 0000 | 1001
mov #0x0008, 2(rPeriph)
mov #0x0008, TONIper
mov #0x0008, &TONIper
mov r4, 2(rPeriph)
mov r4, TONIper
mov r4, &TONIper

5 | 00000 | 00101 | 11001
add #0x0045, 2(rData)
add #0x0045, TONId
add #0x0045, &TONId
bic #0x0045, 2(rData)
bic #0x0045, TONId
bic #0x0045, &TONId

5 | 00101 | 00000 | 11001
add #0x0045, 2(rPeriph)
add #0x0045, TONIper
add #0x0045, &TONId
bic #0x0045, 2(rPeriph)
bic #0x0045, TONIper
bic #0x0045, &TONIper

5 | 00000 | 00001 | 11001
mov #0x0045, 2(rData)
mov #0x0045, TONId
mov #0x0045, &TONId

5 | 00001 | 00000 | 11001
mov #0x0045, 2(rPeriph)
mov #0x0045, TONIper
mov #0x0045, &TONIper

5 | 00000 | 10001 | 10001
mov @rData, 2(rData)
mov @rData, TONId
mov @rData, &TONId

5 | 10000 | 00001 | 10001
mov @rPeriph, 2(rData)
mov @rPeriph, TONId
mov @rPeriph, &TONId

5 | 00001 | 10000 | 10001
mov @rData, 2(rPeriph)
mov @rData, TONIper
mov @rData, &TONIper

5 | 10001 | 00000 | 10001
mov @rPeriph, 2(rPeriph)
mov @rPeriph, TONIper
mov @rPeriph, &TONIper

5 | 00000 | 10101 | 10001
add @rData, 2(rData)
add @rData, TONId
add @rData, &TONId
add @rData+, 2(rData)
add @rData+, TONId
add @rData+, &TONId
bic @rData, 2(rData)
bic @rData, TONId
bic @rData, &TONId
bic @rData+, 2(rData)
bic @rData+, TONId
bic @rData+, &TONId

5 | 00101 | 10000 | 10001
add @rData, 2(rPeriph)
add @rData, TONIper
add @rData, &TONIper
add @rData+, 2(rPeriph)
add @rData+, TONIper
add @rData+, &TONIper
bic @rData, 2(rPeriph)
bic @rData, TONIper
bic @rData, &TONIper
bic @rData+, 2(rPeriph)
bic @rData+, TONIper
bic @rData+, &TONIper

5 | 10000 | 00101 | 10001
add @rPeriph, 2(rData)
add @rPeriph, TONId
add @rPeriph, &TONId
add @rPeriph+, 2(rData)
add @rPeriph+, TONId
add @rPeriph+, &TONId
bic @rPeriph, 2(rData)
bic @rPeriph, TONId
bic @rPeriph, &TONId
bic @rPeriph+, 2(rData)
bic @rPeriph+, TONId
bic @rPeriph+, &TONId

5 | 10101 | 00000 | 10001
add @rPeriph, 2(rPeriph)
add @rPeriph, TONIper
add @rPeriph, &TONIper
add @rPeriph+, 2(rPeriph)
add @rPeriph+, TONIper
add @rPeriph+, &TONIper
bic @rPeriph, 2(rPeriph)
bic @rPeriph, TONIper
bic @rPeriph, &TONIper
bic @rPeriph+, 2(rPeriph)
bic @rPeriph+, TONIper
bic @rPeriph+, &TONIper

6 | 000000 | 010001 | 110001
mov 2(rData), 2(rData)
mov 2(rData), TONId
mov 2(rData), &TONId
mov EDEd, 2(rData)
mov EDEd, TONId
mov EDEd, &TONId
mov &EDEd, 2(rData)
mov &EDEd, TONId
mov &EDEd, &TONId

6 | 000000 | 000001 | 111001
mov 2(rProgram), 2(rData)
mov 2(rProgram), TONId
mov 2(rProgram), &TONId
mov EDEp, 2(rData)
mov EDEp, TONId
mov EDEp, &TONId
mov &EDEp, 2(rData)
mov &EDEp, TONId
mov &EDEp, &TONId

6 | 010000 | 000001 | 110001
mov 2(rPeriph), 2(rData)
mov 2(rPeriph), TONId
mov 2(rPeriph), &TONId
mov EDEper, 2(rData)
mov EDEper, TONId
mov EDEper, &TONId
mov &EDEper, 2(rData)
mov &EDEper, TONId
mov &EDEper, &TONId

6 | 000001 | 010000 | 110001
mov 2(rData), 2(rPeriph)
mov 2(rData), TONIper
mov 2(rData), &TONIper
mov EDEd, 2(rPeriph)
mov EDEd, TONIper
mov EDEd, &TONIper
mov &EDEd, 2(rPeriph)
mov &EDEd, TONIper
mov &EDEd, &TONIper

6 | 000001 | 00000 | 111001
mov 2(rProgram), 2(rPeriph)
mov 2(rProgram), TONIper
mov 2(rProgram), &TONIper
mov EDEp, 2(rPeriph)
mov EDEp, TONIper
mov EDEp, &TONIper
mov &EDEp, 2(rPeriph)
mov &EDEp, TONIper
mov &EDEp, &TONIper

6 | 010001 | 00000 | 110001
mov 2(rPeriph), 2(rPeriph)
mov 2(rPeriph), TONIper
mov 2(rPeriph), &TONIper
mov EDEper, 2(rPeriph)
mov EDEper, TONIper
mov EDEper, &TONIper
mov &EDEper, 2(rPeriph)
mov &EDEper, TONIper
mov &EDEper, &TONIper

6 | 000000 | 010101 | 110001
add 2(rData), 2(rData)
add 2(rData), TONId
add 2(rData), &TONId
add EDEd, 2(rData)
add EDEd, TONId
add EDEd, &TONId
add &EDEd, 2(rData)
add &EDEd, TONId
add &EDEd, &TONId
bic 2(rData), 2(rData)
bic 2(rData), TONId
bic 2(rData), &TONId
bic EDEd, 2(rData)
bic EDEd, TONId
bic EDEd, &TONId
bic &EDEd, 2(rData)
bic &EDEd, TONId
bic &EDEd, &TONId

6 | 000000 | 010001 | 111001
add 2(rProgram), 2(rData)
add 2(rProgram), TONId
add 2(rProgram), &TONId
add EDEp, 2(rData)
add EDEp, TONId
add EDEp, &TONId
add &EDEp, 2(rData)
add &EDEp, TONId
add &EDEp, &TONId
bic 2(rProgram), 2(rData)
bic 2(rProgram), TONId
bic 2(rProgram), &TONId
bic EDEp, 2(rData)
bic EDEp, TONId
bic EDEp, &TONId
bic &EDEp, 2(rData)
bic &EDEp, TONId
bic &EDEp, &TONId

6 | 010000 | 000101 | 110001
add 2(rPeriph), 2(rData)
add 2(rPeriph), TONId
add 2(rPeriph), &TONId
add EDEper, 2(rData)
add EDEper, TONId
add EDEper, &TONId
add &EDEper, 2(rData)
add &EDEper, TONId
add &EDEper, &TONId
bic 2(rPeriph), 2(rData)
bic 2(rPeriph), TONId
bic 2(rPeriph), &TONId
bic EDEper, 2(rData)
bic EDEper, TONId
bic EDEper, &TONId
bic &EDEper, 2(rData)
bic &EDEper, TONId
bic &EDEper, &TONId

6 | 000101 | 010000 | 110001
add 2(rData), 2(rPeriph)
add 2(rData), TONIper
add 2(rData), &TONIper
add EDEd, 2(rPeriph)
add EDEd, TONIper
add EDEd, &TONIper
add &EDEd, 2(rPeriph)
add &EDEd, TONIper
add &EDEd, &TONIper
bic 2(rData), 2(rPeriph)
bic 2(rData), TONIper
bic 2(rData), &TONIper
bic EDEd, 2(rPeriph)
bic EDEd, TONIper
bic EDEd, &TONIper
bic &EDEd, 2(rPeriph)
bic &EDEd, TONIper
bic &EDEd, &TONIper

6 | 000101 | 000000 | 111001
add 2(rProgram), 2(rPeriph)
add 2(rProgram), TONIper
add 2(rProgram), &TONIper
add EDEp, 2(rPeriph)
add EDEp, TONIper
add EDEp, &TONIper
add &EDEp, 2(rPeriph)
add &EDEp, TONIper
add &EDEp, &TONIper
bic 2(rProgram), 2(rPeriph)
bic 2(rProgram), TONIper
bic 2(rProgram), &TONIper
bic EDEp, 2(rPeriph)
bic EDEp, TONIper
bic EDEp, &TONIper
bic &EDEp, 2(rPeriph)
bic &EDEp, TONIper
bic &EDEp, &TONIper

6 | 010101 | 000000 | 110001
add 2(rPeriph), 2(rPeriph)
add 2(rPeriph), TONIper
add 2(rPeriph), &TONIper
add EDEper, 2(rPeriph)
add EDEper, TONIper
add EDEper, &TONIper
add &EDEper, 2(rPeriph)
add &EDEper, TONIper
add &EDEper, &TONIper
bic 2(rPeriph), 2(rPeriph)
bic 2(rPeriph), TONIper
bic 2(rPeriph), &TONIper
bic EDEper, 2(rPeriph)
bic EDEper, TONIper
bic EDEper, &TONIper
bic &EDEper, 2(rPeriph)
bic &EDEper, TONIper
bic &EDEper, &TONIper

A.2.2 Single operand instructions

1 | 0 | 0 | 1
rra r4
rra.b r4
rrc r4
rrc.b r4
sxt r4
swpb r4

2 | 00 | 10 | 01
pop r4

3 | 000 | 101 | 001
rra @rData
rra @rData+
sxt @rData
sxt @rData+
swpb @rData
swpb @rData+

3 | 000 | 001 | 001
push r4
push.b r4

3 | 101 | 000 | 001
rra @rPeriph
rra @rPeriph+
sxt @rPeriph
sxt @rPeriph+
swpb @rPeriph
swpb @rPeriph+

4 | 0000 | 0001 | 1001
push #0x00046

4 | 0000 | 0101 | 1001
rra 2(rData)
rra &EDEd
rra EDEd
sxt 2(rData)
sxt &EDEd
sxt EDEd
swpb 2(rData)
swpb &EDEd
swpb EDEd

4 | 0101 | 0000 | 1001
rra 2(rPeriph)
rra &EDEper
rra EDEper
sxt 2(rPeriph)
sxt &EDEper
sxt EDEper
swpb 2(rPeriph)
swpb &EDEper
swpb EDEper

A.2.3 Jump instructions

2 | 00 | 00 | 11
jmp LABEL
jne LABEL
jeq LABEL
jnc LABEL
jc LABEL
jn LABEL
jge LABEL
jl LABEL

A.3 Dummy instructions

The following table maps the memory trace classes from appendix A to dummy instructions. The ´rData' register denotes any register that we have previously loaded a value into so that we know the `@rData` operand denotes an address in the data memory region, just like `2(rData)` operand does. We use the same method with `rPeriph` and `rProgram`. N is any constant that is not in {-8, -4, -2, -1, 0, 1, 2, 4, 8}. We use r3 as destination register because this register is used as the constant generator and putting a value into it does not change semantics of the program. We use the constant 8 when the instruction needs to use a constant generated with the constant generator.

No.	Class	Dummy instruction
1	1 | 0 | 0 | 1	`mov #8, r3`
2	2 | 00 | 00 | 11	`mov #N, r3`
3	2 | 10 | 00 | 01	`mov @rPeriph, r3`
4	2 | 00 | 10 | 01	`mov @rData, r3`
5	3 | 000 | 010 | 101	`mov 2(rData), r3`
6	3 | 000 | 101 | 001	`swpb @rData`
7	3 | 000 | 001 | 001	`push rData; pop rData`
8	3 | 000 | 100 | 000	RET
9	3 | 000 | 000 | 111	`mov 2(rProgram), r3`
10	3 | 010 | 000 | 101	`mov 2(rPeriph), r3`
11	3 | 101 | 000 | 001	`swpb @rPeriph`
12	4 | 0000 | 0101 | 1001	`rra 2(rData)`
13	4 | 0001 | 0000 | 1001	`mov #8, 2(rPeriph)`
14	4 | 0101 | 0000 | 1001	`rra 2(rPeriph)`
15	4 | 0000 | 0001 | 1001	`mov #8, 2(rData)`
16	5 | 00000 | 00101 | 11001	`add #N, 2(rData)`
17	5 | 00000 | 10101 | 10001	`add @rData, 2(rData)`
18	5 | 00000 | 00001 | 11001	`mov #N, 2(rData)`
19	5 | 00000 | 10001 | 10001	`mov @rData, 2(rData)`
20	5 | 00000 | 10000 | 00000	RETI
21	5 | 10000 | 00001 | 10001	`mov @rPeriph, 2(rData)`
22	5 | 10000 | 00101 | 10001	`add @rPeriph, 2(rData)`
23	5 | 00001 | 10000 | 10001	`mov @rData, 2(rPeriph)`
24	5 | 00101 | 10000 | 10001	`add @rData, 2(rPeriph)`
25	5 | 00001 | 00000 | 11001	`mov #N, 2(rPeriph)`
26	5 | 00101 | 00000 | 11001	`add #N, 2(rPeriph)`
27	5 | 10001 | 00000 | 10001	`mov @rPeriph, 2(rPeriph)`
28	5 | 10101 | 00000 | 10001	`add @rPeriph, 2(rPeriph)`
29	6 | 000000 | 010101 | 110001	`add 2(rData), 2(rData)`
30	6 | 000000 | 010001 | 110001	`mov 2(rData), 2(rData)`
31	6 | 000000 | 000101 | 111001	`add 2(rProgram), 2(rData)`
32	6 | 000000 | 000001 | 111001	`mov 2(rProgram), 2(rData)`
33	6 | 010000 | 000001 | 110001	`mov 2(rPeriph), 2(rData)`
34	6 | 010000 | 000101 | 110001	`add 2(rPeriph), 2(rData)`
35	6 | 000001 | 010000 | 110001	`mov 2(rData), 2(rPeriph)`
36	6 | 000101 | 010000 | 110001	`add 2(rData), 2(rPeriph)`
37	6 | 000001 | 000000 | 111001	`mov 2(rProgram), 2(rPeriph)`
38	6 | 000101 | 000000 | 111001	`add 2(rProgram), 2(rPeriph)`
39	6 | 010001 | 000000 | 110001	`mov 2(rPeriph), 2(rPeriph)`
40	6 | 010101 | 000000 | 110001	`add 2(rPeriph), 2(rPeriph)`

Bibliography

[1] Simulator sancus-sim. URL: https://distrinet.cs.kuleuven.be/software/sancus/doc.php.

[2] The complete MSP430 instruction set of 27 core instructions. URL: https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf.

[3] TEE System Architecture. globalplatform.org, Nov 2018.

[4] GTKWave 3.3 Wave Analyzer User's Guide, November 2020.

[5] J. Agat. Transforming out timing leaks. In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '00, pages 40–53. ACM, 2000.

[6] ARM Limited. ARM Security Technology Building a Secure System using TrustZone® Technology. URL: https://developer.arm.com/documentation/genc009492/c, April 2009.

[7] B. Penuelas. vcdvcd. URL: https://github.com/bmpenuelas/vcdvcd#vcdcat.

[8] M. Bognár. Analyzing side-channel leakage in secure dma solutions, 2020.

[9] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In 2020 IEEE 33rd Computer Security Foundations Symposium (CSF), pages 262–276, 2020.

[10] 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. 2020.

[11] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. Fact: A flexible, constant-time programming language. In 2017 IEEE Cybersecurity Development (SecDev), pages 69–76. IEEE, 2017.

[12] E. De Mulder, P. Buysschaert, S. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic analysis attack on an fpga implementation of an elliptic curve cryptosystem. In EUROCON 2005 - The International Conference on "Computer as a Tool", volume 2, pages 1879–1882, 2005.

[13] Z. Dittia, G. Parulkar, and J. Cox. The apic approach to high performance network interface design: protected dma and other techniques. In Proceedings of INFOCOM '97, volume 2, pages 823–831 vol.2. IEEE, 1997.

[14] J.-E. Ekberg, K. Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. IEEE security privacy, 12(4):29–37, 2014.

[15] Free Software Foundation, Inc. Using the GNU Compiler Collection (GCC). URL: https://gcc.gnu.org/onlinedocs/gcc.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In Operating Systems Review (ACM), volume 37, pages 193–206, New York, NY, 2003. Association for Computing Machinery.

[17] Q. Ge, Q. Ge, Y. Yarom, Y. Yarom, D. Cock, D. Cock, G. Heiser, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of cryptographic engineering, 8(1):1–27, 2018.

[18] Generated by DoxyGen. llvm::BasicBlock Class Reference. URL: https://llvm.org/doxygen/classllvm_1_1BasicBlock.html.

[19] Generated by DoxyGen. llvm::MachineInstr Class Reference. URL: https://llvm.org/doxygen/classllvm_1_1MachineInstr.html.

[20] O. Girard. OpenMSP430, November 2017. Rev. 1.17.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In Detection of Intrusions and Malware, and Vulnerability Assessment, volume 9721 of Lecture Notes in Computer Science, pages 279–299, Cham, 2016. Springer International Publishing.

[22] J. Guilbon. Introduction to trusted execution environment: Arm's trustzone.

[23] Hans WInderix. Benchmark programs for Nemesis Defender. URL: https://github.com/hanswinderix/sllvm/tree/master/test/sancus/, February 2021.

[24] S. L. Harris and D. M. Harris. 6 - architecture. In S. L. Harris and D. M. Harris, editors, Digital Design and Computer Architecture, pages 294–383. Morgan Kaufmann, Boston, 2016.

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

[26] J. Van Bulck, J. T. Mülhberg, M. Bognár, F. Alder. Sancus examples. URL: https://github.com/sancus-tee/sancus-examples.

[27] P. Kankowski. x86 Machine Code Statistics. URL: https://www.strchr.com/x86_machine_code_statistics.

[28] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. Journal of cryptographic engineering, 1(1):5–27, 2011.

[29] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, Advances in Cryptology — CRYPTO '96, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[30] 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. 2019.

[31] U. Lee and C. Park. Softee: Software-based trusted execution environment for user applications. IEEE access, 8:121874–121888, 2020.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE Symposium on Security and Privacy, pages 605–622, 2015.

[33] LLVM Project. The LLVM Compiler Infrastructure. URL: https://llvm.org/.

[34] LLVM Project. LLVM's Analysis and Transform Passes. URL: https://llvm.org/docs/Passes.html, May 2021.

[35] LLVM Project. Tablegen overview. URL: https://llvm.org/docs/TableGen, May 2021.

[36] LLVM Project. Writing an LLVM Pass. URL: https://llvm.org/docs/WritingAnLLVMPass.html, May 2021.

[37] Y. Lyu and P. Mishra. A survey of side-channel attacks on caches and countermeasures. Journal of Hardware and Systems Security, 2, 03 2018.

[38] S. Mangard, E. Oswald, and T. Popp. Power analysis attacks: revealing the secrets of smart cards. Springer, New York, 2007.

[39] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. Open-tee – an open virtual trusted execution environment. In 2015 IEEE Trustcom/BigDataSE/ISPA, volume 1, pages 400–407. IEEE, 2015.

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

[41] D. Ray and J. Ligatti. Defining code-injection attacks. SIGPLAN Not., 47(1):179–190, Jan. 2012.

[42] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen. Coming challenges in microarchitecture and architecture. Proceedings of the IEEE, 89(3):325–340, March 2001.

[43] 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.

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

[45] K. Sen. Concolic testing. In Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, ASE '07, pages 571–572. ACM, 2007.

[46] G. Somani, M. S. Gaur, D. Sanghi, M. Conti, and R. Buyya. Ddos attacks in cloud computing: Issues, taxonomy, and future directions. Computer communications, 107:30–48, 2017.

[47] Stephen St. Michael. What Is a Microarchitecture? Understanding Processors and Register Files in an ARM Core. URL: https://www.allaboutcircuits.com/technical-articles/what-is-a-microarchitecture-processor-register-files-ARM-core/, February 2019.

[48] Texas Instruments. MSP430 Family User Guide, December 2004.

[49] Texas Instruments. Direct Memory Access (DMA) Controller Module, August 2012.

[50] Texas Instruments. Calling Convention and ABI Changes in MSP GCC, February 2015.

[51] V. Costan and S. Devadas. Intel SGX Explained. IACR Cryptology ePrint Archive, 2016(086):1-118, 2016.

[52] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In Proceedings of the 2018 ACM SIGSAC Conference on computer and communications security, CCS '18, pages 178–195. ACM, 2018.

[53] H. Winderix. Security Enhanced LLVM. Master's thesis, KU Leuven, Belgium, 2018.

[54] H. Winderix, J. T. Mühlberg, and f. Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. 2021.

[55] Winderix et al. Attr.td - Attribute Definitions. URL: https://github.com/hanswinderix/llvm-project/blob/master/clang/include/clang/Basic/Attr.td, May 2021.

[56] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, page 719–732, USA, 2014. USENIX Association.