MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling

Marton Bognar
marton.bognar@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Hans Winderix
hans.winderix@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Jo Van Bulck
jo.vanbulck@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Frank Piessens
frank.piessens@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

***Abstract*—Preventing information leakage through microarchitectural side channels is notoriously challenging and, as a result, an important research question. Recent work has shown the viability of compiler-assisted instruction balancing for small, embedded processors with deterministic timing behavior. However, even in such small processors, more subtle microarchitectural side channels continue to be discovered, complicating mitigation efforts.**

We propose a methodology for augmenting an existing instruction set architecture (ISA) specification with instruction-specific microarchitectural leakage traces obtained through principled *microarchitectural profiling*. Building on this *augmented ISA*, it becomes possible to construct software tools to detect and mitigate certain side-channel vulnerabilities. As a case study, we instantiate our methodology on a recently uncovered microarchitectural side channel, which is based on cycle-level timing differences of direct memory access (DMA) requests on 16-bit openMSP430 processors. Using the augmented ISA obtained for this side channel through microarchitectural profiling, we develop practical attack scenarios and extend a state-of-the-art compiler-based mitigation and a binary validation tool, both of which originally targeted a coarser-grained, instruction-granular side channel. Our benchmarks show that our extended compiler mitigation, while still mitigating the instruction-granular leakage, also eliminates the cycle-accurate DMA information leakage without incurring any additional overhead.

1. Introduction

With the rise of the Internet of things (IoT), embedded devices are increasingly used for security- and safety-critical tasks and, as a result, are subject to attacks at different levels of the hardware-software stack. However, to meet stringent cost, power consumption, and real-time requirements, these devices are typically not equipped with established security features that are commonly found in higher-end CPUs. A proposed solution for improving security in embedded systems is trusted execution environments (TEEs) [1]–[8], which offer a lightweight hardware root-of-trust for isolation and attestation of small, security-critical software components, called *enclaves*.

While the architectural isolation guarantees offered by embedded TEEs are well-understood, even to the point of formal verification efforts [5], [8], enclave secrets may still leak through microarchitectural side channels, which are notoriously hard to reason about [9]. Importantly, however, due to the absence of microarchitectural optimizations such as caches, pipelining, and speculative execution, embedded processors commonly feature predictable instruction timings. This results in a significantly reduced attack surface in terms of side channels. Nevertheless, even on embedded processors with fully deterministic instruction timing behavior, it has been shown that secrets may still leak through the overall execution time of secret-dependent branches [10] and even through individual instruction latencies within those branches [11].

The most general way to rule out information leakage from timing side channels is to adopt constant-time programming practices such as control and data flow linearization [12]–[15] to eliminate all secret-dependent code and data transfers directly at the application level, e.g., through the use of vetted cryptographic libraries [5], [16]. Unfortunately, proper constant-time programming requires significant expert developer effort and has been repeatedly shown to be prone to oversights [17]. The constant-time approach, hence, does not scale well to general-purpose applications, highlighting the need for automated side-channel hardening approaches.

In the context of embedded processors with deterministic instruction timing behavior, Winderix et al. [18] recently proposed a transparent, compile-time transformation that carefully balances secret-dependent branches with compensation code, making sure that both sides of a secret-dependent branch exhibit the same leakage. This transformation makes branches indistinguishable, even for advanced adversaries who can observe not only start-to-end execution time but also individual instruction timings in a so-called Nemesis [11] interrupt-latency timing attack. Their work also highlights the potential performance benefit the balancing approach has over linearization in this class of systems. Other works have shown that automated side-channel validation can be performed at the binary level to ensure that programs are free from start-to-end timing [19] and instruction-granular interrupt-latency [20] leakage. Importantly, these works rely on readily available instruction *leakage traces* (i.e., precise descriptions of the instruction's side channel leakage), which in the case of execution times are often included in the instruction set architecture (ISA) specification of embedded, real-time processors [21], [22]. However, relying on such relatively coarse-grained information may not suffice in the face of more advanced and undocumented microarchitectural effects [23].

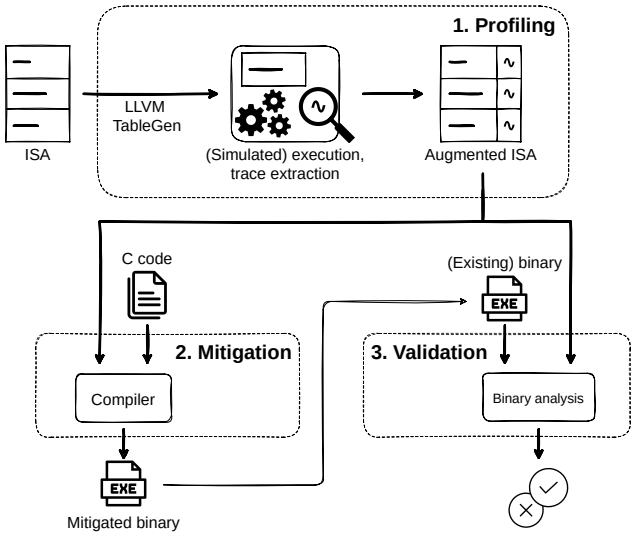


Figure 1: Overview of our approach.

Methodology. In response to these advanced attack vectors, we propose a methodology to detect and mitigate known side channels that expose predictable, instruction-dependent behavior. We systematically *augment* a vendor-provided ISA with instruction-granular, microarchitectural leakage traces that can subsequently be used in tools for mitigating or detecting the side channel. Our three-step approach is outlined in Figure 1.

First, in a *profiling* step, we execute and analyze each instruction supported by the ISA, extracting its side channel leakage trace. This analysis can happen via a cycle-accurate Verilog simulation or on real hardware. To ensure that this profiling is systematic and complete, we use LLVM's TableGen tool [24] to enumerate all valid instructions and their addressing modes. The result of this initial profiling step is an augmented ISA with a microarchitecture- and side-channel-specific leakage trace (denoted $\sim$ in the figure) for every instruction.

Second, we extend an open-source compiler pass [18] that performs instruction-granular balancing to eliminate side-channel leakage in secret-dependent branches. This step is also dependent on the augmented ISA. Concretely, the extracted leakage traces of the instructions determine how to perform the balancing so that the two branches exhibit the same leakage, making them indistinguishable.

Finally, we extend an open-source binary analysis tool [20] with the augmented ISA. The resulting tool can validate whether binaries exhibit secret-dependent leakage under the profiled side channel, making it possible to find vulnerable third-party binaries and validate the correctness of our compiler defense.

DMA side channel. To demonstrate the applicability of our methodology, we instantiate it for a recently described side-channel attack [23] exploiting subtle timing differences of direct memory access (DMA) requests due to contention in openMSP430 processors [25]. The openMSP430 core is an open-source implementation of Texas Instruments' popular, low-power MSP430 [22] microcontroller, which has been the basis of several academic security architectures, such as SMART [1], Sancus [2], and the VRASED family of systems [5]–[7]. Notably, several of these security architectures [2], [5]–[7] support untrusted peripherals by explicitly limiting DMA requests to unprotected parts of the memory. However, it has been recently discovered that such unprivileged DMA requests can trigger contention on the shared memory bus, causing a delay in the request depending on any concurrent memory accesses by the CPU [23]. As a result of this contention, the timing of untrusted DMA requests can reveal a cycle-accurate memory access trace of a victim program executing on the CPU.

We apply our methodology to defend against a capable adversary that combines both the cycle-granular memory access sequences obtained via DMA contention and the instruction timing sequences leaked via interrupt latency differences with Nemesis [11]. First, in the profiling step, we augment the ISA by systematically collecting leakage traces for all instructions under both DMA and Nemesis attacker models. Next, showing the increased power of DMA attackers and underlining the importance of adequate mitigations, we demonstrate several practical, end-to-end attacks on binaries hardened by a state-of-the-art Nemesis compiler mitigation [18]. Afterward, we use our augmented ISA to extend this compiler pass to mitigate leakage not only from overall execution time and interrupt latencies but also cycle-accurate DMA delays. Our experimental evaluation shows that this additional protection does not cause an increased performance impact on the original benchmarks. Finally, by extending an existing binary analysis tool [20] with our augmented ISA, we further increase confidence that the resulting binaries are free from secret-dependent side-channel leakage under the considered adversary model.

During our work, our systematic approach enabled us to uncover oversights in the original binary validation tool and several (documented as well as undocumented) derivations of the openMSP430 [25] core from the base TI MSP430 instruction timing specification [22].

Contributions. In summary, we make the following contributions.

- We provide a principled methodology for profiling and mitigating microarchitectural leakage on embedded processors exhibiting deterministic instruction timing behavior.
- We perform the first in-depth study of a recently described DMA-based side channel [23] on the openMSP430 platform, including several practical end-to-end attacks on programs that were protected by a state-of-the-art compiler mitigation.
- We develop an improved compiler defense, informed by the instruction profiling step, to protect against both instruction-granular Nemesis and cycle-accurate DMA side-channel attacks.
- We extend a binary validation tool that is likewise informed by the instruction profiling step to statically detect DMA-based leakage in programs.
- We evaluate the security and performance of the defense and find that it incurs no additional overhead compared to the original mitigation.

Our profiling toolchain, the case study attacks, our extended tools, and the benchmarks used are available at https://github.com/martonbognar/microprofiler.

2. Background

2.1. The openMSP430 microcontroller

OpenMSP430 [25] is an open-source implementation of Texas Instruments' 16-bit MSP430 microcontroller architecture [22]. OpenMSP430 implements the TI MSP430 ISA with almost cycle-level accuracy, with only a few well-documented differences [25]. The openMSP430 core is implemented in the Verilog hardware description language (HDL), allowing the design to be simulated in software with cycle-level accuracy. It can also be synthesized and flashed onto an FPGA or converted into an ASIC.

The schematic view of the openMSP430 core is shown in Figure 2. There are no caches, branch predictors, or other advanced microarchitectural features. All CPU memory accesses take a single clock cycle to complete, regardless of their origin and destination. The only two stages of the CPU pipeline are the *frontend* (FE), which fetches and decodes instructions from program memory, and the *execution unit* (EU), which handles all arithmetic and memory operations. Application software is presented with a classical Von-Neumann architecture view with a single address space for data and code, even though the physical memory is composed of three separate partitions: data memory (DMEM), program memory (PMEM), and memory-mapped I/O (MMIO). These partitions are connected to the memory backbone through different interfaces and can serve requests in parallel.

2.2. DMA contention side channel

Microarchitectural side-channel attacks [9] often rely on contention for a resource shared between the victim and the attacker. Examples include caches [26], DRAM [27], or a CPU execution unit [11].

The DMA side channel [23] recently demonstrated on openMSP430 platforms results from contention for the memory bus. As illustrated in Figure 2, the shared memory bus connects the three memory partitions to the memory backbone, which in turn serves the memory requests of the connected CPU and any DMA devices. As there is one bus between each memory partition and the memory backbone, if multiple devices issue requests to the same partition in parallel, only one can be handled at a time, the others are delayed. If the CPU has priority, which is the default case on openMSP430, any DMA requests made to the same partition in the same cycle as the CPU will be delayed until the next available cycle (as indicated by the clock symbol in Figure 2 where both the DMA device and the CPU concurrently access data memory). Hence, attackers controlling untrusted peripheral devices can create contention for the shared memory bus and measure delays in their DMA requests to learn in which exact cycles the CPU accessed a specific memory partition.

This attack provides an important advantage over the previous state-of-the-art, instruction-granular Nemesis [11] attack demonstrated on MSP430 platforms. Concretely, while Nemesis exposes individual execution times for every instruction in a victim program through interrupt latencies, the DMA side channel reveals finer-grained, cycle-accurate timings of memory bus accesses *within* individual instructions. As such, even instructions with

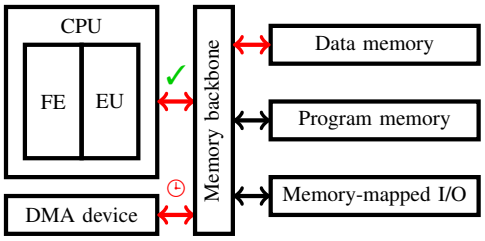


Figure 2: Main components of the openMSP430 core.

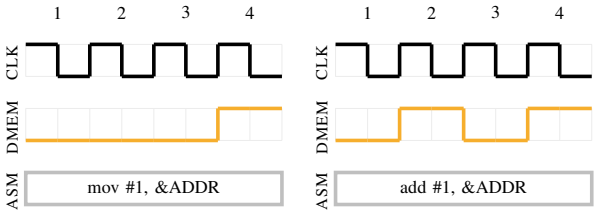


Figure 3: Memory traces of two instructions [23].

the same cycle length might become distinguishable by observing in which cycle they access the memory bus. This is shown in Figure 3, which depicts memory traces of the `mov` and `add` instructions with the same operands. These instructions both take 4 cycles to execute, making them indistinguishable to both start-to-end timing and Nemesis adversaries. With the DMA leakage, however, an attacker can differentiate them based on whether they access memory in the second execution cycle.

While the existence of the DMA side channel has recently been demonstrated [23], until now no in-depth analysis has been carried out to understand the leakage, no practical end-to-end attacks have been demonstrated, and no mitigations have been proposed.

2.3. Security architectures on openMSP430

OpenMSP430 does not support a memory protection mechanism by default. However, it has been an attractive target for building security extensions thanks to its open-source design, extensibility, software support, and the popularity of the base TI MSP430 microcontrollers. Some security architectures which have been built on openMSP430 that consider DMA requests in their threat model are the following:

2.3.1. SMART. SMART [1] adds remote attestation functionality to openMSP430. Its design explicitly assumes DMA to be disabled during the execution of security-sensitive code. However, its implementation is not open-sourced, so the precise impact of the DMA side channel we examine cannot be assessed.

2.3.2. Sancus. The Sancus TEE [2] extends openMSP430 with modified memory access logic and additional instructions to allow hardware-based isolation and attestation of embedded enclaves. While the original Sancus architecture was built on an older version of openMSP430 without DMA support, recent upstream Sancus cores [28] come with additional memory access control logic that allows DMA requests to unprotected memory regions during

enclaved execution. As a result, recent Sancus cores have been the subject of DMA side-channel analysis [23].

2.3.3. VRASED family. VRASED [5] presents a mechanism similar to SMART to provide remote attestation capabilities for openMSP430 microcontrollers. VRASED itself has been used as the basis for several derived systems [6], [7], [29], [30]. In VRASED, DMA requests are explicitly disallowed while protected software performs the attestation. However, it has been shown [23] that timing side-channel leakage from prematurely terminated DMA requests is still possible on VRASED.

3. System model and methodology

In this section, we expand on the properties of the systems and the attacks to which our methodology applies. Afterward, we discuss our methodology in detail.

3.1. System model

In line with previous works [18]–[20] on side-channel mitigations for embedded devices, we assume an elementary IoT processor that exhibits predictable timing behavior. More specifically, we assume that the microarchitectural behavior (e.g., the timing of memory requests or the number of clock cycles until instruction retirement) for executing one instruction is fully determined by that instruction alone. Notably, in contrast to many timing attacks on higher-end platforms [9], this assumption means that the microarchitectural behavior of a program's execution is not in any way influenced by another – possibly untrusted – program executing beforehand or in parallel on the same processor. We, hence, explicitly target low-end IoT processors featuring deterministic execution timings and lacking advanced microarchitectural features such as caches, branch prediction, or out-of-order execution.

As discussed in earlier work [18] and Section 8, there are other candidates for such systems aside from the openMSP430 platform we study, including Atmel AVR, TI MSP430, and ARM Cortex-M23.

3.2. Attacker model

Our approach aims to mitigate known side-channel attacks with specific properties. Therefore, we assume that the target system executes a sensitive program, referred to as the victim *enclave*, whose secrets are protected at the architectural level (e.g., if DMA requests are enabled, the secrets cannot be directly accessed in memory by a malicious device). Our methodology has one important requirement for the side-channel attack it targets: the side channel can only leak information that depends purely on the executed instruction and its operands, as this is the granularity of information in the ISA at which we augment. Otherwise, we have no additional limitations on the profiled attack; our methodology can apply to attacks requiring access to unprotected software, connecting peripherals, or even performing physical measurements on the device.

Nemesis and DMA attack. The two side-channel attacks we focus on in this paper are the Nemesis interrupt-latency attack [11] and the DMA contention attack [23]. Nemesis relies on the attacker precisely timing an interrupt request during the enclave's execution, which implies untrusted code execution and access to a cycle-accurate programmable timer device. The DMA attack relies on controlling a connected peripheral that can issue and time DMA requests but does not necessarily need additional control over the software. This requirement can be achieved either by the attacker having physical access to the target device or by compromising the firmware of an already connected sophisticated peripheral, such as a network controller [31].

3.3. Methodology

The goal of our approach is to systematically augment the description of every instruction in a given ISA with microarchitectural *leakage traces*, representing the leakage of the given instruction in an appropriate format for the given side channel. This augmented ISA can subsequently inform compiler defenses and binary validation tools to ensure that a given program can execute on the target platform without leaking secret-dependent control-flow decisions to a side-channel adversary. For an overview of our approach, see also Figure 1.

Profiling. Our microarchitectural profiling approach complements and refines state-of-the-art embedded side-channel mitigations [18]–[20], which rely solely on vendor-provided, ISA-level timing information. This information is sufficient to protect against previous side-channel attacks, such as Nemesis [11], based on the execution times of instructions, which is part of the MSP430 ISA specification [22]. However, we show that vendor-provided information is insufficient to protect against advanced adversaries who exploit finer-grained microarchitectural side-channel leakage, for example resulting from DMA contention.

In our approach, we augment the ISA with additional microarchitectural leakage traces. To ensure complete coverage, we automate the generation of instructions. The enumeration of all possible instructions could be extracted manually from a vendor-supplied ISA specification. However, for our case study, we use LLVM TableGen [24] to generate all valid MSP430 instruction instances based on a structured description of the MSP430 ISA [22] in the MSP430 backend of the LLVM compiler framework [32]. For each instruction in the ISA, we extract a leakage trace for the microarchitectural side channel under consideration, e.g., the DMA contention side channel in our case. In our experimental setup, we derive the leakage trace by running the openMSP430 instruction in an HDL simulator and extracting the exposed microarchitectural signals directly from the resulting value change dump (VCD) file. Alternatively, however, the leakage trace for an instruction could also be extracted by running it on real hardware and using the same attack technique that would be used by an attacker when trying to extract information from real software. This latter approach is especially applicable when no white-box, cycle-accurate HDL simulator is available for the target system that could model the

side-channel leakage. For example, it is also possible to synthesize commercial hardware designs onto an FPGA or ASIC and collect the leakage traces by loading attacker software or attaching custom measurement hardware.

Compiler mitigation. The resulting augmented ISA, extended with leakage traces for each instruction, can be viewed as a refined contract between the hardware and the compiler. The contract describes the microarchitectural behavior and leakage of instructions, which in turn can be used by the compiler to compile side-channel hardened programs. That is, for secret-dependent branches, the compiler takes care to generate balanced code such that all instructions in both paths exhibit identical leakage traces, making them indistinguishable for a side-channel adversary. The compiler can insert dummy (no-op) instructions as needed and can also take into account leakage traces from different side channels when generating the balanced branches. In Section 6, we show that a single, well-informed compiler pass can simultaneously address information leakage from the Nemesis interrupt latency and DMA contention side channels.

Binary analysis. Lastly, we also use the augmented ISA to extend a standalone, open-source binary analysis tool [20], which was originally developed to detect start-to-end and instruction-granular timing leakage on TI MSP430 platforms. For third-party programs, this extended tool can detect potential vulnerabilities resulting from the analyzed side-channel attack. By analyzing binaries generated by our compiler defense, we can also validate that our defense functions correctly and fully mitigates the studied vulnerability.

4. Instruction profiling and analysis

As a case study to demonstrate our methodology, we instantiated it to mitigate the DMA contention side channel described before. This section describes the profiling step for this attack and an extensive analysis of the extracted leakage traces.

4.1. Instruction generation

We used LLVM's TableGen tool [24] to generate all possible MSP430 assembly instructions to avoid overlooking any instruction or addressing mode during the profiling step. TableGen provides the internal LLVM representation of all instructions, specifically the opcode and the operand types for each supported instruction. This information is (with the notable exception of additional delays when writing to program memory [23]) sufficient to statically determine the Nemesis leakage trace, i.e., the individual execution timings of every MSP430 instruction.

For the considered DMA side channel, however, the TableGen information in itself is insufficient for multiple reasons. First, in contrast to the instruction timings, the DMA leakage trace is not included in the ISA specification [22], [25]; it requires profiling via executing the instruction. Second, the TableGen representation uses an abstract representation for register and memory operands, but as we will show, the concrete operand values also influence the leakage trace. For this reason, we generate multiple assembly instruction instances from one TableGen representation, instantiating it with different register and memory values, for example addressing the three separate memory partitions of openMSP430. We discuss the challenges we encountered with generating the instruction instances in Section 8.

Next, we feed the generated assembly instructions into a testbench program. We extract the memory leakage trace of each instruction instance by running the testbench program in the openMSP430 Verilog simulator (based on Icarus Verilog [33]). The simulation includes the complete microprocessor and all signals with cycle-level accuracy, ensuring that the generated traces are correct and identical to a core synthesized to an FPGA or ASIC.

4.2. Microarchitectural analysis

Although automatically profiling the instructions is enough for constructing the compiler defense and the static analysis tool, it is important to better understand the source of the leakage (see also our discussion in Section 8). In the following, we provide a detailed explanation of the factors that influence the memory leakage trace of a given instruction on the openMSP430 core.

4.2.1. Leakage granularity. Some properties of the openMSP430 design contribute to the high granularity of this side channel's leakage:

- As mentioned in Section 2, the captured leakage traces have a cycle-accurate granularity due to memory accesses on openMSP430 taking one cycle to complete.
- The three memory partitions (Figure 2) have separate buses connecting them to the memory backbone, which means the contention between the CPU and the DMA device only happens if they target the same partition. This enables the attacker to distinguish accesses to the different partitions.
- The lack of a cache or speculative execution on openMSP430 ensures that every memory access is deterministically served from the main memory, and thus visible in the leakage trace.

4.2.2. Instruction fetching. Before an instruction is executed, it needs to be fetched from program memory by the processor frontend. This fetch happens in the cycle before the instruction starts executing. If the instruction contains one or two memory addresses or constant value operands, these are placed next to the instruction in program memory and fetched in the first one or two cycles of the instruction's execution (cf. Section 4.2.4).

4.2.3. Different instructions with identical operands. Figure 3 illustrated that different instructions with the same operands can produce different leakage traces. Note that this figure only showed the data memory access trace since the activity on the other two memory buses for these two instruction instances is identical.

In other cases, different instructions with the same operands can result in identical leakage traces across the three memory buses. Figure 4 illustrates this for an `add` and a `sub` instruction instance with the same operands.

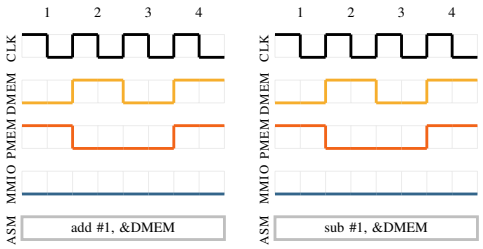


Figure 4: Identical leakage traces for `add` and `sub`.

TABLE 1: Addressing modes in the MSP430 ISA.

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

4.2.4. Addressing modes. The MSP430 instruction set supports seven different addressing modes [22], which are listed in Table 1. Depending on the addressing mode for the source and destination operands, different memory accesses will be made in different cycles. For instance, register mode always reads or writes the content of a register, thus requiring no memory access aside from fetching the instruction word from program memory. All other modes access at least one memory location. Indexed, symbolic, absolute, and immediate modes load a constant value following the instruction word in program memory during the first cycle(s) of execution. The loaded value is subsequently interpreted as either an immediate value or a memory address. In the case of the latter, the memory is once again accessed at the given address. Indirect modes also access the memory once, at the address provided in the source register. Symbolic and absolute modes use the same instruction encoding as indexed mode, with the program counter and the status register used as their register inputs, respectively. As a result, they also have the same TableGen representation, which needs to be considered during the generation of instruction instances for the profiling step.

Using different operand types for the same instruction often changes not only the memory accesses but also the execution time. Figure 5 depicts an example of two `mov` instruction instances that differ in their first operand, resulting in different execution times.

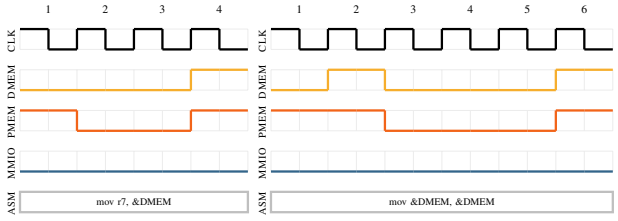


Figure 5: Different operands producing different execution timings.

4.2.5. Value of operands. Even if two instructions have the same operand types, their values might still influence the leakage trace.

Registers. Register `r0` as the destination operand makes instructions one cycle longer, as this register is the program counter. Writing to it delays fetching the next instruction by one cycle, as this instruction depends on the new program counter value.

Memory addresses. Depending on which of the three memory partitions a memory address falls into, a different bus will be active. Figure 6 shows such an example, where the only difference between the instructions is the location of the target memory address.

Writing to program memory is not a standard MSP430 feature; support for this class of instructions was added to Sancus separately [23]. These writes take an extra cycle for the same reason as writing to the program counter register. If the address of the next instruction is overwritten, the next instruction can only be fetched after the write completes. Therefore, these instruction instances delay fetching the next instruction by one cycle.

Immediate values. For the immediate addressing mode, the MSP430 constant generator registers `r2` and `r3` need to be considered. These special registers can efficiently generate some commonly used constants. Most immediate operands are stored adjacent to their instruction in program memory and fetched from there during execution. However, if a constant can be produced via the constant generator registers, the memory access can be avoided and less memory is required to encode the instruction. Figure 7 shows two instructions where the first one takes one cycle less to execute because the number `1` can be produced by the constant generator, while `42` is fetched from program memory. The constant generator hardware can supply the following numbers: $\{-1, 0, 1, 2, 4, 8\}$ [22].

4.3. Leakage classes

After generating the leakage traces, we grouped the instructions with the same leakage trace into *leakage classes*. The 3,389 generated instruction instances could be grouped into 65 separate leakage classes, some of which contain a single instruction instance, while others include more than 70. These leakage classes were later used for constructing our compiler defense (cf. Section 6).

We found that the number of leakage classes can be further reduced when only considering *well-formed programs* that comply with the following reasonable requirements within secret-dependent branch regions:

1) The program only executes code from program memory. This is not a concern on openMSP430, as the specification does not allow executing code from data memory [25]. On other microarchitectures, executing code from data memory is not commonly needed for general-purpose programs.
2) The stack pointer always points to data memory. This is already required for the correct functioning of the protected program. Moreover, stack switching is commonly enforced by TEEs to

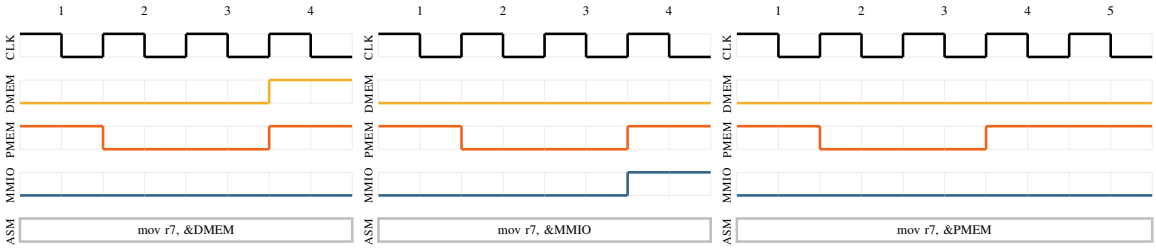


Figure 6: Different memory locations producing different leakage traces and execution times.

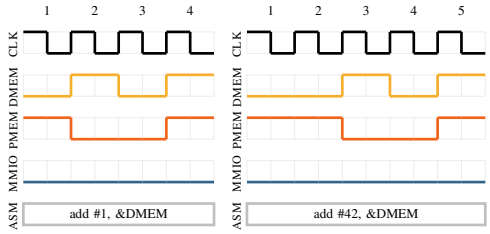


Figure 7: `add` with different constant values as source.

ensure the confidentiality and integrity of enclaved execution. For instance, Sancus [2] and VRASED [5] automatically switch to a protected secret stack (in data memory) on enclave entry.

3) All software pointer dereferences resolve inside the program's protected data memory. This is also a reasonable requirement, as enclave software is supposed to explicitly validate user-provided pointers [34] and never dereference them inside secret-dependent branch regions. That is, accessing attacker-controlled shared memory locations in secret-dependent branch regions exposes greater security risks than possible side-channel leakage, as a DMA attacker or custom MMIO peripheral could directly detect reads and writes to unprotected memory regions. Reading from (protected) program memory may be a valid use case for pointers to program constants, but the compiler can straightforwardly ensure that such constants are also placed in data memory.

Notably, we found that adhering to these requirements drastically reduced the number of generated instruction instances, from 3,389 to 551, and the number of leakage classes, from 65 to 23 (cf. the final leakage classes in Appendix A). These requirements also make the compiler defense (cf. Section 6) simpler and more efficient, requiring less complex analyses and less instrumentation.

5. Attack case studies

To demonstrate the feasibility and relevance of the DMA contention side-channel attack, we describe two end-to-end attacks and one covert channel scenario in this section. The attacks are based on two hardened programs from the third-party benchmark suite of the compiler defense to protect MSP430 programs against instruction-granular Nemesis leakage [18]. The full benchmark suite, including the two selected programs attacked below, is further discussed in Section 7.

5.1. End-to-end attack setup

Our experiments were conducted on the Sancus platform [2], using the openMSP430 simulator. The victim code runs inside a protected Sancus enclave, while the attacker has full control over 1) a DMA-capable peripheral and 2) the untrusted software on the device. The latter orchestrates the attack; it sets up the peripheral and launches the target enclave.

The malicious peripheral is based on the open-source attacker peripheral used in the original DMA side-channel proof-of-concept attack [23], which consists of fewer than 100 lines of Verilog code. The peripheral has a timer, which can be configured from untrusted software. Once this timer expires, the peripheral starts issuing DMA requests in every clock cycle to a specific address. This address should lie in unprotected memory and can fall into any of the three memory partitions depending on the attacker's needs. Once the peripheral is issuing DMA requests, it also records which request is delayed. Based on this information, it can reconstruct whether the target program had accessed the same memory partition in a given cycle, reconstructing the leakage trace for the monitored memory partition.

To reconstruct the full leakage trace of the victim program, it is necessary to execute the attack three times while targeting the three memory partitions with the malicious peripheral. However, we found that monitoring only the program memory is sufficient for our attacks.

5.2. Multiplication routine

Multiplications in MSP430 programs are not natively supported in hardware; the MSP430 ISA [22] does not contain a multiplication instruction. Hence, the compiler is responsible for transparently inserting a call to a software-emulated multiplication routine which computes the result via repeated additions [35]. A simplified version of this code routine, provided by the `Libgcc` compiler runtime library, is shown in Listing 1, where the two multiplication operands are `x` and `y`.

Note that the unmodified multiplication routine already contains a start-to-end timing vulnerability: line 4 is executed for each 1 bit in the operand `y`, thus leaking the Hamming weight of `y` through overall execution time. Furthermore, Nemesis can significantly amplify this leakage by leaking the *exact* bit values in `y` based on individual instruction latencies. After applying the compiler defense for Nemesis, the hardened version was shown to be secure [18].

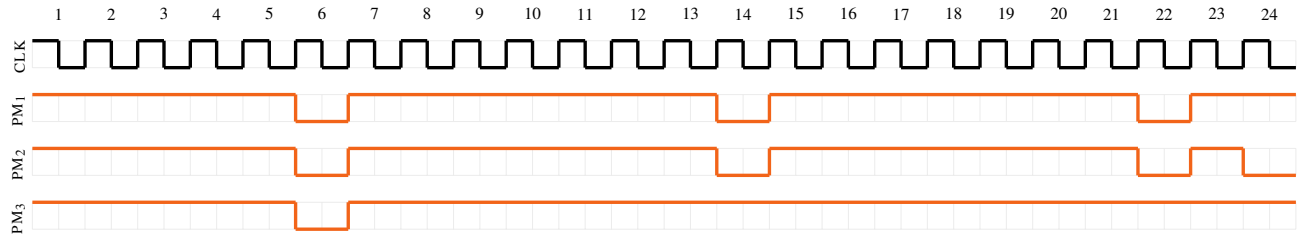


Figure 8: Leakage traces for the multiplication routine.

```
1 for (bit = 0; bit < sizeof(y) * 8; bit++) {
2     if (y) {
3         if (y & 1)
4             result += x;
5         x <<= 1;
6         y >>= 1;
7     }
8 }
```

Listing 1: Multiplication routine by repeated addition.

```
1 for (i=0; i <= IVT_END-IVT_START; i++, ivt++) {
2   if (*ivt != data[i]) {
3     retValue |= 0x40;
4   }
5 }
```

Listing 2: BSL password checking code.

Crucially, while the hardened code is no longer vulnerable to Nemesis, we found that, through the finer-grained DMA side channel, the exact bit values in the `y` operand can be reconstructed once again. Figure 8 shows the program memory access trace of the code after the `if (y)` statement on line 2. The figure shows three different scenarios: the top trace displays the memory accesses when both `if` statements on lines 2 and 3 evaluate to true and `x` is added to the result. This is the case when the current bit of `y` is 1. The middle trace shows the case when only the first `if` statement evaluates to true. This happens when the current bit of `y` is 0, but there are still undiscovered 1 bits left in `y`. Finally, the bottom trace shows the case when both `if` statements evaluate to false, signifying that all remaining bits of `y` are 0.

The `for` loop iterates through all bits of `y`. By capturing the memory trace after the first `if` statement in every iteration, the complete `y` parameter can be reconstructed by a DMA-capable attacker. This has serious ramifications, as this routine is used to transparently replace all multiplications in sensitive code. As a result, a compiler-generated side-channel vulnerability may be introduced even when the conditional control flow is not visible at the application level performing `x*y`. If the parameters of a multiplication are secret, which may be the case, for example, during public-key cryptography operations, leaking one of the operands can break the security objectives of the protected software.

5.3. MSP430 bootstrap loader

The bootstrap loader (BSL) [36] is a protected, privileged component of the TI MSP430 system enabling functionality such as in-field firmware updates. To access the functionality of the bootstrap loader, programs have to supply a secret password. This password-checking routine has been exploited through a timing side channel [10] and was subsequently used as a case study in the original Nemesis [11] attack. A hardened version of this routine is included in the benchmark of the Nemesis-resistant compiler mitigation [18]. The code, shown in Listing 2, iterates over all characters of the provided password and checks whether they are correct. If any character is incorrect, the variable used as the return value is modified.

While the hardened version of this code is not vulnerable to Nemesis, we found that similar to the multiplication routine, it still produces a secret-dependent DMA-based leakage. Figure 9 shows the captured program memory leakage traces after the `if` instruction on line 2. The first trace was captured when the current character of the password was correct, while the second trace shows the leakage with an incorrect password byte.

Although the leakage does not expose the password directly, only the position of correct and incorrect characters, it dramatically decreases the security of the password and the effort required to break it. If the attacker only knows the password length, guessing the full character sequence can take up to 256^{length} tries. With the knowledge of the location of the correct bytes, the attacker can guess the password bytes one by one, reducing the problem to linear complexity, taking at most $256 * length$ guesses.

5.4. DMA contention covert channel

The information leakage through the DMA side channel can also be used as a covert channel to intentionally transfer information. A sandboxed program can communicate with a peripheral by choosing the pattern of its memory accesses for a specific memory partition.

The theoretical upper limit for the bandwidth of this covert channel is one bit of information per clock cycle, where the bit is encoded by whether the monitored partition is accessed. However, according to our experiments, the available leakage classes and the overhead of setting up the channel limits it to approximately one bit per 12 cycles. On a 16 MHz CPU, this leakage rate translates to a bandwidth of approximately 1.3 MBit/sec.

6. Compiler defense

The next step in our methodology is to construct a systematic way of preventing information leakage through both the DMA side channel and the Nemesis attack. While our binary analysis tool can validate whether compiled

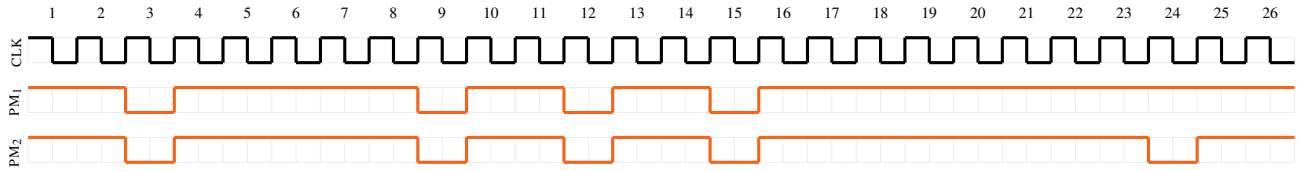


Figure 9: Leakage traces for BSL password check.

```
    cmp @r6, r7      ; if (*r6 != r7) {
    jz 1f            ;
    add #1, 0(r6)    ;   *r6 += 1;
    jmp 2f           ; } else {
1:  add #13, 0(r6)   ;   *r6 += 13;
2:  ...              ; } ...
```

Listing 3: Example vulnerable code.

programs exhibit any secret-dependent leakage (cf. Section 7), constructing a leakage-free program by hand is a tedious and error-prone process. To remove this burden from developers, we implemented an automated approach for hardening sensitive programs, which only requires the developer to add secrecy annotations to the source code.

We extended the LLVM compiler [32] (version 14.0.0) with a pass that hardens secret-dependent branches by inserting *dummy instructions* so that both sides of those branches exhibit the same leakage. We implemented the defense as a `MachineFunctionPass` in the LLVM MSP430 backend, based on the open-source implementation of the Nemesis-hardening compiler pass [18]. We rely on the analysis passes from [18], such as a control-flow analysis (to compute the control-flow graph) and a taint analysis (to identify the secret-dependent branch regions). Consequently, we also inherit the limitations of that implementation. For instance, the secrecy of values loaded from memory is computed conservatively, and therefore the compiler will not accept all valid programs. We maximally reused the implementation from [18] and only replaced the `CompensateInstr` method, which is responsible for generating the actual compensation code. For each instruction in a path that originates from a secret-dependent branch, `CompensateInstr` inserts a suitable *dummy instruction* at the corresponding location in the alternative paths originating from that same branch unless that location already contains an instruction that exhibits the same leakage.

6.1. Balancing with dummy instructions

We demonstrate the balancing approach using the example in Listing 3. This code features a secret-dependent branch and two different additions depending on the outcome of the branch. Figure 10 demonstrates the possible control-flow paths of this code. These branches are clearly not balanced; the left side consists of two instructions, while the right side contains only one. Figure 7 has also shown that the two `add` instructions used in this code have different latencies and memory traces, making this snippet vulnerable to start-to-end timing, Nemesis, and DMA attacks. Figure 11 shows the branches after balancing. The instructions inserted by the compiler, shown in red, prevent the branch outcome from leaking via the control

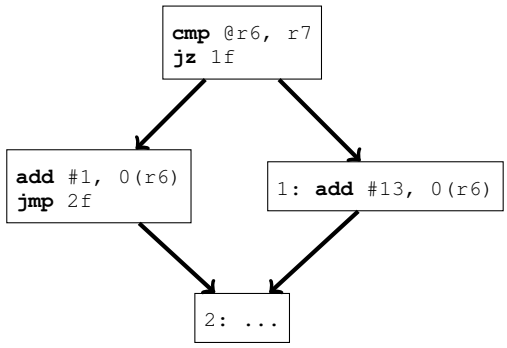


Figure 10: Vulnerable control flow.

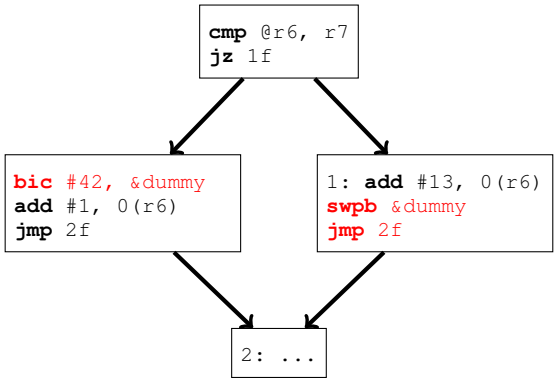


Figure 11: Secure control flow.

flow for the three attacks above. The inserted instructions do not change the behavior of the program: the `bic` and `swpb` instructions modify a specially allocated dummy memory cell which does not contain any useful information, while the inserted `jmp` instruction jumps to label 2, which is the next instruction in the program flow.

6.2. Selecting dummy instructions

The programs that can be hardened by our compiler pass have to satisfy the assumptions outlined in Section 4.3. These restrictions are reasonable for most general-purpose programs, and they help limit the number of possible leakage classes for which dummy instructions have to be selected. Dummy instructions must not affect the live state of executing programs, and they should use minimal resources. To achieve this, we follow the following principles when selecting the dummy instructions:

- When a dummy instruction writes to a register, we select the constant generator register `r3`. Writing to this register has no effect. This way, we do not need to use a general-purpose register for this type of dummy instruction and, importantly, the register pressure does not increase.

- When a dummy reads a memory address from a register (e.g., using the indexed addressing mode), we select the stack pointer register `r1`, as this one always points to a valid data memory address inside the enclave.
- For dummies writing to memory, we reserve a location inside the enclave to serve as the target for these writes.

Dummy memory accesses must always address memory inside the enclave to prevent an attacker from being able to monitor activity at these memory locations. Furthermore, special attention needs to be given to some side effects. For instance, instructions that change the status register (used in branch conditions) might interfere with the intended control flow of the program.

When selecting a dummy counterpart for a given instruction, we need to make sure that we have all the necessary information at the compiler level to be able to associate the instruction with a leakage class. For example, when using indexed or indirect addressing modes, the source register contains an address from where a value is fetched. If the compiler could not statically determine which memory partition the address points to, it would need to instrument the code to determine this at runtime before executing the sensitive instruction. This is necessary since the leakage trace of the instruction depends on the memory partition of the address. However, since we assume well-formed programs that always address data memory, we did not implement this instrumentation.

Using these guidelines, we selected one dummy instruction for almost every considered leakage class. Particularly, we found that the vast majority of sampled MSP430 instructions (527 out of 551) fall in leakage classes for which a dummy instruction could be constructed. For the few remaining leakage classes, it was not possible to select a dummy instruction, as every instruction instance of those classes affects the live state of the program. In general, a compiler has several generic options to work around instruction classes without compensating dummies:

- Move the problematic instruction out of the secret-dependent branch (while preserving the program's functionality).
- Emulate the instruction using others that do have a dummy alternative (e.g., replacing `push r5` with `sub #2, r1; mov r5, 0(r1)`).
- Revert the effects of the instruction in the alternative path (e.g., `push r5; pop r5`), and compensate for the reverting instruction with a dummy in the original path (e.g., `push r5; mov @r1, &dummy`).

Additionally, control-flow instructions, such as `jmp`, `call` or `reti` (return from interrupt), should be balanced with an instance of the same instruction type, although with potentially different operands (e.g., jumping to the next instruction as in the balanced example of Figure 11).

We refer to Appendix A for the complete list of instruction classes with leakage traces and chosen dummy instructions.

Automating the dummy selection. We believe that selecting the dummy instructions for each leakage class could be automated in future work. Based on the ISA specification, a tool could automatically search for a suitable dummy instruction for a given leakage class, optimizing for footprint and side effects. A challenge with this approach is that the functional specification is typically not available in a machine-readable format, making it difficult to automatically determine whether a given instruction will function as a no-op. Tools such as SAIL [37], a language to formally express ISA semantics, may help automate this selection process.

7. Validation and evaluation

After designing and implementing our compiler defense, we evaluated it based on two different aspects. First, as the final step of our methodology, we modified an open-source binary static analysis tool [20] to detect secret-dependent DMA leakage in MSP430 software. This tool, apart from detecting vulnerabilities in third-party code, can also validate the security of the binaries generated by our compiler mitigation. Second, we measured the performance impact of our proposed mitigation in terms of execution time and binary size. We also quantified the scalability of our methodology by measuring the time taken to profile the instructions and to harden applications with our compiler defense.

7.1. Security validation with SCF-MSP

SCF-MSP [20] is a binary analysis tool developed to detect architectural and side-channel (start-to-end and Nemesis timing attacks) leakage on the TI MSP430 platform. We extended this tool to also detect secret leakage via the DMA-based side channel. This implementation also builds on the augmented ISA generated by our instruction profiling step; detecting leakage due to secret-dependent branching is impossible without this information. The tool works on arbitrary MSP430 binaries but requires a description of the function signatures from the developer, including security annotations.

Development process. The systematic development of the validation tool and the compiler mitigation in parallel has enabled us to uncover several interesting findings. First, we discovered a mismatch in the instruction timing specification of openMSP430 [25], which we reported to the maintainers[1]. Specifically, the execution time of the `call` instruction with immediate addressing mode was incorrectly reported to be 5 cycles, while it actually takes 4 cycles. Although this is a minor deviation, it can lead to incorrectly balanced branches in compiler mitigations that rely on the ISA description for instruction timings instead of profiling the instructions.

Furthermore, we discovered several instruction timing mismatches in the static analysis tool. Some of these mismatches result from the original SCF-MSP tool targeting the TI MSP430 ISA, which differs in a small number of documented instances from the openMSP430 timing specification [25]. We also found that the original SCF-MSP tool did not correctly handle the edge case of writing to the program counter register, which always incurs an

1. Available at https://opencores.org/projects/openmsp430/issues/40.

additional penalty cycle according to the TI MSP430 specification (see also Section 4.2.5).

Using the binary analysis tool to validate development versions of the compiler mitigation has also uncovered some issues. Among others, we discovered that the MSP430 assembler [35] optimizes instructions with a zero-indexed register operand `0(rx)` as the source operand into a different instruction that uses the indirect mode addressing `@rx`, changing the leakage trace of the instruction. Since this assembler optimization happens *after* the compilation step, this is not yet visible during our hardening pass. However, the binary analysis tool can show that this optimization makes target programs insecure. This example illustrates the strength of a separate binary validation step in our systematic approach.

Validating the compiler pass. To validate the security of our combined compiler defense, we used the benchmarks from the original Nemesis defense [18]. These benchmarks contain a collection of programs with insecure secret-dependent code. In our experiments, we used the extended binary analysis tool to check for leakage in these programs. First, we compiled them with the original Nemesis defense. This pass left 11 of the 20 programs vulnerable to DMA-based secret leakage. The full results are shown in Table 2. Crucially, these results show that the defense against Nemesis alone is insufficient to mitigate the leakage from DMA. This is unsurprising, as we have already seen that instructions with the same execution time can have different leakage traces. In our second experiment, we compiled the benchmark programs using our improved Nemesis+DMA compiler mitigation. After this pass, we confirmed that none of the programs were reported to have either Nemesis or DMA-based leakage.

During our analysis, we discovered that the over-approximation of SCF-MSP causes a falsely reported Nemesis leakage in the `modexp2` benchmark. We manually checked that this false positive is caused by the conservative taint tracking algorithm, which considers values read from the stack to be always secret [20]. In this benchmark program, a public value is first spilled onto the stack, then read back to be included in a branch condition. This read causes the tool to falsely flag this branch as being secret-dependent. To be able to analyze the rest of the program, we manually patched the tool to ignore this known false positive in this specific benchmark.

Furthermore, due to excessive false positives, we had to disable SCF-MSP's secret-dependent loop condition checks, which is not an issue since none of the benchmark programs contain secret-dependent loop conditions. We also had to leave out one benchmark program from our evaluation, as the original SCF-MSP tool does not support recursion in secret-dependent branches.

7.2. Performance evaluation

To evaluate the performance overhead of our compiler mitigation, we extended the evaluation used for the original Nemesis defense [18]. These benchmarks measure the overhead in terms of compiled binary size and execution time of programs. In our evaluation, we compare the original, non-balanced programs to hardened code by the original Nemesis defense, as well as to code hardened by our mitigation, which defends against both Nemesis and DMA side-channel attacks. We also compare the performance of our compiler against a state-of-the-art linearization [12] technique, which, as opposed to our balancing approach, completely eliminates secret-dependent branches.

TABLE 2: SCF-MSP analysis results on benchmarks.

Benchmark	Nemesis-hardened	Nemesis+DMA-hardened
bsl	✓	✓
keypad	DMA ↯	✓
modexp2 (*)	✓	✓
mulhi3	DMA ↯	✓
mulmod8	DMA ↯	✓
sharevalue	DMA ↯	✓
switch16	DMA ↯	✓
switch8	DMA ↯	✓
call	DMA ↯	✓
diamond	✓	✓
fork	✓	✓
ifcompound	✓	✓
ifthenloop	DMA ↯	✓
ifthenloopif	DMA ↯	✓
ifthenlooploop	DMA ↯	✓
ifthenlooplooptail	DMA ↯	✓
indirect	✓	✓
loop	✓	✓
multifork	✓	✓
triangle	✓	✓

This benchmark set consists of two types of programs: a set of synthetic benchmarks constructed to showcase the original Nemesis defense and a set of third-party programs. The third-party programs also include a variant that has been linearized [12]. This linearization means (manually) rewriting all secret-dependent branches to straight-line code such that there are no more secret-dependent execution paths that could cause any observable leakage.

Changed performance numbers. The results of our evaluation are shown in Tables 3 and 4. Our results for the vulnerable baseline and the Nemesis-hardened code slightly differ from those reported in [18]; the tables contain our results. This difference is most likely a consequence of using different compiler versions. Most notably, the geometric mean of execution times of linearized programs has increased from 76% to 87%, but these differences still allow us to reason about relative overheads.

Synthetic benchmarks. Table 3 shows the performance impact of applying the compiler passes to the vulnerable synthetic benchmark programs. This table contains the program size and execution times for each benchmark program. The different execution times in a given row correspond to the different paths the execution can take depending on the inputs. The overhead is shown in terms of size increase, as well as execution time increase for the different paths. With code balancing, the execution time of the balanced program can never be shorter than the longest possible path in the original program, so this overhead is highlighted in the last column of the table.

Notably, the evaluation resulted in the exact same overhead for our proposed DMA+Nemesis defense as for the Nemesis defense alone without DMA hardening applied. This result shows that our principled microarchitectural profiling approach allowed us to significantly improve the effectiveness of the compiler mitigation *without* incurring additional overhead.

TABLE 3: Performance results for the synthetic benchmark suite [18]. The reported overhead numbers are identical for the original Nemesis defense and our combined mitigation for Nemesis+DMA.

Benchmark	Vulnerable baseline		Overhead of balancing		
	Size (bytes)	*Execution time (cycles)*	*Size*	*Execution time*	*Execution time (longest path)*
call	300	112, 91	1.09x	1.05x, 1.30x	1.05x
diamond	282	102, 101, 103	1.16x	1.13x, 1.14x, 1.12x	1.12x
fork	262	90, 91	1.06x	1.07x, 1.05x	1.05x
ifcompound	382	370, 371, 372	1.06x	1.02x, 1.02x, 1.02x	1.02x
ifthenloop	282	143, 96	1.29x	1.20x, 1.79x	1.20x
ifthenloopif	340	179, 108	1.39x	1.61x, 2.68x	1.61x
ifthenlooploop	306	378, 101	1.57x	1.38x, 5.16x	1.38x
ifthenlooplooptail	348	387, 387, 113	1.66x	1.27x, 1.27x, 4.35x	1.27x
indirect	272	95, 97	1.18x	1.19x, 1.16x	1.16x
loop	398	2841	1.06x	1.02x	1.02x
multifork	288	92, 100, 96, 99	1.19x	1.18x, 1.09x, 1.14x, 1.10x	1.09x
triangle	264	92, 94	1.09x	1.09x, 1.06x	1.06x
Geometric mean			**1.22x**	**1.33x**	**1.16x**

TABLE 4: Performance results for the third-party benchmark suite [18]. The reported balancing overhead numbers are identical for the original Nemesis defense and our combined mitigation.

Benchmark	Vulnerable baseline		Overhead of linearization		Overhead of balancing (this work)	
	Size (bytes)	*Execution time (cycles)*	*Size*	*Execution time*	*Size*	*Execution time*
bsl	392	984	1.28x	1.47x	1.13x	1.20x
keypad	670	1119	1.27x	1.81x	1.31x	1.56x
kruskal	632	2460	1.16x	1.24x	1.13x	1.08x
modexp2	700	23537	1.05x	1.32x	1.05x	1.31x
mulhi3	414	904	1.34x	2.01x	1.36x	1.59x
mulmod8	480	425	1.40x	1.36x	1.49x	1.07x
sharevalue	478	3398	1.05x	1.07x	1.05x	1.04x
switch16	400	115	2.29x	4.65x	1.44x	1.09x
switch8	400	115	2.29x	4.65x	1.44x	1.09x
Geometric mean			1.40x	1.87x	**1.26x**	**1.21x**

Third-party benchmarks. Table 4 shows the benchmark results for the third-party programs. The execution time overhead in this table refers to the overhead when taking the longest possible execution path (arguably, this corresponds to the minimal execution time after hardening). Importantly, the evaluation again showed the same overhead for the two compiler defenses for each benchmark (as with the synthetic suite); thus we conclude that our defense incurs no additional overhead compared to the original Nemesis defense. This table also includes performance numbers for the versions of the third-party programs that were manually linearized. In most cases, our compiler pass outperforms this strategy in terms of code size. For execution times, our compiler defense is more performant in every test case. This result clearly shows the performance benefits of our balancing approach over the more generic yet more costly linearization approach for these side channels. The difference between our approach and the state-of-the-art linearization is especially apparent in benchmark programs with many branches, such as `switch8/16`. In the case of linearization, all branches need to be executed (in a way that only the correct one has an architectural effect), while with our approach, only the correct (balanced) branch executes.

Scalability and overhead. Using our methodology for mitigating programs against the DMA side channel comes with limited overhead. Executing the profiling step for the restricted instruction set of 551 instructions, including collecting the leakage traces from the cycle-accurate simulator, takes less than 5 minutes on a consumer laptop with an Intel Core i7-8665U CPU. Of course, instruction profiling could take longer with significantly more complex ISAs, but this impact is limited by the fact that this step only needs to be performed once to generate the augmented ISA. After creating the compiler pass, the only overhead remaining is applying the mitigation pass to programs, which we also found to be limited: compiling all benchmarks in Tables 3-4 adds less than half a second overhead compared to compiling the vulnerable baseline.

8. Discussion and related work

This paper presented a principled approach to systematically mitigate microarchitectural side-channel leakage on processors with predictable instruction behavior. In this section, we discuss the general applicability of our approach, its connection to related work, and the extent of our case study with the DMA attack.

Deterministic side channels. Our profiling approach associates a leakage trace with every relevant instruction in the ISA. As discussed in Section 3, this implies that the leakage of the analyzed side channel only depends on the instruction and its operands. This restricts the set of applicable side channels considerably. Side channels that depend on the aggregated state in a microarchitectural component, such as the cache or the branch predictor, etc. [9], do not satisfy this requirement. However, the embedded systems we have in scope (such as openMSP430, TI MSP430, Atmel AVR, and ARM Cortex-M23) typically lack stateful microarchitectural components, putting these side channels out of scope. Investigating whether our approach can be extended to consider the side effects of previous instructions in the program is left as future work. Simple examples of such dependencies may include the value of the stack pointer or instruction latencies depending on the previously executed instruction [23]. Future work could also examine whether other known side channels, such as power consumption [38] or electromagnetic radiation [39] satisfy our determinism criteria, and apply our methodology to mitigating them for a given microarchitecture.

Completeness of the approach. Previous work used opaque-box fuzzing approaches [40], [41] to automatically discover microarchitectural vulnerabilities in high-end processors. Our goal is different: we aim to mitigate the leakage of known side channels on small IoT processors where instruction balancing [18]–[20], [42] is feasible. Such balancing mitigations require complete coverage of all relevant instructions, so fuzzing is not a satisfactory option. In our methodology, we automatically enumerate and profile all relevant instructions, which aims to ensure that no leakage can surface as a result of an oversight or undetected instruction or operand combination.

Practical limitations. Our prototype toolchain has practical limitations. Particularly, we build on the open-source Nemesis-hardening compiler pass [18], inheriting its limitations of static analysis techniques (cf. Section 6), which may cause some valid programs to be rejected.

Required expertise. While many steps of our methodology are automated, some domain expertise is required at all steps, and human mistakes can inevitably result in insecurities in our approach. When performing the profiling step in the context of a given side channel, we need to know how exactly the leakage from the side channel manifests and how it can be captured by an attacker, as this will form the leakage trace in the augmented ISA. Moreover, we can optimize the process by anticipating which instruction operands or other variables will affect the leakage traces. For instance, it would be practically infeasible to generate an instruction instance for each possible pair of registers and 16-bit values or to generate an instruction instance for each possible memory address. In our case study, knowing that the leakage only depends on the target memory partition and certain register destinations was crucial expert knowledge that allowed us to optimize our approach. Matching the LLVM TableGen representation of instructions to addressing modes also came with challenges; the symbolic and absolute addressing modes of MSP430 use the same encoding format as the indexed mode (cf. Section 4.2.4) and as a result, have the same TableGen representation. However, the different addressing modes might invoke different side-channel leakage in the hardware, requiring us to generate multiple instruction instances for a TableGen instruction that uses this shared encoding format.

To simplify our approach, some of the steps requiring expert knowledge could be automated. One example is selecting the dummy instructions as outlined above, but partially generating the code of the compiler pass itself and generating parts of the binary validator tool could also be feasible. This automation could further increase confidence in the correctness of the implementation of these tools and reduce the required developer effort.

Hardware-software contracts for security. ISA-level hardware-software security contracts [43]–[45] aim to bridge the security gap between hardware and software by specifying how the hardware leaks information. This security specification can then be used by a secure compiler (e.g., CompCert [46], FaCT [47], and Jasmin [48]) to harden security-critical code, following the recent idea of contract-aware secure compilation (CASCO) [49]. Our work can be seen as a practical way of generating such a security contract for a given microarchitecture. The augmented ISA serves as a description of the hardware's behavior, based on which a compiler can generate *efficient and secure* compensation code for a set of attacks.

To improve source code portability, CASCO encourages decoupling the security policy (e.g., constant-timeness) from the source code. Dinesh et al. [50] explored this idea for writing portable constant-time code. Based on a specification of *safe/unsafe* instructions [51], they developed a tool to automatically create translations for all unsafe instructions in the ISA using only instructions from the safe set. Both the safe/unsafe labeling and the automated translations can be seen as augmentations of the ISA, similar to how our approach automatically augments the ISA with a leakage classification.

9. Conclusion

We presented a principled methodology for profiling and mitigating microarchitectural leakage on embedded processors exhibiting deterministic instruction timing behavior. In a case study of a recently uncovered DMA side channel on openMSP430 platforms, we followed the life cycle of this attack. Following an in-depth understanding of the microarchitectural leakage traces that we used to augment the instruction specifications in the ISA, we presented several practical end-to-end attacks on applications that have been hardened by a state-of-the-art instruction-balancing compiler. Next, building on the augmented ISA, we contributed an improved compiler-based mitigation and a binary validation tool that both take into account the microarchitectural leakage traces of individual instructions in both paths of secret-dependent branches. Our comprehensive experimental evaluation showed that, on top of mitigating the state-of-the-art instruction-granular interrupt-latency leakage, our improved compiler also eradicates the cycle-granular DMA side-channel leakage without incurring any additional overhead.

Data availability

All our materials are available in the repository at https://github.com/martonbognar/microprofiler:

- The profiling scripts and generated leakage traces of openMSP430 instructions.
- Our end-to-end attacks (running in the HDL simulator).
- Our compiler mitigation, implemented as an extension to LLVM.
- Our binary analysis tool, implemented as an extension to SCF-MSP.
- The benchmarks used for our evaluation.

Acknowledgements

We would like to thank Steffie Joosen for exploring the idea of building a compiler pass based on instruction profiling in her master's thesis [52].

This research is partially funded by the Research Fund KU Leuven, the ORSHIN project (Horizon Europe grant agreement No. 101070008), and the Flemish Research Programme Cybersecurity. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

References

[1] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[2] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–33, 2017.

[3] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: a security architecture for tiny embedded devices. In *9th European Conference on Computer Systems (EuroSys)*, pages 10:1–10:14. ACM, 2014.

[4] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *52nd Annual Design Automation Conference (DAC)*, pages 34:1–34:6. ACM, 2015.

[5] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium*, pages 1429–1446, 2019.

[6] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. PURE: using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. ACM, 2019.

[7] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium*, pages 771–788, 2020.

[8] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. Sμv—the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[9] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[10] Travis Goodspeed. Practical attacks against the MSP430 BSL. In *25th Chaos Communications Congress.*, 2008.

[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 178–195, 2018.

[12] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology (ICISC) 2005, 8th International Conference*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.

[13] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy (S&P)*, pages 45–60, 2009.

[14] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26. ACM, 2018.

[15] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 715–733. ACM, 2021.

[16] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806. ACM, 2017.

[17] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *43rd IEEE Symposium on Security and Privacy (S&P)*, pages 632–649. IEEE, 2022.

[18] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 667–682. IEEE, 2021.

[19] Florian Dewald, Heiko Mantel, and Alexandra Weber. Avr processors as a platform for language-based security. In *22nd European Symposium on Research in Computer Security (ESORICS)*, pages 427–445. Springer, 2017.

[20] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. Scf$^{\text{msp}}$: static detection of side channels in MSP430 programs. In *15th International Conference on Availability, Reliability and Security (ARES)*, pages 21:1–21:10. ACM, 2020.

[21] Atmel. *Atmel AVR 8-bit Instruction Set: Instruction Set Manual*, 2016.

[22] Texas Instruments. *MSP430x1xx Family User's Guide*, 2006.

[23] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *43rd IEEE Symposium on Security and Privacy (S&P)*, pages 1638–1655. IEEE, 2022.

[24] LLVM Compiler Infrastructure Contributors. Tablegen overview. https://llvm.org/docs/TableGen/, October 2022. Accessed 2023-04-18.

[25] Olivier Girard. openmsp430 rev 1.17. https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf, November 2017. Accessed 2023-04-18.

[26] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *10th European Workshop on Systems Security (EUROSEC)*, pages 2:1–2:6. ACM, 2017.

[27] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium*, pages 565–581, 2016.

[28] Sancus-core: Minimal openmsp430 hardware extensions for isolation and attestation. https://github.com/sancus-tee/sancus-core. Accessed 2023-04-18.

[29] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the TOCTOU problem in remote attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2921–2936, 2021.

[30] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 641–646. IEEE, 2021.

[31] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *International Workshop on Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

[32] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[33] Stephen Williams. Icarus verilog. https://github.com/steveicarus/iverilog, 2000. Accessed 2023-04-18.

[34] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1741–1758, 2019.

[35] Texas Instruments. Msp430-gcc-opensource. https://www.ti.com/tool/MSP430-GCC-OPENSOURCE, 2021. Accessed 2023-04-18.

[36] Texas Instruments. Msp code protection features. https://www.ti.com/lit/an/slaa685/slaa685.pdf, 2015. Accessed 2023-04-18.

[37] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages (POPL)*, 3:71:1–71:31, 2019.

[38] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[39] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2002.

[40] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium*, pages 1427–1444, 2020.

[41] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: testing black-box cpus against speculation contracts. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 226–239, 2022.

[42] Majid Salehi, Gilles De Borger, Danny Hughes, and Bruno Crispo. Nemesisguard: Mitigating interrupt latency side channel attacks with static binary rewriting. *Computer Networks*, 205:108744, 2022.

[43] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *42nd IEEE Symposium on Security and Privacy, (S&P)*, pages 1868–1883. IEEE, 2021.

[44] Gernot Heiser. For safety's sake: We need a new hardware-software contract! *IEEE Design & Test*, 35(2):27–30, 2018.

[45] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In *49th Annual International Symposium on Computer Architecture (ISCA)*, pages 72–86. ACM, 2022.

[46] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proceedings of the ACM on Programming Languages (POPL)*, 4:7:1–7:30, 2020.

[47] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A Flexible, Constant-Time Programming Language. In *IEEE Cybersecurity Development (SecDev)*, pages 69–76. IEEE Computer Society, 2017.

[48] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1807–1823. ACM, 2017.

[49] Marco Guarnieri and Marco Patrignani. Contract-aware secure compilation. *CoRR*, abs/2012.14205, 2020.

[50] Sushant Dinesh, Grant Garrett-Grossman, and Christopher W. Fletcher. Synthct: Towards portable constant-time code. In *29th Annual Network and Distributed System Security Symposium (NDSS)*, 2022.

[51] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[52] Steffie Joosen. Hardening enclave programs against side-channel vulnerabilities at compile-time. Master's thesis, KU Leuven, 2021.

A. Leakage classes

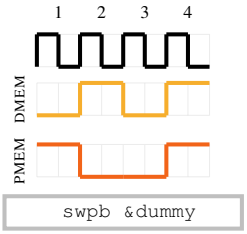
In this appendix, we provide the 23 complete leakage classes obtained via our principled profiling methodology for well-formed programs, as described in Section 4.3. For every class, we first provide the extracted (common) leakage trace as a waveform diagram that includes the number of execution cycles (i.e., the Nemesis [11] side-channel leakage), as well as the exact clock cycles in which program (PMEM) and data (DMEM) memory is accessed (i.e., the DMA [23] side-channel leakage). Note that we do not display accesses to the MMIO partition, as well-formed programs should only address protected data memory. We, furthermore, provide the selected dummy instruction for that class under the waveform. Next, we list all MSP430 instructions that are a member of the respective class, including their opcode, source and destination addressing modes, and operand values (cf. Section 4.2). Instructions listed as members of the same class exhibit identical leakage traces and are, hence, indistinguishable to a Nemesis or DMA side-channel adversary.

Tables 5 to 8 provide the first 12 leakage classes, which include the vast majority of sampled MSP430 instructions (527 out of 551). For these leakage classes, we selected dummy instructions that behave as a no-operation, as explained in Section 6.2.

For the remaining ten leakage classes, listed in Tables 9 to 11, we could not construct a suitable dummy instruction that acts as a no-op. It is, however, possible to get around this problem for all instructions in these classes using the techniques outlined in Section 6. Specifically, the control-flow instructions in Tables 10 and 11 (`br`, `call`, `reti` and `ret`) must be balanced with an instance of the same instruction type. The instructions in the remaining classes in Table 9 are disallowed from secret-dependent regions and have to be emulated by one or more instructions for which a dummy instruction exists.

TABLE 5: OpenMSP430 instruction leakage classes with dummies, resulting from microarchitectural profiling. For every class, all member instructions listed in the rightmost columns exhibit the leakage trace depicted in the left column – both in terms of total execution cycles (i.e., Nemesis side-channel leakage) and in terms of program and data memory access traces (i.e., DMA side-channel leakage). Leakage classes are continued on the next pages.

	Dummy and leakage trace	Members		

Class #1		`add #0x1, 42(r6)`
		`add r10, &dmem`
		`add.b r10, 42(r6)`
		`addc #0x1, &dmem`
		`addc.b #0x1, 42(r6)`
		`addc.b r10, &dmem`
		`and r10, 42(r6)`
		`and.b #0x1, &dmem`
		`bic #0x1, 42(r6)`
		`bic r10, &dmem`
		`bic.b r10, 42(r6)`
		`bis #0x1, &dmem`
		`bis.b #0x1, 42(r6)`
		`bis.b r10, &dmem`
		`bit r10, 42(r6)`
		`bit.b #0x1, &dmem`
		`cmp #0x1, 42(r6)`
		`cmp r10, &dmem`
		`cmp.b r10, 42(r6)`
		`dadd #0x1, &dmem`
		`dadd.b #0x1, 42(r6)`
		`dadd.b r10, &dmem`
		`rra.b 42(r6)`
		`rrc &dmem`
		`sxt 42(r6)`
		`sub #0x1, &dmem`
		`sub.b #0x1, 42(r6)`
		`sub.b r10, &dmem`
		`subc r10, 42(r6)`
		`subc.b #0x1, &dmem`
		`swpb 42(r6)`
		`xor #0x1, &dmem`
		`xor.b #0x1, 42(r6)`
		`xor.b r10, &dmem`
			`add #0x1, &dmem`
			`add.b #0x1, 42(r6)`
			`add.b r10, &dmem`
			`addc r10, 42(r6)`
			`addc.b #0x1, &dmem`
			`and #0x1, 42(r6)`
			`and r10, &dmem`
			`and.b r10, 42(r6)`
			`bic #0x1, &dmem`
			`bic.b #0x1, 42(r6)`
			`bic.b r10, &dmem`
			`bis r10, 42(r6)`
			`bis.b #0x1, &dmem`
			`bit #0x1, 42(r6)`
			`bit r10, &dmem`
			`bit.b r10, 42(r6)`
			`cmp #0x1, &dmem`
			`cmp.b #0x1, 42(r6)`
			`cmp.b r10, &dmem`
			`dadd r10, 42(r6)`
			`dadd.b #0x1, &dmem`
			`rra 42(r6)`
			`rra.b &dmem`
			`rrc.b 42(r6)`
			`sxt &dmem`
			`sub r10, 42(r6)`
			`sub.b #0x1, &dmem`
			`subc #0x1, 42(r6)`
			`subc r10, &dmem`
			`subc.b r10, 42(r6)`
			`swpb &dmem`
			`xor r10, 42(r6)`
			`xor.b #0x1, &dmem`
				`add r10, 42(r6)`
				`add.b #0x1, &dmem`
				`addc #0x1, 42(r6)`
				`addc r10, &dmem`
				`addc.b r10, 42(r6)`
				`and #0x1, &dmem`
				`and.b #0x1, 42(r6)`
				`and.b r10, &dmem`
				`bic r10, 42(r6)`
				`bic.b #0x1, &dmem`
				`bis #0x1, 42(r6)`
				`bis r10, &dmem`
				`bis.b r10, 42(r6)`
				`bit #0x1, &dmem`
				`bit.b #0x1, 42(r6)`
				`bit.b r10, &dmem`
				`cmp r10, 42(r6)`
				`cmp.b #0x1, &dmem`
				`dadd #0x1, 42(r6)`
				`dadd r10, &dmem`
				`dadd.b r10, 42(r6)`
				`rra &dmem`
				`rrc 42(r6)`
				`rrc.b &dmem`
				`sub #0x1, 42(r6)`
				`sub r10, &dmem`
				`sub.b r10, 42(r6)`
				`subc #0x1, &dmem`
				`subc.b #0x1, 42(r6)`
				`subc.b r10, &dmem`
				`xor #0x1, 42(r6)`
				`xor r10, &dmem`
				`xor.b r10, 42(r6)`

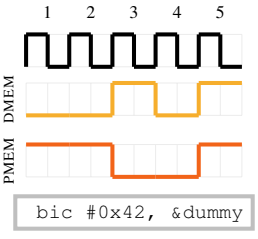
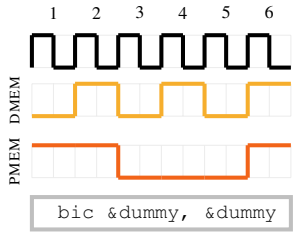
Class #2		`add #const, 42(r6)`
		`add.b #const, &dmem`
		`addc.b #const, 42(r6)`
		`and #const, &dmem`
		`bic #const, 42(r6)`
		`bic.b #const, &dmem`
		`bis.b #const, 42(r6)`
		`bit #const, &dmem`
		`cmp #const, 42(r6)`
		`cmp.b #const, &dmem`
		`dadd.b #const, 42(r6)`
		`sub #const, &dmem`
		`subc #const, 42(r6)`
		`subc.b #const, &dmem`
		`xor.b #const, 42(r6)`
			`add #const, &dmem`
			`addc #const, 42(r6)`
			`addc.b #const, &dmem`
			`and.b #const, 42(r6)`
			`bic #const, &dmem`
			`bis #const, 42(r6)`
			`bis.b #const, &dmem`
			`bit.b #const, 42(r6)`
			`cmp #const, &dmem`
			`dadd #const, 42(r6)`
			`dadd.b #const, &dmem`
			`sub.b #const, 42(r6)`
			`subc #const, &dmem`
			`xor #const, 42(r6)`
			`xor.b #const, &dmem`
				`add.b #const, 42(r6)`
				`addc #const, &dmem`
				`and #const, 42(r6)`
				`and.b #const, &dmem`
				`bic.b #const, 42(r6)`
				`bis #const, &dmem`
				`bit #const, 42(r6)`
				`bit.b #const, &dmem`
				`cmp.b #const, 42(r6)`
				`dadd #const, &dmem`
				`sub #const, 42(r6)`
				`sub.b #const, &dmem`
				`subc.b #const, 42(r6)`
				`xor #const, &dmem`

TABLE 6: OpenMSP430 instruction leakage classes with dummies (continued).

	Dummy and leakage trace	Members		

Class #3		add 42(r6), 42(r6)
		add &dmem, &dmem
		add.b 42(r6), &dmem
		addc &dmem, 42(r6)
		addc.b 42(r6), 42(r6)
		addc.b &dmem, &dmem
		and 42(r6), &dmem
		and.b &dmem, 42(r6)
		bic 42(r6), 42(r6)
		bic &dmem, &dmem
		bic.b 42(r6), &dmem
		bis &dmem, 42(r6)
		bis.b 42(r6), 42(r6)
		bis.b &dmem, &dmem
		bit 42(r6), &dmem
		bit.b &dmem, 42(r6)
		cmp 42(r6), 42(r6)
		cmp &dmem, &dmem
		cmp.b 42(r6), &dmem
		dadd &dmem, 42(r6)
		dadd.b 42(r6), 42(r6)
		dadd.b &dmem, &dmem
		sub 42(r6), &dmem
		sub.b &dmem, 42(r6)
		subc 42(r6), 42(r6)
		subc &dmem, &dmem
		subc.b 42(r6), &dmem
		xor &dmem, 42(r6)
		xor.b 42(r6), 42(r6)
		xor.b &dmem, &dmem
			add &dmem, 42(r6)
			add.b 42(r6), 42(r6)
			add.b &dmem, &dmem
			addc 42(r6), &dmem
			addc.b &dmem, 42(r6)
			and 42(r6), 42(r6)
			and &dmem, &dmem
			and.b 42(r6), &dmem
			bic &dmem, 42(r6)
			bic.b 42(r6), 42(r6)
			bic.b &dmem, &dmem
			bis 42(r6), &dmem
			bis.b &dmem, 42(r6)
			bit 42(r6), 42(r6)
			bit &dmem, &dmem
			bit.b 42(r6), &dmem
			cmp &dmem, 42(r6)
			cmp.b 42(r6), 42(r6)
			cmp.b &dmem, &dmem
			dadd 42(r6), &dmem
			dadd.b &dmem, 42(r6)
			sub 42(r6), 42(r6)
			sub &dmem, &dmem
			sub.b 42(r6), &dmem
			subc &dmem, 42(r6)
			subc.b 42(r6), 42(r6)
			subc.b &dmem, &dmem
			xor 42(r6), &dmem
			xor.b &dmem, 42(r6)
				add &dmem, 42(r6)
				add.b &dmem, 42(r6)
				addc 42(r6), 42(r6)
				addc &dmem, &dmem
				addc.b 42(r6), &dmem
				and &dmem, 42(r6)
				and.b 42(r6), 42(r6)
				and.b &dmem, &dmem
				bic 42(r6), &dmem
				bic.b &dmem, 42(r6)
				bis 42(r6), 42(r6)
				bis &dmem, &dmem
				bis.b 42(r6), &dmem
				bit &dmem, 42(r6)
				bit.b 42(r6), 42(r6)
				bit.b &dmem, &dmem
				cmp 42(r6), &dmem
				cmp.b &dmem, 42(r6)
				dadd 42(r6), 42(r6)
				dadd &dmem, &dmem
				dadd.b 42(r6), &dmem
				sub &dmem, 42(r6)
				sub.b 42(r6), 42(r6)
				sub.b &dmem, &dmem
				subc 42(r6), &dmem
				subc.b &dmem, 42(r6)
				xor 42(r6), 42(r6)
				xor &dmem, &dmem
				xor.b 42(r6), &dmem

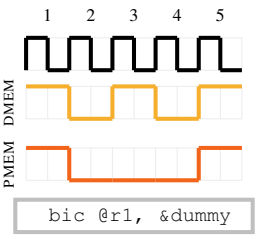
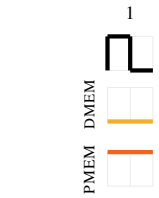
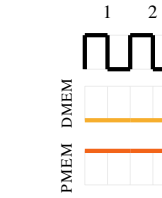
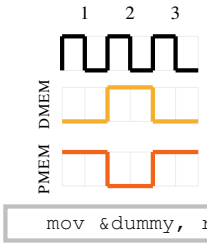
Class #4		add @r6, 42(r6)
		add @r6+, &dmem
		add.b @r6+, 42(r6)
		addc @r6, &dmem
		addc.b @r6, 42(r6)
		addc.b @r6+, &dmem
		and @r6+, 42(r6)
		and.b @r6, &dmem
		bic @r6, 42(r6)
		bic @r6+, &dmem
		bic.b @r6+, 42(r6)
		bis @r6, &dmem
		bis.b @r6, 42(r6)
		bis.b @r6+, &dmem
		bit @r6+, 42(r6)
		bit.b @r6, &dmem
		cmp @r6, 42(r6)
		cmp @r6+, &dmem
		cmp.b @r6+, 42(r6)
		dadd @r6, &dmem
		dadd.b @r6, 42(r6)
		dadd.b @r6+, &dmem
		sub @r6+, 42(r6)
		sub.b @r6, &dmem
		subc @r6, 42(r6)
		subc @r6+, &dmem
		subc.b @r6+, 42(r6)
		xor @r6, &dmem
		xor.b @r6, 42(r6)
		xor.b @r6+, &dmem
			add @r6, &dmem
			add.b @r6, 42(r6)
			add.b @r6+, &dmem
			addc @r6+, 42(r6)
			addc.b @r6, &dmem
			and @r6, 42(r6)
			and @r6+, &dmem
			and.b @r6+, 42(r6)
			bic @r6, &dmem
			bic.b @r6, 42(r6)
			bic.b @r6+, &dmem
			bis @r6+, 42(r6)
			bis.b @r6, &dmem
			bit @r6, 42(r6)
			bit @r6+, &dmem
			bit.b @r6+, 42(r6)
			cmp @r6, &dmem
			cmp.b @r6, 42(r6)
			cmp.b @r6+, &dmem
			dadd @r6+, 42(r6)
			dadd.b @r6, &dmem
			sub @r6, 42(r6)
			sub @r6+, &dmem
			sub.b @r6+, 42(r6)
			subc @r6, &dmem
			subc.b @r6, 42(r6)
			subc.b @r6+, &dmem
			xor @r6+, 42(r6)
			xor.b @r6, &dmem
				add @r6+, 42(r6)
				add.b @r6, &dmem
				addc @r6, 42(r6)
				addc @r6+, &dmem
				addc.b @r6+, 42(r6)
				and @r6, &dmem
				and.b @r6, 42(r6)
				and.b @r6+, &dmem
				bic @r6+, 42(r6)
				bic.b @r6, &dmem
				bis @r6, 42(r6)
				bis @r6+, &dmem
				bis.b @r6+, 42(r6)
				bit @r6, &dmem
				bit.b @r6, 42(r6)
				bit.b @r6+, &dmem
				cmp @r6+, 42(r6)
				cmp.b @r6, &dmem
				dadd @r6, 42(r6)
				dadd @r6+, &dmem
				dadd.b @r6+, 42(r6)
				sub @r6, &dmem
				sub.b @r6, 42(r6)
				sub.b @r6+, &dmem
				subc @r6+, 42(r6)
				subc.b @r6, &dmem
				xor @r6, 42(r6)
				xor @r6+, &dmem
				xor.b @r6+, 42(r6)

TABLE 7: OpenMSP430 instruction leakage classes with dummies (continued).

	Dummy and leakage trace	Members		

Class #5		`add #0x1, r10`
		`add.b r10, r10`
		`addc.b #0x1, r10`
		`and r10, r10`
		`bic #0x1, r10`
		`bic.b r10, r10`
		`bis.b #0x1, r10`
		`bit r10, r10`
		`cmp #0x1, r10`
		`cmp.b r10, r10`
		`dadd.b #0x1, r10`
		`mov r10, r10`
		`mov.b r10, r10`
		`rrc r10`
		`sub #0x1, r10`
		`sub.b r10, r10`
		`subc.b #0x1, r10`
		`xor #0x1, r10`
		`xor.b r10, r10`
			`add r10, r10`
			`addc #0x1, r10`
			`addc.b r10, r10`
			`and.b #0x1, r10`
			`bic r10, r10`
			`bis #0x1, r10`
			`bis.b r10, r10`
			`bit.b #0x1, r10`
			`cmp r10, r10`
			`dadd #0x1, r10`
			`dadd.b r10, r10`
			`mov.b #0x1, r10`
			`rra r10`
			`rrc.b r10`
			`sub r10, r10`
			`subc #0x1, r10`
			`subc.b r10, r10`
			`xor r10, r10`
			`mov.b r10, r10`
				`add.b #0x1, r10`
				`addc r10, r10`
				`and #0x1, r10`
				`and.b r10, r10`
				`bic.b #0x1, r10`
				`bis r10, r10`
				`bit #0x1, r10`
				`bit.b r10, r10`
				`cmp.b #0x1, r10`
				`dadd r10, r10`
				`mov #0x1, r10`
				`mov.b r10, r10`
				`rra.b r10`
				`sxt r10`
				`sub.b #0x1, r10`
				`subc r10, r10`
				`swpb r10`
				`xor.b #0x1, r10`

Class #6		`add #const, r10`
		`addc.b #const, r10`
		`bic #const, r10`
		`bis.b #const, r10`
		`br r10`
		`dadd #const, r10`
		`jmp const`
		`sub #const, r10`
		`subc.b #const, r10`
			`add.b #const, r10`
			`and #const, r10`
			`bic.b #const, r10`
			`bit #const, r10`
			`cmp #const, r10`
			`dadd.b #const, r10`
			`mov #const, r10`
			`sub.b #const, r10`
			`xor #const, r10`
				`addc #const, r10`
				`and.b #const, r10`
				`bis #const, r10`
				`bit.b #const, r10`
				`cmp.b #const, r10`
				`jn const`
				`mov.b #const, r10`
				`subc #const, r10`
				`xor.b #const, r10`

Class #7		`add 42(r6), r10`
		`add.b &dmem, r10`
		`addc.b 42(r6), r10`
		`and &dmem, r10`
		`bic 42(r6), r10`
		`bic.b &dmem, r10`
		`bis.b 42(r6), r10`
		`bit &dmem, r10`
		`cmp 42(r6), r10`
		`cmp.b &dmem, r10`
		`dadd.b 42(r6), r10`
		`mov &dmem, r10`
		`mov.b 42(r6), r10`
		`sub &dmem, r10`
		`subc 42(r6), r10`
		`subc.b &dmem, r10`
		`xor.b 42(r6), r10`
			`add &dmem, r10`
			`addc 42(r6), r10`
			`addc.b &dmem, r10`
			`and.b 42(r6), r10`
			`bic &dmem, r10`
			`bis 42(r6), r10`
			`bis.b &dmem, r10`
			`bit.b 42(r6), r10`
			`cmp &dmem, r10`
			`dadd 42(r6), r10`
			`dadd.b &dmem, r10`
			`mov.b 42(r6), r10`
			`mov.b &dmem, r10`
			`sub.b 42(r6), r10`
			`subc &dmem, r10`
			`xor 42(r6), r10`
			`xor.b &dmem, r10`
				`add.b 42(r6), r10`
				`addc &dmem, r10`
				`and 42(r6), r10`
				`and.b &dmem, r10`
				`bic.b 42(r6), r10`
				`bis &dmem, r10`
				`bit 42(r6), r10`
				`bit.b &dmem, r10`
				`cmp.b 42(r6), r10`
				`dadd &dmem, r10`
				`mov 42(r6), r10`
				`mov.b &dmem, r10`
				`sub 42(r6), r10`
				`sub.b &dmem, r10`
				`subc.b 42(r6), r10`
				`xor &dmem, r10`

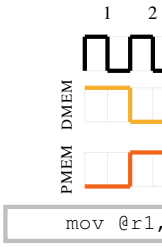
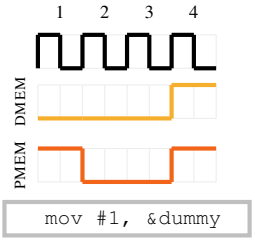
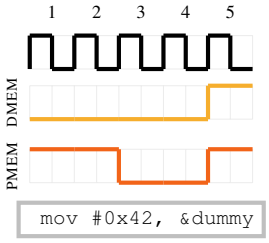
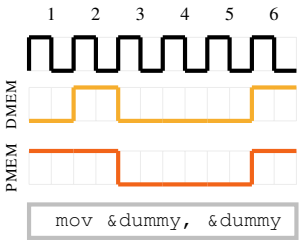
Class #8		`add @r6, r10`
		`add.b @r6+, r10`
		`addc.b @r6, r10`
		`and @r6+, r10`
		`bic @r6, r10`
		`bic.b @r6+, r10`
		`bis.b @r6, r10`
		`bit @r6+, r10`
		`cmp @r6, r10`
		`cmp.b @r6+, r10`
		`dadd.b @r6, r10`
		`mov @r6+, r10`
		`pop r10`
		`sub.b @r6, r10`
		`subc @r6+, r10`
		`xor @r6, r10`
		`xor.b @r6+, r10`
			`add @r6+, r10`
			`addc @r6, r10`
			`addc.b @r6+, r10`
			`and.b @r6, r10`
			`bic @r6+, r10`
			`bis @r6, r10`
			`bis.b @r6+, r10`
			`bit.b @r6, r10`
			`cmp @r6+, r10`
			`dadd @r6, r10`
			`dadd.b @r6+, r10`
			`mov.b @r6, r10`
			`sub @r6, r10`
			`sub.b @r6+, r10`
			`subc.b @r6, r10`
			`xor @r6+, r10`
				`add.b @r6, r10`
				`addc @r6+, r10`
				`and @r6, r10`
				`and.b @r6+, r10`
				`bic.b @r6, r10`
				`bis @r6+, r10`
				`bit @r6, r10`
				`bit.b @r6+, r10`
				`cmp.b @r6, r10`
				`dadd @r6+, r10`
				`mov @r6, r10`
				`mov.b @r6+, r10`
				`sub @r6+, r10`
				`subc @r6, r10`
				`subc.b @r6+, r10`
				`xor.b @r6, r10`

TABLE 8: OpenMSP430 instruction leakage classes with dummies (continued).

	Dummy and leakage trace	Members		

Class #9		`mov #0x1, 42(r6)`
		`mov r10, &dmem`
		`mov.b r10, 42(r6)`
			`mov #0x1, &dmem`
			`mov.b #0x1, 42(r6)`
			`mov.b r10, &dmem`
				`mov r10, 42(r6)`
				`mov.b #0x1, &dmem`
				`push #const`

Class #10		`mov #const, 42(r6)` `mov.b #const, &dmem`	`mov #const, &dmem`	`mov.b #const, 42(r6)`

Class #11		`mov 42(r6), 42(r6)`
		`mov &dmem, &dmem`
		`mov.b 42(r6), &dmem`
			`mov &dmem, 42(r6)`
			`mov.b 42(r6), 42(r6)`
			`mov.b &dmem, &dmem`
				`mov 42(r6), &dmem`
				`mov.b &dmem, 42(r6)`

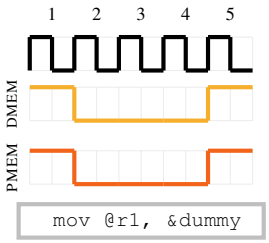
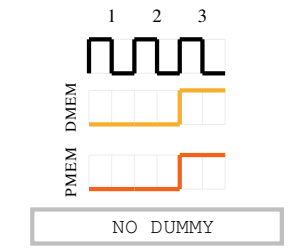
Class #12		`mov @r6, 42(r6)` `mov.b @r6, &dmem`	`mov @r6, &dmem`	`mov.b @r6, 42(r6)`

TABLE 9: OpenMSP430 instruction leakage classes without dummies (disallowed; need to be emulated).

	Dummy and leakage trace	Members		
Class #20		`push #0x1`	`push r10`	`push.b r10`

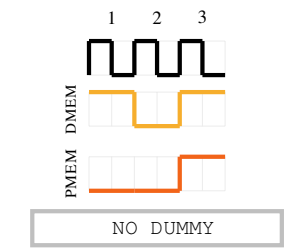
Class #23		`rra @r6`
		`rra.b @r6+`
		`rrc.b @r6`
		`sxt @r6+`
			`rra @r6+`
			`rrc @r6`
			`rrc.b @r6+`
			`swpb @r6`
				`rra.b @r6`
				`rrc @r6+`
				`sxt @r6`
				`swpb @r6+`

TABLE 10: OpenMSP430 instruction leakage classes without dummies. The listed `call` instructions need to be balanced with an identical `call` instruction in the compensation path.

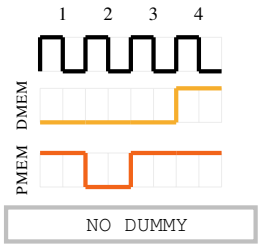
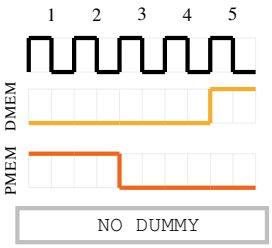
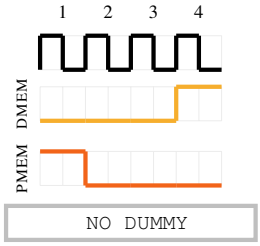
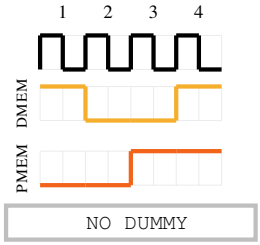
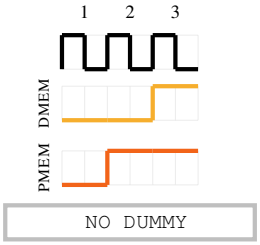
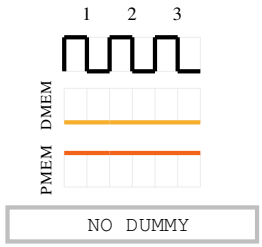
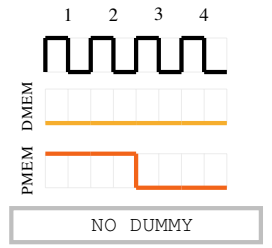
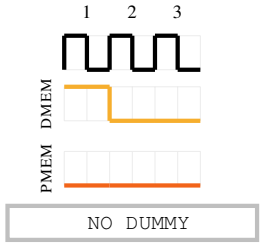
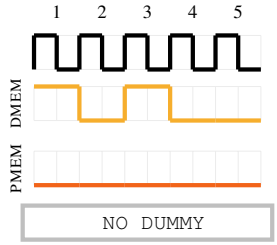
	Dummy and leakage trace	Members
Class #15		`call #const`
Class #16		`call 42(r7)`
Class #17		`call @r7`
Class #18		`call @r6+`
Class #19		`call r10`

TABLE 11: OpenMSP430 instruction leakage classes without dummies. The respective control-flow-transfer instructions need to be balanced with an identical instruction in the compensation path.

	Dummy and leakage trace	Members
Class #13		`br #const`
Class #14		`br 42(r7)`
Class #21		`ret`
Class #22		`reti`