BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect

Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto
Centro ALGORITMI / LASI, Universidade do Minho
id9492@uminho.pt, {daniel.oliveira, sandro.pinto}@dei.uminho.pt

***Abstract*—Spectre and Meltdown have pushed the research community toward an otherwise-unavailable understanding of the security implications of processors' microarchitecture. Notwithstanding, research efforts have concentrated on high-end processors (e.g., Intel, AMD, Arm Cortex-A), and very little has been done for microcontrollers (MCU) that power billions of small embedded and IoT devices. In this paper, we present BUSted. BUSted is a novel side-channel attack that explores the side effects of the MCU bus interconnect arbitration logic to bypass security guarantees enforced by memory protection primitives. Side-channel attacks on MCUs pose incremental and unforeseen challenges, which are strictly tied to the resource-constrained nature of these systems (e.g., single-core CPU, stateless bus). We devise a unique approach that relies on the concept of hardware gadgets. We present practical attacks on state-of-the-art Armv8-M MCUs with TrustZone-M, running the Trusted Firmware-M (TF-M). In contrast to the Nemesis attack, our attack is practical on Arm Cortex-M MCUs, and our findings suggest that it can scale across the full MCU spectrum.**

***Index Terms*—Side-Channels, Microarchitecture, Bus, Microcontrollers, TEE, TrustZone-M.**

1. Introduction

Microarchitectural side-channel attacks rely on secret-dependent traces (e.g., timing differences) left in the microarchitecture of a computer to extract otherwise-unavailable secret information [1, 2]. Spectre [3] and Meltdown [4] have shed light on an untapped potential and practicality of leveraging microarchitectural elements (e.g., caches, branch target buffers) and design techniques (e.g., speculative execution) to leak sensitive information and secrets. Since then, we have witnessed the rise of a plethora of powerful software-based microarchitectural timing side-channel attacks capable of breaking and bypassing the security (isolation) boundaries of numberless processors and technologies from mainstream central processing unit (CPU) vendors (e.g., Intel [5–10], AMD [2, 11, 12], Arm [13–16]).

Despite the abundance of research and literature on microarchitectural side channels affecting high-end application processors (APU) used in servers, desktops, and mobiles, very little has been done and is known regarding the existence and practicality of side-channel attacks in the opposite side of the computing spectrum, i.e., small embedded and IoT systems powered by microcontrollers (MCUs). MCUs (e.g., Arm Cortex-M family) are resource-constrained in computing power and memory, have highly simplistic microarchitectures (e.g., no caches, 2-3 pipeline stages), and lack support for virtual memory; notwithstanding, MCUs are shipped in billions annually and are at the heart of our infrastructures, transportation systems, and consumer devices [17].

Despite the pervasive use of MCUs at scale, the research community is reluctant to understand the potential security implications of MCUs' microarchitecture. To the best of our knowledge, the Nemesis attack [18] is the single existing work reporting on how to explore the microarchitecture of an MCU (MSP430) to leak sensitive information; however, the attack is tied to the CPU interrupt logic of the MSP430 MCU and not proven replicable on Cortex-M MCUs (e.g., Cortex-M0+), which dominates the market. We hypothesize that this lack of motivation may be justified by a general assumption that many microarchitectural elements and design techniques available in high-end APUs (e.g., highly parallelized pipelines, virtual memory) are absent on MCUs, precluding the existence of microarchitectural side-channel attacks and justifying a bias towards this assumption. Notwithstanding, we decided to challenge the status quo and answer the still open question: *Which unique microarchitectural elements on MCUs may create new channels, and how can they be used to mount effective attacks?*

In this paper, we contribute to answering this question by unveiling BUSted. BUSted is a novel class of microarchitectural side-channel attacks that leverage the timing differences exposed in the arbitration logic of the MCUs bus interconnect. Our key observation states that when multiple bus *mains*[1] (e.g., CPU, DMA) perform simultaneous accesses to the same *secondary* port (e.g., memory controller), one will be delayed in time, resulting in subtle timing differences. To corroborate our observation, we validate the existence of this channel by mounting a simple covert channel in 11 MCUs, encompassing CPUs from five different ISAs and five different silicon manufacturers. Evolving from a simple channel observation to a successful attack has proven challenging, mainly due to the foreseen MCU limitations. Firstly, MCUs are mostly powered by a single CPU, hindering the concurrent execution of the traditional spy and victim logic. And lastly, the bus interconnect is stateless,

1. Authors replaced *master* and *slave* terminology used in technical documentation and manuals with the keywords *main* and *secondary*.

thus requiring timing differences to be assessed on-the-fly (i.e., in real time).

To address these challenges, we introduce the concept of *hardware gadgets*. Hardware gadgets are widely available MCU memory-mapped peripherals (e.g., timers, DMAs) that can be programmed to perform a specific logic without CPU intervention. These gadgets can be leveraged to automate concurrent spy logic (e.g., periodic memory accesses) while the victim code executes in the CPU. One interesting observation is that the simple and highly deterministic microarchitecture of MCUs, which intuitively precludes the existence of side channels, facilitates the synchronization of the spy and victim at the clock cycle level. This property enables the spy to use these gadgets that precisely monitor the victim's activity and perform conditional actions based on the victim's control flow.

To mount the BUSted attack, we resort to the so-called *smart gadget network*, i.e., a combination of hardware gadgets interconnected to perform a specific spy logic. Similarly to Nemesis, we emulate a smart lock IoT use case, embedding a reference security-critical application that interfaces a trusted keypad [19]. We implement the attack on two state-of-the-art Armv8-M MCU platforms with TrustZone-M, i.e., ST NUCLEO-L552ZE-Q and Microchip ATSAML11E16A. The trusted execution environment (TEE) kernel encompasses the open-source Trusted-Firmware-M (TF-M) [20], configured for Platform Security Architecture (PSA) isolation levels 2 and 3. We completely bypassed the TrustZone-M isolation guarantees and successfully stole the 4-digit PIN.

We focus on Armv8-M MCUs with TrustZone-M. However, our findings suggest that the attack can be used to bypass isolation enforced via memory protection units (MPUs) on legacy Armv7-M MCUs (and RISC-V), thus breaking the protections enforced among applications in RTOS (e.g., Zephyr [21]) or enclaves in TEEs (e.g., MultiZone [22]). Compared to Nemesis, our attack has proven effective in bypassing security-oriented technologies on state-of-art Arm MCUs and suggests being scalable across the full MCU spectrum.

Contributions. In summary, the contributions of this work are:

- We disclose the MCU bus interconnect arbitration logic as a novel, non-conventional side-channel to bypass hardware memory isolation primitives and leak information from protected / trusted applications;
- We show the prevalence of the side-channel across the full MCU spectrum, providing evidence about the existence of the side-channel in 11 MCUs from five different ISAs (including the complete Arm Cortex-M family) and five different silicon manufacturers;
- We introduce a unique approach based on the concept of hardware gadgets (widely available on MCUs) and leverage them to mount the most powerful and scalable microarchitectural side-channel attack for MCUs to date;
- We demonstrate and evaluate the effectiveness of the attack to bypass state-of-the-art commercial-graded MCU hardware TEE technologies (TrustZone-M).

Responsible disclosure. We first shared our findings with the Trusted Firmware (TF) security team and the STMicroelectronics Product Security Incident Response Team (PSIRT). Both teams acknowledged the successful exploit. The TF security team argues no vulnerability exists in the TF-M code (TF-M is simply a convenient framework for demonstrating the exploit), and the vulnerability is in the microarchitecture of the platform and the custom victim secure partition – we agree with them. Surprisingly, the ST PSIRT argues that the vulnerability is not in the ST's product but rather related to Arm's TrustZone-M (Armv8-M) – we do not necessarily agree with their view. We also reported to Arm which acknowledge the attack and released a public statement on their website [23]. We have reported our findings to other silicon manufacturers listed in Table 1. There are ongoing discussions with Microchip, NXP, and Silicon Labs. As for GigaDevice, we have not received any response.

2. Background and Attack Overview

In this section, we identify the source of the channel, provide an overview of the attack, and assess the pervasiveness of the channel in a large spectrum of MCU.

2.1. Scope and Adversary Model

Scope. We target tiny embedded and IoT devices powered by MCUs. MCUs typically have a single CPU[2], feature a wide range of peripherals (e.g., UART, SPI, timers, DMAs, I2C, etc.), and do not support virtual memory (thus are not capable of running Linux). Some devices have MPUs, and the most recent Armv8-M MCUs have support for two security states, i.e., secure and non-secure world (a.k.a. TrustZone-M). MCUs are highly deterministic computing units designed to consistently produce the same outputs for the same inputs under well-defined timing constraints.

Adversary model. We consider the attacker's[3] main goal is – from an isolated environment (e.g., TEE enclave, RTOS task) – to bypass the memory isolation security mechanisms enforced by privileged software (e.g., trusted kernel, RTOS) and steal sensitive information from another isolated environment. We assume the attacker has full access and control over one isolated domain and its resources, i.e., peripherals (e.g., timer) and bus mains (e.g., DMA). For example, in Armv8-M MCUs with TrustZone, we assume the attacker has control over the (unprivileged) normal world and its resources. We also assume that the attacker knows the victim's code, and this code contains secret-dependent control flows. The spy and the victim run from the same

2. A few modern MCUs already embedded dual-core configurations – e.g., STM32H7, i.MX RT1170, and PSoC 64 series

3. Throughout the paper, we will refer to the person who mounts the attack as the attacker, and the code that the attacker develops as the spy.

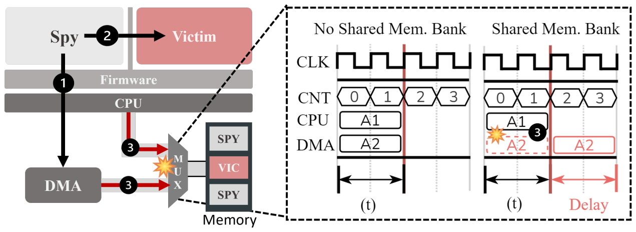
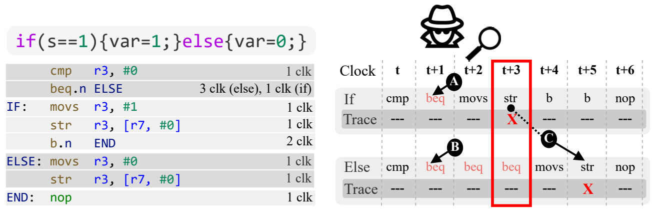


Figure 1: Two bus mains, i.e., CPU (BM1) and DMA (BM2), simultaneously accessing a shared memory bank.



```
if(s==1){var=1;}else{var=0;}
```

```
       cmp    r3, #0                       1 clk
       beq.n  ELSE     3 clk (else), 1 clk (if)
IF:    movs   r3, #1                       1 clk
       str    r3, [r7, #0]                 1 clk
       b.n    END                          2 clk
ELSE:  movs   r3, #0                       1 clk
       str    r3, [r7, #0]                 1 clk
END:   nop                                 1 clk
```

Clock	t	t+1	t+2	t+3	t+4	t+5	t+6
If	cmp	beq	movs	str	b	b	nop
Trace	---	---	---	X	---	---	---
Else	cmp	beq	beq	beq	movs	str	nop
Trace	---	---	---	---	---	X	---

Figure 2: Left, balanced If-else statement compiled for Arm Cortex-M33 (-O0). Right, memory access pattern and monitoring of clock $t + 3$.

physical memory bank, and the victim executes upon requests (remote procedure calls) from the attacker. We do not consider software attacks exploiting vulnerabilities in the most privileged software (e.g., trusted kernel) [24, 25]. We assume those are correct and belong to the system's TCB. The platform's secure boot initializes the system to a known state. We do not consider physical attacks that tamper with hardware, e.g., fault injection [26–29].

2.2. Root Cause: Bus Interconnect

In an MCU, architectural elements have to be interconnected, e.g., CPU, memory, and peripherals. This is typically done through a central interconnect, i.e., bus matrix, that, accordingly to the address broadcasted by a bus main (BM), creates a communication channel between the main and secondary. Assuming that the target bus secondary is different, the bus can perform several concurrent non-blocking full-bandwidth transfers between multiple BMs and secondary ports; however, when two data transfers target the same bus secondary, the bus resorts to an arbitration policy to issue the transfers in a specific order.

Arbitration policies. We survey 11 MCUs from different ISAs and vendors and found that there are two main arbitration policies: *priority-based* (5) or *round-robin* (7). For the priority-based policy, access to the bus secondary is granted to the main with the highest priority. For the round-robin policy, secondary port access is fairly multiplexed in time between competing bus mains. The arbitration policy does not hinder our ability to establish a channel, as shown in §2.5.

2.3. Key Observation

Our key observation states that when multiple bus mains (e.g., CPU, DMA) issue simultaneous accesses to the same secondary (either for priority-based or round-robin policies), the access of one BM will be delayed in time, resulting in subtle timing differences. By carefully analyzing these timing differences, a BM can determine if another BM accessed the same secondary during a specific time frame.

Figure 1 illustrates a scenario with two BMs: a CPU and a DMA peripheral. The CPU is multiplexed in time between the spy and victim, while the DMA is under the exclusive control of the spy. Both the CPU (BM1) and DMA (BM2) operate (i.e., read, write) over the same memory bank but in isolated physical memory spaces (assuming memory protection mechanisms are correctly set). Despite enforcing access control for memory regions, the memory bank (and the memory slave port) is still shared between BM1 and BM2. By observing the timing drifts caused by the matrix arbitration on memory transactions, BMs can extract information about access patterns. Thus, without breaking security isolation boundaries, a malicious BM can spy on bus activity and determine when another BM accessed a specific slave.

2.4. Attack Overview

Our attack leverages the timing differences exposed in the arbitration logic of the MCUs bus interconnect, i.e., an attacker resorts to the bus arbitration side-effects to obtain the victim's memory access pattern. First, the spy triggers the DMA to perform memory accesses (1) and then invokes the victim (2). Using an external wall clock, the spy can measure the BM transfer latency to assess the time of the transaction, thus concluding if there was contention. As depicted in Figure 1, when there is contention, DMA access (A2) takes $t + Delay$ instead of t (3). A spy can observe those differences by leveraging the MCUs load-store architecture, which has two types of instructions: (i) instructions that interface with memory, i.e., loads and stores; and (ii) arithmetic-logic unit (ALU) operations. If the victim code has a secret-dependent control flow, i.e., loads/stores are executed at different clock offsets on conditional paths, producing different memory access patterns. The spy may leverage it to infer and steal a particular secret.

"Toy" attack. Figure 2 presents a toy example of the overall attack. We assume the victim code has a balanced conditional statement that branches on a secret. A careful analysis of assembly code (compiled with -O0 for the Arm Cortex-M33) shows that an external observer would not see any difference in execution time since both execution paths have the same execution time (6 clock cycles, starting at $t + 1$, i.e., after the `cmp` instruction). Additionally, this if-else statement fits within a single instruction cache line (16 bytes), so there is no difference in the instruction cache access pattern between paths. While the example code snippet may be hardened against timing and cache attacks, a spy can still observe a different access pattern to the memory between execution paths. In this case, the notable difference between the two execution paths is due to the

Platforms	PIC18F16Q41	GD32VF103	STM32L0	SAMD21	EFM32GG	SAML11	STM32L5	LPC5500	STM32f4	SAME54	STM32F7/H7
Vendor	MC	GD	ST	MC	SL	MC	ST	NXP	ST	MC	ST
MCU	P18	RVM	M0+	M0+	M3	M23	M33	M33	M4	M4	M7
MHz	64	108	32	48	48	32	110	150	180	120	216
ISA	PIC	RV32	v6M	v6M	v7M	v8M	v8M	v8M	v8M	v7M	v7M
Arch.	H	H	VM	VM	H	VM	H	H	H	H	H
Isolation	—	PMP	MPU	—	MPU	TZ	TZ	TZ	TZ	MPU	MPU
Policy	P	R	R	P	R	PR	R	P	PR	P	R
Ch. (bits)	6.16/8	6.16/8	6.16/8	6.16/8	6.19/8	6.16/8	6.16/8	6.16/8	6.16/8	6.16/8	6.16/8

TABLE 1: Evaluation of the extensibility of the vulnerability. Architectures: Von Neumann (VM) or Harvard (H). Vendors: ST, NXP, Silicon Labs (SL), Microchip (MC) and Gigadevice (GD). MCUs: Cortex-Mx (Mx), PIC18 (P18) and RISC-V Microcontroller Profile (RVM). ISAs: Armv6-M (v6M), Armv7-M & Armv7E-M (v7M), Armv8-M (v8M), PIC, and RISC-V RV32IMAC (RV32). Interconnect arbitration policy: Priority-based (P), Round-robin (R) and Priority-based or Round-robin (PR). MCUs with any hardware isolation primitive (–).

branch instruction. If the branch (`beq`) is not executed, it takes only one clock cycle Ⓐ; but, if executed, it takes three clock cycles Ⓑ. Overall, this changes the relative position of the `str` instruction Ⓒ to the `beq` instruction and unveils the secret. When the secret is 1, the `str` occurs in clock cycle $t + 3$; otherwise, it happens in clock cycle $t + 5$. A spy monitoring one of these two clock cycles can derive the secret by observing whether or not there is contention on the data memory bus.

2.5. Pervasiveness of the Channel

The bus interconnect is a microarchitectural element pervasive in MCUs. To assess the extensibility of this microarchitectural channel, we mounted a simple covert channel in 11 low-end embedded platforms from 5 vendors, and evaluated the existence of the channel. Covert channels leverage microarchitectural elements to create unauthorized communication channels between isolated domains, i.e., encoding and transmitting information from an evil actor to a spy; this also holds for benign actors (side-channel victims). There is a common assumption that when there is a covert channel, the system is vulnerable to side-channel attacks [30].

The covert channel aims to transmit 8 bits of information. It encompasses three steps: (i) the spy triggers a DMA to initiate the transfer of N bytes of data from a shared memory bank and starts a timer; (ii) it then invokes a malicious code (a trojan) injected into the victim domain, which generates a specific number of memory accesses, depending on the secret it wants to transmit; lastly, (iii) the spy reads the timer, measuring the time it took for the DMA to transfer all the N bytes of data. The total measured time varies depending on the number of observed contentions at the memory secondary port (each contention results in an additional delay), which is closely related to the value sent by the trojan.

Results. From the 11 platforms assessed, we were able to establish a channel in all of them. We found no additional challenges related to differences in the arbitration policies. The results are summarized in Table 1, where we present the channel capacity[4] measured for each platform/MCU.

4. The channel capacity was measured with leakiest tool [31], which is based on the concepts of information theory presented in [32, 33].

3. Attack Methodology

Template attacks are a powerful subset of side-channel attacks, where the spy creates a victim execution-related profile used to correlate with the side-channel data and steal a secret [34]. These attacks are composed of two phases: (i) a *profiling phase*; and (ii) an *exploitation phase*. In the profiling phase, the victim runs in a spy-controlled environment to record several side-channel traces. After collecting the traces, the template is generated, i.e., side-channel patterns that unequivocally identify secret-dependent control flow. In the exploitation phase, the spy correlates the incoming side-channel data with the patterns on the template to identify a victim execution path associated with a secret. Inspired by former template attacks [35–37], we established a very similar attack methodology.

In the (i) *profiling phase*, the spy constructs a template matrix by tracing each victim's vulnerable execution path (see Section §4.2). We consider a vulnerable code (vC) any secret-dependent control flow presented in the victim binary (vB). The attacker must manually analyze the vB to identify a vC. The process, conducted offline, involves correlating, at the instruction level, the vB with the recorded trace (a process similar to the one depicted in Figure 2). In the (ii) *exploitation phase*, the spy matches the trace of the victim's execution with the template, aiming to derive the secret. There are two variants, depending on the target victim application. Firstly, for applications that can operate multiple times over the same secret, the attacker can leverage traditional template attacks, i.e., get a victim trace through multiple runs and correlate it with the template matrix to identify the secret. A second (and more sophisticated) variant is related to applications where the secret is ephemeral, i.e., it changes per victim run. To retrieve the secret, the attacker has to collect all side-channel data in a single victim run, which requires a more complex attack infrastructure and levels of expertise.

3.1. Profiling Phase

The profiling aims to obtain the memory access template matrix of a specific vC presented in the vB. This template matrix can take two forms: (i) raw template matrix ($rT \in \mathbb{N}^{N\times M}$), used in the exploitation phase to find the first instruction of the vC (see Section §3.2); or (ii) pruned

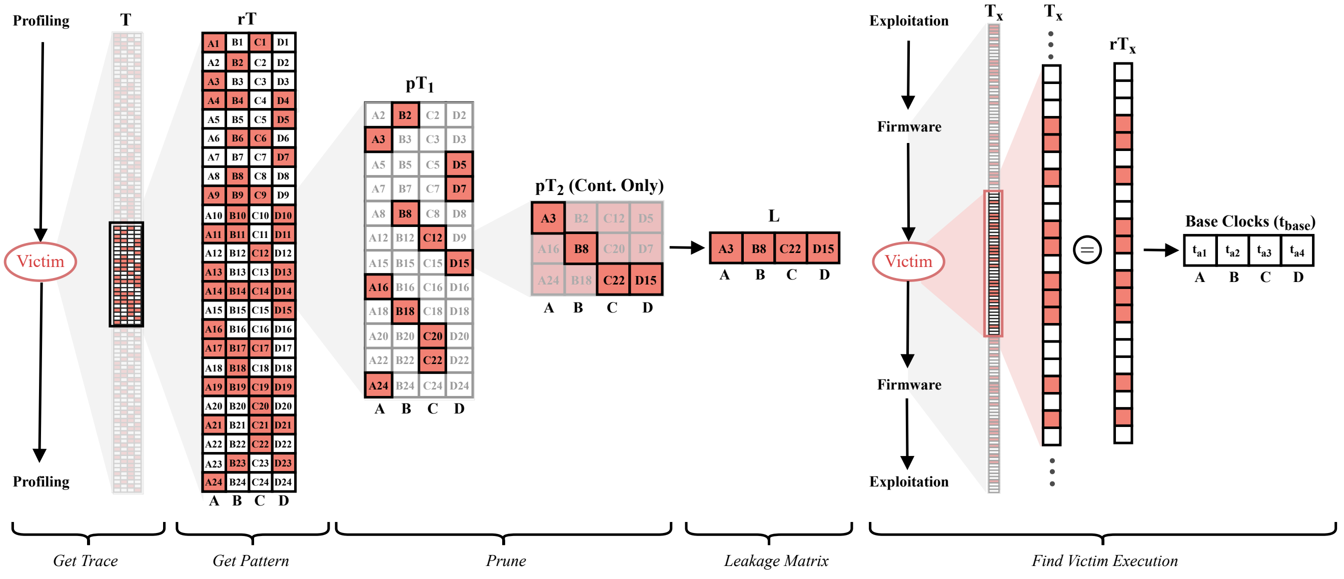


Figure 3: Attack methodology: core steps to identify "vulnerable" clocks cycles. First, the attacker profiles the victim (represented with four execution paths A, B, C, and D) to obtain the trace T. Next, the attacker identifies the pattern rT for the victim's vC. The rT matrix is then pruned to yield the pT_1 matrix, which is further reduced to include only the clock cycles that result in contention, pT_2. Using pT_2, the attacker selects the clocks to be monitored (e.g., A3, B8, C22, and D15) and constructs the leakage matrix L. In the exploitation phase, the spy traces the victim's execution paths and matches them with T_x to obtain the pattern rT_x (collected in the profiling phase). It yields the base clock cycle t_{base} for each execution path.

template matrix ($pT \in \mathbb{N}^{N\times M}$), used in the exploitation phase to infer the secret. Both matrices (represented on the left side of Figure 3) have the same number of columns (M), one per vC execution path, but differ in the number of rows (N). The rT has one row per each vC clock cycle, and the pT only has the rT rows with useful information, i.e., rows with different memory access latencies across vC execution paths. The profiling phase (Algorithm 1), has four stages: (i) first, the vC is identified; (ii) second, the spy gets a victim trace ($\vec{T}$) for each execution path of the vC; (iii) third, the spy gets the rT by searching in $\vec{T}$ the pattern for each vC execution path; and (iv) finally, the pT matrix is computed. To generate the template matrices, i.e., rT and pT, for each vC execution path, the spy needs to get a victim trace $\vec{T}$, which is a vector of memory access latencies per MCU clock cycle.

Identify** vC **and execution paths. The attacker starts by analyzing the victim's code and identifying all secret-dependent code structures. From the constructed poll of vulnerable code, the attacker chooses a snippet to be monitored. This is highly dependent on the target victim application and is performed manually. After clear identification of the vulnerable code, the attacker determines the execution paths that will lead to the leakage of the secret and store them in the vector of execution paths to profile ($\vec{E}$). For instance, in the conditional statement depicted in Figure 2, there are two possible execution paths, i.e., if $S = 1$ or $S \neq 1$.

Get victim trace $\vec{T}$. For each x execution path in $\vec{E}$, the attacker has to check for contention on each victim clock cycle. However, the spy can only assess one clock cycle per

Algorithm 1: Profiling Phase.

```
input  : Victim Binary, $vB$
output: Template Matrices, $rT$ and $pT$
begin Profiling $vB$
    Identify the $vC$ and their execution paths $\vec{E}$
    foreach Execution path $x$ in $\vec{E}$ do
        foreach Victim clock cycle $t$ do
            Generate contention in clock cycle $t$
            Call victim to execute execution path $x$
            $T_t \leftarrow$ Record contention in clock cycle $t$
        end
        $r\vec{T}_x \leftarrow$ Get $vC$ pattern in trace $\vec{T}$
    end
    $pT \leftarrow$ Prune raw template matrix $rT$
end
```

victim run; thus, the profile requires running the victim multiple times (for each x execution path in $\vec{E}$), equal to the number of clock cycles needed for that specific execution path to execute. The spy obtains a trace using a hardware mechanism that creates contention in a particular clock cycle for each victim run and records whether or not there was contention (Section §4.2).

Get pattern $r\vec{T}_x$. For each x execution path in $\vec{E}$, the attacker manually searches for the vC in the trace $\vec{T}$ and stores it in the rT matrix. The vC pattern is within the victim trace, but the attacker has no pre-existing knowledge about the exact location. To discover that information, the attacker correlates the vB with a victim trace by matching the microarchitectural profile of the code with the memory accesses in the trace. To do that, the attacker emulates vB execution at the instruction level; however, this process is highly dependent on the target MCU's microarchitecture. Information about instruction timing and behavior is required but often unavailable due to MCU design secrecy, requiring reverse engineering. This process can be challenging because microarchitectural components may affect code execution, e.g., cache and pipeline hazards. Notable, MCUs are highly deterministic; thus, the victim code generates the same memory fingerprint across different executions.

Prune rT. In this step, the pT matrix is computed from the rT matrix by removing entries without useful side-channel information, i.e., entries with the same memory access latency across all execution paths. To identify N execution paths, the pT matrix needs to have at least N unique entries.

3.2. Exploitation Phase

As previously explained, the exploitation phase can have two variants, i.e., (i) victims that can run multiple times over the same secret and (ii) victims whose secret is ephemeral. We focus on the latter and consider the former out of the scope. We argue that mounting a multiple-run attack is, in principle, easier because the attacker can acquire multiple victim traces and perform most of the attack offline.

The exploitation phase (Algorithm 2) for the single-run variant of the attack encompasses three stages: (i) obtaining the victim trace and finding a specific execution path's pattern ($r\vec{T_x}$); (ii) launching the attack by observing contention in key points in time (clock cycles) of the victim pattern ($r\vec{T_x}$); and (iii) comparing the resulting contention pattern ($\vec{C}$) with a leakage template matrix ($L \in \mathbb{N}^{N\times M}$), to infer the secret ($S$). All of these stages are performed automatically.

Find execution pattern. The target victim is typically an application running atop privileged firmware (e.g., TEE kernel), which switches execution between spy and victim. To determine the exact time occurrence of the vC first instruction, the spy must account for the elapsed time between the spy call and the victim execution, i.e., the first instruction of the vC. We refer to this specific point in time as the base clock cycle (t_{base}). To find all instances of the vC pattern, the spy follows these steps: (i) first, the spy invokes the victim (through the firmware APIs) and traces all the full execution (firmware + victim); (ii) second, the spy searches for the vulnerable code pattern $r\vec{T_x}$ on the trace, and tracks all pattern matches. The $r\vec{T_x}$ pattern may appear more than once in the $\vec{T}$ because a vC may be executed multiple times. For example, if the vC is a piece of code within a loop, it will appear once in $\vec{T}$ for every iteration of the loop. Each pattern match returns a unique clock cycle (t_{base}), which is stored in a vector ($\vec{t}_{base}$), and marks the beginning of each vC execution.

Monitor victim execution. This step is the core of the attack. The spy uses specific hardware mechanisms (see Section §4.2) to automate the attack while the victim executes in the CPU. To do this, the spy monitors specific points in time (clock cycles) within the victim code, which may leak secret-related information. They are defined by two factors: (i) a base clock cycle, $\vec{t}_{base}$, which indicates specific points in time within the victim code where the target vC is executed; (ii) a set of offsets, $\vec{t}_{offset}$, which are relative to $\vec{t}_{base}$ and identify potential contention points within vC. These contention points ($\vec{t}_{offset}$) are selected based on the minimum number of pT matrix entries that unequivocally identify an execution path. The spy launches this phase by repeatedly checking if a specific point in time (t) in the victim code is one of the contention points to be monitored, i.e., t_{base} + t_{offset}. When a match occurs, the mounted hardware spy logic automatically records the result in the $\vec{C}$ vector.

Leak the secret. Finally, the spy has to identify the execution path associated with the recorded $\vec{C}$ pattern to retrieve the secret data. Thus, the spy compares the $\vec{C}$ pattern with the pre-recorded patterns L_x in the leakage matrix L. These patterns L_x are unique fingerprints that unequivocally identify a specific execution path x, and the matrix L is a subset of the $p\vec{T}$ matrix, holding only the matrix entries associated with the monitored $\vec{t}_{offset}$ clock cycles. When there is a match between vectors $\vec{C}$ and $\vec{L}_x$, the spy uncovers that the vC

Algorithm 2: Exploitation Phase Single-Run Variant.

```
input  : Template Matrices, $rT$ and $pT$
input  : Execution path to monitor, $x$
output: Secret or Secret-Related Information, $S$
begin Find Execution Pattern
    $\vec{T}$ ← Invoke victim and get trace
    $\vec{t}_{base}$ ← Find pattern $r\vec{T_x}$ in $\vec{T}$
end
begin Attack
    $L$, $\vec{t}_{offset}$ ← Select $pT$ entries to monitor
    Launch the attack and invoke the victim
    while Victim Executes do Simultaneously
        foreach $t_{base}$ in $\vec{t}_{base}$ do
            foreach $t_{offset}$ in $\vec{t}_{offset}$ do
                if $t = t_{base} + t_{offset}$ then
                    Generate contention in t
                    $C_t$ ← Check contention in t
                end
            end
        end
    end
    foreach Pattern $\vec{L}_x$ in $L$ do
        if $\vec{C} = \vec{L}_x$ then
            $S \leftarrow x$
        end
    end
end
```

executed path x. The leaked information directly exposes the secret, i.e., x is the secret or is closely related to the secret. How the secret is inferred depends on the victim, i.e., the target application.

4. Attack Building Blocks

This section discusses the challenges we faced in designing the attack and how we address them by introducing the concept of *hardware gadgets*. We propose two elementary hardware gadgets as key building blocks to mounting the attack.

4.1. Challenges

To successfully mount an attack, we have to fulfill three main requirements. (**R1**) The spy must be able to create contention in a memory bank shared between the spy and victim (i.e., the spy needs arbitrary control over a BM). (**R2**) The spy must be able to record the shared secondary port access pattern to obtain the execution trace. (**R3**) The spy must be able to detect contention points on-the-fly. These requirements need specific core building blocks, and their materialization imposes four main challenges. These four challenges combined create an unforeseen set of conditions that, to our knowledge, have yet to be addressed so far.

***C1:* Spy lacks access to past microarchitecture states.** Unlike other microarchitectural elements (such as I-/D-caches and branch predictor caches), the bus does not retain any state over time. The contention must be assessed and recorded on-the-fly, i.e., in real-time.

***C2:* Spy is unable to run concurrently with the victim.** MCUs are typically powered by a single CPU[5], thus concurrent execution of spy and victim code is not possible.

***C3:* Victim execution cannot be interrupted.** We assume that the victim domain implements mitigations against Nemesis attacks [18, 38] and has interrupts disabled. Hence, the spy cannot preempt the victim's execution.

***C4:* Spy only has one chance to steal the secret.** The attack must be performed during a single execution of the victim since we are focusing on the single-run variant of the attack.

4.2. Hardware Gadgets

We introduce the concept of *hardware gadgets* to overcome the aforementioned challenges. These gadgets are interconnected peripherals (e.g., timers and DMAs) that can execute specific tasks in the background without requiring CPU intervention. The peripherals are self-contained and have direct communication channels, enabling them to operate in parallel without interfering with each other. Thus, hardware gadgets enable concurrent execution between the victim code (CPU) and the spy logic (hardware gadgets), addressing **C2**. The hardware gadgets can automate logic (such as periodic memory access) without interrupting the victim's code execution, addressing **C3**. The highly deterministic nature of MCUs also enables these gadgets to be synchronized with the victim code at the clock cycle level. It allows the spy to use these gadgets to monitor the victim's activity (e.g., memory accesses) and execute conditional actions based on the victim's control flow, addressing **C1** and **C4**, respectively.

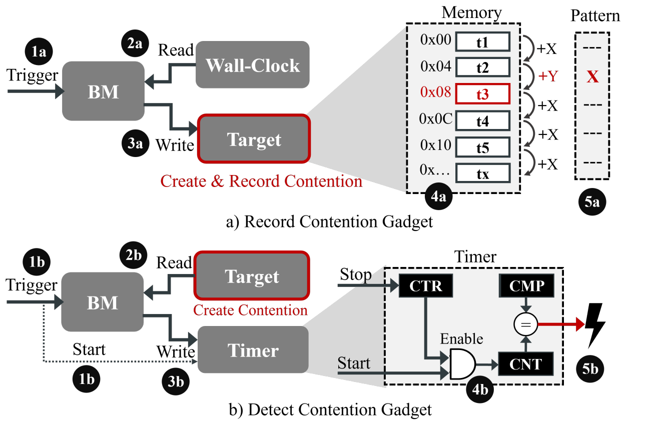


Figure 4: Illustration of two gadgets: a) *record contention gadget* and b) *detect contention gadget.*

To meet the requirements of the attack (creating (**R1**), recording (**R2**), and detecting contention (**R3**)), we devise two elementary hardware gadgets (for advanced gadgets, see Section §5): the *record contention gadget* (Figure 4a) and the *detect contention gadget* (Figure 4b). The *record contention gadget* is used to obtain the victim's memory access pattern, while the *detect contention gadget* is used to detect, on the fly, accesses to the shared memory bank. These gadgets are activated by an external trigger, i.e., software or another hardware gadget. Both require control of a BM and access to a timer (operating at the CPU frequency).

Record contention gadget. The *record contention gadget*, Figure 4a, can record all bus contentions. It can assess the victim memory access pattern and re-create victim execution control-flow traces. The *record contention gadget* consists of three peripherals: a BM for data transfer, a free-running timer acting as a wall clock, and a target memory. When triggered 1a, the BM reads the value of the wall clock 2a and writes it to the target memory 3a, creating and recording contention (**R1** and **R2**). Contention created during consecutive data transfers will result in delays in the BM transfers, causing an observable timing difference. This time difference is recorded by the wall clock. To reconstruct the bus contention history, the spy calculates the difference between recorded transactions 4a after the victim execution, resulting in a pattern 5a similar to the one shown in Figure 2. This gadget is a powerful tool for attackers performing the profiling phase of the single-run variant or mounting the multiple-run variant of the attack.

5. While dual-core configurations can be found in modern microcontrollers, e.g., STM32H7 and i.MX RT1170, single-core MCUs still make up the majority of the market share.

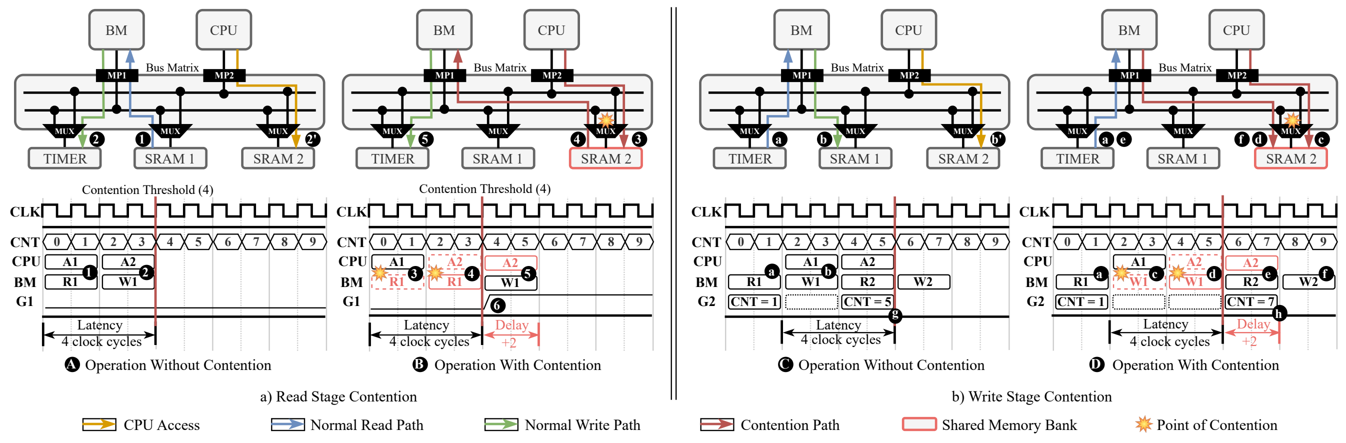


Figure 5: Illustration of two possible contention scenarios: a) `read` stage contention and b) `write` stage contention. Both scenarios are depicted with and without contention at the system-level, with the MCU diagram, and at bus-level, with the timing diagram.

Detect contention gadget. The *detect contention gadget* enables on-the-fly bus contention detection, although it cannot record the contention history. Thus, it is a perfect fit for the single-run variant of the attack. Figure 4b depicts the gadget operation and the timer peripheral. This gadget also consists of three peripherals: the BM for data transfer, the timer for detecting contention, and the target (memory) secondary port. To observe contention, the gadget uses a timer to measure BM latency (4b) and to generate an interrupt (5b) in case contention is detected (which can be handled by software or fed into other hardware gadgets). The detection timer is started at the same time the BM starts the transfer (1b). The BM then reads from the target memory (2b) and writes (3b) the value to the timer control register (CTR), stopping the timer. Both the BM `read` and `write` have a dual goal: the BM `read` creates contention (**R1**) and reads the timer CTR stop value, while the BM `write` stops the timer and detects contention (**R3**). The contention is detected if the BM transfer latency exceeds a certain threshold (minimal BM transfer latency for contention). The compare register (CMP) is set with this threshold value, and when the counter (CNT) overflows, the contention is detected. If there is no contention, the timer will be stopped before the counter overflows, and nothing will happen; otherwise, it will generate an interrupt (5b). By precisely monitoring the start of the transfer (1b), the spy can make the contention happen at a specific time. Using this gadget, a spy can monitor memory accesses with clock cycle granularity (e.g., clock cycle $t + 3$ in Figure 2) and detect contention on the fly without the CPU's intervention.

4.3. Read and Write Stage Contention

At the bus level, data transfers consist of dual-stage access, i.e., data is read from one memory address and written to another. A notable exception is the CPU, which can perform single-stage accesses by reading data from core registers and writing to memory or vice-versa. The spy can use read and/or write stages to create contention, depending on the characteristics of the peripheral.

Figure 5 depicts the bus view of a `read` and `write` stage contention. In this case, the bus arbitration relies on the *round-robin policy*, with a bus quantum of 2 clock cycles. In this example, the spy-controlled bus main (BM block in Figure 5) performs, in the `read` stage, one BM transfer, and in the `write` stage, two BM transfers. In both contention scenarios, the CPU does two consecutive memory accesses, i.e., A1 and A2 accesses. In this example, the first arbitration is granted to the CPU, and all subsequent BM memory accesses have an equal bus quantum. As shown in Figure 5, the bus quantum (3) and (c) are assigned to the CPU's A1 access, and the BM transfers are postponed to the next bus quantum (4) and (d). A contention happens between the second CPU memory access, A2, and the delayed BM operation, R1 (4) and W1 (d). However, the CPU A2 access is precluded by the BM R1 and W1 accesses, incurring an additional bus quantum delay to complete the A2 operation (5) and (e).

Read stage contention. Figure 5a illustrates the timing and bus level interactions at the `read` stage. In a memory transaction without contention, the BM reads from SRAM 1 (1) and then writes the read value to the timer control register (2). At the same time, the CPU does two consecutive accesses to SRAM 2 (2'). In this case, there is no contention because there is no dispute for the SRAM access. When CPU and BM issue concurrent accesses, the bus matrix round-robin arbitration process the CPU A1 and delays the BM R1 read (3). Therefore, R1 is delayed by one bus quantum (4), and thus the W1 write operation (5). This contention point is captured with the detect contention gadget (G1), which triggers an interrupt upon the occurrence of a certain threshold. On an operation without contention (Figure 5a - timing diagram A), the timer starts counting upon the BM transfer A1 (1) and is immediately stopped by the BM write W1 (2), i.e., the BM itself writes to the timer control register to stop counting. The interrupt is not triggered in this case because no contention was detected. In a scenario under contention (Figure 5a - timing diagram

B), the BM write W1 is delayed ⑤, and, therefore, the threshold is met, and the timer interrupt is triggered ⑥.

Write stage contention. Figure 5b illustrates the timing and bus level interactions at the `write` stage. Contrary to the `read` stage contention, in this case, the BM needs to perform two consecutive transactions per bus contention. For an operation without contention (Figure 5b - timing diagram C), the CPU accesses the SRAM 2 ⓑ' while the BM performs two memory transactions. First, the BM reads a free-running timer ⓐ and then writes the read value to SRAM 1 ⓑ. No contention occurs in this case since both transactions do not share the same memory banks. In a contention scenario (Figure 5b - timing diagram D), the BM attempts to write the timer value to SRAM 2 while the CPU simultaneously accesses the same memory ⓒ. At the bus matrix, the BM W1 ⓓ operation is delayed, which introduces a cascade effect delay on all the remaining operations, i.e., BM R2 ⓔ and BM W2 ⓕ. The spy leverages this delay (recorded in memory) to identify a contention point in the memory access trace by calculating the time difference between two consecutive BM transfers. If there is no contention (Figure 5b - timing diagram C), the BM will read a value of 1 and 5, with a transfer latency of 4 ⓖ. If there is contention (Figure 5b - timing diagram D), the BM will read a value of 1 and 7, with a transfer latency of 6 ⓗ.

5. BUSted Attack

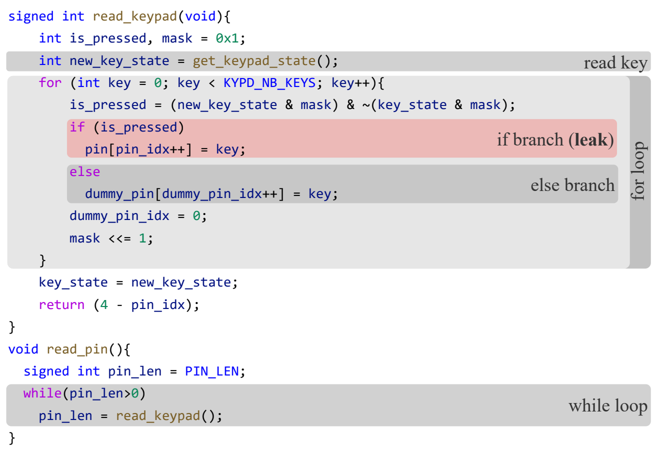
In this section, we introduce the BUSted[6] attack. Like other side-channel attacks [5, 18, 39–42], we focus on single-run applications whose secret depends on asynchronous events, e.g., a user input. To demonstrate the effectiveness of the attack in a realistic setup, we used an off-the-shelf smart lock application which we justify and describe next. This application uses an external keypad to interface with the user. The attacker's goal is to steal the PIN entered by the user.

5.1. Off-the-Shelf Smart Lock Application

The selected smart lock application is a component of Vulcan [43], a vehicular authentication system that adheres to a reference implementation of a trusted keypad[7] provided by Texas Instrument [19]. This very same application was used in the Nemesis attack [18]. There are also popular open-source firmwares such as QMK [47] and TMK [48] that use a similar logic to interface with keyboards, i.e., larger keypads.

We embedded this off-the-shelf logic in a trusted application (TA) deployed in the secure domain of a TustZone-enabled MCU. It receives service requests from the main

6. https://github.com/ESCristiano/BUSted
7. Texas Instruments offers several products based on the trusted keypad reference implementation [19], e.g., the Low-Power Hex Keypad [44], the BOOST-IR BoosterPack Plug-in Module [45], and a USB Keyboard [46].



```
signed int read_keypad(void){
    int is_pressed, mask = 0x1;
    int new_key_state = get_keypad_state();
    for (int key = 0; key < KYPD_NB_KEYS; key++){
        is_pressed = (new_key_state & mask) & ~(key_state & mask);
        if (is_pressed)
          pin[pin_idx++] = key;
        else
          dummy_pin[dummy_pin_idx++] = key;
        dummy_pin_idx = 0;
        mask <<= 1;
    }
    key_state = new_key_state;
    return (4 - pin_idx);
}
void read_pin(){
  signed int pin_len = PIN_LEN;
  while(pin_len>0)
    pin_len = read_keypad();
}
```

Figure 6: We split the `read_keypad` code snippet into five parts. There are two potential execution paths, depending if a key is pressed. Both paths include three common parts: (i) the `while` loop, (ii) the `read_key` function, and (iii) the `for` loop. The key difference between the two paths is the secret-dependent `if` statement, which causes a variation in the memory access patterns based on user input. When a key is pressed, the `if branch` is executed; otherwise, it is the `else branch`.

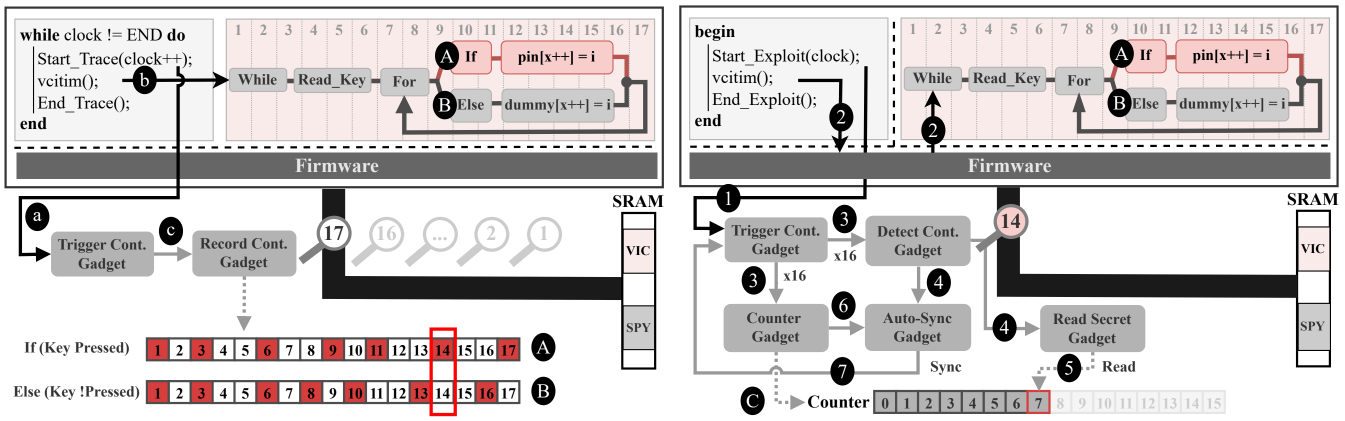
application in the non-secure domain, e.g., to authenticate a user. Upon a request, it loops over all 16 keys of the keypad (i.e., 4x4 key matrix) until all four digits of the pin have been pressed. The TA then returns a status message indicating whether the entered pin was valid or not. The code snippet of the smart lock `read_keypad` function is presented in Figure 6.

To steal the PIN, the spy monitors the victim's control flow and checks whether the victim executes the secret-dependent path (Figure 6). Apparently, the code logic is not vulnerable to timing attacks since the code is balanced for a constant time. However, the assembly implementation of the `if` statement (we refer the interested reader to Appendix A) indicates that memory accesses occur at different points in times, suggesting the secret-dependent path and the secret.

With the identification of the vulnerable code (`if` statement), the spy only needs to monitor a specific timing (similar to $t+3$ clock cycle in Figure 2) in each `for` loop iteration. For example, if the spy observes that the `if` branch was executed in the 7th iteration of the `for` loop, it indicates that the user pressed the key 7. The spy logic is simple: (i) keep track of the loop and record the number of iterations it has executed, and (ii) monitor a specific point in time (clock cycle) to observe contention. When a key is pressed, the spy will observe the resulting contention. To acknowledge the pressed key, the spy resorts to the internal counter that keeps track of the `for` loop.

5.2. Smart Gadget Network

To track execution and assess the victim control flow, we devised a combination of hardware gadgets called *smart gadget network* (SGN). SGN interconnects multiple hardware gadgets to perform a specific logic. To implement



a) Profiling Phase b) Exploitation Phase

Figure 7: Profiling and Exploitation SGNs. The gray arrows connecting the hardware gadgets of both SGNs represent direct communication channels between the gadgets' peripherals. These communication channels are used to send signals to each other, e.g., trigger their execution or acknowledge a particular action.

the attack methodology (see Section §3), we resort to two SGNs: one for the profiling phase and another for the exploitation phase.

The *profiling SGN* (pSGN), Figure 7a, is used to trace the victim's execution. To achieve this, the pSGN uses two hardware gadgets: (i) the *record contention gadget* that generates and records contention; and (ii) the *trigger contention gadget* that instructs the *record contention gadget* to execute and create contention at specific points in time (clock cycle level). The *exploitation SGN* (eSGN), Figure 7b, is used to track the smart lock application execution and steal the secret. This SGN automatically generates and detects contention on the secret-dependent `if` branch and keeps track of the `for` loop iteration number. To achieve that, the eSGN uses five hardware gadgets: (i) the *detect contention gadget*, to generate and automatically detect contention; (ii) the *trigger contention gadget* (same as the pSGN), to trigger the *detect contention gadget* and generate contention in arbitrary clock cycles; (iii) the *counter gadget*, to track the iteration index of the `for` loop; (iv) the *read secret gadget*, to automatically read the *counter gadget* when there is contention (i.e., a key was pressed); and (v) the *auto-sync gadget*, to keep the eSGN in-sync (at the clock cycle granularity) with the victim.

5.3. Complete Attack Flow

In the *profiling* phase, the attacker uses the pSGN to capture the memory access patterns. During the (online) attack, the attacker uses the eSGN to retrieve the secret. Figure 7 shows an overview of the attack.

Offline profiling phase. In this phase, Figure 7a, the spy has full control over the victim, which is executed within the spy's domain. The spy gets the victim's execution profile (traces Ⓐ and Ⓑ) by running the victim multiple times and, in each execution, incrementing the clock cycle where contention is created. To profile an execution path, the spy forces the victim to execute only one path per run by forcing a constant input. This can be done either by pressing the same key repeatedly – we mimic this behavior by instrumenting the keypad with a shunt resistor – (to trace path Ⓐ) or by not pressing any key at all (to trace path Ⓑ). The profiling involves three steps. Firstly (**step 1**), the spy instructs the pSGN to generate contention at a specific clock cycle ⓐ. Then (**step 2**), the spy invokes the victim ⓑ. Finally (**step 3**), after a predetermined amount of time has elapsed (indicated by the `clock` variable in Figure 7a), the *trigger contention gadget* activates the *record contention gadget* ⓒ. This process is repeated (loop) until the victim executes the last instruction. The profiling is conducted for each of the 17 potential execution paths in the `read_keypad` code (16 distinct keys plus one path for the absence of key press). Each execution path may generate one of two patterns: one when a key is pressed (`if` path) Ⓐ, and another when there is no key press (`else` path) Ⓑ.

Online exploitation phase. In the exploitation phase, Figure 7b, the spy monitors specific points in time ($t_{base}+t_{offset}$), to detect the executed path and thus inferring the secret. The spy and the victim run in isolated domains. The victim's execution is initiated by the spy's exploitation code that issues a request to the secure firmware, which then performs a domain switch ② and runs the victim. This context switch results in a timing offset (t_{base}) of the profiling phase patterns.

(**Step 1**) To account for the context-switch (firmware) overhead, the spy traces the victim domain (victim + firmware), and searches for the patterns (rT) obtained in the profiling phase. It enables the spy to determine t_{base}, i.e., the time elapsed from when the spy invokes the victim until the first victim instruction is executed. The spy uses this offset to adjust the patterns (rT and pT) from the profiling phase.

(**Step 2**) After selecting the target point in time ($t_{base}+t_{offset}$), the spy launches the attack. First, the spy triggers ① the eSGN to generate contention in the selected clock cycle and invokes the victim ②. Then, the *trigger gadget* creates 16 triggers ③, one per iteration of the `for` loop. This causes the *detect contention gadget* to generate contention at the first clock cycle (t_{base}) and in multiples of that ($t_{base} + t_{offset}$). Each trigger generated ③ is fed into the *counter gadget*, which will keep track of the number of triggers, from 0 to 15 Ⓒ. Upon reaching the count of 15, the *counter gadget* signals the completion of the `for` loop ⑥.

(**Step 3**) When contention is detected, a signal ④ is sent to both the *read secret gadget* and the *auto-sync gadget*. The *read secret gadget* then accesses the value of the *counter gadget* Ⓒ to reveal the key (secret).

(**Step 4**) The *auto-sync gadget* is activated by either the completion of the `for` loop ⑥ or by the detection of contention ④. Upon activation, it adjusts ⑦ timing for the next clock cycle in which contention will occur, thereby re-synchronizing the eSGN with the victim code.

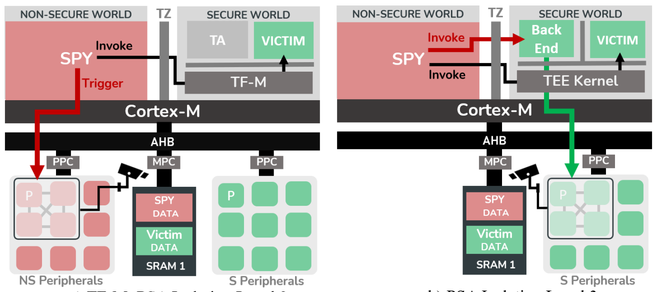
Finally, as an example, let us assume the initial instruction of the victim executes at the clock cycle 1000 (t_{base}), and the spy chooses the 14th clock cycle (t_{offset}), from traced patterns Ⓐ and Ⓑ. Then, the 14th clock cycle shifts to 1014, and the spy monitors clock 1014 ($t_{base} + t_{offset}$). Assuming that the `for` loop requires 50 cycles to complete, the contention will be generated in the clock cycle 1014 for key0, 1014 + 50 for key1, 1014 + 100 for key2, and so on. If the *counter gadget* holds a value of 7 when contention is detected, i.e., clock cycle 1014 + 7x50, it indicates that the key corresponding to the 7th iteration (key7) of the `for` loop was pressed.

6. Implementation and Evaluation

We implemented BUSted on the NUCLEO-L552ZE-Q [49] and ATSAML11E16A [50] (Armv8-M/TrustZone-M platforms available when we started this work). From now on, we will refer to NUCLEO-L552ZE-Q as ST and ATSAML11E16A as SAM. As secure TEE kernel (firmware), we used the open-source TF-M [20]. Although there are other TEE kernel options, we either were not granted an evaluation license (e.g., ProvenCore-M [51]) or we decided to exclude them due to limited adoption (e.g., mTower [52], and uTango [53]). Furthermore, despite our findings suggesting that it may be possible to implement the attack targeting other ISAs (e.g., Armv6/7-M, and even RISC-V) and firmware (e.g., MultiZone [22], Zephyr [21]), we focused on security-oriented TEE hardware technologies for MCUs (TrustZone-M).

6.1. TF-M: PSA Isolation Level 2

Attack setup. We leveraged TrustZone-M to create a trusted and an untrusted domain (secure and normal worlds, respectively). As depicted in Figure 8a, the spy operates in the normal world, while the victim (smart lock application) is encapsulated in a trusted application (TA). The TA runs



a) TF-M: PSA Isolation Level 2 b) PSA Isolation Level 3

Figure 8: BUSted attack setup.

in secure unprivileged thread mode on top of the privileged trusted kernel (TF-M). TF-M [20] was configured at PSA isolation level 2 [54], which provides isolation between the secure and non-secure world (TrustZone), as well as between trusted kernel and TAs (secure MPU). To create contention on the shared memory banks, the spy uses a DMA as a malicious BM and leverages a set of peripherals that are accessible to the non-secure world to build the SGNs.

Attack implementation. We mounted this attack scenario on the ST platform. The ST features a Cortex-M33 that implements the Harvard architecture, i.e., instruction and data fetches occur on separate buses, and has 8KB of I-cache and no D-cache. The platform's bus interconnect uses round-robin as the arbitration policy. It has a 256KB of SRAM memory, divided into two banks: one with 192KB (where the NS data is located) and another with 64KB (exclusively assigned to TF-M). The spy cannot observe memory accesses to the 64KB bank; however, TF-M allocates the TA data to the 192KB memory bank by default. Out of the 73 peripherals available on the ST platform, only 6 (i.e., 4 timers, 1 DMA, and 1 DMA Multiplexer) were used in this exploit, which represents 8.2% of the available peripherals. We successfully bypassed all isolation barriers at PSA level 2 (TrustZone and secure MPU) and retrieved the secret of the smart lock TA with a 100% success rate, i.e., we always recover the user's PIN.

6.2. PSA Isolation Level 3

Attack setup. Figure 8b expands the system configuration to a more restrictive (but realistic) scenario: all MCU peripherals are assigned to the secure world, i.e., enforcement via the so-called peripheral protection controller (PPC), and not directly accessible on the normal world to the attacker. In this case, device access is mediated via a TA in the secure world. We acknowledge the deployment of 3rd party TAs as part of the thread model of TrustZone-M [20]. Thus, the TEE kernel should be configured with PSA isolation level 3 [54], which extends PSA level 2, enforcing protection between TAs via the secure MPU. To validate BUSted attack, we devised a stripped-down version of TF-M. We reduced the code base to the bare minimum set of features to provide: (i) peripherals assignment/protection via PPC to the secure world; (ii) memory protection enforcement

Gadgets	**Timers** *SAM*	*ST*	**DMAs** *SAM*	*ST*	**DMA MUX** *SAM*	*ST*	**Event System** *SAM*	*ST*	**LUTs** *SAM*	*ST*
Rec. & Det.	1	1	1 (same)	1	Na	2	4	Na	–	Na
Trigger	1	1	–	–	Na	1	1	Na	–	Na
Counter	–	1	1 (same)	–	Na	1	–	Na	–	Na
Read Secr.	–	–	1	1	Na	–	–	Na	2	Na
Auto-Sync	–	1	3	3	Na	1	–	Na	–	Na
Total	2/3	4/14	5/8ch	5/16ch	Na	5/16ch	5/8ch	Na	2/2	Na

Platform	**Core** CPU	Freq.	**Bus** Arch.	Arb.	**Memory** SRAM	FLASH	**I-Cache** Size	Status	**N Peri.** Total	Used
SAM	M23	32 MHz	VN	P	16KB	64KB	512B	Ena	29	5
ST	M33	110 MHz	H	R	256KB	512KB	8KB	Ena	73	6

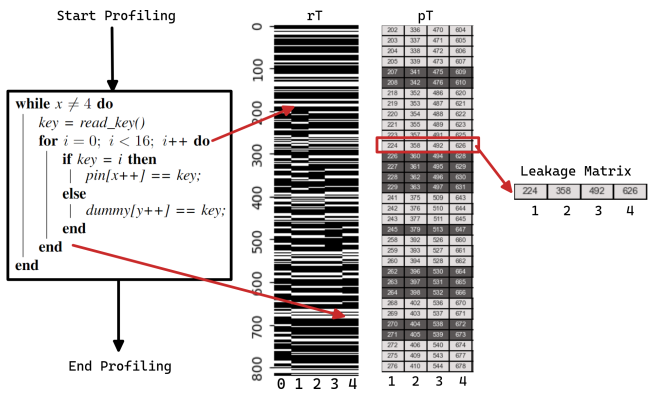
TABLE 2: Resources used by the BUSted attack by gadget and the characteristics of each platform. CPUs: Cortex-M23 (M23) and Cortex-M33 (M33). Architectures: Von Neumann (VN) and Harvard (H). Arbitration Policies: priority-based (P) and round-robin (R). Resource not used (–).

between trusted kernel and multiple TAs via the secure MPU; security gate (SG) entry/exit logic for TAs. We argue that this code base presents the smallest TCB to implement a system following PSA isolation level 3. Any extra firmware would increase the TCB and, therefore, the attack surface. On top of this stripped version of TF-M, we ran a 3rd party TA as a back-end service acting as an interface to secure peripherals.

Attack implementation. We mounted this attack scenario on the SAM platform. The SAM features a Cortex-M23, implementing the Von Neumann architecture, i.e., instruction and data fetch occur on the same bus, and has 512B of I-cache and no D-cache. The platform's bus interconnect uses a priority-based arbitration policy. The SAM platform has a single bank of 16KB SRAM, shared between secure and non-secure software, i.e., the spy can monitor all the secure world memory accesses. Out of the 29 peripherals available on the SAM platform, only 5 (i.e., 2 timers, 1 DMA, 1 configurable custom logic (CCL), and 1 Event System) were used in this exploit, which represents 17.2% of the available peripherals. Even under a very restrictive system configuration, we also successfully bypassed all isolation barriers at PSA level 3 (secure MPU, TrustZone-M, and PPC) and retrieved the secret of the smart lock TA with a 100% success rate. Despite the SAM bus interconnect implementing a priority-based policy, we found no additional implementation challenges. By default, the highest priority is assigned to the CPU, i.e., the bus arbitration logic will always grant access first to the CPU and only when the CPU is idle to the remaining bus mains - end-result, contention always exists.

6.3. Platform Constraints on SGNs

Table 2 summarizes the platform resources (hardware gadgets) required to successfully mount the BUSted attack. The number of gadgets varies across platforms due to the difference in features and properties of the peripherals. While some gadgets, e.g., *record*, *detect*, and *trigger* gadgets, use the same peripherals on both platforms (just slight variations), others, such as the *counter gadget*, require entirely different peripherals (i.e., a timer in ST and a DMA in SAM). Despite having the same design, SGNs implementations can be different.



```
while $x \neq 4$ do
    key = read_key()
    for $i = 0;\ i < 16;\ i$++ do
        if key = $i$ then
            pin[x++] == key;
        else
            dummy[y++] == key;
        end
    end
end
```

Figure 9: Smart lock application trace recorded in SAM. Both boards generate similar traces. We limit the trace to 4 execution paths, i.e., 4 keys.

The pSGN in ST uses the write stage contention, while the eSGN uses the read stage contention. On the SAM platform, both the pSGN and eSGN utilize the read stage contention. The timers used to implement the *record gadget* wall-clock (see §4) do not have the counter register synchronized with the internal counter, which means we cannot read a register to get the timer counter value. This drives the *record gadget* to be implemented using the read stage contention, i.e., a trigger initializes a timer, and the DMA stops the timer, which measures the DMA transfer latency. In post victim run, the spy manually synchronizes the timer to read its counter value.

The differences in the DMAs from ST and SAM is another major implementation constraint. In the former, the DMA configuration is static, meaning that once it is set up, the DMA always performs the same transfer. In contrast, the DMA channel on the SAM platform is much more versatile, allowing for multiple transfers, each with a different configuration. As a result, on the ST, each gadget requires a dedicated DMA channel. In contrast, the DMA channel on the SAM platform can multiplex operations. This allows the *detect* and *counter gadgets* to share the same DMA channel and the *auto-sync gadget* to be implemented exclusively with DMA channels. In contrast, on the ST, the *counter* and *auto-sync gadgets* require an additional timer. Furthermore, SAM features a peripheral (the CCL), which enables the implementation of basic logic gates such as AND and OR. These gates can be used in signals generated by gadgets. The *read secret gadget* utilizes this feature to execute a logical condition and determine whether it should execute its function.

6.4. Real Victim trace

Figure 9 shows a real victim trace obtained from SAM. Due to space limitations, we only present the trace for five execution paths. The figure includes a representation of the rT and pT matrices obtained from profiling the `read_pin` code (Figure 6), as well as the L matrix for the exploitation phase. Columns 1 to 4 of rT represent four key presses. Column 0 is provided as a visual reference point (no key pressed). After all pruning operations (see §3.1), we obtain the pT matrix with all the unique contention points from rT matrix, from which we select four points in time (clock cycles) to monitor in the exploitation phase (note that any clock or combination of clocks from pT would be effective). The secret is immediately revealed upon the existence of contention in one of these monitored timing points.

7. Countermeasures

The most effective countermeasure to mitigate BUSted is to avoid sharing memory banks between sensitive and non-sensitive execution domains. However, this may not always be a realistic possibility. MCUs are typically resource-constrained, and among the entire spectrum, many devices feature a single memory bank, e.g., the evaluated SAM platform. Furthermore, in many cases, code is loaded from the flash directly into data memory (SRAM) to boost performance and minimize power consumption [55]. This creates difficulties even on devices that feature multiple memory banks. Thus, below, we point out other potential countermeasures.

Avoid secret-dependent code. Eliminating secret-dependent code can prevent the spy from detecting timing differences resulting from simultaneous memory accesses. Developers may balance memory accesses across execution paths, but it requires significant engineering effort and expertise. Alternatively, this process can be automated at the compiler level [56]. However, this approach would still have severe scalability limitations due to the intrinsic dependencies on the microarchitecture and the exponential design options.

Disable DMA during the victim execution. DMA peripherals (or other bus mains), a key element in the overall attack methodology, may be disabled during the execution of sensitive applications. However, this may not be a realistic approach due to the widely-establishment of DMAs in MCU platforms and the severe impact on these applications' functionality, usability, and performance.

Enforce priority-based arbitration policy. For platforms with the priority-based policy (e.g., the SAM platform discussed in §6.2), it is possible to program priorities and thus prioritize other bus mains, e.g., DMA devices, instead of CPUs. In this case, it is possible to eliminate the contention from the DMA (attacker) perspective since those transactions are always prioritized. Notwithstanding, this approach has a significant drawback since it comes at the cost of enabling the attacker to perform a complete denial of service attack over the main CPU (CPU starvation).

Add random delays to the victim code. Adding random delays while mediating the victim execution (e.g., on the TEE kernel while switching between execution domains) introduces unpredictability, resulting in non-deterministic patterns. This hampers the spy's ability to monitor memory accesses and build the template matrix. One challenge with this approach is related to the source of randomness since not all MCUs feature random number generators.

8. Discussion

BUSted generalization to other applications. We acknowledge that BUSted is highly tailored to the target application; however, we argue that our attack methodology (see §3) is generic and can be used to attack other applications with some engineering effort. BUSted is a proof-of-concept attack that demonstrates the effectiveness of our methodology. While the SGN requires a re-design per application, hardware gadgets may be reused, considerably reducing the generalization effort to attack other applications. Furthermore, we argue that BUSted was designed under a pessimistic scenario, i.e., single-run, single-core, and non-preemptable execution. Relaxing just one of those conditions would significantly enhance the attacker's capabilities. For instance, in a multi-run attack scenario (e.g., software-based cryptographic application), the complexity of the SGN would be considerably reduced due to the possibility of obtaining multiple traces. Alternatively, in a single-run attack with preemption, the attacker could use an approach similar to the ones reported in Nemesis [18] and SGX-Step [57] (i.e., interrupt the victim at each instruction, launch the record contention gadget, return to the victim instruction, and observe if there was contention), to yield a very accurate trace of all the victim instructions while, once again, highly reducing the complexity of the SGNs.

BUSted portability to other platforms. Given the pervasiveness of the bus interconnect across the MCU spectrum, we strongly believe that similar variants of the attack may be replicable on a plethora of platforms powered by Armv6/7/8-M, RISC-V (and other ISA) MCUs. We support our claims with the empirical findings from Table 1. Notwithstanding, the high heterogeneity of the MCU spectrum in terms of architectural and microarchitectural elements may create some challenges. While the SGN design is agnostic from the MCU, its implementation depends on the available peripherals and microarchitecture. We argue that (i) MCUs typically have enough peripherals to create the SGN and (ii) the SGN design is flexible enough to cope with the expected heterogeneity.

BUSted and hardware gadgets. The number and type of hardware gadgets an attacker can use to implement an SGN are limited by the overall peripherals available on the target platform. Moreover, hardware gadgets can only automate simple operations such as memory reads/writes, comparisons, tracking time, and simple combinational logic operations (e.g., AND, OR, NOT). Although it is possible to use hardware gadgets that implement conditional actions (as we did with the *detect contention gadget*), the conditions must be simple (e.g., $>$, $=$, $<$). These restrictions impose inherent limitations on the spy logic. Notwithstanding, we argue it is possible to combine multiple hardware gadgets to implement a reasonably complex algorithm, as demonstrated with the eSGN, using only a fraction of the platform peripherals (see Section §6).

BUSted and multi-core MCUs. BUSted leverages a DMA as the malicious bus main; however, it is possible to mount

BUSted resorting to other bus mains. For instance, there is an ongoing trend to embed dual-core configurations in modern MCUs (e.g., STM32H7 and PSoC 64 series). Inclusively, the PSoC 64 leverages a dual-core (Cortex-M4 + Cortex-M0+) to implement isolated execution environments and is Arm PSA certified. Thus, for this MCU series, we envision a scenario where the spy runs on one CPU (e.g., Cortex-M4) and the victim on the other CPU (e.g., Cortex-M0+). This design configuration empowers the attacker since the spy logic can run concurrently and a significant portion of it can be implemented in software, i.e., easing the reliance on SGNs and the complexity and limitations they induce (aforementioned).

BUSted (full-)attack automation. To successfully mount BUSted, we acknowledge the need for a skilled attacker and a large manual effort; however, we argue that automating a significant part (if not all) of the process is possible. For example, mounting hardware gadgets and SGNs can be reflected in a puzzle-solving problem that considers the constraints and functionalities of the peripherals. Since both are well-defined, it is realistic to develop a tool that takes the target application (algorithm) as input and outputs an SGN design (and potential implementation). Furthermore, target applications with less restrictive attack conditions (e.g., multi-run crypto-related applications) may allow for a fully automated template attack.

9. Related work

Side-channel attacks are not a new endeavor. These attacks can leverage several physical properties, e.g., power [58–61], sound [62–64], and electromagnetic emanations [65–68], to leak secret information. Microarchitectural side-channel attacks are a particular class of these attacks introduced by Kocher [69] which achieved mainstream attention with the disclosure of Spectre [3] and Meltdown [4]. At a high level, there are two major classes of software-based microarchitectural side-channel attacks [70]: *persistent-state* and *transient-state* attacks. However, transient execution attacks [2, 71] are not transient-state attacks, but rather a combination of both.

Persistent-state attacks, also called stateful or residual-state attacks, modify the state of a microarchitectural element and preserve this state over time (at least after the victim's execution). Prominent examples include those targeting the cache [36, 72–78], the address translation [10, 79–83], and branch predictors [42, 84–88]. Transient-state attacks, also called stateless or contention-based attacks, [89], exploit transient information that does not leave any microarchitectural trace. These channels arise due to limited bandwidth when multiple bus mains compete for the same microarchitectural resource. There are multiple transient-state side-channel attacks, leveraging different microarchitectural elements, e.g., cache banks [70, 90], execution ports [91–93], and bus interconnects [89, 94–96]. BUSted can be seen as a transient-state attack, similar to attacks leveraging the interconnect of high-end processors [89, 94, 96]. Nevertheless, multiple-run variants of our attack (using hardware gadgets to record a contention trace) may fall under the persistent-state classification. BUSted is also close to controlled-channel attacks [5, 18, 39, 40], a subset of side-channel attacks that leverages deterministic and noiseless channels to extract a victim's secret in a single-run.

Software-base side-channel attacks on MCUs. Despite the significant research effort on high-end processors, only a few software-based side-channel attacks have been reported on MCUs [18, 97–99]. In [98], authors exploit a timing side-channel to construct a leakage module of the Cortex-M4 and Cortex-M7 MCUs, which is then exploited through a power side-channel. From a different perspective, in [99], D. Gnad et al. leverage the correlation between the ADC noise and the power consumption of the MCU (STM32 Cortex-M4) to collect from software power consumption traces, which are then used to leak the secret key of an AES implementation. Similarly, in [97], authors also use the ADC to mount a remote power side-channel attack to bypass the TrustZone-M protection of a SAM L11 (Cortex-M23) MCU and retrieve the secret key. Finally, the Nemesis attack [18] explore the microarchitecture of the MSP430 MCU to leak sensitive information. In summary, most existing works are software-based power side-channel attacks. To the best of our knowledge, the Nemesis attack [18] is the single research work focused on microarchitectural side-channels on MCUs; however, the attack is tied to the particularities of the CPU interrupt logic of the MSP430 MCU and has not proven scalable across the MCU spectrum (in particular in Arm Cortex-M).

10. Conclusion

In this paper, we presented BUSted, a novel side-channel attack that leverages timing differences on the MCU bus interconnect to undermine security guarantees enforced by memory protection primitives, including MPU, TrustZone-M, and other platform-level protection controllers (e.g., PPC). To tackle the challenges posed by resource-constrained MCUs, we introduced a novel concept named hardware gadgets. We mounted the attack on two modern TrustZone-M hardware platforms, i.e., the ST NUCLEO-L552ZE-Q and the Microchip ATSAML11E16A, running the TF-M TEE kernel configured at PSA isolation levels 2 and 3.

Acknowledgments

We would like to thank the reviewers for their valuable feedback and suggestions. Cristiano Rodrigues was supported by FCT grant 2020.08729.BD. Daniel Oliveira was supported by FCT grant 2020.04585.BD. This work is supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope UIDB/00319/2020, and partially by the European Union's Horizon Europe research and innovation program under grant agreement No 101070537, project CROSSCON (Cross-platform Open Security Stack for Connected Devices).

References

[1] Qian Ge et al. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journ. Cryptogr. Eng.* 2018.

[2] Claudio Canella et al. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *Proc. of USENIX Security*. 2019.

[3] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proc. of S&P*. 2019.

[4] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Proc. of USENIX Security*. 2018.

[5] Daniel Moghimi et al. “COPYCAT: Controlled Instruction-Level Attacks on Enclaves”. In: *Proc. of USENIX Security*. 2020.

[6] Michael Schwarz et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *Proc.s of ACM CCS*. 2019.

[7] Stephan van Schaik et al. “CacheOut: Leaking Data on Intel CPUs via Cache Evictions”. In: *Proc. of S&P*. 2021.

[8] Stephan van Schaik et al. “RIDL: Rogue In-Flight Data Load”. In: *Proc. of S&P*. 2019.

[9] Jo Van Bulck et al. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *Proc. of S&P*. 2020.

[10] Claudio Canella et al. “Fallout: Leaking Data on Meltdown-Resistant CPUs”. In: *CCS ’19*. 2019.

[11] Daniel Katzman et al. “The Gates of Time: Improving Cache Attacks with Transient Execution”. In: *Proc. of USENIX Security*. 2023.

[12] Moritz Lipp, Daniel Gruss, and Michael Schwarz. “AMD Prefetch Attacks through Power and Time”. In: *Proc. of USENIX Security*. 2022.

[13] Joseph Ravichandran et al. “PACMAN: Attacking ARM Pointer Authentication with Speculative Execution”. In: *Proc. of ACM ISCA*. 2022.

[14] Keegan Ryan. “Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm’s TrustZone”. In: *Proc. of ACM CCS*. 2019.

[15] Jose Rodrigo Sanchez Vicarte et al. “Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest”. In: *Proc. of S&P*. 2022.

[16] Gregor Haas, Seetal Potluri, and Aydin Aysu. “iTimed: Cache Attacks on the Apple A10 Fusion SoC”. In: *Proc. of IEEE HOST*. 2021.

[17] Statista. *DIGITAL & TRENDS Microcontrollers*. Tech. rep. Statista, 2023.

[18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *Proc. of ACM CCS*. 2018.

[19] Texas Instruments. *Implementing An Ultra-Low-Power Keypad Interface With MSP430™ MCUs*. Tech. rep. Texas Instruments, Fev 2002/18.

[20] Arm. *Arm Trusted Firmware*. www.trustedfirmware.org. Accessed: 2023-02-09.

[21] The Linux Foundation. *Zephyr Project*. www.zephyrproject.org. Accessed: 2022-12-01.

[22] Sandro Pinto and Cesare Garlati. “Multi Zone Security for Arm Cortex-M Devices”. In: *Proc. of Embed. World Conf.* 2020.

[23] Arm. *Clarification of Timing Side Channel Attacks on TrustZone enabled Cortex-M based systems*. https://developer.arm.com/documentation/ka005578/latest. Accessed: 2023-07-09.

[24] David Cerdeira et al. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: *Proc. of S&P*. 2020.

[25] Jo Van Bulck et al. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *Proc. of CCS*. 2019.

[26] Marvin Saß, Richard Mitev, and Ahmad-Reza Sadeghi. “Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M”. In: *arXiv*. 2023.

[27] Johannes Obermaier, Marc Schink, and Kosma Moczek. “One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers”. In: *Proc. of USENIX WOOT*. 2020.

[28] Carlton Shepherd et al. “Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis”. In: *Journ. Comp. & Sec.* 2021.

[29] Chong Hee Kim and Jean-Jacques Quisquater. “Faults, Injection Methods, and Fault Attacks”. In: *Journ. IEEE Design & Test of Comp.* 2007.

[30] Qian Ge et al. “Your processor leaks information — and there’s nothing you can do about it”. In: *arXiv: Cryptography and Security*. 2016.

[31] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. “A Tool for Estimating Information Leakage”. In: *Proc of. Comp. Aid. Verific.* 2013.

[32] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell Sys. Tech. Jour.* 1948.

[33] Jonathan K. Millen. “Covert Channel Capacity”. In: *Proc. of S&P*. 1987.

[34] Rohatgi Pankaj Chari Suresh Rao Josyula R. “Template Attacks”. In: *Proc. of CHES*. 2002.

[35] Billy Bob Brumley and Risto M. Hakala. “Cache-Timing Template Attacks”. In: *Proc. of ASIACRYPT*. 2009.

[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *Proc. of USENIX Security*. 2015.

[37] Michael Schwarz, Florian Lackner, and Daniel Gruss. “JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits”. In: *Proc. of NDSS*. 2019.

[38] Matteo Busi et al. “Securing Interruptible Enclaved Execution on Small Microprocessors”. In: *Journ. ACM Trans. Program. Lang. Syst.* 2021.

[39] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side

Channels for Untrusted Operating Systems". In: *Proc. of S&P*. 2015.

[40] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *Proc. of USENIX ATC*. 2017.

[41] Zili Kou et al. "Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict". In: *Proc. of DAC*. 2021.

[42] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *Proc. of USENIX Security*. 2017.

[43] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks". In: *Proc. of ACSAC*. 2017.

[44] Texas Instruments. *Low-Power Hex Keypad Using MSP430™ MCUs*. Tech. rep. Texas Instruments.

[45] Texas Instruments. *BOOST-IR Infrared (IR) BoosterPack™ Plug-in Module*. Tech. rep. Texas Instruments.

[46] Texas Instruments. *USB Keyboard Using MSP430™ Microcontrollers*. Tech. rep. Texas Instruments.

[47] QMK. *QMK Firmware*. https://qmk.fm/.

[48] TMK. *tmk_keyboard*. https://github.com/tmk/tmk_keyboard.

[49] STMicroelectronics. *RM0438 Reference manual: STM32L552xx and STM32L562xx advanced Arm®-based 32-bit MCUs*. Tech. rep. STMicroelectronics, Dec. 2020.

[50] Microchip. *SAM L10/L11 Family Data Sheet*. Tech. rep. Microchip, June 2020.

[51] ProvenRun. *ProvenCore-M: A secure Real Time Operating System*. https://provenrun.com/products/provencore-m/. Accessed: 2023-02-09.

[52] Taras A. Drozdovskyi and Oleksandr S. Moliavko. "mTower: Trusted Execution Environment for MCU-based devices". In: *Journ. Open Sour. Soft.* 2019.

[53] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. "uTango: An Open-Source TEE for IoT Devices". In: *Journ. IEEE Access*. 2022.

[54] Arm. *Arm® Platform Security Architecture Trusted Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M 2.0*. Tech. rep. Dec. 2019.

[55] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. "Hypnos: An Ultra-Low Power Sleep Mode with SRAM Data Retention for Embedded Microcontrollers". In: *Proc. of ACM CODES*. 2014.

[56] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks". In: *Proc. of EuroS&P*. 2021.

[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proc. of ACM SysTEX*. 2017.

[58] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proc, of CRYPTO*. 1999.

[59] T.S. Messerges, E.A. Dabbish, and R.H. Sloan. "Examining smart-card security under the threat of power analysis attacks". In: 2002.

[60] Louis Goubin. "A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems". In: *Proc. of PKC*. 2002.

[61] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *Proc. of S&P*. 2021.

[62] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Proc. of CRYPTO*. 2014.

[63] Li Zhuang, Feng Zhou, and J. D. Tygar. "Keyboard Acoustic Emanations Revisited". In: *Journ. ACM Trans. Inf. Syst. Secur.* 2009.

[64] Daniel Genkin et al. "Synesthesia: Detecting Screen Content via Remote Acoustic Side Channels". In: *Proc. of S&P*. 2018.

[65] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards". In: *Proc. of E-SMART*. 2001.

[66] Daniel Genkin et al. "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels". In: *Proc. of CCS*. 2016.

[67] Giovanni Camurati et al. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers". In: *Proc. of CCS*. 2018.

[68] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Proc. of CHES*. 2001.

[69] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Proc. of CRYPTO*. 1996.

[70] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *Proc. of CHES*. 2016.

[71] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks and Their Mitigations". In: *Journ. ACM Comput. Surv.* 2022.

[72] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Proc. of CT-RSA*. 2006.

[73] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proc. of USENIX Security*. 2014.

[74] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Proc. of DIMVA*. 2016.

[75] Colin Percival. "Cache Missing for Fun and Profit". In: *Proc. of BSDCan*. 2005.

[76] F. Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *Proc. of S&P*. 2015.

[77] Moritz Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". In: *Proc. of USENIX Security*. 2016.

[78] Ning Zhang et al. “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices”. In: *IACR Cryptol. ePrint Arch.* 2016.
[79] Ben Gras et al. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *Proc. of USENIX Security*. 2018.
[80] Jo Van Bulck et al. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *Proc. of USENIX Security*. 2017.
[81] Ben Gras et al. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *Proc. of NDSS*. 2017.
[82] Stephan van Schaik et al. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think”. In: *Proc. of USENIX Security*. 2018.
[83] Jakob Koschel et al. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs”. In: *Proc. of EuroS&P*. 2020.
[84] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Understanding and Mitigating Covert Channels Through Branch Predictors”. In: *Journ. ACM Trans. Archit. Code Optim.* 2016.
[85] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “Predicting Secret Keys Via Branch Prediction”. In: *Proc. of CT-RSA*. 2007.
[86] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *Proc. of ASIACCS*. 2007.
[87] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *Proc. of IEEE/ACM MICRO*. 2016.
[88] Dmitry Evtyushkin et al. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *Proc. of ASPLOS*. 2018.
[89] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *Proc. of USENIX Security*. 2021.
[90] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack against Constant-Time Crypto Implementations”. In: *Journ. CoRR*. 2017.
[91] Ben Gras et al. “ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures.” In: *Proc. of NDSS*. 2020.
[92] Alejandro Cabrera Aldaya et al. “Port Contention for Fun and Profit”. In: *Proc. of S&P*. 2019.
[93] Atri Bhattacharyya et al. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: *Proc. of ACM CCS*. 2019.
[94] Junpeng Wan et al. “MeshUp: Stateless Cache Side-channel Attack on CPU Mesh”. In: *Proc. of S&P*. 2022.
[95] M. Bognar, J. Van Bulck, and F. Piessens. “Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures”. In: *Proc. of S&P*. 2022.
[96] Zhenyu Wu, Zhang Xu, and Haining Wang. “Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud”. In: *Proc. of USENIX Security*. 2012.
[97] Colin O’Flynn and Alex Dewar. “On-Device Power Analysis Across Hardware Security Domains.: Stop Hitting Yourself.” In: *Journ. IACR Trans. on Crypt. Hard. and Embed. Syst.* 2019.
[98] Alessandro Barenghi et al. “Exploring Cortex-M Microarchitectural Side Channel Information Leakage”. In: *Journ. IEEE Access*. 2021.
[99] Dennis R. E. Gnad, Jonas Krautter, and Mehdi B. Tahoori. “Leaky Noise: New Side-Channel Attack Vectors in Mixed-Signal IoT Devices”. In: *Journ. IACR Trans. on Crypto. Hard. and Embed. Syst.* 2019.

Appendix A. Victim Application Vulnerable Instructions

We compiled the target victim application for Cortex-M33 using different compilation optimizations (O0, O1, and O2). In all outputted binaries, we observed the existence of at least one conditional branch (e.g., `bne` or `beq`) that leaks the secret. All compilation options resulted in code vulnerable to our side-channel attack. Below, we present code snippets with the assembly instructions generated by each compilation for the vulnerable if-else statement in the `read_keypad` function (see Figure 6). We used the `arm-none-eabi-gcc` compiler, `v9.3.1`, with the compilation flags `-mcpu=cortex-m33` and `-Ox` (x = 0, 1, 2).

A.1. Victim Application Compiled With O0

Listing 1 presents the code snippet with the assembly instructions for the vulnerable `read_keypad` function, i.e., the if-else statement, compiled with optimization `-O0`. The `beq .L5` instruction (line 16) represents the if-else statement which unintentionally leaks the secret. Similar to the code snippet shown in Figure 2, when the branch is taken (to execute code at label `.L5`, i.e., the else path), there is a delay of two clock cycles, allowing to distinguish from the if path, i.e., when the `beq .L5` is not taken, and execution continues at label line 17.

```
1  read_keypad:
2  (...)
3  .L7:
4          ldr r2, [r7, #4]
5          ldr r3, [r7, #12]
6          ands r2, r2, r3
7          ldr r3, .L9
8          ldr r1, [r3]
9          ldr r3, [r7, #12]
10         ands r3, r3, r1
11         mvns r3, r3
12         ands r3, r3, r2
13         str r3, [r7]
```

```
14        ldr r3, [r7]
15        cmp r3, #0
16        beq .L5 /*Instruction That Leaks*/
17        ldr r3, .L9+4
18        ldr r3, [r3]
19        adds r2, r3, #1
20        ldr r1, .L9+4
21        str r2, [r1]
22        ldr r2, [r7, #8]
23        uxtb r1, r2
24        ldr r2, .L9+8
25        strb r1, [r2, r3]
26        b .L6
27  (...)
```

Listing 1: Vulnerable `read_keypad` if-else statement compiled with `-O0`.

A.2. Victim Application Compiled With O1

Listing 2 presents the code snippet with the assembly instructions for the vulnerable `read_keypad` function, i.e., the if-else statement, compiled with optimization `-O1`. The `beq .L3` instruction (line 7) represents the if-else statement which unintentionally leaks the secret. Similar to the code snippet shown in Figure 2, when the branch is taken (to execute code at label `.L3`, i.e., the else path), there is a delay of two clock cycles, making it clearly distinguishable from the if path, i.e., when the `beq .L3` is not taken, and execution continues at label line 8.

```
1  read_keypad:
2  (...)
3  .L5:
4         and r2, r5, r1
5         uxth r2, r2
6         cmp r2, #0
7         beq .L3 /*Instruction That Leaks*/
8         strb r3, [lr, r0]
9         adds r0, r0, #1
10        mov r7, ip
11        b .L4
12 (...)
```

Listing 2: Vulnerable `read_keypad` if-else statement compiled with `-O1`.

A.3. Victim Application Compiled With O2

Listing 3 presents the code snippet with the assembly instructions for the vulnerable `read_keypad` function, i.e., the if-else statement, compiled with optimization `-O2`. The `bne .L12` instruction (line 7) represents the if-else statement which unintentionally leaks the secret. Similar to the code snippet shown in Figure 2, when the branch is taken (to execute code at label `.L12`, i.e., the if path), there is a delay of two clock cycles, making it clearly distinguishable from the else path, i.e., when the `bne .L12` is not taken, and execution continues at label line 8.

```
1  read_keypad:
2  (...)
3  .L6:
4         and r2, r5, r1
5         uxth r2, r2
6         cmp r2, #0
7         bne .L12 /*Instruction That Leaks*/
8         strb r3, [ip, r4]
9         adds r3, r3, #1
10        cmp r3, #16
11        mov r4, #0
12        lsl r1, r1, #1
13        bne .L6
14 (...)
```

Listing 3: Vulnerable `read_keypad` if-else statement compiled with `-O2`.

Appendix B. Meta-Review

B.1. Summary

This paper presents BUSted, a new side channel vector to bypass memory protection by exploiting the bus contention as a timing side channel. Measuring the bus state in a single-core CPU is the major technical challenge, addressed by using hardware gadgets composed of peripheral devices (e.g., DMA, timer) that work in parallel to the CPU. BUSted is demonstrated by attacking a target program protected by TrustZone-M on the two state-of-the-art ARMv8-M MCU platforms. The target program implements a keypad device for entering a secret 4-digit PIN, and BUSted launched from the non-secure world successfully stole the PIN.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field;
- Independent Confirmation of Important Results with Limited Prior Research;
- Identifies an Impactful Vulnerability.

B.3. Reasons for Acceptance

1) Conducted successful attack on two TEE platforms;
2) The adversarial model focuses on attacks that have a single shot at stealing the secret (a challenging threat model);
3) Paper discusses potential mitigations.

B.4. Noteworthy Concerns

- As acknowledged and discussed as a limitation in the paper, the attack is evaluated only with a particular implementation of a simple keypad application.