White Paper
Software-based Microarchitectural Timing Side-Channels Attacks on TrustZone-M MCUs

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

Abstract. This white paper reports a novel software-based timing side-channel attack that leverages a key microarchitectural structure pervasive in all microcontrollers (MCUs), i.e., the bus interconnect. By mounting this attack, we made it possible to bypass hardware-enforced isolation mechanisms available on modern MCUs, i.e., Trustzone-M world isolation. Leveraging a non-privileged malicious application in the non-secure world, we can leak information from a trusted application (TA) and steal secrets from the secure world. We provide a proof-of-concept demonstration of our attack emulating a secure smart lock IoT application: the smart lock TA interfaces with a trusted keypad, and the TA runs on top of a reference TEE kernel, Trusted Firmware-M (TF-M). We demonstrate this attack on a real hardware platform powered by an Armv8-M MCU (TrustZone-M), i.e., the NUCLEO-L552ZE-Q , and we can recover the PIN entered by the user in real-time.

1 Root Cause: Interconnect Arbitration

In an MCU, there are several bus entities that need to be interconnected, such as the CPU, memory, and peripherals. Usually, it is done through a central interconnect, also known as a bus matrix, that accordingly to the address broadcasted by a BM, creates a communication channel between the master and slave port. An interconnect is able to perform several concurrent non-blocking full-bandwidth transfers between several BMs and bus slaves, as long as the target destination is different among all transfers. However, when two data transfers target the same component, by following a specific arbitration policy, the interconnect decides in which order the BMs will access the target peripheral. There are two possible arbitration policies, i.e., *priority-based* or *round-robin*. In priority-based arbitration, access to the slave is granted to the master with the highest priority. In round-robin arbitration, slave access is fairly multiplexed in time between competing masters. Our attack relies on one simple key observation: when two BMs want to access the same target component, the access of one BM has to be delayed (both in priority-based and round-robin policies). By observing the presence or absence of a delay in its own access to the target slave (e.g., memory bank), a BM can determine whether or not another BM accessed the slave during the same time slot.

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 domains, while the DMA is exclusively assigned to the spy domain. Both the CPU (BM 1) and DMA (BM 2) operate (i.e., read, write) over the same

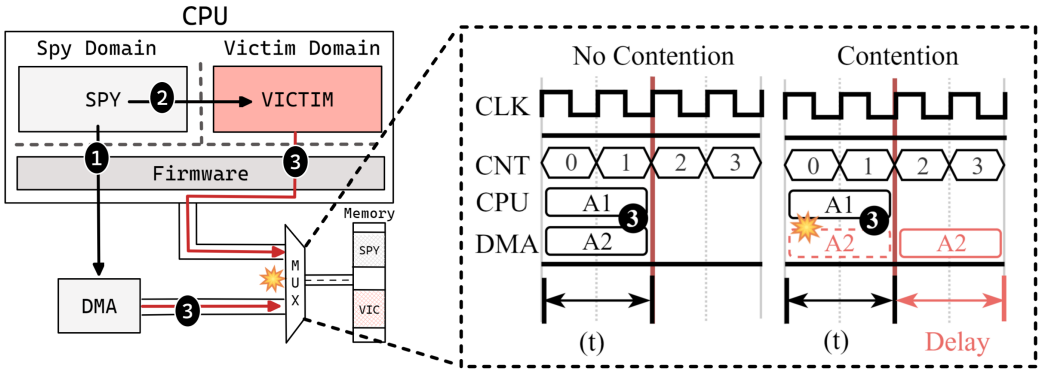


Figure 1: Two bus masters, i.e., CPU and DMA, simultaneously accessing a shared memory bank.

memory bank, but in separated memory spaces. Although the memory spaces are fully isolated, the bus interconnect linking BM 1 and BM 2 to the memory bank is shared. By observing the temporal changes caused by the matrix arbitration in their memory transactions the BMs can extract information about each other's memory accesses. Without breaking physical security isolation boundaries, a malicious BM can spy on bus activity and determine when a specific component was accessed by another BM. As long as the bus slave is shared between BMs, memory protections and security states cannot prevent this type of arbitration-related information leakage.

2 Threat Model and Assumptions

An attacker has the goal to subvert the system hardware isolation and bypass security barriers to get sensitive information from an otherwise isolated domain. In a side-channel attack, this secret is derived from unintended information leakage which is the result of the victim's interaction with the system.

We assume that:

1. The attacker [1], during the attack build-up phase, has access to the victim's code;
2. The attacker has full control over an isolated domain and its resources;
3. During the exploitation phase, the spy is able to invoke the victim code.

The attack exploits a microarchitectural timing channel on the bus interconnect. To exploit this channel, there are a minimal set of assumptions:

1. A shared memory bank between the spy (the attacker's code) and the victim;
2. A secret-dependent control flow in the victim code;
3. A set of peripherals and a bus master (BM) that can be controlled by the spy.

[1]Throughout the paper, we will refer to the person who mounts the side-channel attack as the attacker, and the code that the attacker develops to perform the attack as the spy.

The shared memory bank will create visible contention when the spy and victim try to access it simultaneously. The secret-dependent control flow structures will create unbalanced memory accesses and visible timing differences between execution paths. The bus master, i.e., the direct memory access (DMA), will be used to generate contention and the peripherals (e.g., timers) will be used to detect it.

3 Vulnerability Overview: a Toy Example

The spy can leverage this arbitration-related leakage to obtain the victim's access pattern to the shared memory. First, the spy triggers the DMA to do memory accesses ❶ and then invokes the victim ❷. Leveraging any external wall-clock, the spy can measure the BM transfer latency to detect if the transaction lasts more than expected, hence if there was contention or not. As depicted in Figure 1, when there is contention, DMA access (A2) takes $t + delay$ instead of t ❸.

Using this attack method, a spy can detect whether a specific memory was accessed at a given time. The spy takes advantage of this information leak to detect secret-dependent differences in the victim's access to the monitored memory. A spy can observe those differences by leveraging the MCUs load-store architecture which have two types of instructions: (i) instructions that interface with memory, i.e., loads and stores; and (ii) ALU operations. A spy can detect when the victim executes loads and stores and infer its execution pattern, as these are the only instructions that access data memory. If the victim code has a secret-dependent control flow, i.e., loads or stores are executed at different clock offsets between execution paths, it will produce different memory access patterns; the spy may leverage it to infer and steal a particular secret.

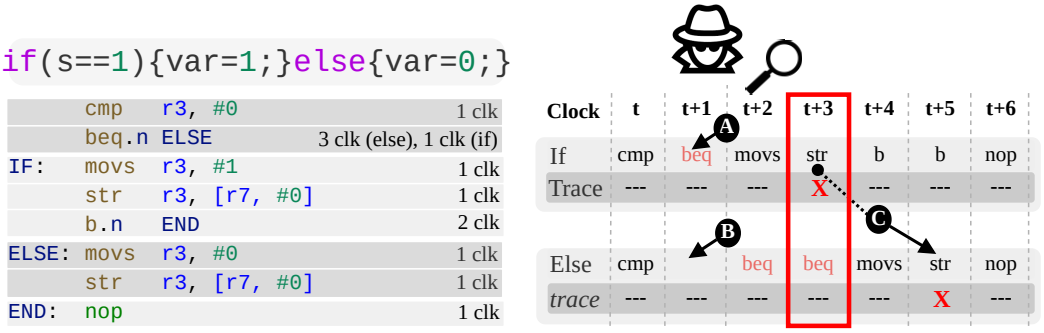


Figure 2: Left, balanced If-else statement compiled for Arm Cortex-M33 (-O0). Right, memory access pattern and monitoring of clock $t + 3$.

Figure 2 depicts an example of how a spy can leverage the load-store architecture as well as bus contention to spy on a victim and steal a secret. Although, both execution paths take the same time to execute, i.e., 6 clocks, the accesses to memories are done in different clock cycles ($t + 3$ and $t + 5$). In this case, the difference in execution between the two paths is due to the branch instruction. If the branch (`beq` instruction) is not taken, it takes only 1 clock cycle Ⓐ, but if it is taken, it takes 3 clock cycles Ⓑ. This changes the relative position of the `str` instruction Ⓒ to the `beq` instruction and reveals the secret. When the secret is 1, the `str` occurs in clock cycle $t + 3$, otherwise, it occurs in clock cycle $t + 5$. A spy monitoring either of these two clock cycles can derive the secret by observing whether or not there is contention on the data memory bus.

4 Exploiting the Toy Vulnerability

In this section, we assume that the previous code is executing in a separated and isolated domain and targets a single-core MCU. The spy can invoke the code to be executed, however, the code is instantiated in another domain. The spy's goal is to retrieve if the secret code is 1 (s==1).

4.1 Challenges

To carry out the attack, there are three main requirements that must be met.

(R1): The spy and the victim must share the very same memory bank;

(R2): The spy must be able to record the shared memory bank access pattern to obtain an 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:

C1: **Spy lacks access to past microarchitecture states.** Unlike other microarchitectural components (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.** Low-end MCUs are typically powered by a single CPU[2], thus concurrent execution of spy and victim code is not possible.

C3: **Victim execution cannot be interrupted.** We assume that the victim isolated domain implements mitigations against Nemesis attacks [6] and has the interrupts disabled.

C4: **Spy only has one chance to steal the secret.** The attack must be performed during a single execution of the victim code, as we are targeting non-cryptographic applications where the secret is processed in a single run.

4.2 Hardware Gadgets

To overcome the aforementioned challenges, we introduce the concept of *hardware gadgets*. These gadgets are interconnected peripherals, such as timers and DMAs, that can execute specific tasks in the background without requiring CPU's intervention. Hardware gadgets allow for some level of concurrent execution between the victim code running on the CPU and the spy logic, addressing **C2**. The hardware gadgets can execute specific tasks (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's code at the clock cycle level. This allows the spy to use these gadgets to monitor the victim's activity (such as access to a shared memory) and execute conditional actions based on the victim's control-flow, addressing **C1** and **C4** respectively.

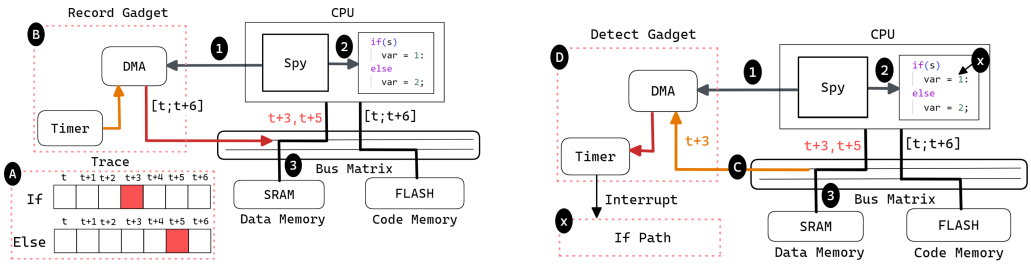
To meet the requirements of the attack, i.e., creating (**R1**), recording (**R2**), and detecting contention (**R3**), we devise two basic hardware gadgets (more advanced gadgets can be seen in Section 6): the record contention gadget and the detect contention gadget. The record contention gadget is 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. Both gadgets are activated by an external

[2]While dual-core configurations can be found in modern microcontrollers, like STM32H7, MUSCA-B1, and i.MX RT1170, single-core MCUs still make up the majority of the market share.

trigger, which can be sent by software or other hardware gadgets. Both gadgets require arbitrary control over a BM, e.g., DMA, and access to a peripheral timer operating at the CPU frequency.

4.3 Attack Phases

Inspired by former template attacks [2, 3, 5], our attack is composed of two phases: (i) a profiling phase; and (ii) an exploitation phase. In the profiling phase, the victim runs in an spy-controlled environment, where several side-channels traces are recorded. 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.



(a) Profiling phase.

(b) Exploitation phase.

Figure 3: Example of the attack phases using the toy example from Figure 2.

In the profiling phase, depicted in Figure 3a, the attacker constructs a template matrix, which has one trace per victim execution path. In the example represented in Figure 3a, the template matrix has two traces (A). One for the `If` path other for the `else` path. These traces are obtained using the record contention gadget (B). The spy triggers a DMA to execute (1) (i.e., read a timer) and invokes the victim code (2). This is repeated several times until the spy has generated contention in all clocks cycles. For the toy example, the code accesses the SRAM (3) in clock cycle $t + 3$ when the `if` path is executed, and in clock cycle $t + 5$ when the `else` path is executed.

In the exploitation phase, depicted in Figure 3b, the spy monitors a specific clock cycle and compares it with the template matrix obtained in the profiling phase. From the profiling phase, we know that contention in clock cycle $t+3$ means `if` path was executed and contention in clock cycle $t + 5$ means `else` path was executed. As illustrated in Figure 3b, to get the secret the spy uses the detect contention gadget (D) to monitor one of those clocks (C), e.g, $t + 3$. When there is contention in $t + 3$ (x) it means the $Secret = 1$, and when there is no contention in $t + 3$, it means $Secret \neq 1$.

5 Attacking a "Smart Lock" Application

To demonstrate the effectiveness of the attack in a more realistic setup, we used a smart lock application[3] as our use case, which is in line with state-of-the-art side-channel attacks for MCUs [6]. The smart lock deals with security-critical information (the user's pin) and interfaces with an external secure keypad to detect key presses in real-time. This application serves as a good example of typical low-end applications that operate in real-time, reading, processing, and actuating over asynchronous events.

[3]https://github.com/sancus-tee/vulcan/blob/master/demo/ecu-tcs/sm_tcs_kypd.c

Attacker's Goal

The attacker's goal is to steal the Smart Lock pin entered by the user.

5.1 Smart Lock Application / Setup in a nutshell

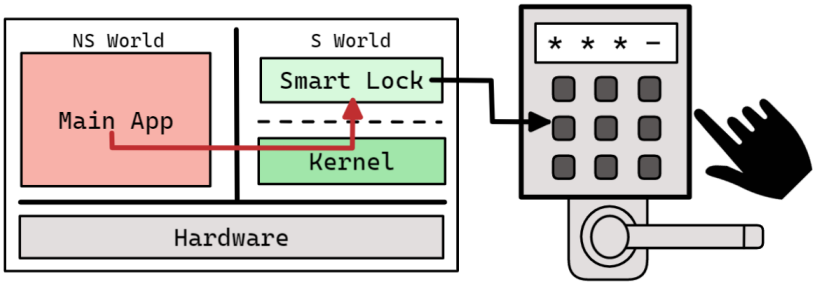


Figure 4: Smart lock application setup.

In TrustZone-based systems, there are two domains: a secure and a non-secure one. As shown in Figure 4, the main application runs in the non-secure domain, while the smart lock trusted application (TA), which processes security-sensitive information, is deployed in the secure domain and interfaces with an external keypad for user input. The smart lock application is triggered by the main application in the non-secure domain, e.g., to authenticate a user, and reads the keypad until all four digits of the pin have been pressed. The smart lock then returns a status message indicating whether the entered pin was valid or not. The code snippet of the Smart Locks `read_keypad` function, is presented in Listing 1.

```
signed int read_keypad(void){
    int is_pressed, mask = 0x1;
    int new_key_state = get_keypad_state();
    for (int key = 0; key < 16; key++)    {
        // detect rising edge
        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; //avoid buffer overflow
        mask <<= 1;
    }
    key_state = new_key_state;
    return (4 - pin_idx);
}

void read_pin(){
  signed int pin_len = 4;
  while(pin_len >0)
    pin_len = read_keypad();
}
```

Listing 1: `Read_Keypad` code based on Nemesis [6] attack use case, which follows a reference implementation of a keypad interface from Texas Instruments [4].

The goal of the spy is to uncover the pressed key. To do this, the spy monitors the victim's control flow and checks whether or not the victim executes the secret-dependent path. The `read_keypad` code continually iterates over all 16 keys in a loop, verifying if the key was pressed and registering it when there is a match. Apparently, this is not vulnerable to typical timing attacks, since the code is carefully balanced for a constant time. However, a closer examination of the `if` statement in assembly (similar to Figure 2) reveals that the code accesses memories in different points in times (different execution paths), revealing the secret-dependent path and thereby the secret (i.e., the pressed key).

With the identification of the vulnerable code, i.e., the `if` statement, the spy only needs to be set to monitor a specific clock cycle (similar to $t + 3$ clock cycle in Figure 2) in each `for` loop iteration. If the spy is able to do that, it will be possible to indirectly understand which key was pressed. For example, if the spy observes that the `if` branch was executed in the 7th iteration of the `for` loop, it instantly realizes that the user pressed key 7.

The task of the spy is straightforward: 1) keep track of the for loop and record the number of iterations it has executed; 2) continuously monitor a specific clock cycle for any signs of contention. When a key is pressed, the spy will observe the resulting contention. To determine which key was pressed, the spy can then look at its internal counter that is keeping track of the `for` loop.

5.2 Smart Gadget Network

To track the code execution and get the victim control flow we need to use a combination of hardware gadgets, what we call a smart gadget network (SGN). SGN is a combination of hardware gadgets interconnected to perform a specific algorithm/logic. To implement the attack, we need two SGN, i.e., one for the profiling phase and other for the exploitation phase.

The profiling SGN is used to trace the victim execution. It needs two hardware gadgets. One gadget to generate and record contention, i.e., the *record contention gadget*, and another to instruct the record contention gadget to execute and create contention in a specific clock cycle, *trigger contention gadget*.

The exploitation SGN is used to track the smart lock code 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 be able to do that, the exploitation SGN needs 5 hardware gadgets:

(G1): The *detection gadget*, to generate and automatically detect contention;

(G2): The *trigger contention gadget* (same as the profiling SGN), to trigger the detection gadget, and generate contention in arbitrary clock cycles;

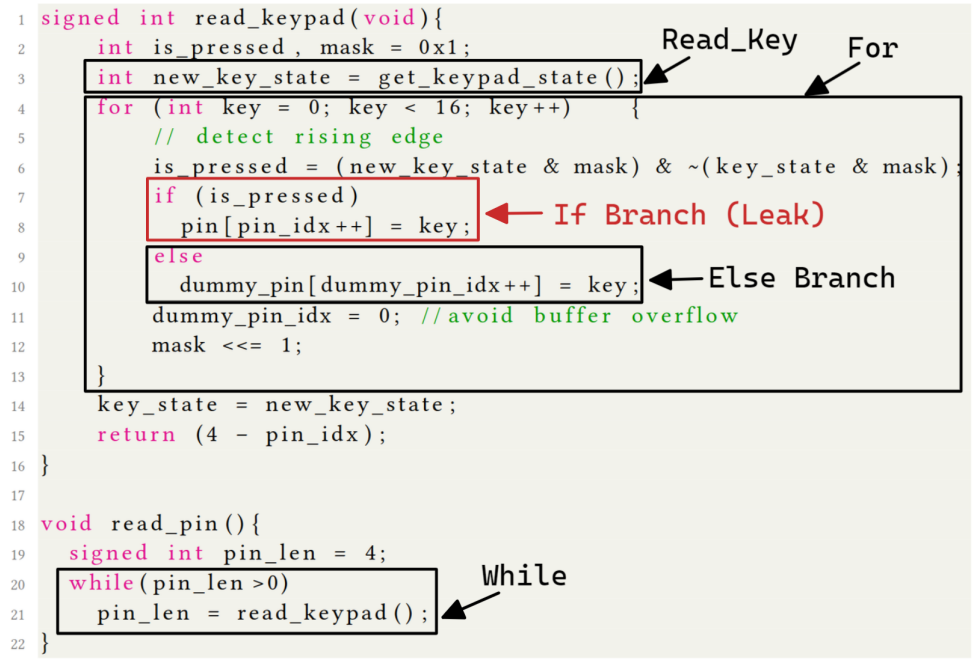
(G3): The *counter gadget*, to track the iteration index of the `read_keypad` `for` loop;

(G4): The *read secret gadget*, to automatically read the counter gadget when there is contention (i.e., the secret);

(G5): The *auto-sync gadget*, to keep the exploitation SGN in synchronization at the clock cycle with the victim code.

5.3 Attack Overview

To better understand the `read_keypad`, we split the code into 5 parts, as shown in Figure 5. There are two potential execution paths, depending on whether a key is pressed or not. Both paths include three common parts: (i) the `while` loop, (ii) the `read_key` function, and (iii) the `for` loop.




```
signed int read_keypad(void){
    int is_pressed, mask = 0x1;
    int new_key_state = get_keypad_state();
    for (int key = 0; key < 16; key++)    {
        // detect rising edge
        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; //avoid buffer overflow
        mask <<= 1;
    }
    key_state = new_key_state;
    return (4 - pin_idx);
}

void read_pin(){
  signed int pin_len = 4;
  while(pin_len>0)
    pin_len = read_keypad();
}
```

Figure 5: Key parts of the `read_keypad` function highlighted.

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, the `else branch` is executed. In the *profiling* phase, the attacker uses the *profiling* SGN to capture the memory access patterns. During the actual attack, the attacker uses the *exploitation* SGN to retrieve the secret by using the memory access traces obtained in the *profiling* phase.

5.3.1 Offline Profiling Phase

During the profiling phase, the spy has complete control over the victim code, which is executed within the spy's domain, i.e., the non-secure world. To obtain the victim's execution profile (please refer to traces Ⓐ and Ⓑ from Figure 6), the spy creates contention in every clock cycle of the victim code. This is done by executing the victim multiple times and in each execution, incrementing the clock cycle where the contention is created. To profile each execution path, the spy forces the victim to only execute one path per run by sending a constant input. This can be done either by pressing the same key repeatedly (to trace path Ⓐ) or by not pressing any key at all (to trace path Ⓑ).

The profiling is carried out in three steps:

1. First, the spy code executes in a `while` loop, incrementing a clock variable for each victim execution until the final clock cycle is reached. During each iteration, the code triggers the Profiling SGN to generate contention at a specific clock cycle ⓐ.

2. Then the spy invokes the victim ⓑ;

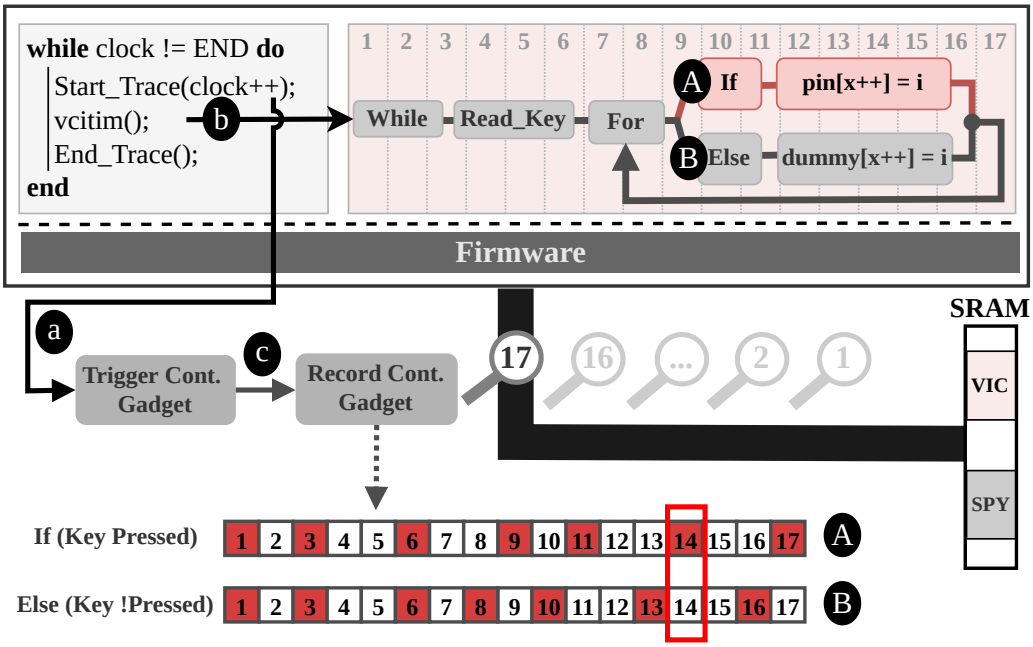


Figure 6: Profiling SGN.

3. Finally, the trigger gadget ⓒ, after a predetermined amount of time has passed, initiates the record gadget to create and record contention.

Step 3 generates two execution patterns: one when a key is pressed and the `if` path is executed Ⓐ, and another when there is no keypress and the `else` path is executed Ⓑ. The profiling is repeated for each possible execution path of the `read_keypad` code, of which there are 17 in total, one for each possible key (16 possibilities) plus one additional execution path for when there is no key press.

5.3.2 Online Exploitation Phase

In the exploitation phase, the spy monitors a set of clock cycles to determine the executed path and infer the secret. Spy and the victim run in separate domains, with the spy operating in the non-secure world and the victim in the secure world. To initiate the victim's execution, the spy's exploitation code sends a request to the firmware, which then performs a domain switch ❷ and runs the victim code. This results in an offset in the clock cycles of the profiling phase patterns. To account for this firmware overhead, the spy traces the secure world, including both the firmware and the victim, and searches for the patterns obtained in the profiling phase. This allows the spy to determine the amount of time elapsed from the moment the spy invokes the victim until the first victim instruction is executed. This spy uses this offset to correct the patterns from the profiling phase. For the sake of example, let us consider that the first instruction of the victim code is executed at clock cycle 1000 and that the spy selects clock cycle 14 from the traced patterns Ⓐ and Ⓑ. As a result, clock cycle 14 will be shifted to 1014 and the spy will monitor clock 1014 instead of 14.

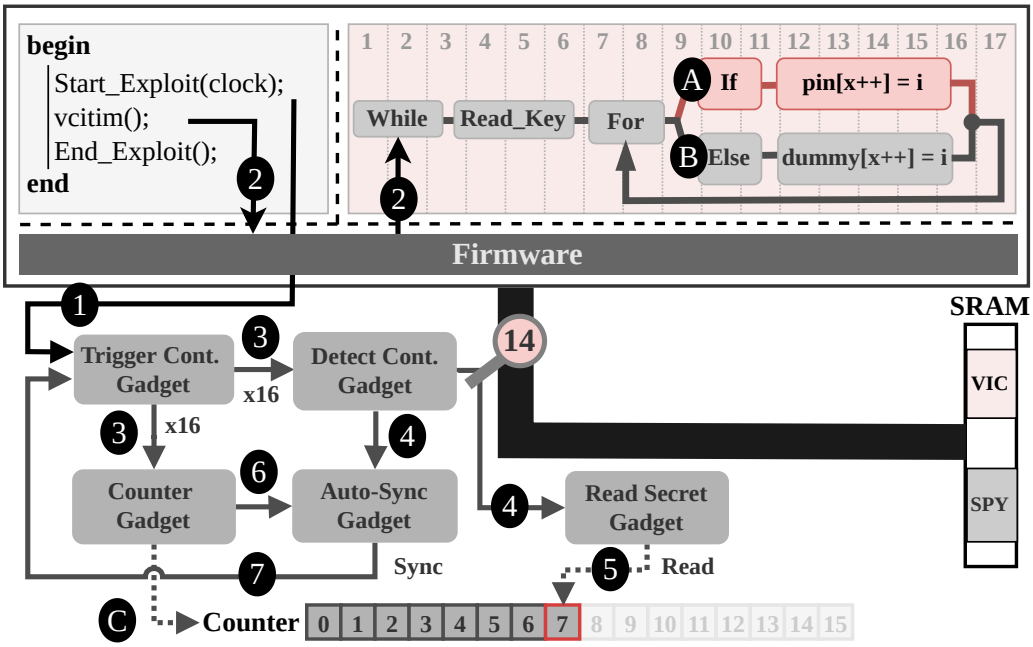


Figure 7: Exploitation SGN.

After the target clock cycle to monitor is selected, the spy launches the attack:

1. The spy triggers ❶ the exploitation SGN to generate contention in the selected clock cycle;
2. The spy invokes the victim through a firmware call ❷;
3. The trigger gadget creates 16 triggers ❸, one for each iteration of the `read_keypad` `for` loop. This causes the detect contention gadget to generate contention at the first clock cycle, and in multiples of that. For example, if the `for` loop takes 50 cycles to execute, then contention will be generated in clock cycles 1014 (`for key=0`), 1014 + 50 (`for key=1`), 1014 + 100 (`for key=2`), and so on;
4. Each trigger generated ❸ 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 ❻;
5. When contention is detected, a signal ❹ 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). For example, if the counter gadget holds a value of 7 and contention is detected, it indicates that the key corresponding to the 7th iteration of the `for` loop was pressed.
6. The auto-sync gadget is activated by either the completion of the `for` loop ❻ or by the detection of contention ❹. Upon activation, it adjusts ❼ timing for the next clock cycle in

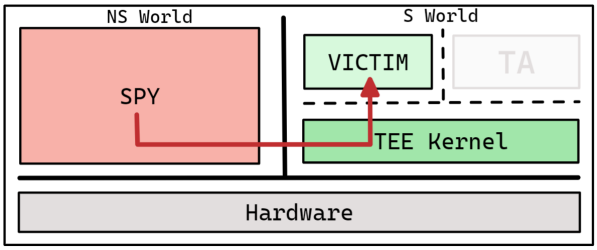


Figure 8: TF-M attack setup.

which contention will occur, thereby re-synchronize the exploitation SGN with the victim code.

6 Attack Demonstration

We mounted the attack against a TEE kernel leveraging the TrustZone-M (Armv8-M), i.e., TF-M [1] . The spy runs in the non-secure world while the victim, i.e., the smart lock, is a TA running on top of a trusted kernel in the secure world, as shown in Figure 8.

We successfully exploit the smart lock TA (bypassing TrustZone-M) on a NUCLEO-L552ZE-Q platform by leveraging the profiling and exploitation SGNs. This platform features an STM32L552 MCU powered by an Arm Cortex-M33, implementing the Harvard architecture, i.e., instruction and data fetch occur on separate buses. The bus interconnect uses a round-robin arbitration policy. NUCLEO-L552ZE-Q has a total of 72 peripherals, including 14 timers, 2 DMAs with 8 channels each, and a 16-channel DMA Multiplexer, which enables direct communication between peripherals and the DMA channels. The profiling and exploitation SGNs require 4 out of the 14 timers, 5 out of the 16 DMA channels, and 5 out of the 16 DMA Multiplexer channels. In total, only 6 (i.e., 4 timers, 1 DMA, and 1 DMA Multiplexer) out of the 72 available peripherals were used in this exploit, which represents 8.3% of the available peripherals.

7 Generalization to other MCUs

In this white paper, we demonstrate the viability and practicality of the attack in the STM32L552 MCU. Notwithstanding, given the widespread availability of the leveraged microarchitectural source, i.e., the bus interconnect, in Armv6-M, Armv7-M, and Armv8-M (and even other ISAs) MCUs, we strongly believe that it is possible to mount similar variants of the attack in a large number of platforms, including Armv7-M MCUs without TrustZone but with memory protection unit (MPU).

To have a feeling about the real extent of the vulnerability, we mounted a simple covert channel (to transmit 8 bits of information) on three additional ST MCUs: the STM32L0 (Cortex-M0+), the STM32L4 (Cortex-M4), and the STM32F7 (Cortex-M7). As shown in Figure 9, all of the evaluated boards exhibit a clear, 45-degree, noise-free channel, indicating that they may be vulnerable to the same microarchitectural timing side-channel attack.

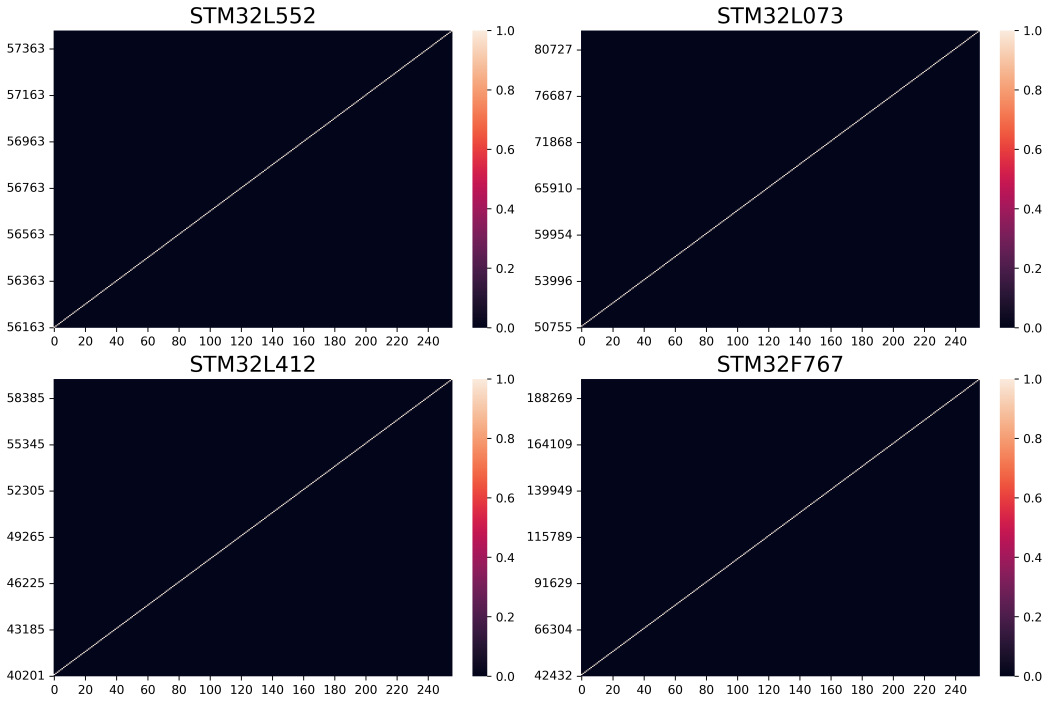


Figure 9: Raw MCUs' channel matrix, measured in a baremetal setting.

8 Security Incident Process

8.1 Timeline

- The microarchitectural channel was preliminary observed on the STM32L552 in February 2022.
- The microarchitectural channel was fully validated on the STM32L552 in April 2022.
- We successfully mounted the Toy example attack on the STM32L552 in June 2022.
- We successfully mounted the Smart Lock use case (with TF-M [1]) attack in October 2022.
- The microarchitectural channel was fully validated on three additional ST MCUs (STM32L073, STM32L412, STM32F767) in November 2022.
- We completed the White Paper and initiated responsible disclosure of the vulnerability in January 2023.
- We submitted an academic paper with an in-depth description of the attack to the IEEE Security & Privacy Oakland (IEEE S&P) in April 2023.
- We disclosed the attack at Black Hat Asia on May 12th, 2023[4].

[4]https://www.blackhat.com/asia-23/briefings/schedule/hand-me-your-secret-mcu-microarchitectural-timing-attacks-on-microcontrollers-are-practical-30579

References

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

[2] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks." In: *Advances in Cryptology – ASIACRYPT 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 667–684. ISBN: 978-3-642-10366-7.

[3] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912. ISBN: 978-1-939133-11-3.

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

[5] Michael Schwarz, Florian Lackner, and Daniel Gruss. "JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits." In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019).

[6] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 178–195. ISBN: 9781450356930.