

Software-based Microarchitectural Attacks on Low-End IoT Devices

Cristiano Rodrigues

UMinho | 2026

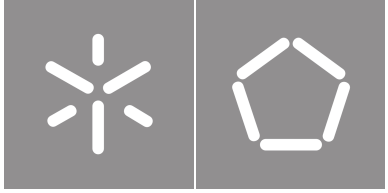


Universidade do Minho
Escola de Engenharia

Cristiano António Azevedo Rodrigues

Software-based Microarchitectural Attacks on Low-End IoT Devices

Abril de 2026



Universidade do Minho
Escola de Engenharia

Cristiano António Azevedo Rodrigues

Software-based Microarchitectural Attacks on Low-End IoT Devices

Tese de Doutoramento
Programa Doutoral em Engenharia Eletrónica e Computadores

Trabalho efetuado sob a orientação de:
Investigador Principal Doutor Sandro Emanuel Salgado Pinto
Professor Doutor Adriano José Conceição Tavares

Abril de 2026

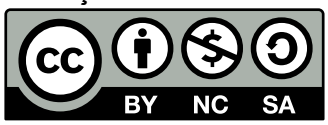
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

`https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en`

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.2) .

Acknowledgements

This thesis would not have been possible without the support and contributions of many people, to whom I owe a great deal of gratitude. As the saying goes, "*if you want to go fast, go alone; if you want to go far, go together.*" This sentiment resonates deeply with me. The roughly five years of my PhD have been an emotional rollercoaster, spanning COVID-19 lockdown periods and the challenges of living abroad away from my loved ones, to the great privilege of discovering a little more of the world. Along the way, I traveled to new places, experienced new cultures, and presented my/our work at leading academic venues. All of this was possible because I was surrounded by remarkable people who enabled me to go far beyond what I ever thought I was capable of. Had I chosen to navigate this path alone, none of this would have been possible. While this thesis was written by me, it was made possible by all of you.

First and foremost, I would like to thank my advisors, Sandro Pinto and Adriano Tavares from the University of Minho, Portugal, and Mongkol Ekpanyapong from the Asian Institute of Technology, Thailand, for their guidance, patience, and encouragement throughout this journey. Their insights and dedication have been instrumental in shaping both this thesis and my growth as a researcher.

In particular, I wish to dedicate a separate note of gratitude to my primary advisor Sandro Pinto. Beyond his role as an academic advisor, Sandro has been a mentor, a friend, and one of the people who has had the greatest impact on me throughout this journey. He consistently pushed me beyond what I believed I was capable of, challenging me to operate at the edge of my technical limits and to think beyond conventional boundaries. That relentless drive is, in no small part, responsible for the contributions the reader will encounter throughout this thesis. I also carry with me one of his mottos: "*always aim for the stars, even if you end up landing on the moon, that is still a very good outcome*". For all of this, I am deeply grateful.

This thesis has also greatly benefited from fruitful collaborations that, beyond their research impact, allowed me to meet remarkable people abroad. I would like to thank Renato Mancuso and Shahin Roozkhosh from Boston University for making my exchange period such a rewarding experience. Shahin, thank you for the many deep and thought-provoking conversations, and for the exceptional writing and communication skills I have been fortunate to learn from you. Renato, thank you for your relentless technical guidance, your contagious curiosity, and for making me feel at home in your lab (BU Cyber-Physical Systems Lab).

I was also fortunate to meet and work alongside a group of researchers whose work has been a source of inspiration since the very beginning of this thesis. Thank you, Jo Van Bulck and Márton Bognár from KU Leuven, for your invaluable contributions and the many stimulating technical discussions we shared. To you, I wish to borrow Newton's words: "*If I have seen further, it is by standing on the shoulders of giants*". Working alongside such talented researchers has been an immense privilege. More than admiring you as outstanding researchers, I admire you for being genuinely wonderful people. At the start of my PhD, I never imagined that, a few years later, I would be sharing beers at cozy Belgian pubs with the very people whose work I had long looked up to.

I would also like to extend my gratitude to my fellow lab mates, Ivo Marques, Afonso Santos, Diogo Costa, and Ricardo Roriz, among others whom space does not allow me to list here, whose camaraderie and support made this experience far more enjoyable. In particular, Daniel Oliveira, a big thank you for your patience with me at the start of my PhD. I know that onboarding me and shaping my academic writing was, at times, a non-trivial task, but without your effort, it would have taken me much longer to reach the technical and writing quality I have today. Thank you for all the knowledge you passed on to me.

To my friends, "malta da pesada", "bando dos 5", "segunda parte", "rota dos tascos", all fellow music seekers, and kickboxing mates who have shared this journey during my time off with great moments of laughter, adventure, and a much-needed escape from the academic grind. Thank you for keeping me grounded and reminding me that there is life beyond the lab. Without you, it would have been a much harder journey.

To my family, who have carried me through this period with unwavering support: I have not always been as present as I would have wished, yet you have remained a constant source of strength. I am deeply grateful for your patience and understanding.

Last but not least, a heartfelt thank you to you, my beloved Gabi. While you joined this journey in its later stages, your encouragement, drive, and support through one of the most demanding phases, the writing of this thesis, have been invaluable. You have provided me with unconditional support and pushed me to be the best version of myself, and for that, I am truly grateful. I look forward to keep growing, embracing new challenges, and exploring the world with you. Love you!

To all of you who have been part of this journey and made it a more enjoyable and rewarding experience, a sincere "Obrigado / Thank You / Khop khun khrap".

Cristiano Rodrigues
April 2026, Guimarães, Portugal.

This work was supported by FCT, Fundação para a Ciência e Tecnologia, which provided financial support throughout this thesis (grant 2020.08729.BD), and by FLAD, Fundação Luso-Americana para o Desenvolvimento, which provided complementary financial support for my exchange period at Boston University (grant Proj. 2023/0221).

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____

(Place) (Date)

_____

(Cristiano António Azevedo Rodrigues)

”

“God loads the gun, you pull the trigger.”

— Author’s adaptation of Dr. George Bray

Resumo

Ataques Microarquiteturais Baseados em Software em Dispositivos IoT

Os *Trusted Execution Environments* (TEEs) permitem que enclaves executem operações de forma confidencial, garantindo isolamento por *hardware* mesmo que o sistema operativo seja comprometido. Apesar destas garantias, a literatura tem demonstrado que, em plataformas de alto desempenho, como Intel SGX, AMD SEV e Arm TrustZone-A, os TEEs permanecem vulneráveis a ataques microarquiteturais. Os TEEs têm vindo igualmente a consolidar a sua presença em dispositivos com recursos limitados alimentados por microcontroladores (MCUs), com a Arm TrustZone-M a emergir como a solução de referência. No entanto, os esforços de investigação permanecem assimétricos: enquanto os TEEs de alto desempenho foram exaustivamente analisados, a investigação em MCUs continua bastante limitada.

Nesta tese, demonstramos que, embora os MCUs possuam microarquiteturas mais simples, as suas características permitem a exploração de canais microarquiteturais específicos destes dispositivos. Embora estas plataformas sejam frequentemente consideradas intrinsecamente seguras devido à sua simplicidade, demonstramos que essas mesmas características introduzem novas superfícies de ataque. Em particular, os MCUs apresentam uma execução altamente determinística, o que facilita a recolha de padrões de informação microarquitetural com baixo ruído. Adicionalmente, os MCUs disponibilizam inúmeros periféricos concebidos para aliviar o processador de tarefas computacionalmente intensivas; estes periféricos podem ser explorados por um atacante para contornar limitações arquiteturais, nomeadamente a ausência de execução concorrente e de memória virtual.

Em suma, esta tese avança o estado da arte em ataques microarquiteturais, demonstrando que estes afetam todo o espetro de MCUs, apesar da sua microarquitetura mais simplista. Concretamente, apresentamos três contribuições principais: *(i)* revelamos o BUSted, um canal microarquitetural não convencional específico para MCUs, demonstrando como pode ser explorado para comprometer o isolamento da tecnologia TrustZone-M; *(ii)* desenvolvemos o M-Step, a primeira *framework* de *single-stepping*, permitindo a recolha de padrões microarquiteturais com elevada resolução; e *(iii)* apresentamos um ataque *end-to-end* que permite extrair uma chave privada RSA de uma implementação protegida pela TrustZone-M a partir de um único trace. Esta tese desafia a premissa de que os MCUs são imunes a ataques microarquiteturais, evidenciando o seu potencial para atacar dispositivos baseados em MCUs da Arm.

Palavras-chave: Ataques Colaterais por *Software*, Ataques Microarquiteturais, MCU, TrustZone-M.

Abstract

Software-based Microarchitectural Attacks on Low-End IoT Devices

Trusted Execution Environments (TEEs) provide strong confidential computing guarantees for hardware-enforced enclaves, ensuring isolation even in the presence of a compromised operating system. Despite these guarantees, extensive research on high-end platforms, such as Intel SGX, AMD SEV, and Arm TrustZone-A, has demonstrated that TEEs remain vulnerable to software-based microarchitectural side-channel attacks. TEEs are also being deployed on resource-constrained devices powered by microcontroller units (MCUs), with Arm TrustZone-M emerging as a leading solution. However, current research efforts remain asymmetrical. While high-end TEEs have been thoroughly analyzed, very little has been done for MCUs that power billions of embedded and Internet of things (IoT) devices.

This thesis addresses this gap. We demonstrate that while MCUs are simpler than their high-end counterparts, their unique characteristics enable the exploitation of specific microarchitectural side-channels. In particular, these devices are optimized for highly deterministic and predictable execution, which significantly eases trace collection and yields a high signal-to-noise ratio for the attacker. Moreover, MCUs feature numerous advanced peripherals designed to offload computation from the main processor. We show that an attacker can leverage these peripherals to overcome inherent MCU limitations, such as the lack of concurrent execution and virtual memory, to mount MCU-specific side-channel exploits.

Ultimately, this thesis advances the state-of-the-art in microarchitectural side-channel attacks by demonstrating pervasive leakage across the MCU spectrum, despite their simplistic microarchitecture. Concretely, we make three main contributions: *(i)* we unveil BUSted, an MCU-specific non-conventional microarchitectural leakage which we leverage to mount the first microarchitectural attack capable of subverting TrustZone-M isolation; *(ii)* we develop M-Step, the first single-stepping framework, which allows an attacker to streamline the gathering of microarchitectural traces on TrustZone-M-enabled MCUs; and *(iii)* we demonstrate the most powerful end-to-end microarchitectural side-channel attack against MCUs to date, capable of extracting a full RSA private key from a TrustZone-M-protected cryptographic library in a single trace. Overall, this thesis challenges the assumption that MCUs are immune to software-based microarchitectural attacks and highlights the critical threat they pose to state-of-the-art Arm devices.

Keywords: MCUs, Microarchitectural Attacks, Software-based Side-Channels, TrustZone-M.

Contents

List of Figures **xiii**

List of Tables **xv**

List of Listings **xvi**

Acronyms **xvii**

1 Introduction **1**

1.1 The Exotic World of Side-Channel Attacks. 2

1.2 MCUs, The Elephant in the Room. 3

1.3 Scope and Threat Model 4

1.4 Thesis Contributions 5

1.4.1 BUSted, Non-Conventional Microarchitectural Leakage on MCUs. 5

1.4.2 M-Step, A Single-Step Framework for Side-Channel Analysis on TrustZone-M 6

1.4.3 End-to-End Attacks on a TrustZone-M-Protected Cryptographic Library. . . . 6

1.4.4 Other contributions 7

1.4.5 Open Science 10

1.4.6 Ethical Considerations 11

1.4.7 List of Publications 11

1.5 Document Outline 12

2 Background **14**

2.1 Computing Spectrum and Computer Architecture 14

2.1.1 The Constraints that Shape the Device. 15

2.1.2 Processing Unit. 16

2.1.3 Memory Subsystem. 19

2.1.4 I/O Interface. 22

2.1.5 Trusted Execution Environments. 22
2.1.6 What is a MCU? 23
2.2 Arm Cortex-M Architecture 24
2.2.1 Arm Cortex-M family. 24
2.2.2 CPU Characteristics. 25
2.2.3 Memory Subsystem. 26
2.2.4 Interrupt Logic. 27
2.2.5 Arm TrustZone-M. 28
2.2.6 Secure Software Stack. 29
2.3 Side-Channel Attacks "101" 30
2.3.1 Side-Channel Attacks 30
2.3.2 The Power of Software-Based Side-Channel Attacks 31
2.3.3 The Root Cause: Sharing is not Caring. 32
2.3.4 Attacks' Phases 33

3 State Of The Art **35**
3.1 Software-based Microarchitectural Attacks on APUs 35
3.1.1 Cache Attacks 35
3.1.2 Controlled-Channel Attacks 38
3.1.3 Transient Execution Attacks 40
3.2 Software-based Microarchitectural Attacks on MCUs 42
3.2.1 Interrupt Latency Microarchitectural Attacks 42
3.2.2 Bus Contention Microarchitectural Attacks 44
3.2.3 Microarchitectural Attacks on Cryptographic Applications 45
3.2.4 Other Software-Based Side-Channel Attacks. 45
3.3 Gap Analysis 46
3.3.1 Thesis Contributions 46

4 Arm Cortex-M Microarchitectural Analysis **48**
4.1 Deep Dive on Cortex-M 49
4.1.1 Arm-based microcontroller units (MCUs) Microarchitectural Landscape . . . 49
4.1.2 Cache 51
4.1.3 Interrupt Logic 51
4.1.4 Bus Interconnect 52
4.2 First Evidence of Microarchitectural Leakage 53
4.2.1 Cache Covert-Channel 54
4.2.2 Interrupt Logic Covert-Channel 56
4.2.3 Bus Interconnect Covert-Channel 57

4.3 Toying with Microarchitectural Leakage . . . 59
4.3.1 Toy Attack Setup . . . 59
4.3.2 Toying with Cache . . . 60
4.3.3 Toying with Interrupt Logic . . . 61
4.3.4 Toying with Bus Interconnect . . . 62
4.4 Discussion . . . 63

5 BUSted: Non-Convention Microarchitectural Leakage on MCUs 65
5.1 Attack Methodology . . . 67
5.1.1 Profiling Phase . . . 68
5.1.2 Exploitation Phase . . . 70
5.2 Attack Building Blocks . . . 71
5.2.1 Challenges . . . 71
5.2.2 Hardware Gadgets . . . 72
5.2.3 Read and Write Stage Contention . . . 73
5.3 BUSted Attack . . . 76
5.3.1 Off-the-Shelf Smart Lock Application . . . 76
5.3.2 Smart Gadget Network . . . 77
5.3.3 Complete Attack Flow . . . 78
5.4 Evaluation . . . 79
5.4.1 TF-M: PSA Isolation Level 2 . . . 80
5.4.2 PSA Isolation Level 3 . . . 80
5.4.3 Platform Constraints on SGNs . . . 81
5.4.4 Real Victim trace . . . 82
5.5 Countermeasures . . . 83
5.6 BUSted Limitations. . . . 84
5.7 Summary . . . 85
5.7.1 Ethical Considerations . . . 86

6 M-Step: Single-Step Framework for TrustZone-M. 87
6.1 Problem Statement . . . 88
6.1.1 Attack-Specific Assumptions . . . 88
6.1.2 Single-Stepping Frameworks . . . 88
6.1.3 Single-Stepping Challenges on Arm MCUs . . . 90
6.2 Single-Stepping on Cortex-M . . . 91
6.2.1 Profiling Instruction IRQ Behavior (C1) . . . 91
6.2.2 Single-Stepping Instruction Streams . . . 93
6.3 M-Step Framework . . . 95

6.3.1 Implementation Aspects . . . 95
6.3.2 Framework Plugins . . . 96
6.3.3 Framework Architecture (C5) . . . 97
6.3.4 M-Step Profiles . . . 99
6.3.5 Rapid Prototyping & Trace Analysis . . . 99
6.4 Evaluation . . . 100
6.4.1 Single-Step Reliability Microbenchmarks . . . 100
6.4.2 Covert-Channel Leakage Analysis . . . 101
6.4.3 Off-the-Shelf Smart Lock Application . . . 103
6.5 M-Step Limitations . . . 105
6.6 Summary . . . 106

7 Breaking Cryptographic Applications on MCUs 108
7.1 Unleashing M-Step on Mbed TLS . . . 109
7.1.1 Recovering Mbed TLS binary extended Euclidean algorithm (BEEA) Control-Flow 110
7.2 End-to-end Attacks on Mbed TLS. . . . 112
7.2.1 Mbed TLS Vulnerabilities. . . . 112
7.2.2 Case-Study Attacks. . . . 114
7.3 Countermeasures . . . 115
7.3.1 Naive Approaches . . . 115
7.3.2 Secure Interrupt Sanitization . . . 116
7.4 Summary . . . 117
7.4.1 Ethical Considerations . . . 118

8 Conclusion and Future Work 119
8.1 Conclusions . . . 119
8.1.1 Concluding Remarks. . . . 121
8.2 Future Work . . . 121

Bibliography 125

List of Figures

1 Illustration of traditional attacks versus side-channel attacks. 2
2 Thesis threat model overview. 5

3 High-level view of an APU (left) versus an MCU (right). 15
4 High-level overview of a CPU pipeline (No pipeline vs pipelined CPU). 17
5 High-level overview of a superscalar OoO pipeline.. 18
6 High-level overview of a memory subsystem with memory hierarchy of depth 4. 20
7 High-level overview of an APU with virtual memory versus a MCU with flat memory. . . 21
8 Overview of Cortex-M family process and execution modes and the software stack. . . . 26
9 Overview of the two types of memory subsystem featured in the Cortex-M family. 27
10 Armv8-M modes and security states and Armv8-M memory protection controllers. . . . 28
11 Trusted Firmware-M (TF-M) implemented PSA isolation levels as per PSA guidelines [48]. 29
12 Taxonomy of side-channel attacks, adapted from [29]. 31
13 Types of microarchitectural components. 33
14 microarchitectural attack phases. 34

15 Illustration of a Prime+Probe cache attack. 37
16 Illustration of a Flush+Reload cache attack. 38
17 Illustration of a controlled-channel attack. 39
18 Illustration of a transient execution attack. 41
19 Illustration of the interrupt latency leakage. 43
20 Illustration of the bus contention leakage. 44

21 Channel matrix of all six instruction cache (ICache)-based covert-channels. 55
22 Stream of interleaved `MOV` and `LDR` instructions and the respective IRQ latency. 56
23 Illustration of the interrupt logic microarchitectural channel. 57
24 Channel matrix of all twelve bus interconnect covert-channels. 58
25 Code of the synthetic victims with a secret-dependent conditional branch. 60

26 Control-flow of a secret-dependent conditional branch and its Prime+Probe ICache trace. 61
27 Control-flow of a secret-dependent conditional branch and its interrupt-latency trace. . . 62
28 Control-flow of a secret-dependent conditional branch and its memory contention trace. 63

29 Attack methodology: core steps to identify "vulnerable"clock cycles. 68
30 Illustration of two gadgets used to record contention and profile each victim run. 73
31 Illustration of the `read` stage contention with and without contention. 74
32 Illustration of the `write` stage contention with and without contention. 75
33 Illustration of the `read_keypad` code snippet. 77
34 Profiling and Exploitation SGNs. 78
35 Illustration of the BUSted attack setup. 81
36 Smart lock application trace recorded in SAM. 83

37 Cortex-M interrupt behavior for different instruction types and interrupt arrival times. . . 92
38 Simplified M-Step algorithm for dynamically adjusting the single-step timer interval. . . 95
39 Reconstructed Cortex-M interrupt handling process during an M-Step single-step. 97
40 M-Step rapid prototyping framework architecture. 98
41 Example of the Mstp-Visualizer depicting the a trace of the `printf` function. 100
42 M-Step leakage analysis experiments. 102
43 M-Step PoC attack on a smart lock application. 104

44 Template matrices for detecting U and V shifts and subtraction type. 111
45 Mbed TLS `mbedtls_mpi_gcd` vulnerable code to count the leading zeros of variable X. 113
46 Illustration of the secure interrupt sanitization mechanism. 117

List of Tables

1 Summary of APU vs MCU differences. 23
2 Summary of current Arm Cortex-M family (2026). 25
3 Overview of software-based power and timing side-channel attacks on MCUs. 47
4 Survey of microarchitectural characteristics from a distinct pool of MCUs. 50
5 Resources used by the BUSted attack. 82
6 Comparison of single-stepping frameworks. 89
7 Interrupt latencies for memory loads targeting different memory partitions. 93
8 List of current M-Step plugins. 96
9 M-Step's single-stepping reliability metrics across various instruction streams. 101
10 List of Mbed TLS vulnerable functions. 113
11 Reference Mbed TLS applications using RSA with vulnerable code paths. 115
12 Representative survey of MCUs integrating specialized processing units (xPUs). 122
13 Representative survey of heterogeneous multiprocessing platforms. 123

List of Listings

BUSted Profiling Phase. 69
BUSted Exploitation Phase Single-Run Variant. 70
Mbed TLS's BEEA algorithm to compute $X = A^{-1} \pmod{B}$. 110

Acronyms

ADC	analog-to-digital
AI	artificial intelligence
ALU	arithmetic logic unit
APIC	Advanced Programmable Interrupt Controller
APU	application processor unit
ARoT	Application Root of Trust
BEEA	binary extended Euclidean algorithm
BHT	branch history table
BM	bus main
BTAC	branch target address cache
BTB	branch target buffer
CCA	Confidential Compute Architecture
CCI	coherent central interconnect
CLI	command-line interface
CoVE	Confidential VM Extension
CPU	central processing unit
CRT	chinese remainder theorem
DCache	data cache
DMA	direct memory access
FPGA	field-programmable gate array
FPU	floating-point unit

GCD greatest common divisor
GIC Generic Interrupt Controller
GPU graphics processing unit
GUI graphical user interface

I/O input/output
ICache instruction cache
ICI interrupt-continuable instruction
IDAU implementation defined attribution unit
IoT Internet of things
IRQ interrupt request
ISA instruction set architecture
ISR interrupt service routine

LLC last-level cache

MCU microcontroller unit
MDS microarchitectural data sampling
ML machine learning
MMU memory management unit
MPU memory protection unit
MSI message-signaled interrupts

NPU neural processing unit
NSC Non-Secure Callable
NSPE Non-Secure Processing Environment

OoO out-of-order
OS operating system

PC program counter
PCIe Peripheral Component Interconnect Express
PMP physical memory protection
PoC proof-of-concept
PRoT Privilege Root of Trust
PSA Platform Security Architecture
PSIRT Product Security Incident Response Team

QoS quality of service

RPC remote procedure call
RTOS real-time operating system

SAU security attribution unit
SEV-SNP Secure Encrypted Virtualization-Secure Nested Paging
SG secure gate
SGX Software Guard Extensions
SoC system-on-chip
SPE Secure Processing Environment
SPM secure partition manager
SRAM static random access memory
ST ST Microelectronics

TA trusted application
TCB trusted computing base
TCM Tightly Coupled Memory
TDX Trust Domain Extensions
TEE trusted execution environment
TF Trusted Firmware
TF-M Trusted Firmware-M
TLB translation lookaside buffer
TRM technical reference manual

USB Universal Serial Bus
uTEE micro-TEE

VCD value change dump
VM virtual machine
VTOR vector table register

XOM execute-only memory

1 Introduction

Modern society relies heavily on interconnected digital systems, ranging from wearable devices to ubiquitous global communication networks. While these capabilities have become pervasive, they are fundamentally underpinned by a massive infrastructure of heterogeneous computing units. This infrastructure was largely established in the pre-Internet era, when security requirements were less stringent because attackers typically required local, physical access to their targets. However, the threat landscape shifted as the industry connected millions of previously isolated devices to the Internet, a trend that became known as the Internet of things (IoT). Devices originally designed to operate offline suddenly became globally accessible, exposing them directly to remote attackers.

Concretely, microcontroller units (MCUs) lie at the core of this computational paradigm shift. In contrast to the complex processors powering smartphones, laptops, and servers, MCUs are resource-constrained computing units designed for less compute-intensive workloads. They excel at deterministic operations under strict power budgets, but historically lacked robust secure-by-design features, as their attack surface was largely restricted to local adversaries. However, as the threat landscape evolved, major MCUs vendors began incorporating memory-based protection mechanisms to enforce security. Today, even basic MCUs [1, 2] feature hardware mechanisms for code compartmentalization, while more advanced models implement trusted execution environments (TEEs) [3, 4] tailored to these resource-constrained environments. In particular, all Arm Cortex-M MCUs released since 2016 support TrustZone-M [5], a hardware-based security extension designed to enforce strict isolation between untrusted and security-sensitive code. TrustZone-M physically compartmentalizes the core into two execution domains, the secure and non-secure world, ensuring that secure operations remain protected even if an attacker fully compromises non-secure ones [6].

In this thesis, we present novel attacks [7–11] against state-of-the-art MCU security extensions, in particular Arm TrustZone-M, and demonstrate how an attacker can bypass these hardware-enforced isolation mechanisms and compromise security-sensitive code. While these architectures continue to provide strong protection against traditional memory-based vulnerabilities, such as buffer overflows and unauthorized direct memory accesses, our research reveals that their resilience against software-based side-channel attacks is poorly understood. Concretely, we show that MCUs, despite their simplistic designs,

are also susceptible to microarchitectural attacks [12, 13], a threat vector long assumed to be confined to high-end computational units [12–16].

1.1 The Exotic World of Side-Channel Attacks.

Security is a perpetual arms race, as defenders enforce stronger protections, attackers continually devise novel and more creative exploits to circumvent them. In this thesis, we explore an exotic class of attacks, known as *side-channel attacks* [29], which has emerged from this arms race to compromise systems that are, by all conventional measures, "secure." Unlike conventional attack vectors, which target direct weaknesses such as design errors, implementation flaws, or hidden backdoors, side-channel attacks extract secrets from the unintended byproducts of computation itself. Similar to any other physical system, computing units produce not only the intended result but also observable side products [29], such as heat, electromagnetic emanations, and sound, which vary depending on the executed computation.

To build intuition about side-channel attacks, consider the abstract scenario in Fig. 1: an attacker who wants to recover a highly confidential piece of data stored in a secure hard drive. To steal the target piece of data the attacker can follow two approaches: *(i)* exploit a known vulnerability in the computing network responsible for handling the data, bypass all security measures, and steal it **directly**; or *(ii)* assume

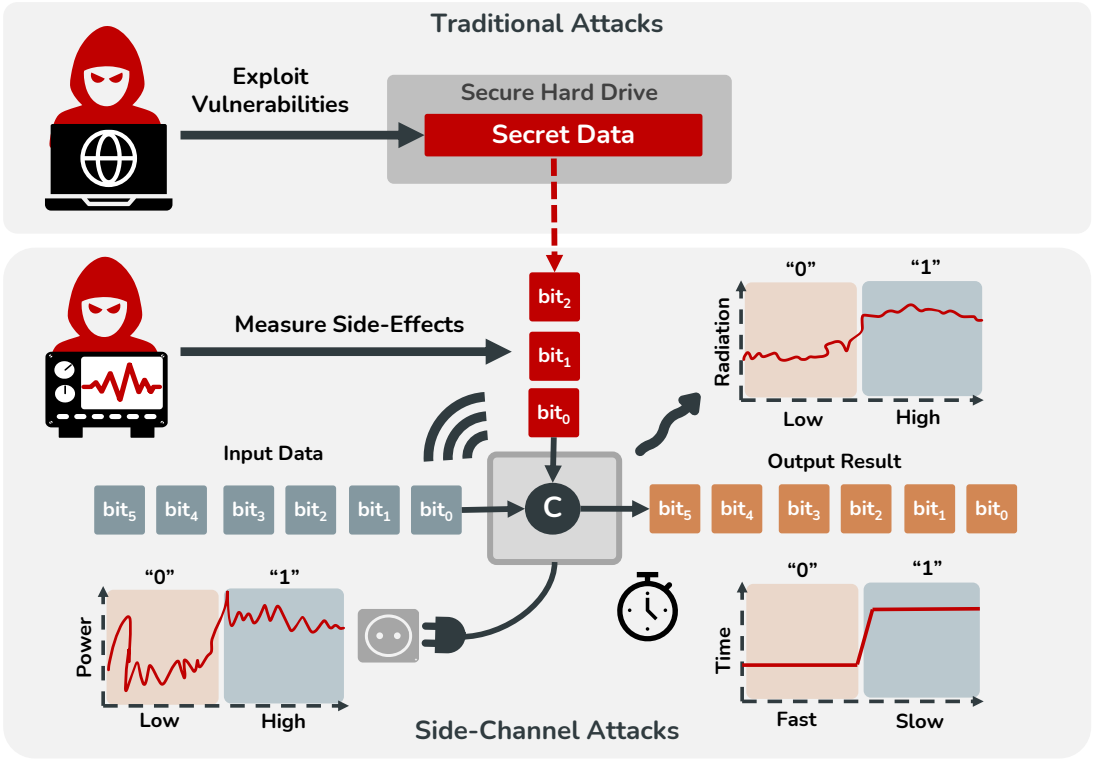


Figure 1: Illustration of traditional attacks (top), which exploit system vulnerabilities to access confidential data, versus side-channel attacks (bottom), which extract secret data by gathering metadata from measuring side-effects (such as sound [17–19], electromagnetic emanations [20–23], power consumption [24–27], or timing [28]) of computations performed over secret data.

the system is completely secure with no known vulnerabilities, and try to extract the secret by **indirect** means, using a side-channel. To execute the second strategy, the attacker can place a measurement device near the computing unit processing the data to record fluctuations in physical side effects, such as heat [30], sound [17–19], electromagnetic emanations [20–23], power consumption [24–27], or execution time [28]. Then the attacker can correlate these observations with publicly known information, e.g., the executed code and attacker-controlled inputs, where the only unknown variable is the secret being processed[1]. Concretely, the attacker may know that, given a fixed input bit set at "1", a computation with a secret bit set "0" consumes less energy, generates less acoustic or electromagnetic noise, and completes faster than a corresponding computation with a secret bit set at "1". By repeatedly measuring these computations over time, the attacker collects traces of these side effects and applies statistical analysis to infer whether the computing unit processed a "0" or a "1", indirectly extracting the full secret. This stealthiness, combined with the deep technical understanding to mount the exploit, earns side-channel attacks their reputation as "exotic" threats.

While side-channel attacks exploit a wide range of physical leakage sources, such as power consumption [24–27], electromagnetic emissions [20–23], and acoustic emanations [17–19], this thesis focuses on a specific class known as software-based microarchitectural attacks [12–16]. These attacks leverage timing variations caused by contention on shared CPU microarchitectural components (e.g., caches [15], execution ports [31–33], and branch predictors [34–36]) to infer secrets from the observable side effects of computation. While software-based microarchitectural attacks [12, 13] have been extensively studied on high-end application processor units (APUs) [12–16], i.e., processors powering smartphones, laptops, and servers, they remain largely overlooked on resource-constrained embedded and IoT devices powered by MCUs [37].

1.2 MCUs, The Elephant in the Room.

Despite the pervasive use of MCUs at scale (with billions of units shipped annually [38, 39]), the research community has been reluctant to investigate the potential security implications of MCUs microarchitecture. At the onset of this thesis, Nemesis [37] stood as the sole microarchitectural attack targeting an MCU, in stark contrast to the vast array of side-channel attacks already unveiled on APUs at the time [12–16]. We hypothesize that this lack of motivation stems from 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.

However, MCUs are becoming increasingly complex. Advances in semiconductor manufacturing allow designers to integrate vastly more transistors within strict power and cost budgets. Moreover, the performance demands of emerging use cases (e.g., shifting Capsule Networks and machine learning

[1] While this scenario may appear far-fetched, it captures the threat model of many cryptographic computations: the algorithm is known, the input can be controlled by the attacker, and only the secret key is unknown and therefore the target of the attack.

(ML) models to the deep edge [40–42]) have driven MCU designers to adopt advanced microarchitectural features. While the microarchitectural gap between APUs and MCUs remains significant, we argue that it is not as pronounced as it once was. Modern MCUs incorporate performance-oriented features that increasingly resemble those of APUs (e.g., deeper pipelines, branch prediction, and built-in caches). Notably, MCUs are now deployed in environments where trusted and untrusted software coexist in isolated domains (e.g., under the TrustZone-M threat model) while sharing the same underlying hardware. As with APUs, this sharing of hardware resources across security boundaries creates the fundamental conditions for cross-domain microarchitectural leakage.

As a result, while the microarchitectural complexity of MCUs has steadily increased, the security community still holds the classic (but historically reasonable) assumption that MCUs are too simple to leak. Motivated by this observation, we formulate two hypotheses. First, we hypothesize that, although simpler, modern Arm-based MCUs are likely to exhibit some form of microarchitectural leakage previously studied on APUs. Second, we hypothesize that, because of their distinct design constraints, MCUs may also expose leakage sources specific to this class of devices. In light of these hypotheses, we formulate the primary research question that guides the remainder of this thesis:

Research Question: Which microarchitectural components, whether previously exploited on APUs or unique to MCUs, can be leveraged to mount real-world microarchitectural side-channel attacks on Arm-based MCUs?

1.3 Scope and Threat Model

In this thesis, we study software-based microarchitectural side-channel attacks on MCUs, with a particular focus on Arm-based platforms. Arm-based MCUs dominate the embedded market (with reports indicating around 70% market share as of 2026 [43–45]) and offer a mature ecosystem with increasingly advanced security features. In particular, the latest Arm Cortex-M architecture (Armv8-M) natively supports TrustZone-M, a hardware security extension that enables the realization of TEEs on resource-constrained microcontrollers. Moreover, Arm maintains TF-M [46], a reference TEE implementation tailored for TrustZone-M which follows Arm's Platform Security Architecture (PSA) security guidelines [47, 48]. While high-end processors feature multiple competing TEE technologies (e.g., Intel SGX/TDX [49, 50], AMD SEV-SNP [51], or Arm TrustZone-A [6]), in the MCU domain, TrustZone-M [6] has emerged as a *de facto* standard for securing embedded applications. Consequently, TrustZone-M proves to be a highly relevant target. Its widespread deployment and strong privilege separation enable more capable software attackers than those traditionally considered on MCUs equipped with simpler memory-based isolation, such as Arm Cortex-M4 or Cortex-M7 processors that rely solely on memory protection units (MPUs). Therefore, we focus on resource-constrained embedded and IoT devices powered by TrustZone-M-enabled Armv8-M processors, such as the Arm Cortex-M23 and Cortex-M33.

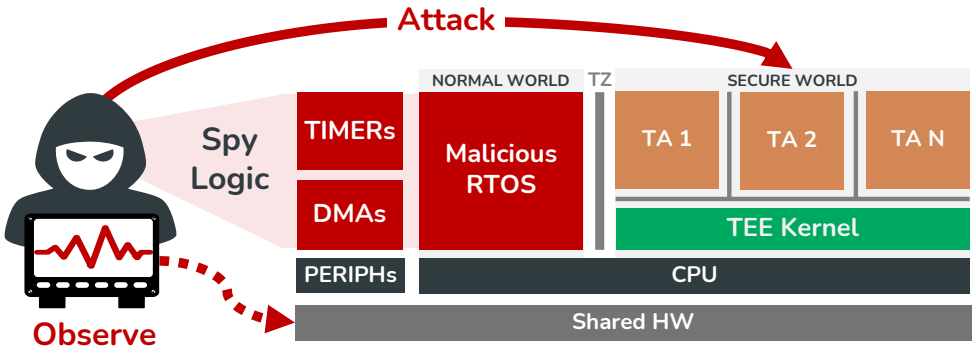


Figure 2: Thesis threat model overview: a software-only adversary with full control over the Normal World (red) attempts to extract sensitive information from a TrustZone-M-protected Secure World application through microarchitectural side-channel observations.

Regarding attacker capabilities, our threat model, illustrated in Fig. 2, assumes a software-only adversary whose primary goal is to leak sensitive information from a TrustZone-M-protected victim application through the indirect observation of microarchitectural behavior. Following the standard TrustZone threat model [6], we assume the attacker has full control over the Normal World and its assigned resources, such as timers and direct memory access (DMA) peripherals. We assume the Secure World trusted computing base (TCB) is correct and free of architectural software vulnerabilities [52, 53], but it is assumed to exhibit subtle non-constant-time behavior that leads to secret-dependent control- or data-flow variations. Physical attacks requiring hardware tampering, such as fault injection [54–57], are explicitly out of scope.

1.4 Thesis Contributions

In light of the research question raised at the onset of this thesis (**RQ**), we advance the state-of-the-art in microarchitectural side-channel attacks by demonstrating that this attack vector extends to MCU-powered embedded devices. In particular, we show that microarchitectural leakage is not confined to high-end APUs and that the perceived simplistic microarchitecture of a MCUs can actually be a key enabler of microarchitecture attacks on MCU-powered devices. Concretely, we make three main contributions, which we detail below.

1.4.1 BUSted, Non-Conventional Microarchitectural Leakage on MCUs.

We start our work by conducting a comprehensive microarchitecture leakage analysis of the Arm Cortex-M processor family. In particular, we evaluate seventeen MCUs from multiple vendors and instruction set architectures (ISAs), demonstrating for the first time pervasive microarchitectural leakage on resource-constrained devices. In our preliminary experiments (Chapter 4), we successfully establish covert channels targeting multiple microarchitectural components and validate their exploitability by mounting toy attacks on synthetic victims. Building upon this initial evidence, we introduce BUSted (Chapter 5), a non-conventional side-channel attack that exploits the side effects of the MCU bus interconnect arbitration

logic to bypass the security guarantees enforced on state-of-the-art Armv8-M MCUs. To mount this attack, we introduce the concept of hardware gadgets to overcome key challenges strictly tied to the resource-constrained nature of MCUs, such as *(i)* a single-core central processing unit (CPU) with no concurrent execution, and *(ii)* the stateless nature of the bus interconnect, which, unlike caches, does not preserve its microarchitectural state post victim execution. These hardware gadgets leverage widely available MCU memory-mapped peripherals, such as timers and DMAs, to implement spy logic that executes concurrently with the victim without CPU intervention. Using this novel technique, we demonstrate the first end-to-end microarchitectural side-channel attack on TrustZone-M, successfully extracting a secret PIN in a single run from a smart lock application deployed inside the secure world.

1.4.2 M-Step, A Single-Step Framework for Side-Channel Analysis on TrustZone-M

As a second contribution, we introduce M-Step (Chapter 6), the first single-stepping framework tailored for side-channel analysis on Arm TrustZone-M. Building upon the preliminary evidence of interrupt-based leakage observed during our initial experiments (Chapter 4), M-Step leverages the processor's interrupt logic to simulate single-stepping on production code without requiring any debug capabilities. This allows an attacker to continuously sample the microarchitectural state at an instruction-level granularity. While multiple single-stepping frameworks have been deployed targeting all major TEEs, such as Intel SGX/TDX [58–61], AMD SEV-SNP [62], and Arm TrustZone-A [63, 64], enabling over 50 precise side-channel attacks on such devices [59, 60, 62, 65], M-Step is the first such framework targeting MCUs. However, MCU present specific challenges not present in single-step frameworks targeting high-end TEEs, in particular, *(i)* the lack of virtual memory and *(ii)* the specific interrupt handling behavior of Arm Cortex-M MCUs. We propose novel techniques to overcome these challenges and have demonstrated that M-Step can reliably single-step generic instruction streams, allowing precise gathering of fine-grained traces from distinct microarchitectural structures. To substantiate this capability, we successfully establish four covert channels leveraging distinct microarchitectural components and mount an end-to-end proof-of-concept attack to extract a 4-digit PIN from a TrustZone-M-protected smart lock application, debunking prior claims that Cortex-M platforms are "immune" to interrupt-latency attacks [37].

1.4.3 End-to-End Attacks on a TrustZone-M-Protected Cryptographic Library.

As a third contribution, we present the first single-trace, end-to-end microarchitectural attack capable of recovering RSA private keys from TrustZone-M. While in BUSted (Chapter 5) and M-Step (Chapter 6) we demonstrated that microarchitectural leakage poses a tangible threat to TrustZone-M, we previously targeted simplified proof-of-concept scenarios by extracting secrets from general-purpose applications, such as dummy secure keypad implementations [8, 37]. To bridge this gap, we leverage our single-stepping

framework, M-Step, to uncover previously unknown vulnerabilities in the latest[2] version of Arm's widely used Mbed TLS cryptographic library, integrated within the TF-M project. Notably, we deterministically extract full RSA private keys from a single execution trace of Mbed TLS's complex BEEA implementation by accurately probing multiple secret-dependent branches in close succession. Moreover, we show that this vulnerability extends to platforms relying on hardware-based cryptographic accelerators. Following our responsible disclosure, Arm issued software patches for Mbed TLS (v3.6.5) to mitigate the vulnerabilities exploited by our attack, tracked under CVE-2025-54764. Ultimately, this final contribution provides a definitive answer to the main research question of this thesis (**RQ**), demonstrating that high-resolution, software-based side-channel attacks extend beyond high-end platforms and are practical even on TrustZone-M-enabled MCUs.

1.4.4 Other contributions

During the course of this thesis, driven by my own curiosity and the enthusiasm of fellow researchers, I participated in several parallel research efforts that diverged from the primary focus of this thesis. Below, I briefly outline these additional contributions both as first-author and in collaboration with other researchers.

1.4.4.1 What the TrustZone-M Doesn't See, the MCU Does Grieve Over: Lessons Learned from Assessing a Microcontroller TEE.

During the development of the BUSted attack [8] on the SAM L11 MCU, we uncovered an orthogonal vulnerability that allows an unprivileged secure partition to bypass the micro-TEE (uTEE) isolation boundaries. In particular, we demonstrate that the Trustonic Kinibi-M uTEE [66] implements PSA isolation level 3 [48], which aims to sandbox unprivileged third-party applications from the rest of the system; however, the underlying hardware lacks the necessary support for such granular enforcement. While TrustZone-M provides CPU-level security primitives that allow Kinibi-M to achieve PSA isolation level 3 from the CPU's perspective, e.g., separating non-secure from secure assets via the security attribution unit (SAU) and implementation defined attribution unit (IDAU) and privileged from unprivileged assets via the MPU, the SAM L11 system-level protection controllers (PPCs) only enforce PSA isolation level 1. Consequently, these system-level controllers only distinguish between non-secure and secure memory accesses, failing to protect isolated assets within the secure world itself. An attacker can configure a DMA peripheral to read and write arbitrary secure-world memory from an unprivileged third-party trusted application (TA). Concretely, we exploit this system-level vulnerability to bypass all hardware and software isolation boundaries of the Trustonic Kinibi-M uTEE, ultimately retrieving cryptographic keys from other secure partitions and the trusted kernel itself. We responsibly disclosed this issue to the affected vendors and presented our findings at Black Hat Asia 2024.

[2] Mbed TLS v3.6.3.1, the most up-to-date version at the time of our research.

C. Rodrigues and S. Pinto, *"What the TrustZone-M Doesn't See, the MCU Does Grieve Over: Lessons Learned from Assessing a Microcontroller TEE"*. In: BlackHat ASIA, 2024. *Link*. *Video*. *Slide deck*.

1.4.4.2 Devil-in-FPGA: Coherent Trojans in Reconfigurable Heterogeneous Systems.

This work was conducted in collaboration with researchers from Boston University specializing in real-time systems. Real-time properties (concerning interference) and security (concerning side-channel attacks) offer distinct perspectives on the same underlying challenge: the sharing of microarchitectural components. Following the release of BUSted, we began exploring whether the interference observed at the coherent central interconnect (CCI) level could be exploited to mount side-channel attacks. We concluded that while establishing such channels is possible, there is a significantly larger attack surface that could be used to exploit these systems. Consequently, our research evolved from investigating interference as a source of side-channel attacks to exploiting the CCI to mount powerful memory-based attacks. Specifically, we exposed the threat of latent trojans hidden within custom hardware integrated into the coherence system. We introduced a Coherent Trojan model systematizing the behavior of any malicious module that can stealthily interact with the coherence subsystem by mimicking coherence protocols once added to a coherence domain. To this end, we designed and implemented a runtime-configurable framework, the Trojan Tool, with a dual purpose: *(i)* advancing practical security research and *(ii)* serving as an exploration framework for attackers. Using this tool, we demonstrated that coherent trojans can execute stealthy attacks that compromise the confidentiality, integrity, and availability of commercially available platforms deploying cache-coherent FPGAs.

C. Rodrigues, S. Roozkhosh, S. Pinto, and R. Mancuso, *"Devil-in-FPGA: Coherent Trojans in Reconfigurable Heterogeneous Systems"*. (Under submission) In: Annual Computer Security Applications Conference (ACSAC), 2026.

1.4.4.3 Burning Fetch Execution: A Framework for Zero-Trust Multi-party Confidential Computing.

This work was conducted concurrently with our research on Devil-in-FPGA. We extended the capabilities of the Devil-in-FPGA Trojan Tool, incorporating additional custom field-programmable gate array (FPGA) modules to design an initial proof-of-concept for a computational paradigm aimed at supporting zero-trust multi-party confidential computing. Specifically, we introduce Burning Fetch Execution (BFX), a hardware mechanism that ensures sensitive data materializes only at the exact moment of the CPU's data fetch and is destroyed immediately after execution. By keeping data continuously encrypted in memory and decrypting it on the fly exclusively during the CPU data fetch stage, BFX ensures that decrypted data exists only transiently. Once processed, the data is irreversibly "burned," preventing decrypted materials from residing elsewhere in the system. Notably, BFX does not require redesigning existing processing units or modifying application software. Instead, it plugs directly into the coherence subsystem, operating

transparently to other coherent processing units as a man-in-the-middle. For heterogeneous CPU+FPGA platforms, BFX can be enabled out of the box. On non-reconfigurable architectures, it can be integrated during the system-on-chip (SoC) development cycle like any other coherent hardware module.

S. Roozkhosh, B. El Mabsout, **C. Rodrigues**, P. Carpanedo, D. Hoornaert, S. M. Tan, B. Lubin, M. Caccamo, S. Pinto, and R. Mancuso, *"Burning Fetch Execution: A Framework for Zero-Trust Multi-party Confidential Computing"*. In: Proceedings of 1st GENZERO Workshop, 2024. *Link*.

1.4.4.4 BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices.

This work was conducted in collaboration with a master's student as a follow-up to BUSted [8]. Concretely, it preceded and set the stage for M-Step (Chapter 6) and the end-to-end attacks we evaluate in Chapter 7. While in BUSted [8] we introduced two attack variants, the single-run and multi-run approaches, only the former was validated in a proof-of-concept attack against a smart lock application. However, we found that the single-run variant does not scale to complex victims, namely cryptographic libraries, due to its high profiling overhead. To address this limitation, we extended the multi-run BUSted variant by introducing an interrupt-driven multi-run methodology that traces victim execution at a per-interrupt granularity. Our technique leverages a trampoline and a sliding-window approach to minimize stack interference, improving the profiling granularity. We validated the effectiveness of this methodology by successfully profiling the TinyCrypt implementation of AES-CBC.

A. Barbosa, **C. Rodrigues**, T. Gomes, and S. Pinto, *"WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices"*. In: International Conference on Embedded Wireless Systems and Networks, 2024. *Link*.

1.4.4.5 MCU-Wide Timing Side Channels and Their Detection.

This work was conducted in collaboration with researchers from RPTU Kaiserslautern-Landau, Siemens EDA, and Stanford University. We introduce a formal verification method to detect BUSted-like microarchitectural timing side-channel attacks in MCUs. In particular, we model a holistic view of the SoC to capture both CPU and system-level microarchitectural interference on the shared interconnect. This system-level analysis is crucial to detect BUSted-style leakage, since both the CPU and other DMA-capable masters can create and observe contention on the bus. To validate our formal verification method, we evaluate it against an open-source RISC-V-based MCU, Pulpissimo, and uncover multiple sources of microarchitectural leakage. Based on this analysis, we implement a countermeasure that mitigates the detected vulnerabilities. Ultimately, this contribution presents a practical first step toward designing MCUs resilient against BUSted-like microarchitectural attacks.

J. Müller, A. Antón, L. Deutschmann, D. Mehmedagić, **C. Rodrigues**, D. Oliveira, M. Fadiheh, K. Devarajegowda, S. Pinto, D. Stoffel, and W. Kunz. *"MCU-Wide Timing Side Channels and Their Detection"*. In: Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24), 2024. *Link*.

1.4.4.6 Rethinking Trusted Execution Environments in the Age of Reconfigurable Computing.

This work was conducted in collaboration with fellow researchers from our laboratory. We explore the programmable logic embedded in CPU+FPGA platforms to devise a system where the trust anchor is moved to the FPGA. Specifically, we introduce Trusted Execution Environments On-Demand (TEEOD), a paradigm that enables the physical creation and destruction of enclaves on a per-request basis. These enclaves are materialized using soft-cores implemented on the reconfigurable logic. In contrast to conventional TEEs, which allocate enclaves on static hardware and may reuse the same hardware across multiple TA sessions, TEEOD provisions each TA session on cold hardware with a clean microarchitectural state. This approach provides a highly flexible architecture that can be configured on the fly and scaled in tandem with user needs. Moreover, because the FPGA fabric is physically isolated from the main processing system, no microarchitectural components are shared between untrusted and security-critical domains. This physical separation reduces the attack surface for microarchitectural side-channel attacks, which have been extensively exploited on high-end TEEs, such as Intel SGX, AMD SEV-SNP, and Arm TrustZone-A [13, 14, 67–69].

S. Pereira, D. Cerdeira, **C. Rodrigues**, and S. Pinto, *"Rethinking Trusted Execution Environments in the Age of Reconfigurable Computing"*. In: IEEE Access, 2025. *Link*.

1.4.5 Open Science

We open-source all research artifacts presented in this thesis to foster further research on microarchitectural side-channel attacks on MCUs. Specifically, we release the scripts used to generate figures and plots, the datasets required to reproduce our experiments, and the source code for all experimental evaluations. For each chapter detailing experiments, tool development, or proof-of-concept (PoC) attacks (Chapters 4 to 7), we provide a direct link to the corresponding artifacts released alongside the original publications. Moreover, we provide a dedicated repository containing the source code for previously unpublished experiments and thesis-specific artifacts (**A1**).

Summary of Open-Sourced Artifacts:

- **A1 - Artifacts Chapter 4:** https://github.com/ESCristiano/uarch-attacks-mcus.
- **A2 - Artifacts Chapter 5 (BUSted):** https://github.com/ESCristiano/BUSted.
- **A3 - Artifacts Chapters 6 and 7 (M-Step):** https://github.com/M-Step-Framework.

1.4.6 Ethical Considerations

We followed responsible disclosure best practices, proactively disclosing all vulnerabilities to the affected parties. When embargo periods were requested, we provided sufficient time for vendors to address the reported issues. Moreover, we engaged with Arm engineers on multiple occasions to provide feedback and assist in patching the vulnerable code exploited in this thesis. We release attacker source code only when we believe users are not at risk, i.e., after coordinating with vendors and ensuring security fixes are available, preventing the code from being weaponized. All provided artifacts and PoC attacks are released solely for research purposes to ultimately strengthen the overall security of MCUs.

Summary of Security Advisories:

S1 - Security advisory Chapter 5 (BUSted): Arm. *"Clarification of Timing Side Channel Attacks on TrustZone enabled Cortex-M based systems"*, 2023. *Link*.

S2 - Security advisory Chapter 5 (BUSted): wolfSSL. *"BUSted: Side-channel attacks to TrustZone-M separation"*, 2023. *Link*.

S3 - Security advisory Chapters 6 and 7 (M-Step): Arm Mbed TLS. *"Side channel in RSA key generation and operations (SSBleed, M-Step)"*, 2025. *Link*.

S4 - Security advisory Chapters 6 and 7 (M-Step): CVE Program. *"CVE-2025-54764"*, In: National Vulnerability Database, 2025. *CVE-2025-54764*.

1.4.7 List of Publications

This section summarizes the publications that underpin the main contributions of this thesis (**P1**, **P2**, and **W1**), including presentations at major security venues. Moreover, we list additional work, both as first author and in collaboration with other researchers, which constitutes secondary contributions developed during the course of this thesis.

1.4.7.1 Summary of publications as primary contributions:

T1 - Talk: C. Rodrigues and S. Pinto, *"Hand Me Your SECRET, MCU! Microarchitectural Timing Attacks on Microcontrollers Are Practical"*. In: BlackHat ASIA, 2023. *Link*. *Video*. *Slide deck*.

W1 - White paper: C. Rodrigues, D. Oliveira, and S. Pinto, *"White Paper: Software-based Microarchitectural Timing Side-Channel Attacks on TrustZone-M MCUs"*, In: www.bustedattack.com, 2023. *Link*.

P1 - Publication: C. Rodrigues, D. Oliveira, and S. Pinto, *"BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect"*, In: IEEE Symposium on Security and Privacy (S&P), 2024. *Link*.

P2 - Publication: C. Rodrigues, M. Bognár, S. Pinto, and J. Van Bulck *"M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M"*, (To appear) In: 35th USENIX Security Symposium, 2026. *Link*.

T2 - Talk: C. Rodrigues, M. Bognár, J. Van Bulck, and S. Pinto *"Tiny Chips, Big Leaks: Breaking TrustZone-M with Single-Stepping Attacks"*. (To appear) In: BlackHat USA, 2026. *Link*.

1.4.7.2 Summary of publications as secondary contributions:

T3 - Talk: C. Rodrigues and S. Pinto, *"What the TrustZone-M Doesn't See, the MCU Does Grieve Over: Lessons Learned from Assessing a Microcontroller TEE"*. In: BlackHat ASIA, 2024. *Link*. *Video*. *Slide deck*.

P3 - Publication: C. Rodrigues, S. Roozkhosh, S. Pinto, and R. Mancuso *"Devil-in-FPGA: Coherent Trojans in Reconfigurable Heterogeneous Systems"*, (Under submission) In: Annual Computer Security Applications Conference (ACSAC), 2026.

P4 - Publication: S. Roozkhosh, B. El Mabsout, **C. Rodrigues**, P. Carpanedo, D. Hoornaert, S. M. Tan, B. Lubin, M. Caccamo, S. Pinto, and R. Mancuso, *"Burning Fetch Execution: A Framework for Zero-Trust Multi-party Confidential Computing"*. In: Proceedings of 1st GENZERO Workshop, 2024. *Link*.

P5 - Publication: A. Barbosa, **C. Rodrigues**, T. Gomes, and S. Pinto, *"WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices"*. In: International Conference on Embedded Wireless Systems and Networks, 2024. *Link*.

P6 - Publication: J. Müller, A. Antón, L. Deutschmann, D. Mehmedagić, **C. Rodrigues**, D. Oliveira, M. Fadiheh, K. Devarajegowda, S. Pinto, D. Stoffel, and W. Kunz. *"MCU-Wide Timing Side Channels and Their Detection"*. In: Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24), 2024. *Link*.

P7 - Publication: S. Pereira, D. Cerdeira, **C. Rodrigues**, and S. Pinto, *"Rethinking Trusted Execution Environments in the Age of Reconfigurable Computing"*. In: IEEE Access, 2025. *Link*.

1.5 Document Outline

The remainder of this thesis is structured into eight additional chapters. The first two chapters provide the necessary technical background and review prior work (Chapters 2 and 3), while the remaining chapters present our core contributions (Chapters 4 to 7). Concretely, in Chapter 2, we establish the foundations of the current computing architecture landscape, review key features of the Arm Cortex-M processor family, and introduce side-channel attacks. Next, Chapter 3 reviews prior work on microarchitectural attacks targeting high-end CPUs, contrasting it with the limited literature on low-end MCUs.

In Chapter 4, we analyze the Arm Cortex-M processor family and present the first evidence of microarchitectural leakage on MCUs through covert-channels and toy attacks. In Chapter 5, we present BUSted, a novel side-channel attack that exploits the timing side effects of the MCU bus-interconnect arbitration logic. With BUSted, we mount a highly scalable microarchitectural side-channel attack, demonstrating its pervasiveness across the whole MCU computing spectrum. In Chapter 6, we present M-Step, the first single-stepping framework targeting TrustZone-M-enabled MCUs. We show how this framework can extract high-resolution traces capable of capturing subtle timing differences across execution paths, down to a single instruction. In Chapter 7, we provide definitive evidence that side-channel attacks on MCUs pose a severe real-world threat. Concretely, we mount the first side-channel attack capable of extracting cryptographic keys directly from TrustZone-M.

Finally, Chapter 8 summarizes our findings, reflects on the lessons learned, and outlines future research directions for microarchitectural side-channel attacks on MCUs.

2

Background

In this chapter, we provide the basic background and context on the core concepts explored throughout this thesis. First, in Section 2.1, we will provide an overview of the computing spectrum to position MCUs within the broader computing landscape and review core computer architecture concepts. Then, in Section 2.2, we will narrow down to Arm-based MCUs and explain in depth the Arm Cortex-M architecture. To conclude the chapter, in Section 2.3, we will provide a high-level explanation of the target attack class explored in this thesis, i.e., side-channel attacks.

2.1 Computing Spectrum and Computer Architecture

The modern computing spectrum encompasses a vast array of use cases, each addressed by specialized computational units. Servers handle high computational demands, while personal computers and smartphones offer mid-range performance for daily use. Smaller devices, e.g., smartwatches, rely on highly energy-efficient but low-performance computational units. All these devices from high- to low-end are powered by processors, which are shaped by distinct design constraints depending on their specific use cases, making them fundamentally different in nature. The processors powering the high- to mid-end of the computing spectrum are typically designated as APUs[1], while the chips powering the low-end of the spectrum are typically referred to as MCUs. However, while APUs primarily consist of processor logic, with memory and peripherals being deployed at the system-level, as depicted in Fig. 3, MCUs are all-in-one computing units, packing a processor, memory, and peripherals into a single chip. While there are other core characteristics that vastly differ between both types of processors, which we explain next, this system-level integration provides the primary clear-cut distinction that historically separates both computing units' definitions. Although the distinction between APUs and MCUs may seem straightforward, it is not always well understood. Therefore, while in this thesis we focus on MCUs, this section provides a broader view of the compute spectrum and its core computer architecture concepts covering both computing units.

[1] These processors are also historically known as microprocessors. Throughout this thesis, we use the term APU instead, as it better conveys the nature of these devices and provides a clearer distinction from microcontrollers.

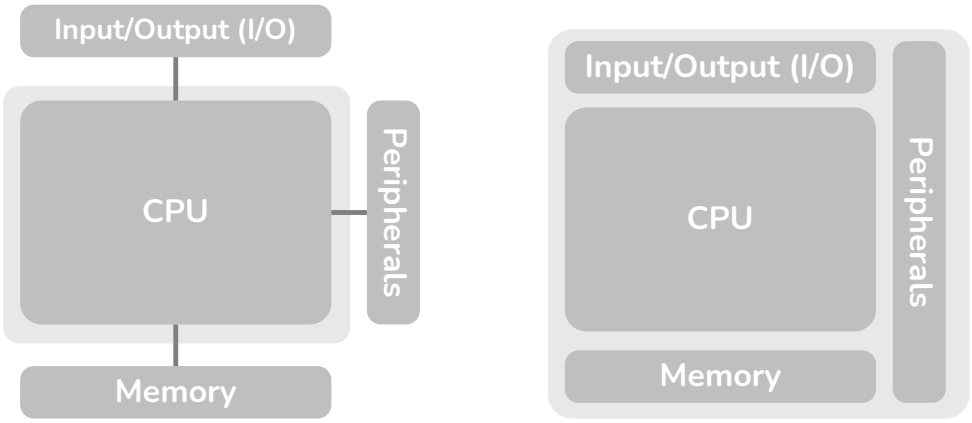


Figure 3: High-level view of an APU (left) versus an MCU (right).

This comparison will later help the reader to better understand the MCU-specific challenges we had to overcome to develop the side-channel attacks presented in this thesis.

2.1.1 The Constraints that Shape the Device.

Computer architecture relies on multiple abstraction layers to manage the physical complexity of hardware. Through these layers, transistors are arranged into logic gates, which are controlled by a precise set of instructions dictated by an ISA. These instructions can be assembled in countless configurations by translating high-level programming languages into machine code, enabling complex computations.

While the underlying abstractions are similar across the computing spectrum, e.g., all computing units are built around an ISA, their implementations vary significantly. Naturally, the target applications of APUs and MCUs dictate the design constraints that shape these devices at both the architectural and microarchitectural levels[2]. Concretely, MCUs are designed to reliably perform a small set of actions in a deterministic fashion for years, whereas APUs are designed to handle the unpredictable demands of a human user as fast as possible, such as seamlessly managing dozens of browser tabs. In other words, MCUs are engineered for real-time, deterministic behavior, while APUs prioritize maximum throughput. For an MCU, the system must know exactly how many microseconds an action will take, such as reacting to a brake press to activate a car's anti-lock braking system before it skids. In contrast, in an APU, the precise execution time of a single action, e.g., rendering a video frame, is less important than executing millions of such actions as quickly as possible to maintain a smooth user experience.

[2]Architecture refers to the functional definition of a system, typically captured by the ISA (e.g., x86-64 or Armv8-M), while the microarchitecture refers to the specific implementation of these architectural functions (e.g., Intel Skylake or Arm Cortex-M33). For instance, while the architecture defines instructions for memory access, the microarchitecture implements structures like caches, branch predictors, and pipeline stages to optimize these accesses. Crucially, the architecture serves as the interface between software and hardware, providing the final layer from which an attacker can directly derive hardware information. The microarchitecture of a CPU (such as the current state of its caches) is inherently hidden from the user and cannot be directly accessed. This distinction is vital for side-channel attacks, which attempt to make microarchitectural information architecturally visible through indirect means.

Moreover, power consumption serves as a strict design constraint shaping these systems. While lower power consumption is desirable for APUs, especially those powering laptops and smartphones, power management is far stricter in MCUs. Since MCUs are frequently deployed in systems without a direct power supply, they must minimize energy draw. For context, an AMD Ryzen AI 400 Series processor for laptops [70] typically consumes between 15 W and 54 W, whereas a widespread MCU like the STM32L4 [71], which features an Arm Cortex-M4 CPU, consumes from a few μW in standby mode up to a few mW at full capacity. These stark differences in power budget restrict the maximum operating speed of an MCU and require highly optimized, energy-efficient designs. Ultimately, these power constraints, combined with the inherent trade-off between real-time predictability and raw throughput, represent the primary design forces shaping the architectural and microarchitectural implementations of both types of computing units.

Anatomy of a Computing Unit. Before we start comparing in more detail APUs and MCUs, we first must define what a computer unit is. Regardless of its target application, a computer unit performs a set of actions based on a set of inputs and produces a predictable outcome when the same set of instructions are performed. To do so, it needs three essential building blocks: *(i)* a *central processing unit (CPU)*, to perform the desirable operations; *(ii)* a *memory subsystem*, to store data involved in the computation; and *(iii)* an *input/output (I/O) interface* to interact with the external world. These three main building blocks characterize how both APUs- and MCUs-powered devices are built. However, despite the similarity in structure organization, these three building blocks have vastly different implementations, at both the architecture and microarchitectural level, between APUs and MCUs, which we review next.

2.1.2 Processing Unit.

CPUs process data stored in memory based on a predefined set of instructions. Each instruction encodes a specific action that the CPU must perform, such as fetching data from memory or adding two numbers. A sequence of these instructions forms the program to be executed. While APUs prioritize maximum throughput, MCUs rely on real-time, deterministic behavior. Consequently, these primary design goals dictate how a CPU handles instructions and shapes its microarchitectural implementation. In APUs, the microarchitecture is optimized to execute a stream of instructions as quickly as possible and to minimize pipeline stalls. In contrast, the microarchitecture of MCUs is traditionally optimized for deterministic operation, often accepting performance penalties, such as stalls, to avoid unpredictable execution behavior.

2.1.2.1 CPU Pipelining.

To maximize instruction throughput, modern CPUs rely on pipelining, as illustrated in Fig. 4. Pipelining overlaps the execution of multiple instructions by dividing the instruction lifecycle into distinct, independent stages. In a strictly sequential, non-pipelined architecture, the CPU must wait for an instruction to

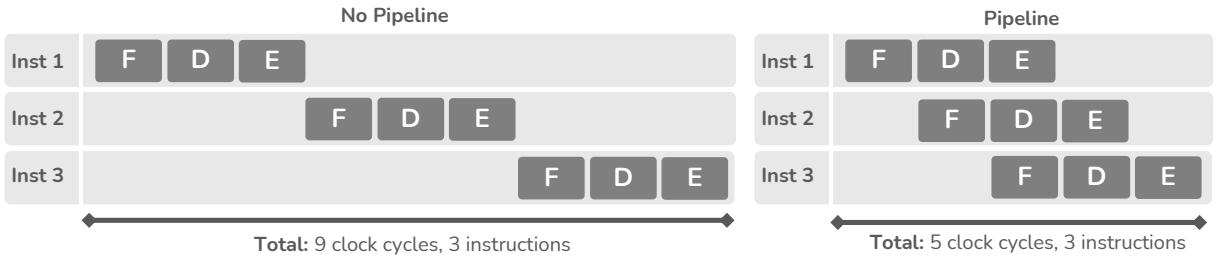


Figure 4: High-level overview of a CPU pipeline. We contrast the execution of three instructions in a non-pipelined CPU (left) and a three-stage pipelined CPU (right), comprising fetch (F), decode (D), and execute (E) stages. Pipelining overlaps instruction stages to improve throughput; once full, the pipelined CPU provides a peak throughput of one instruction per clock cycle.

fully complete before fetching the next. Pipelining overcomes this bottleneck by inserting pipeline registers between stages to store intermediate results; this allows the hardware to process different stages of separate instructions simultaneously. For example, early pipelines used a three-stage model (Fetch, Decode, and Execute), enabling up to three instructions to be in-flight concurrently. Under ideal conditions, once the pipeline is full, this architecture achieves a throughput of completing one instruction every clock cycle. While modern CPUs feature significantly deeper pipelines, the fundamental principle of overlapping instruction execution remains the same regardless of the pipeline length.

APUs vs. MCUs Distinction. The CPU pipeline length provides the first major architectural distinction between APUs and MCUs. Modern APUs feature highly complex pipelines with multiple stages (some exceeding 20 stages). Conversely, even the most powerful MCUs (e.g., Arm Cortex-M7 and Cortex-M85) do not exceed 7 pipeline stages. The vast majority of MCUs exhibit simple pipelines with 2 or 3 stages; for instance, out of the 12 MCUs in the Arm Cortex-M family, 8 have pipelines with fewer than 4 stages.

2.1.2.2 Superscalar Execution.

While pipelines improve instruction throughput, they are still susceptible to pipeline hazards. When an instruction has to wait for a specific action to complete, e.g., the result of an arithmetic logic unit (ALU) computation or a high-latency memory access, the entire pipeline freezes. These stalls are known as hazards, which arise from conditions such as data and control-flow dependencies. To mitigate these hazards and further exploit instruction-level parallelism, CPUs evolved to use superscalar architectures. By replicating hardware execution units, a superscalar processor can issue and execute multiple instructions without direct dependencies during the same clock cycle. Early superscalar CPUs relied on an in-order execution model. However, this approach still suffered from a fundamental bottleneck, i.e., if one instruction stalled, all subsequent instructions in the stream were also blocked to maintain the original program order, even if independent execution units were sitting idle.

To overcome this limitation, out-of-order (OoO) execution was introduced, as illustrated in Fig. 5. This paradigm allows instructions with ready operands to execute immediately, effectively bypassing older stalled instructions waiting in independent execution units. However, while instructions in a OoO CPU

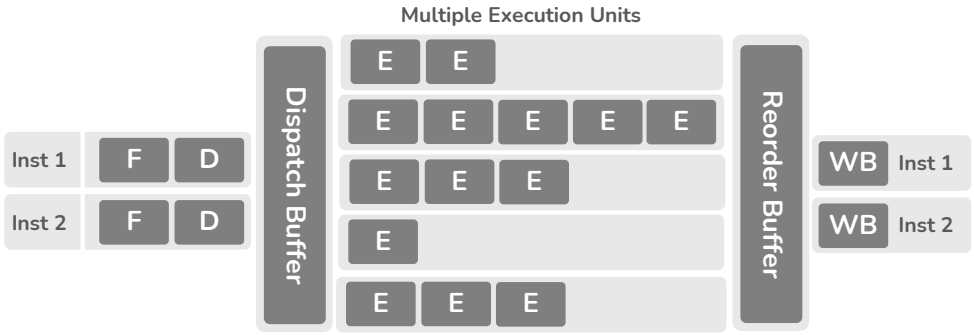


Figure 5: High-level overview of a superscalar OoO pipeline. In this illustrative example, the pipeline fetches two instructions per cycle and features five parallel execution units, each optimized for distinct instruction types; pipeline stage latencies vary accordingly.

execute out of their original program sequence, they must be committed, i.e., their results become architecturally visible, strictly in order. To enforce this requirement, OoO execution leverages a dispatch buffer that issues instructions to the execution units as soon as their operands are available, alongside a reorder buffer (ROB), that gathers the executed instructions and precisely reorders them to be committed in their original program sequence.

APUs vs. MCUs Distinction. Superscalar execution provides the second major distinction in CPU architecture between APUs and MCUs. In fact, the optimization techniques described here are almost exclusively used in APUs. While a small set of high-performance Arm Cortex-M CPUs (Arm Cortex-M7 and Cortex-M85) implement a limited version of in-order superscalar execution capable of dual-issuing certain instruction pairs, this is an exception rather than the norm. The vast majority of MCUs lack any capability to process more than one instruction at a time.

2.1.2.3 Branch Predictors.

While CPUs with highly optimized pipelines achieve high throughput by parallelizing instruction execution, they must still fetch instructions that are non-contiguous in memory due to control-flow branches. To prevent the CPU from waiting for a branch outcome, microarchitectural functional units called branch predictors speculate on the result and fetch instructions ahead of time to avoid pipeline stalls. Specifically, branch prediction strategies fall into two categories: *(i) static predictors*, which always perform the same prediction based on the instruction type, e.g., predicting that backward branches are taken while forward branches are not, and *(ii) dynamic predictors*, which leverage previously executed instructions to make an educated prediction. Dynamic predictors primarily rely on two microarchitectural structures: (i) a branch history table (BHT) to predict a branch outcome based on historical execution, and (ii) a branch target buffer (BTB) to predict the branch target address.

In modern CPUs, branch predictors are essential for speculative execution, allowing the CPU to execute instructions ahead of time based on the predictor's state. These speculatively executed instructions are only committed architecturally if the prediction is correct; otherwise, they are discarded. While discarding

instructions incurs a performance penalty, accurate predictions avoid extensive pipeline stalls and provide substantial performance improvements.

APUs vs. MCUs Distinction. Branch prediction and speculative execution provide the third major distinction in CPU architecture between APUs and MCUs. While there are MCUs with branch predictors (Cortex-M7, Cortex-M52, Cortex-M55, and Cortex-M85), these are mainly used to prefetch instructions ahead of time and do not provide speculative execution capabilities.

2.1.3 Memory Subsystem.

The memory subsystem is used to store the data to be processed by the computing unit, as well as the instructions that guide it to perform the desired actions. At the architectural level, data is indexed by its memory address, and access is controlled by specific permission bits. However, at the microarchitectural level, there is a complex memory hierarchy, holding multiple copies of the same data across different memory buffers with varying access latencies. These buffers must be automatically synchronized to maintain coherency whenever a data copy is modified. Furthermore, while data location is fixed by its address architecturally, its microarchitectural placement depends on spatial and temporal locality relative to data previously processed by the CPU. Although these mechanisms exist in both APUs and MCUs, their implementations vastly differ due to the inherent characteristics of each computing unit. In APUs, the main data is typically stored off-chip in larger, slower memories. This physical distance significantly increases latency, forcing the memory subsystem to employ complex strategies to sustain the high throughput required by the CPU. In contrast, MCUs utilize small, fast, on-chip memories located close to the CPU. This integration keeps access latency low and greatly reduces the need for complex microarchitectural buffers.

2.1.3.1 Memory Hierarchy.

There is an inherent inverse correlation between the size of a memory and its access latency; the faster the memory, the smaller it must be[3], mainly due to the hardware cost of implementing such fast memory. To bring data closer to the CPU, i.e., to reduce memory access latency, modern memory subsystems implement a multi-layer memory hierarchy. Data that is highly likely to be accessed in the near term is kept as close as possible to the CPU in microarchitectural buffers called *caches*. Conversely, infrequently accessed data is relegated to larger, slower memories until it is explicitly needed. Concretely, modern memory subsystems employ multiple layers of these microarchitectural structures to progressively stage recently used data. For instance, level 1 caches store very recently used data and are strictly private to a single core, while level 2 caches are typically shared across multiple cores within a CPU. Level 3 caches,

[3]While in-memory processing [72–74] circumvents this for specific parallel tasks (e.g., matrix multiplications), most operations are still routed to the host CPU. Consequently, the fundamental need for a traditional, low-latency cache hierarchy remains intact.

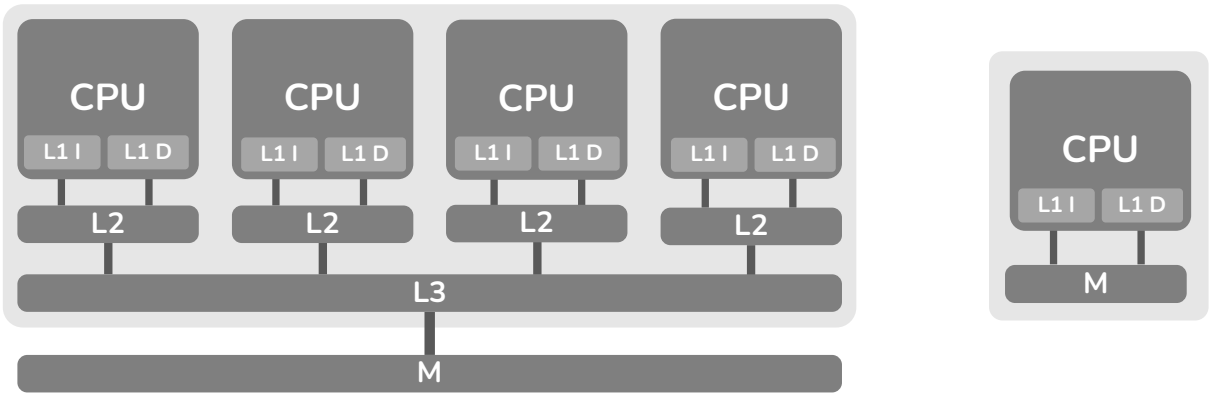


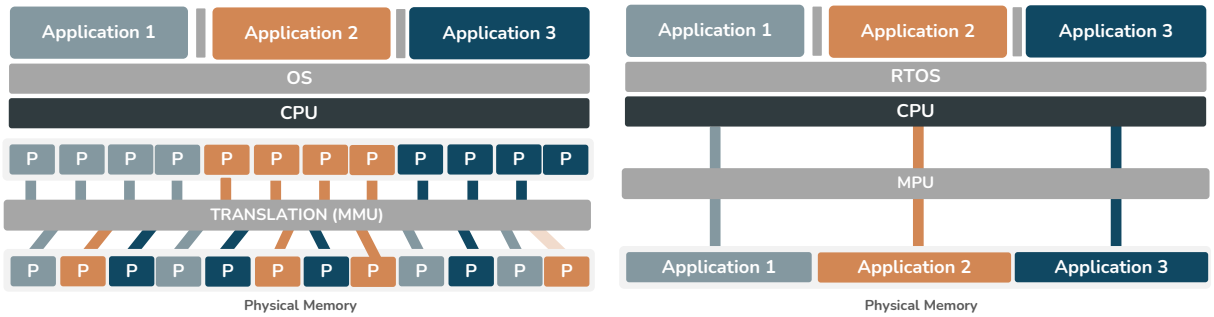
Figure 6: High-level overview of memory hierarchies: an APU with depth 4 (L1, L2, L3, and main memory) versus advanced MCUs with depth 2 (only L1 and main memory).

commonly known as last-level caches (LLCs), are typically shared across multiple computing units. Moving down the memory hierarchy, each subsequent layer is strictly larger but slower than the preceding one.

APUs vs. MCUs Distinction. The memory hierarchy provides the fourth major distinction between APUs and MCUs, as illustrated in Fig. 6. Although cache-like structures can be found in both types of computing units, the depth and complexity of the memory hierarchy vastly differ. Native caches in MCUs are not the norm, and even high-performance MCUs that do feature them, such as the Arm Cortex-M7, Cortex-M55, and Cortex-M85, do not extend beyond first-level caches [75]. Instead, they feature very fast memories that can sometimes deliver data in a single clock cycle, such as Tightly Coupled Memorys (TCMs) and static random access memorys (SRAMs). In contrast, APUs implement multiple layers of cache (typically at least two) and utilize various other cache-like structures, e.g., translation lookaside buffers (TLBs), to bring different types of data closer to the CPU.

2.1.3.2 Virtual Memory.

Virtual memory abstracts the physical limitations of hardware memory, enabling multiple programs to execute securely and efficiently within a shared, theoretically infinite memory space. This abstraction is crucial for feature-rich operating systems (OSes), such as Linux or macOS, to enforce process isolation and prevent malicious or faulty applications from corrupting the memory of other processes. Concretely, an OS can execute the same application multiple times at identical virtual addresses, as the underlying hardware maps these isolated virtual memory spaces to distinct physical locations. For instance, two concurrent applications may both access the virtual address `0x00000000`; however, the system natively maps these accesses to entirely different physical addresses. In contrast, MCUs typically rely on flat memory address spaces without virtual memory support, where instructions and data reside at fixed, hard-coded physical locations (e.g., Flash and SRAM). Consequently, reusing addresses between isolated applications is impossible. Each application must be explicitly deployed at a distinct physical address, and executing multiple instances of the same application requires recompiling them with different base addresses, as illustrated in Fig. 7b.



(a) Virtual memory space, APUs.

(b) Flat memory space, MCUs.

Figure 7: High-level overview of an APU with virtual memory (left) versus a MCU with flat memory (right). On the left, applications see a contiguous virtual address space with pages (P) mapped in succession; however, these pages are translated to arbitrary physical addresses and scattered throughout physical memory. In contrast, on the right, applications on a MCU are placed directly into physical memory within a single, unified address space, ensuring that all code and data remain physically contiguous.

Virtual memory achieves this abstraction by splitting the virtual address space into fixed-size slices called pages, which the hardware maps to corresponding physical pages. While the virtual memory appears contiguous from the application's perspective, the underlying physical pages can be scattered across the physical memory, as illustrated in Fig. 7a. When the CPU accesses memory, the system translates the virtual address into the exact physical address where the data is physically stored. To manage this translation, the architecture leverages page tables, which act as a comprehensive lookup structure for physical-to-virtual mappings. Because page tables are large and reside in slow main memory, processors use a cache-like structure known as a TLB to store recent address translations, accelerating the translation process. The system accesses the slow in-memory page tables only during a TLB miss, i.e., when the requested translation is not cached, triggering a page table walk. Concretely, a centralized memory management unit (MMU) manages this translation logic and simultaneously enforces access permissions, such as applying read-only, no-execute, or privileged-only constraints to specific memory pages. In contrast, while MCUs with flat memory spaces lack an MMU, they still use an MPU to enforce access policies and isolate applications based directly on physical addresses.

APUs vs. MCUs Distinction. Virtual memory provides the fifth major distinction between APUs and MCUs, as illustrated in Fig. 7. All MCUs rely on a flat memory address space and use physical address-based isolation primitives, such as the MPU, which only restrict applications from accessing memory outside their delineated boundaries. This lack of virtual memory allows MCUs to maintain deterministic memory access behavior, as there are no unpredictable page table walks, and ensures low access latency since memory can be addressed directly without navigating multiple levels of address translation. While virtual memory enables APUs to run feature-rich OSes like Linux, flat memory address spaces restrict MCUs to simpler OSes that are better suited for real-time operations, i.e., real-time operating systems (RTOSs).

2.1.4 I/O Interface.

The I/O block is the component of a computing unit responsible for interfacing the digital domain with the physical world. It receives data and signals from external entities or events and communicates the results of system computations. To manage this interface, it utilizes peripherals capable of performing actions independently of the main CPU, such as moving data via direct memory access (DMA) or transmitting it through specific protocols like Universal Serial Bus (USB).

APUs vs. MCUs Distinction. The management and functional range of these peripherals represent the sixth major distinction between APUs and MCUs. MCUs feature a high density and variety of integrated peripherals designed to meet real-time and low-power requirements. These peripherals offload tasks from the CPU to more energy-efficient, specialized hardware. In an MCU, peripherals are memory-mapped to the flat physical address space, allowing the CPU to program them directly with extremely low latency. Moreover, these peripherals trigger immediate hardware interrupts via dedicated physical wires to a local vectored interrupt controller, ensuring highly deterministic responses to external events.

In contrast, while APUs also use memory-mapped I/O, the hardware interaction is heavily abstracted. The OS must rely the MMU to map physical peripheral registers into virtual memory. Furthermore, rather than interfacing directly with simple endpoints, an APU typically communicates with complex intermediate host controllers, e.g., Peripheral Component Interconnect Express (PCIe) or USB controllers, over a high-performance system interconnect. Interrupt handling is similarly abstracted; peripherals generate message-signaled interruptss (MSIs) by performing direct memory writes rather than relying on dedicated physical wires. An advanced, system-level interrupt controller, e.g., the Advanced Programmable Interrupt Controller (APIC) in x86 or Generic Interrupt Controller (GIC) in Arm architectures, intercepts these messages, translates them into interrupt signals, and dynamically routes them to CPU cores.

2.1.5 Trusted Execution Environments.

TEEs provide isolated environments where security-critical code can be safely deployed and executed alongside untrusted, potentially malicious code. While data at rest and in transit are protected by well-established cryptographic primitives, data in use must be processed in its raw, unprotected format, making it vulnerable to attacks. To mitigate this potential vulnerability, TEEs enforce the protection of data while it is being processed. Concretely, the system runs non-security-critical code in a rich execution environment, i.e., an environment with extensive capabilities, and deploys security-critical tasks, such as key storage or cryptographic algorithms, in a highly restrictive, trusted environment. The TEE threat model assumes that the entire rich environment is untrusted and **cannot** interfere with the trusted environment, even if compromised. To enforce this security boundary, TEEs use hardware blocks that physically isolate the rich and trusted environments across the entire system, including the CPU, memory subsystem, and I/O interfaces.

Currently, there are two generations of this security technology. The first generation isolates individual

processes or applications, such as Intel Software Guard Extensions (SGX) [49], Arm TrustZone-A [6], and RISC-V physical memory protection (PMP) used in a Keystone-like fashion [76]. The second generation provides isolation for an entire virtual machine (VM), such as Intel Trust Domain Extensions (TDX) [50], AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) [51], Arm Confidential Compute Architecture (CCA) [77], and RISC-V Confidential VM Extension (CoVE) [78]. While initially developed for APUs, TEEs are now integrated into MCUs through technologies such as Arm TrustZone-M [6].

2.1.6 What is a MCU?

Table 1: Summary of APU vs MCU differences.

	Feature	Application Processor Unit (APU)	Microcontroller Unit (MCU)
SYSTEM	**Typical Devices**	Servers, PCs, Smartphones	ECUs, Wearables, IoT, Sensors
	Standardization	Highly standardized	Heterogeneous, Vendor-specific
	Core Count	Multi-core (2+)	Single- or Dual-Core
	Security (TEEs)	Process/VM isolation	Hardware-backed (TZ-M)
	Primary Design Goal	Maximum throughput	Real-time / Deterministic Execution
	System Integration	External memory/peripherals	All-in-one SoC
	Power Consumption	High (W–100W+)	Ultra-low (μW–mW)
	Major Distinction	System Integration	
CPU	**CPU Pipeline**	Deep (20+ stages)	Short (2–7 stages)
	Superscalar Execution	Superscalar, Out-of-Order	Scalar, In-Order
	Branch Prediction	Dynamic, speculative	None or prefetching
	Major Distinctions	Pipeline length, Instruction Parallelism, Prediction	
MEM	**Memory Hierarchy**	Deep (L1–L3)	None or Shallow (L1)
	Memory Management	Virtual memory (MMU)	Flat Memory Space (MPU)
	Data Location	Off-Chip Slow Memories	On-chip Fast Memories
	Main Memory Latency	Hundreds of Clock Cycles	Few Clock Cycles
	Memory Size	Gigabytes to Terabytes	Kilobytes to Megabytes
	Major Distinctions	Virtual Memory and Memory Hierarchy Complexity	
I/O	**I/O Interfaces**	Virtual MMIO / Host Controllers	Physical MMIO
	Interrupts	MSI via APIC/GIC	Direct Injection (Physical Wires)
	Major Distinctions	Number of Peripherals and Type of Interfaces	

In this section, we reviewed core computer architecture principles and contrasted APUs with MCUs, as summarized in Table 1. Building upon this comparison, we can now establish a concrete definition of an MCU. Essentially, MCUs are highly deterministic computing units designed to deliver consistent

outputs for a specific set of inputs under strict, predictable timing constraints. Typically, an MCU comprises a standardized, single-core[4] CPU[5] paired with a diverse, vendor-specific array of non-standardized peripherals (e.g., UART, SPI, timers, and DMAs). Unlike APUs, where processors from vendors like Intel or AMD are highly standardized and largely interchangeable, MCUs are inherently heterogeneous. The exact same Cortex-M CPU can power vastly different MCUs, each tightly coupled with custom peripherals and proprietary memory backends. Lacking virtual memory support, MCUs cannot run rich OSes such as Linux. Instead, they rely on MPUs for memory isolation and typically execute bare-metal software or a RTOS. Finally, the latest Armv8-M MCUs integrate hardware-backed TEEs via TrustZone-M, partitioning the system into isolated secure and non-secure execution environments.

2.2 Arm Cortex-M Architecture

Modern MCUs are shipped in the billions every year [38, 39], with a staggering projection of one trillion total accumulated shipments by 2035 [79]. Arm is the leader in this domain [43–45], maintaining the lion's share of the market with its Arm Cortex-M processor family. Additionally, Arm provides the most advanced and widely adopted security features for embedded devices, i.e., Arm TrustZone for MCUs (TrustZone-M) [6]. Consequently, this thesis focuses exclusively on Arm-based MCUs[6], the Arm Cortex-M family [5]. This section provides an overview of this processor family, detailing its core architectural and microarchitectural features, as well as its security primitives.

2.2.1 Arm Cortex-M family.

The Cortex-M family is one of Arm's three primary processor profiles, alongside the Cortex-A series for high-performance applications and the Cortex-R series for high-performance real-time processing. Unlike the Cortex-A processors typically found in mobile devices (and more recently in laptops and servers) or Cortex-R processors used in complex real-time systems, the Cortex-M family is specifically tailored for low-power, low-cost, and resource-constrained MCUs. Originally designed for use cases demanding high energy efficiency and deterministic execution, low-latency interrupt handling, these processors have seen widespread deployment across embedded systems. Currently, the Cortex-M lineup features 12 distinct processor models [80], spanning three architectural generations: *(i)* Armv6-M, *(ii)* Armv7-M, and *(iii)* Armv8-M. Table 2 provides an overview of the Cortex-M processor family, highlighting its primary architectural and microarchitectural characteristics.

[4] While less common, some modern MCUs feature dual-core configurations (e.g., the STM32H7, i.MX RT1170, and PSoC 64 series).

[5] Throughout this thesis, we will use MCU to refer to the entire SoC, CPU to the computational unit within a MCU, and a core to the processing units within the CPU.

[6] However, while we focus on Arm-based MCUs, in Chapter 4 we demonstrate that the microarchitectural side-channel exploited in this thesis is not confined to the Cortex-M family; it affects other non-Arm-based MCUs, including PIC- and RISC-V-based platforms.

Table 2: Summary of current Arm Cortex-M family (2026). T16/32, Thumb 16- and 32-bit instructions. Information gathered from Arm Cortex-M Processor Comparison Table [5], and Armv6/7/8-M technical reference manuals (TRMs) [2, 4, 81].

	CM0	CM0+	CM1	CM23	CM3	CM33	CM35P	CM4	CM52	CM55	CM7	CM85
Year	2009	2012	2007	2016	2004	2016	2018	2010	2023	2020	2014	2022
Arch.	VN	VN	VN	VN	H	H	H	H	H	H	H	H
ISA	Armv6-M	Armv6-M	Armv6-M	Armv8-M	Armv7E-M	Armv8-M	Armv8-M	Armv7-M	Armv8.1-M	Armv8.1-M	Armv7E-M	Armv8.1-M
Pipeline	3-stages	2-stages	3-stages	2-stages	3-stages	3-stages	3-stages	3-stages	4-stages	5-stages	6-stages	7-stages
Dual-Issue	✗	✗	✗	✗	✗	Limited T16	Limited T16	✗	Limited T16	Limited T16	T16/32	T16/32
Branch Predictor	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Inst. Restart	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Inst. Resume	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
I-TCM	✗	✗	0-1MB	✗	✗	✗	✗	✗	0-16MB	0-16MB	0-16MB	0-16MB
D-TCM	✗	✗	0-1MB	✗	✗	✗	✗	✗	0-16MB	0-16MB	0-16MB	0-16MB
FPU	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
I-Cache	✗	✗	✗	✗	✗	✗	2-16kB	✗	0-64Kb	0-64Kb	0-64Kb	0-64Kb
D-Cache	✗	✗	✗	✗	✗	✗	✗	✗	0-64Kb	0-64Kb	0-64Kb	0-64Kb
MPU	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
TZ-M	✗	✗	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓
Store Buffer	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Spec. Reads	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Spec. Execution	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Inst. Queue	✗	✓	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓

2.2.2 CPU Characteristics.

While the majority of Cortex-M processors feature a simplistic pipeline with minimal performance-oriented microarchitectural optimizations, the most powerful models, primarily from the latest Armv8-M architecture, incorporate some instances of advanced features, most notably superscalar execution. Currently, two Cortex-M processors are capable of dual-issuing instructions, i.e., the Cortex-M7 and the Cortex-M85. However, instruction-level parallelism is not exclusive to these high-performant CPUs. Mid-range Cortex-M processors, such as the Cortex-M33 and Cortex-M55, have the capability to dual-issue certain 16-bit Thumb instruction pairs. Arm refers to this process as *Instruction Folding*, which essentially enables these instructions to execute simultaneously without incurring extra clock cycles. Furthermore, several models implement branch prediction, using either static (Cortex-M52 and Cortex-M55) or dynamic (Cortex-M7, and Cortex-M85) prediction techniques. The dynamic predictors utilize a branch target address cache (BTAC) to predict branch outcomes based on execution history, similar to branch predictors found in APUs. To support these branch prediction capabilities, the Cortex-M7, Cortex-M55, and Cortex-M85 can speculatively perform memory reads. However, per Arm documentation [82], they do not support speculative execution (i.e., they do not propagate the results of speculatively executed instructions to subsequent operations).

Privilege Levels and Execution Modes. As illustrated in Fig. 8, Cortex-M family defines two privilege levels (privileged and unprivileged) to isolate the RTOS kernel from application tasks, alongside two execution modes (thread and handler) to separate exception handlers from regular code execution. Thread mode can operate in either privileged or unprivileged state, whereas handler mode is always privileged (i.e., controlled by the RTOS).

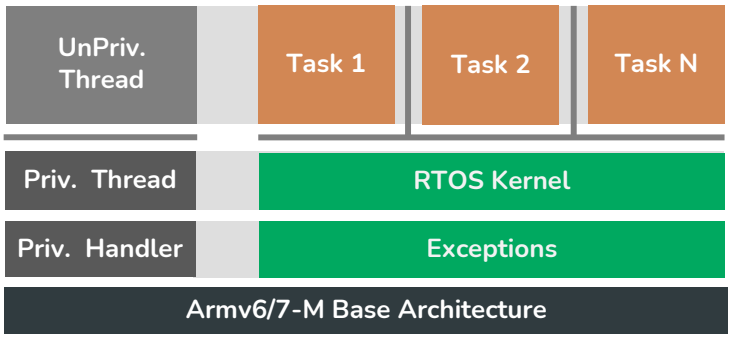


Figure 8: Overview of Cortex-M family process and execution modes (left) and the software stack leveraging those modes (right).

2.2.3 Memory Subsystem.

The Cortex-M memory system, as depicted in Fig. 9, is significantly simpler than those found in APUs, as detailed in Section 2.1.3. However, certain Cortex-M processors feature optional integrated microarchitectural components, such as caches and TCMs, to enhance performance. Built-in caches are typically reserved for high-performance MCUs that often interface with larger, slower external memories, such as the Cortex-M7, Cortex-M55, and Cortex-M85. In these models, caches hide memory latency by providing single-cycle access times for cached data. This performance gain comes at the cost of execution determinism due to potential cache misses, making caches best suited for general-purpose or soft real-time applications. For hard real-time applications deployed on these models, Arm provides a special type of memory, TCMs. These offer single-cycle access latencies similar to caches, but without the drawback of cache miss penalties. Consequently, TCMs are ideal for storing critical code and data subject to strict real-time constraints. Cache sizes can scale up to 64 KB, while TCMs capacities can reach up to 16 MB. Arm Cortex-M processors follow one of two distinct memory architectures: *(i)* Von Neumann (where instruction and data fetches share a single bus) or *(ii)* Harvard (where instruction and data fetches operate concurrently on separate buses). These architectures present different trade-offs between gate count and performance. The Von Neumann architecture is exclusively used in ultra-resource-constrained MCUs (e.g., Cortex-M0, Cortex-M0+, and Cortex-M23) to minimize hardware complexity and gate count. Conversely, the Harvard architecture is implemented in more powerful MCUs to maximize throughput, once it allows the CPU to fetch data and instructions simultaneously.

Memory Isolation. The majority of the Cortex-M family processors (excluding the Cortex-M0 and Cortex-M1) feature an MPU to enforce memory isolation and restrict access rights (cf. Table 2). The MPU regulates access based on several region attributes: *(i)* base address and size; *(ii)* access permissions (e.g., read-only, read-write, or no access); *(iii)* execution privilege (i.e., privileged or unprivileged access); and *(iv)* memory traits (e.g., cacheable, bufferable, shareable, or execute-never). MPUs[7] are used to

[7]The Cortex-M family implements two distinct MPU variants: one for Armv7-M and another for Armv8-M, which differ substantially, with the latter not being backward compatible. For further details, we refer the reader to Yiu's definitive guide on Armv8-M CPUs [83].

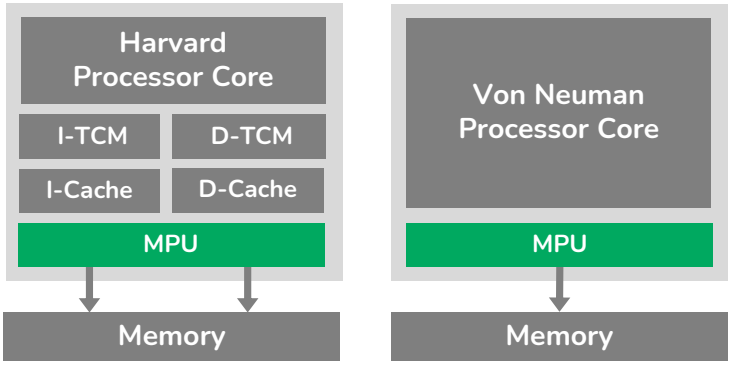


Figure 9: Overview of the two types of memory subsystem featured in the Cortex-M family. Left: the most powerful Cortex-M CPUs include I-/D-TCMs and I-/D-Caches, accessing memory over two distinct buses (Harvard architecture), one for instructions and one for data. Right: basic Cortex-M CPUs access memory via a single bus (Von Neumann architecture) and lack microarchitectural buffers. All accesses pass through an MPU to enforce memory isolation between privileged modes.

enforce task isolation and sandbox the execution of unprivileged applications. This prevents faulty or malicious software from corrupting or tampering with the data and code of other tasks, as well as the RTOS kernel itself.

2.2.4 Interrupt Logic.

To minimize interrupt latency, all Cortex-M processors implement *automatic stacking*, a hardware mechanism that automatically saves and restores the CPU context upon interrupt entry and exit without requiring software intervention. The Arm Cortex-M family supports several stack frame layouts, ranging in size from 8 to 34 registers, depending on the current execution context. Specifically, there are four stack frame types, which are determined by the status of the floating-point unit (FPU) (enabled or disabled) and the security state (secure or non-secure) of the interrupted context. If the interrupt source shares the same security state as the executing context, the processor stacks only the caller-saved registers (R0–R3, R12, LR, and xPSR), resulting in the minimum stack frame size of 8 data words. Conversely, during cross-world interrupts (e.g., when an non-secure interrupt arrives while the secure world is executing), the CPU saves the entire integer register file to prevent potential data leakage between security states. Furthermore, if the FPU is active, the stack frame expands by 17 data words to accommodate the floating-point register file. This additional state can be saved unconditionally or, if lazy stacking is enabled, deferred until the interrupt service routine (ISR) actually executes an FPU instruction. Although each stack frame format exhibits a deterministic interrupt latency, this latency varies depending on the required stack frame type and the underlying CPU architecture. In Harvard-based MCUs, automatic stacking and interrupt vector fetching occur concurrently, thereby minimizing the overall interrupt latency. Conversely, in Von Neumann architectures, the interrupt vector is fetched only after the CPU completes the automatic stacking process, resulting in increased latency.

Interruptible Instructions. To further reduce interrupt latency, the Cortex-M architecture introduces

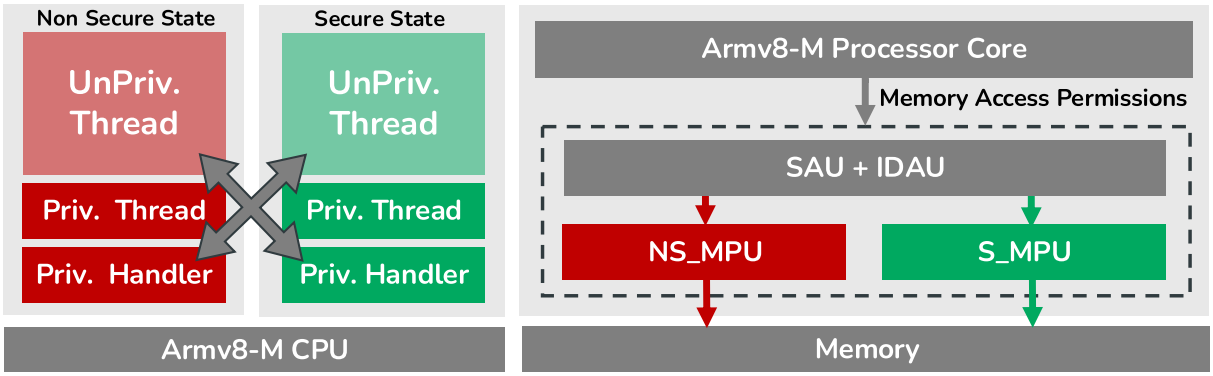


Figure 10: Armv8-M modes and security states (left) and Armv8-M memory protection controllers (right).

a microarchitectural feature that allows multi-cycle load and store instructions to be interrupted mid-execution. This theoretically enables the CPU to service incoming interrupts with minimal timing penalties. Arm refers to these as interrupt-continuable instructions (ICIs) [2, 4]. When an ISR completes, an interrupted ICI instruction exhibits one of two behaviors: *(i)* it either resumes execution from the exact point of interruption or *(ii)* restarts execution from scratch. This feature is only fully implemented in a subset of the Cortex-M family. Entry-level processors, such as the Cortex-M0, Cortex-M0+, Cortex-M1, and Cortex-M23, only support the restart-from-scratch behavior (cf. Table 2). This design choice avoids saving the mid-execution instruction context, thereby minimizing hardware complexity, gate count, and power consumption.

2.2.5 Arm TrustZone-M.

In the latest generation (Armv8-M), Arm introduced support for TrustZone-M, as illustrated in Fig. 10. This feature enables the deployment of TEEs on MCUs, commonly referred to as uTEEs. While TrustZone bears the same name across Arm Cortex-A and Cortex-M platforms, the TrustZone-M implementation is fundamentally different, having been designed from the ground up specifically for MCUs [6]. Concretely, Armv8-M extends the earlier Armv6/7-M architectures with two new security states (worlds): secure and non-secure. These states are orthogonal to existing processor modes and coexist with the standard two privilege levels (privileged and unprivileged) and two execution modes (thread and handler). Moreover, Armv8-M introduces new memory protection controllers, namely the SAU and IDAU. These components complement the existing MPU by enforcing isolation between secure and non-secure memory regions.

To enable cross-world communication, Armv8-M defines a new memory region, the Non-Secure Callable (NSC) region, which serves as the gateway through which non-secure code can invoke secure-world services. While this region is accessible to both worlds, the non-secure world can only execute the secure gate (SG) instruction within it. Executing SG transitions the processor to the secure world and establishes a secure entry point; any attempt to enter the secure world via other means triggers a security fault. In contrast to TrustZone-A, TrustZone-M performs all world transitions entirely in hardware, removing the need for a secure monitor. Consequently, when secure-world execution is interrupted by an non-secure

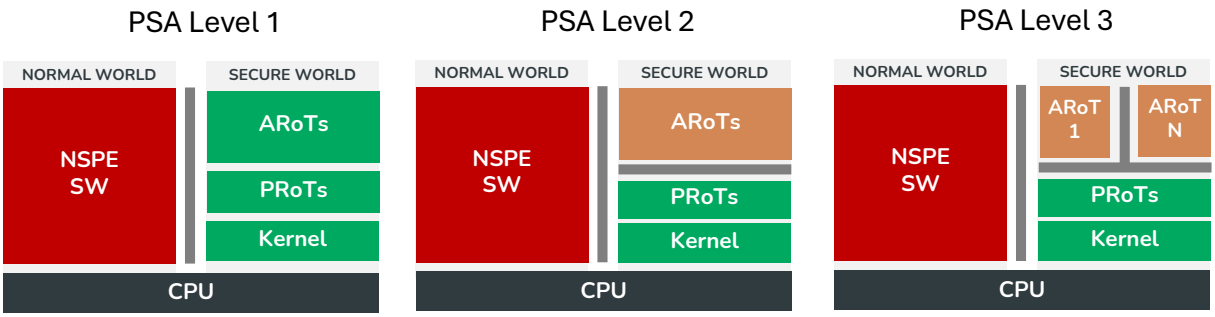


Figure 11: TF-M implemented PSA isolation levels as per the PSA security model guidelines [48].

world interrupt, the processor transitions directly from the secure context to the non-secure ISR. We exploit this behavior in one of the attacks introduced in this thesis (cf. Chapter 6).

2.2.6 Secure Software Stack.

Arm maintains an open-source reference implementation of secure world software for Armv8-M MCUs, known as TF-M [46], which adheres to PSA guidelines [47]. As of this writing, TF-M supports a wide array of platforms (18) across multiple vendors (8). Although there are other uTEEs, TF-M has become the *de facto* standard. Alternative solutions are often closed-source, such as ProvenCore-M [84] and Trustonic Kinibi-m [66], requiring royalty payments, or academic initiatives with limited industry adoption, such as mTower [85] and uTango [86].

TF-M is built upon three main components: *(i)* secure boot software, supported by the MCUBoot open-source bootloader [87]; *(ii)* the TF-M kernel, designed by Arm to act as the secure partition manager (SPM); and *(iii)* privileged and unprivileged secure services (conceptually similar to TAs), referred to as Privilege Root of Trusts (PRoTs) and Application Root of Trusts (ARoTs), respectively. The ARoTs serve as the deployment environment for third-party applications, whereas PRoTs are used to implement core TF-M secure services, such as attestation and cryptographic primitives, and are always part of the TF-M TCB. The TF-M kernel manages these secure services in an OS-like fashion. Upon a service request from the Non-Secure Processing Environment (NSPE) or another secure service, the SPM sends a signal to the target secure partition to execute. All secure services remain in an idle state until an remote procedure call (RPC) is issued. Additionally, the TF-M SPM is responsible for interrupt management; it always traps and emulates interrupts, ensuring they are never directly injected into the secure partitions.

Regarding isolation, TF-M adheres to the PSA security model and implements three of the four isolation levels defined by the standard [48], as depicted in Fig. 11: *(i) PSA Isolation Level 1:* establishes two security domains, where the Secure Processing Environment (SPE) is protected from the NSPE, and all secure world software is part of the TCB; *(ii) PSA Isolation Level 2:* establishes three security domains, where in addition to the Level 1 isolation, the PRoTs and the TEE kernel are isolated from the ARoTs; and *(iii) PSA Isolation Level 3:* establishes three or more security domains, where the ARoTs are individually sandboxed and isolated from one another, preventing malicious applications from tampering with other

ARoTs. TF-M uses the SAU to enforce isolation between the SPE and NSPE software (PSA Level 1), and uses the secure world MPU to isolate PRoTs from ARoTs (PSA Level 2) as well as to separate individual ARoTs (PSA Level 3). These protection levels can be configured according to application requirements, bounded only by the underlying hardware capabilities of the target board.

2.3 Side-Channel Attacks "101"

In this section, we introduce side-channel attacks, describing their mechanisms and contextualizing them within the broader landscape of security threats. This serves as a high-level overview to help the reader understand this class of attacks. In Chapter 3, we delve deeper into side-channel attacks, specifically focusing on the ones researched in this thesis, i.e., software-based microarchitectural attacks.

2.3.1 Side-Channel Attacks

Traditional attacks circumvent system security through *(i)* fault injection, *(ii)* exploiting design flaws, or *(iii)* leveraging zero-day vulnerabilities to directly access the target *data* of interest. In contrast, side-channel attacks operate under restrictive threat models without exploiting any specific software bug or functional design error. Instead, they extract *metadata* from the side effects of computation itself, which, through complex signal processing and statistical algorithms, can be used to recover the underlying data. Side-channel attacks leverage a fundamental physical principle: any energy expended in performing a computation produces both useful work (energy directed toward the intended action) and wasted byproducts (energy dissipated through interaction with the surrounding environment). By positioning themselves in proximity to the victim computation, side-channel attackers can harness these wasted byproducts, such as heat, power variations, electromagnetic emanations, or sound, to gather information about computations operating over confidential data. When a measurable byproduct of computation correlates with the data being processed, we designate this leakage as a side-channel[8].

The literature demonstrates creative attacks leveraging multiple byproducts to extract secrets. Researchers have used thermal cameras to perform thermal attacks, where heat traces from authentication operations can reconstruct passwords [30]. Others have leveraged light to retrieve keystrokes and sound due to light-dimming effects inflicted by vibrations [88–90]. Acoustic attacks [17–19] extract secret information from sound emanations during computation. Notably, Genkin et al. [18] recovered full 4096-bit RSA keys by analyzing acoustic signatures of decryption operations. These examples illustrate the breadth

[8]Throughout this thesis, we distinguish between side-channel attacks and covert-channel attacks, which operate under similar leakage mechanics but differ fundamentally in intent. A side-channel attack exploits *unintentional* leakage from normal system operation, whereas a covert-channel uses an attacker-controlled mechanism to *intentionally* encode and transmit secrets. The latter is used when an attacker is sandboxed without a direct communication channel; the attacker establishes a covert-channel by modulating the byproducts of computation (e.g., power consumption, thermal dissipation, cache state) to encode information and communicate with the external entity.

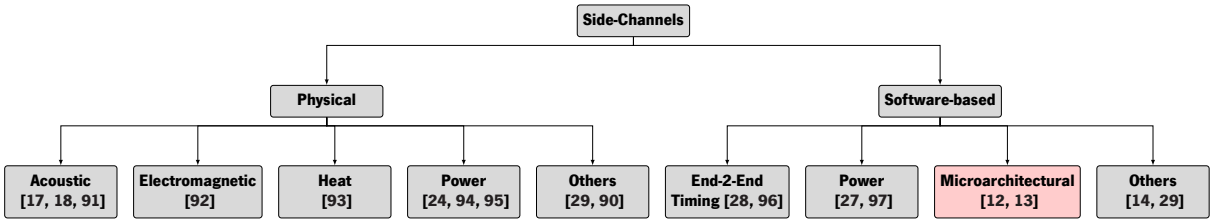


Figure 12: Taxonomy of side-channel attacks, adapted from [29]. We group local and vicinity attacks into a single class, i.e., physical side-channels. This taxonomy highlights the most common types of physical and software-based side-channels; we refer the interested reader to [29] for a comprehensive taxonomy. Red highlights the class of side-channels that we focus on in this thesis.

of observable byproducts attackers can leverage. For a comprehensive taxonomy, we refer the reader to the systematic survey by Spreitzer et al. [29].

2.3.2 The Power of Software-Based Side-Channel Attacks

There are two classes of side-channel attacks, as depicted in Fig. 12: *(i)* physical side-channel attacks and *(ii)* remote side-channel attacks, also known as software-based side-channel attacks[9]. Physical side-channel attacks require the attacker to be close to the victim or even have direct physical access to the target device to place measurement probes and capture traces such as heat, power, sound, or electromagnetic emanations [29]. In contrast, software-based side-channel attacks rely on malicious code running on the target device itself to gather side-channel traces remotely. This fundamentally changes the attacker model, from a local adversary that must physically reach the device to a remote adversary that can deploy an exploit over a network connection. As a result, the potential attack surface expands substantially while the practical barrier to launch the attack is reduced.

Historically, physical side-channel attacks were difficult and expensive to mount because they typically required specialized laboratory equipment, e.g., high-end oscilloscopes and custom probes. For instance, an attack on a Google Titan security key, while extremely powerful, required equipment costing more than €10k[10] [99, 100]. This does not make physical side-channel attacks impractical, but it does constrain them primarily to highly motivated attackers, such as corporations seeking trade secrets or nation states conducting espionage.

Software-based side-channel attacks, by contrast, are often easier to deploy because the attacker needs only to execute malicious software on the target platform to retrieve secret-dependent traces. With the proliferation of Internet-connected personal computers, smartphones, and embedded IoT devices, this has become a serious and highly realistic threat model. These devices are increasingly used for high-stakes computations, such as online banking, cryptocurrency management, and encrypted communications, which further raises the impact of successful remote attacks.

[9] We will use this term from now on throughout this thesis.

[10] We note that some attacks can be mounted with far less expensive equipment, such as the ChipWhisperer toolkit [98]. Nevertheless, even in these cases the attacker must still acquire dedicated hardware, which raises the engineering and financial bar to implement the attack.

Software-based Microarchitectural Attacks. Software-based side-channel attacks encompass all side-channel leakage that can be observed and exploited entirely from software. This includes end-to-end timing attacks [96, 101–104], attacks leveraging microarchitectural interference [61, 105–112], and even software-observable power variations [27, 113, 114]. Among these attack classes, we focus on software-based **microarchitectural** side-channel attacks (cf. Fig. 12), i.e., attacks that exploit timing variations caused by interference on shared microarchitectural components. This class of attacks has been extensively studied on APUs, where prior work has shown that multiple microarchitectural components can be exploited to mount successful real-world attacks. Prominent examples include attacks targeting the cache [104, 115–120], the address translation [121–126], and branch predictors [34–36, 127–129]. We review this class of attacks in more detail in Chapter 3.

Trusted Execution Environments, A Paradoxical Driver. The threat model of TEEs (cf. Section 2.1.5) assumes that malicious code may safely coexist on the same device as security-sensitive modules because both domains are architecturally isolated. Paradoxically, this very deployment model has become a key enabler of software-based side-channel attacks [130]. While the hardware-enforced isolation of TEEs prevents untrusted code from directly reading trusted data, it does not prevent the side effects of trusted computation from propagating through shared hardware resources and becoming observable by a colocated software entity. Consequently, although TEEs are highly effective against architectural attacks, they remain susceptible to microarchitectural leakage. This makes software-based attacks particularly attractive. Unlike physical side-channels, their cost is largely bounded by the engineering effort required to develop the exploit, and they can be deployed remotely against a significantly larger population of potential victims.

2.3.3 The Root Cause: Sharing is not Caring.

Modern computing units meet growing computational demands by integrating increasingly sophisticated microarchitectural features, such as branch predictors and internal buffers (cf. Section 2.1). These structures are shared across multiple software entities and, with the advent of TEEs, often remain shared even across isolated domains that may host mutually distrusting code. As a result, the execution of one entity changes the microarchitectural state of the shared component, creating observable side effects for other entities that use the same resource. An attacker can correlate these side effects with the victim's underlying data- or control-flow and thereby infer sensitive information. Kocher [28] first exposed the security consequences of this phenomenon in 1996 by showing that secret-dependent timing variations in RSA implementations can reveal the private key.

Type of Sharing. Microarchitectural components can be shared in two fundamental ways: *(i)* spatial sharing, where the victim and attacker execute concurrently, or *(ii)* time sharing, where they use the same component sequentially. Under spatial sharing, the attacker observes the shared microarchitectural component state while both entities are active. Under time sharing, the victim and attacker access the

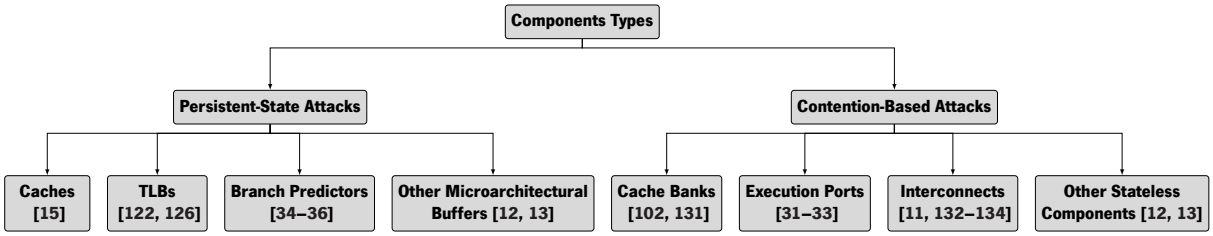


Figure 13: Types of microarchitectural components used in by persistent-state (left) and contention-state attacks (right).

component in different time slots. This distinction matters because the exploitability of microarchitectural leakage depends on whether the component preserves state over time.

Types of Microarchitectural Components. Microarchitectural components fall into two broad classes, as systematized in Fig. 13: *(i)* those that preserve their state over time and *(ii)* those that do not. Attacks targeting state-preserving components can exploit both spatial and time sharing. These attacks are designated as persistent-state attacks[11]. Persistent-state attacks modify a microarchitectural component's state, a change which persists even after the victim finishes execution. Prominent examples include attacks on the cache [104, 115–120], address translation [121–126], and branch predictors [34–36, 127–129]. These attacks exploit the limited capacity of microarchitectural components, where data already stored on the component must be evicted to accommodate newly incoming data.

Conversely, attacks on non-state-preserving components require spatial sharing once the attacker must observe leakage while it is being created. These attacks are designated as contention-based attacks[12]. Contention-based attacks [132] exploit transient information that leaves no persistent microarchitectural footprint, consequence of multiple software entities competing for the same shared resource with limited bandwidth. Prominent examples leverage cache banks [102, 131], execution ports [31–33], and bus interconnects [132–135].

2.3.4 Attacks' Phases

Regardless of the targeted microarchitectural component, sharing method, or attack type, a microarchitectural attack proceeds in three major phases, as illustrated in Fig. 14. First, during the *attack setup* phase (❶), the attacker prepares the microarchitectural context of the CPU either to interfere with the victim's execution, in contention-based attacks, or to prepare the shared microarchitectural state for later measurement, in persistent-state attacks. Second, during the *victim execution* phase (❷), the attacker either invokes the victim via an RPC, a common pattern in TEEs deployments, or waits for it to be scheduled by lower-level firmware. The attacker then captures the effect of the victim's execution on the shared microarchitectural component, either concurrently under spatial sharing or after victim execution under

[11] Some authors use the terms stateful or residual-state attacks interchangeably.

[12] Some authors use the terms transient-state or stateless attacks; however, transient-state is easily confused with transient execution attacks [14, 16], so we explicitly avoid that terminology.

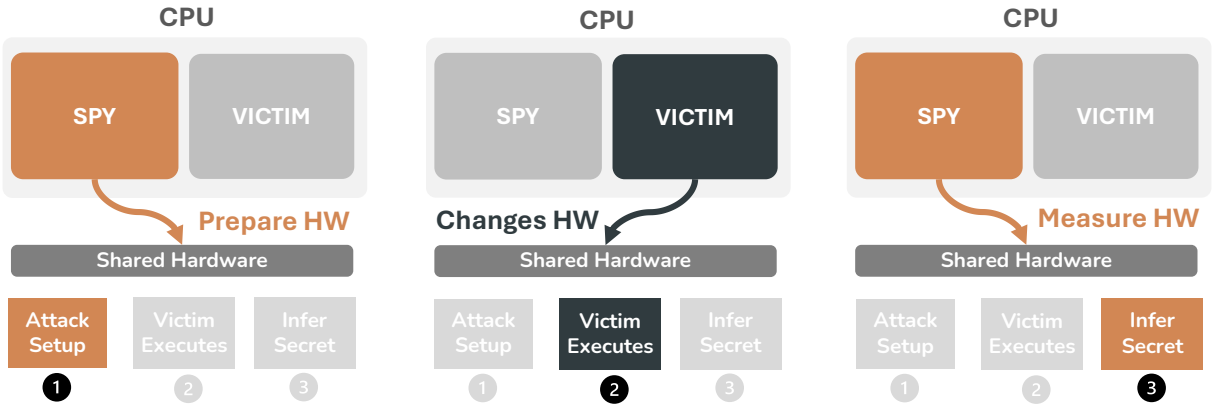


Figure 14: Microarchitectural attack phases: the attacker prepares the hardware to a known state (❶); the victim executes and leaves microarchitectural traces (❷); the attacker measures the hardware state and detects changes (❸) compared to the first phase (❶), inferring the secret.

time sharing. Notably, time sharing is not always feasible, which forces the attacker to rely exclusively on concurrent measurements, i.e., to mount a contention-based attack. Finally, in the *infer the secret* phase (❸), the attacker analyzes the collected side-channel traces, which are obtained either during active victim execution or by probing the shared microarchitectural state after the victim finishes execution.

3

State Of The Art

In this chapter, we review the literature on software-based microarchitectural attacks, focusing on both established attack vectors and emerging threats on resource-constrained devices. Firt, in Section 3.1, we survey the extensive body of work on microarchitectural attacks targeting APUs. Then, in Section 3.2, we examine the comparatively limited set of attacks demonstrated on MCUs. Finally, in Section 3.3, conduct a gap analysis of attacks most relevant to this thesis and positions our contributions within the state of the art.

3.1 Software-based Microarchitectural Attacks on APUs

Given the extensive body of work on software-based microarchitectural attacks on APUs, we focus on three attack classes that together provide a broader view of the microarchitectural attack landscape. We begin with cache attacks (Section 3.1.1), the most extensively studied attack class and a foundation for many subsequent techniques. While work in this area dates back to 1996 [28] and cache attacks have since been exploited multiple times [67, 101, 104, 115, 117–119, 124, 136–146], new attacks and exploitation techniques keep emerging (2025 and 2026) [61, 105–112]. We then review controlled-channel attacks [147] (Section 3.1.2), a subset of microarchitectural attacks that is closely related to the attack class we study in this thesis. These attacks enable an attacker to obtain deterministic and noiseless traces of the victim's execution. Finally, we review transient execution attacks [14, 16] (Section 3.1.3). While transient execution attacks do not directly apply to MCUs, as they rely on sophisticated microarchitectural features not available in these devices, mainly speculative execution and OoO execution, we still review them since they are among the most powerful software-based side-channel attacks unveiled to date.

3.1.1 Cache Attacks

Cache attacks represent the most extensively studied class of software-based microarchitectural attacks and constitute the foundation upon which many subsequent attack techniques are built. These attacks exploit the inherent performance characteristics of the memory hierarchy, where memory access latency

depends critically on whether data is cached. Modern processors feature multiple cache levels, small, fast memories that buffer recently accessed data to hide the latency of slower, lower-level memories (cf. Section 2.1.3). Caches rely on the principles of temporal and spatial locality, which are based on the empirical observation that recently accessed memory is likely to be reused in the near future (temporal locality) and adjacent memory locations are likely to be accessed sequentially (spatial locality). When a requested piece of data, i.e., a cache line, is already buffered in these fast memories, a *cache hit* occurs, and the CPU experiences a fast memory access. Conversely, when the cache line has not yet been allocated in the cache, a *cache miss* occurs, resulting in a significantly slower memory access.

While this greatly improves performance, it introduces an inherent security cost by making memory accesses non-constant time by design. All CPU memory accesses go through caches, and every level in the memory hierarchy experiences a different latency. While some caches are private to a single core (e.g., L1 Instruction and Data caches), others are shared across cores (e.g., L2) or even across the entire system (e.g., the Last Level Cache, or LLC). Furthermore, caches preserve their microarchitectural state after the victim's execution, and these caches are often shared with other software entities. Thus, if the victim code performs secret-dependent memory accesses, an external observer can query the microarchitectural states of the shared cache and correlate those states with the victim's execution. If those states form patterns correlated with the processed secret, the attacker can use them to retrieve confidential data.

While there is a plethora of attack techniques that can be leveraged to derive information from cache microarchitectural states (e.g., Prime+Probe [101], Flush+Reload [115], Evict+Reload [110, 117, 119]), they all follow the same underlying attack methodology (recall Section 2.3.4). First, during the *attack setup* phase, the cache is placed into an attacker-known microarchitectural state. Then, the victim executes and modifies those microarchitectural states. Finally, the attacker infers the secret by probing the cache and comparing its microarchitectural state before and after the victim's execution to detect the cache lines that were accessed by the victim. Next, we explain the two most prominent cache attack techniques, namely Prime+Probe [104] and Flush+Reload [115].

Prime+Probe. Prime+Probe, first introduced by Osvik et al. [104], is a powerful technique capable of detecting which cache sets a victim has accessed by monitoring the time it takes to load a cache-sized buffer before and after the victim's execution, as illustrated in Fig. 15. During the prime phase, this buffer is loaded into the cache, occupying all available cache sets. When the victim executes, it evicts some of these attacker-controlled cache lines. Subsequently, when the attacker probes the cache by reloading the buffer, the evicted cache lines will experience cache misses (slower access times), while the other will result in cache hits (faster access times). By carefully accessing data mapped to each cache set and measuring the associated memory access latency, the attacker learns exactly which cache sets were used by the victim. This information can then be leveraged to infer the victim's data access patterns or control-flow, ultimately leading to the extraction of sensitive secrets. Prime+Probe does not require any special privileged operations or shared memory, allowing it to be mounted from entirely unprivileged user-space applications. This is the attack of last resort when no advanced features (e.g., cache flush instructions) are

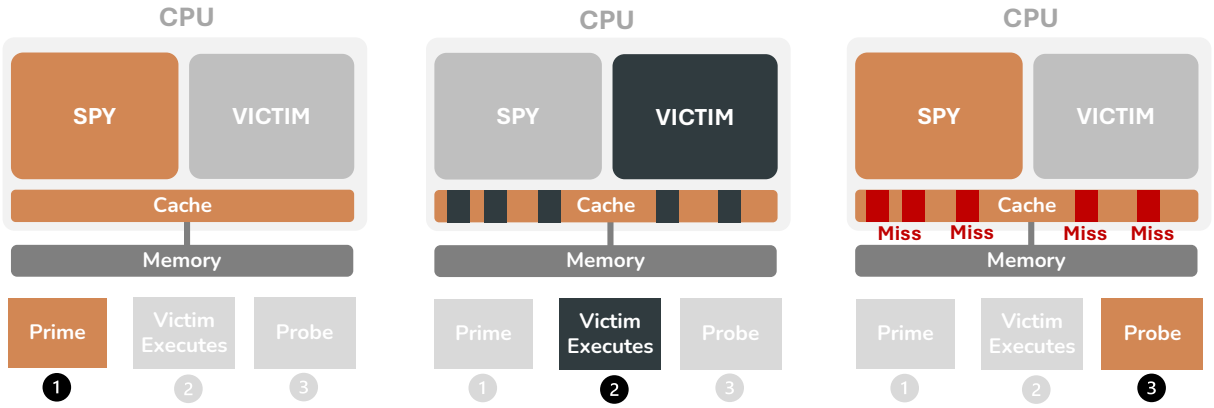


Figure 15: Illustration of a Prime+Probe cache attack. First the spy primes the cache by loading a cache-size buffer (❶). Then the victim executes and evicts cache lines (❷). Lastly, the spy probes for cache misses (❸), revealing which cache lines the victim accessed.

available to the attacker. However, it comes with a significant challenge: to perform the prime phase, the attacker must construct eviction sets. This task is significantly complicated by modern caches that feature advanced, often undocumented, cache replacement policies[1]. While traditional policies like Least Recently Used (LRU) are relatively easy to target, modern APUs have much more sophisticated and less predictable replacement mechanisms, complicating the construction of reliable eviction sets. Prime+Probe has been leveraged to recover secret keys by exploiting multiple cryptographic algorithms, including RSA [101, 118, 136, 148–152], AES [103, 104, 138, 153–157], and ElGamal [118, 152, 158], as well as more generic, non-cryptographic applications [119, 159, 160].

Flush+Reload. Flush+Reload, introduced by Yarom and Falkner [115], is generally easier to mount than Prime+Probe. However, it requires the attacker to have access to cache flush instructions, which on some CPU architectures, such as Arm, are not available to unprivileged user-space applications. Furthermore, this technique is only applicable if the attacker and the victim share virtual memory. As illustrated in Fig. 16, Flush+Reload uses a flush instruction to evict a specific address within a shared library, such as `libc`, from the cache hierarchy. The attacker then lets the victim execute and subsequently reloads the shared address, measuring the access time. If the reload results in a cache hit, the attacker learns that the victim accessed the data mapped to that exact cache line. In contrast to Prime+Probe, which only leaks the accessed cache set, Flush+Reload can pinpoint the exact cache line accessed by the victim. However, the requirements of shared virtual memory and flush instructions restrict the applicability of Flush+Reload to a narrower set of scenarios than Prime+Probe. Flush+Reload has been widely exploited in systems that use shared memory across security boundaries, successfully targeting cryptographic implementations such as RSA [115], AES [137, 138, 161], and ECDSA [162, 163], as well as a variety of generic, non-cryptographic applications to infer user behavior, such as keystrokes [117, 119].

Other Cache Attack Techniques. Multiple variants of Prime+Probe [104] and Flush+Reload [115]

[1]When all ways of a cache set are occupied, the cache must make space for new cache line by evicting an existing one. The algorithm used to determine which way to evict is known as the cache replacement policy.

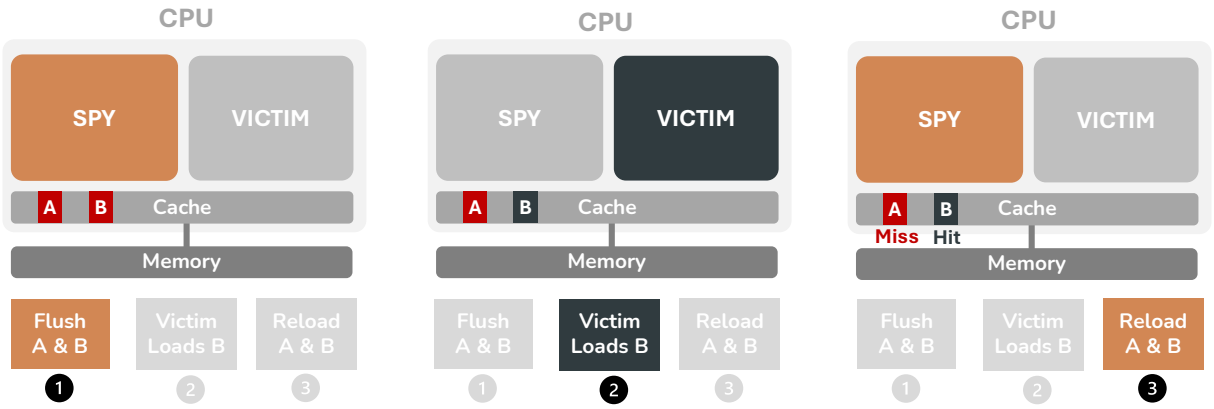


Figure 16: Illustration of a Flush+Reload cache attack. The attacker flushes shared cache lines A and B (❶), the victim accesses B (❷), and the attacker reloads both lines (❸). A cache hit on B reveals that the victim accessed it.

have been proposed over the years. Evict+Time [96, 101–104], an end-to-end timing attack introduced alongside Prime+Probe, evicts cache lines of interest and then measures the total execution time of the victim. If the evicted cache line was accessed by the victim, the attacker will theoretically observe an increase in the victim's overall execution time. In Prime+Count [164], the attacker focuses only on the number of cache sets used by the victim. The authors used this technique to build a cross-world covert channel on Arm TrustZone-A. In Flush+Flush [110, 116, 119], the authors leverage timing differences in the flush instruction itself, which vary depending on the cacheability state of the target address. By avoiding the need to reload the shared memory, this attack becomes significantly faster. Evict+Reload [110, 117, 119] substitutes the flush instruction with an eviction set, similar to Prime+Probe, making the Flush+Reload attack applicable to more restricted environments while still targeting specific shared cache lines. In Prime+Abort [165], instead of timing the probe phase, the attacker uses the Intel TSX abort feature. When the victim touches a cache line previously loaded into a TSX transaction by the attacker, the transaction implies an abort, immediately notifying the attacker. This technique removes the requirement for a high-resolution timer. There are other techniques, such as Reload+Reload [109] and Reload+Refresh [166], that eliminate the need to evict data from caches altogether, allowing these attacks to evade online cache attack detection mechanisms. Most recent techniques like Cohere+Reload [105] can even leak encrypted memory from AMD SEV-SNP by exploiting its cache coherency mechanisms.

3.1.2 Controlled-Channel Attacks

Controlled-channel attacks [53, 68, 147, 167–169] are a subset of software-based microarchitectural attacks that leverage deterministic and (largely) noiseless channels to extract a victim's secret in a single run. These attacks typically leverage attacker-controlled events, such as page faults [147], to gather deterministic traces about the victim's execution, as illustrated in Fig. 17. These attacks leverage a key characteristic of the Intel SGX TEE: the untrusted OS can use virtual memory mechanisms to manipulate the page tables used by the SGX enclave, and revoke or change the access rights to enclave pages. Then,

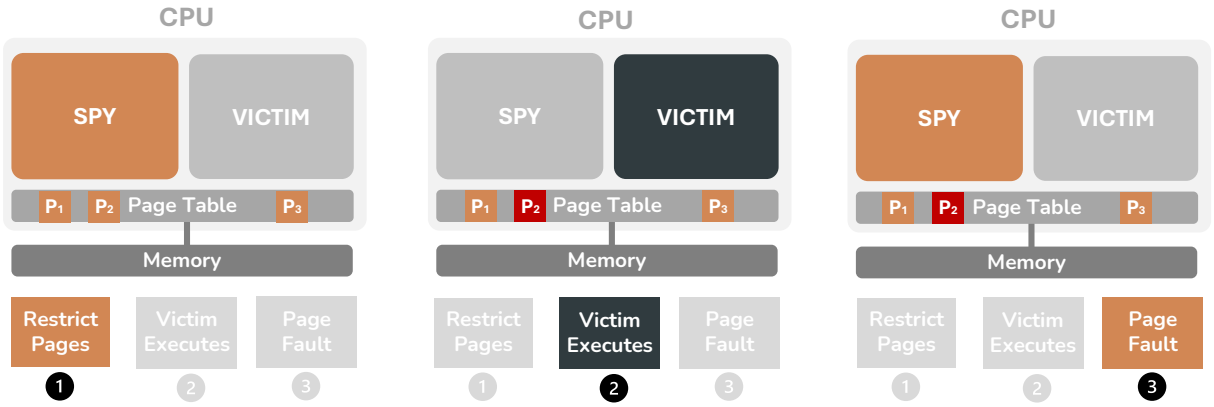


Figure 17: Illustration of a controlled-channel attack. The spy (malicious OS or hypervisor) revokes access rights to victim code and data pages (❶). Then, when the victim accesses a restricted page (❷), a page fault is generated, informing the attacker which page was accessed (❸), i.e., page P_2 in this example.

when the enclave executes code or accesses data from those pages, the untrusted OS will be notified (for instance, via a page fault) that those pages were accessed. This allows an attacker in control of the untrusted OS to deterministically leak data- or control-flow information of the victim enclave code, and thereby leak secrets.

The concept was first introduced by Xu et al. [147], who leveraged page faults to retrieve enclave control flow with page-size granularity, successfully extracting full text documents and JPEG images from widely used word processing applications. Following this seminal work, researchers exploited page tables to detect page accesses by probing the Accessed and Dirty page-table bits [167, 168]. To avoid detection through high-frequency page fault monitoring, other researchers flush the TLB [167] and measure the caching behavior of untrusted page-table memory, thereby circumventing page faults altogether. While these controlled-channel attacks provide deterministic and noiseless traces, their granularity is limited to 4 KiB (the page size). Subsequent work reduced this granularity to single-instruction [60, 68] and even single-byte [53, 169] precision by combining page faults with deterministic interrupt counts [60, 68] or by leveraging other architectural faults, such as segmentation faults [169] and misaligned data accesses [53].

Controlled-channel attacks have also proven effective against newer TEE technologies like Intel TDX. Researchers demonstrated that, even with coarse page-level traces, an attacker could use page faults to extract a 4096-bit RSA private key from MbedTLS [170] executing inside a confidential virtual machine (CVM). Furthermore, controlled-channel attacks have been a key enabler of single-step frameworks, such as SGX-Step [58], SEV-Step [62], and TDX-Step [59–61]. These frameworks leverage page faults to iteratively advance the victim's execution and use page-table accessed bits to monitor its progress, serving as a fundamental building block for numerous subsequent attacks [65]. While these single-step frameworks are not controlled-channel attacks *per se*, they heavily rely on controlled-channel primitives to reliably single-step through generic code and accurately detect precise victim events, such as specific executed instructions.

Controlled-channel attacks are exceptionally powerful due to their ability to extract secrets deterministically from a single execution trace. This capability significantly expands the set of vulnerable victims. First, they can target non-cryptographic applications, such as password entry or keystroke monitoring, where execution is ephemeral and the attacker obtains only a single opportunity to capture the secret. We demonstrate two attacks of this nature in Chapters 5 and 6 [8]. Second, they evade mitigations designed to defeat multi-trace side-channel attacks in cryptographic implementations, such as nonce-based input blinding, and can attack algorithms that process a secret only once, including key generation routines. We present an attack demonstrating this in Chapter 7 [11].

3.1.3 Transient Execution Attacks

Spectre [171] and Meltdown [172] exposed the practical security implications of leveraging microarchitectural elements together with performance mechanisms, such as speculative execution and OoO execution, to leak sensitive information. Since then, numerous transient execution attacks have bypassed security and isolation boundaries across processors and technologies from mainstream CPU vendors, including Intel [69, 121, 173–175], AMD [14, 146, 176], and Arm [67, 142, 177, 178]. While transient execution attacks share similarities with traditional software-based microarchitectural attacks, they constitute a distinct attack class. Traditional microarchitectural attacks typically leverage the side-effects of concurrent computations competing for shared hardware to leak metadata, which can yield the underlying data through post-mortem analysis. In contrast, transient execution attacks can directly leak the underlying data by transiently encoding it in stateful microarchitectural components, such as caches.

This leakage occurs due to the long streams of instructions that are speculatively executed on modern OoO processors. These processors reorder instructions that lack apparent data dependencies to increase throughput and avoid unnecessary pipeline stalls. These reordered instructions are executed speculatively, but they are only architecturally committed in their original program order. When there is a misspeculation (as leveraged by Spectre [171]) or an exception (as leveraged by Meltdown [172]), the processor flushes the pipeline and reverts the operations performed by the speculatively executed instructions. Because these instructions were never architecturally committed, the misspeculation leaves no directly observable architectural effects, other than a modest performance penalty. However, speculative and OoO execution still perturb the microarchitectural state. An attacker can subsequently observe this state by probing the affected microarchitectural components, most notably the cache.

In contrast to traditional software-based microarchitectural attacks [28], which can only infer a target program's data or control flow from traced metadata, transient execution attacks manipulate transient execution to directly leak the contents of a processor register used by a speculatively executed instruction. This capability allowed Meltdown [172] to directly read arbitrary kernel memory and Spectre [171] to leak memory across isolation boundaries. Although Meltdown and Spectre exploit different microarchitectural behaviors to compel the CPU into leaking data, exceptions and misspeculation, respectively, their underlying attack methodologies are remarkably similar. Both attacks induce an out-of-bounds memory read



Figure 18: Illustration of a transient execution attack. This example, based on Spectre attack [171], captures the core phases common to such attacks. The attacker first prepares the microarchitecture (❶), in Spectre by mistraining the branch predictor, then triggers an illegal memory access (❷) that loads a secret value into CPU registers; these registers are subsequently used to index an attacker-controlled array in a transient execution window. When the CPU detects the misprediction (Spectre) or fault (Meltdown), it flushes the pipeline (❸). However, the cache remains intact, preserving the side effects of the transient window, which are later recovered through a cache-based side-channel attack (❹). In this example, the illegal access (`S=X[3]`) reads value 2, touching attacker-controlled array cache line A3.

that is executed transiently. The CPU detects this unauthorized or misspeculated read only later in the transient execution window, after the speculatively executed instructions have already brought the target memory into the cache. By using a cache-based covert-channel gadget, the attacker can then extract the secret by observing which cache line was accessed.

Figure 18 illustrates how this attack is executed. First, the attacker prepares the microarchitecture to induce speculative execution (❶), for example by training the branch predictor. Then, the attacker triggers an unauthorized memory access by reading from an address to which they lack access rights (❷). This instruction and the subsequent ones are transiently executed at the microarchitectural level but never architecturally committed. While these transiently executed instructions leave no architectural effects observable to software, the attacker carefully constructs a gadget that encodes the secret in the cache during the transient window, which can subsequently be recovered and made architecturally visible through a simple cache attack (cf. Section 3.1.1). To accomplish this, the attacker uses the transiently read secret value as an index into an attacker-controlled buffer. This buffer contains 256 elements, each encoding a possible byte value and each mapped to a distinct cache line. Accessing one of the 256 elements brings the corresponding cache line into the cache, directly leaking the secret value (❹). For example, as shown in Fig. 18, if the secret value is 2, the third element of the indexed buffer is fetched, bringing cache line 3 (A3 in Fig. 18) into the cache. Although the CPU flushes the pipeline (❸) upon detecting a misspeculation (Spectre-type attacks) or fault (Meltdown-type attacks), the microarchitectural side effects of the speculation are already encoded in the cache state, which the attacker can later recover.

Following the disclosure of Spectre [171] and Meltdown [172], subsequent work unveiled numerous variations of these attacks. These attacks generally fall into one of two categories [14]: *(i)* Spectre-type attacks, which rely on data- or control-flow misprediction to induce transient execution and *(ii)* Meltdown-type attacks, which exploit transient execution arising from the delayed handling of exceptions. Spectre-type attacks that target data flow[2] abuse address speculation to create a transient window for data exfiltration. Multiple attacks of this nature have been unveiled, including LVI [175], Fallout [121], RIDL [174], ZombieLoad [173], CrossTalk [180], and CacheOut [69]. Other Spectre-type attacks abuse various control-flow predictors that speculate on the outcome of conditional branches (Spectre-PHT) [181–185], branch destination addresses (Spectre-BTB) [33, 186], or function return addresses (Spectre-RSB) [187, 188] to create the transient window and retrieve the transiently processed data. Analogous to Spectre-type attacks, multiple variations of Meltdown have been unveiled since its initial release [184, 185, 189–192], demonstrating increasingly advanced exploitation techniques across a broader set of microarchitectural features.

3.2 Software-based Microarchitectural Attacks on MCUs

While low-end MCUs have long been targets of physical side-channel analysis, such as electromagnetic leakage [99, 100], their susceptibility to software-based side-channels has received far less attention. This line of work gained traction only recently, largely alongside the emergence of low-end TEEs for these devices. However, compared to the extensive literature on software-based side-channel attacks targeting high-end processors [12, 14, 16], microarchitectural attacks on MCUs remain sparse, with only a handful of demonstrations reported to date [37, 135, 193–196]. Existing work clusters around two main leakage vectors: *(i)* interrupt-latency leakage [37, 135, 193–195], and *(ii)* bus-contention leakage [135, 193]. We review both lines of work next, then briefly discuss recent attacks on cryptographic code and other software-based side channels on MCUs.

3.2.1 Interrupt Latency Microarchitectural Attacks

While classical end-to-end timing attacks are well known [96, 101–104], including on microcontrollers [197], Nemesis [37] was the first to demonstrate instruction-level microarchitectural leakage on MCUs by exploiting secret-dependent variations in interrupt latency to amplify subtle timing differences in non-constant-time code. Nemesis leverages the fact that MSP430 devices service interrupts only at instruction boundaries, forcing any pending interrupt to be delayed until the current instruction completes. Initial demonstrations [37, 135, 193] targeted the Sancus research TEE [198], originally on a 16-bit openMSP430 softcore and later on real TI MSP430 MCUs [194, 195]. However, the authors [37, 135, 193] argue that this microarchitectural leakage was not directly portable to all MCUs. In particular, Arm-based MCUs incorporate special interrupt-handling mechanisms that can abandon, restart, or resume multi-cycle instructions

[2] These attacks are also known as microarchitectural data sampling (MDS) [121, 174, 179].

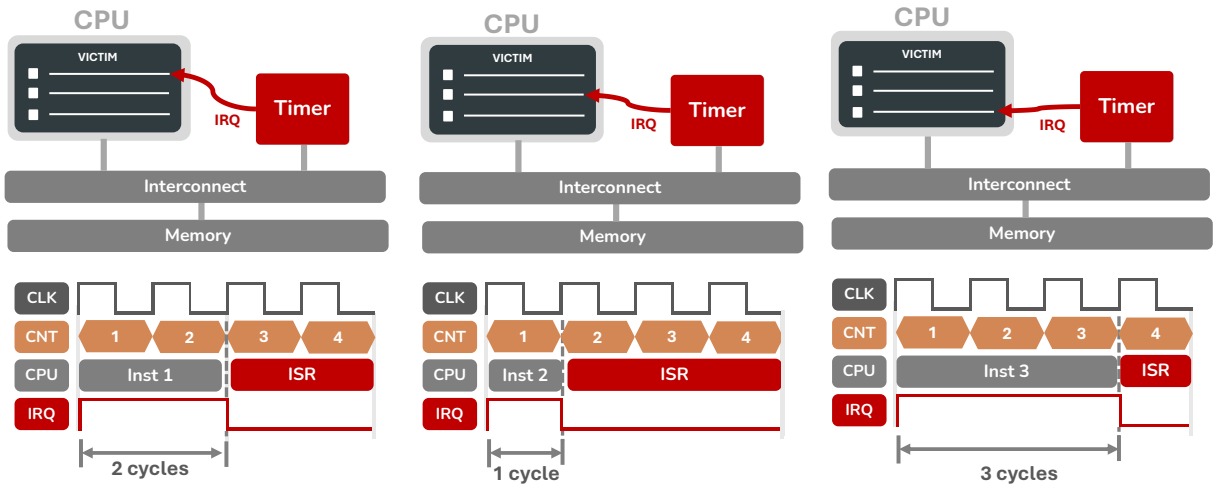


Figure 19: Illustration of the interrupt latency leakage. The top diagram illustrates an attacker-controlled timer, landing one interrupt on each victim instruction. The bottom diagram shows the effects of that interrupt on the CPU pipeline (CPU row), as well as the variation in the IRQ latency (IRQ row) depending on the execution time of the interrupted instruction. The ISR signal encompasses the time to context switch to the ISR and start executing the first ISR instruction. This overhead is constant, thus only the instruction length influences the variation of the interrupt latency. Red identifies attacker-controlled peripherals and signals.

(cf. Section 2.2.4), which led the Nemesis authors [37] to initially conclude that interrupt-latency attacks would not be feasible on Arm-based devices.

Interrupt Latency Leakage Overview. We now explain how interrupt latency can be leveraged to distinguish different instructions and infer microarchitectural events that affect execution time. The attacker measures interrupt latency by configuring a timer to inject an interrupt immediately after the victim instruction executes, then recording the elapsed time until the interrupt service routine begins execution. Assuming the attacker interrupts each victim instruction of interest, the length of the interrupted instruction dictates the observed interrupt latency. As illustrated in Fig. 19, instruction one (*Inst 1*) incurs a two-cycle delay on the interrupt latency, whereas instruction two (*Inst 2*) incurs a single-cycle delay. Moreover, this latency can be further affected by microarchitectural events, such as cache misses or bus contention, which increases the instruction execution time. For instance, if instruction three (*Inst 3*) is identical to instruction two (*Inst 2*) but experiences an microarchitectural event delaying the memory access by two clock cycles, the attacker can distinguish between both executions of the same instruction.

Attack Flow. The attacker leverages attacker-controlled peripherals to generate precise interrupts, capturing distinct microarchitectural behaviors based on the observed interrupt latency. Concretely, the attacker uses a timer to generate interrupts that land on the target instruction, while concurrently measuring the time elapsed from when the interrupt is asserted until the ISR begins execution. To extract an instruction-granular trace, the attacker configures the timer to interrupt the victim on the first instruction immediately after its invocation. Once the interrupt is injected into the CPU and the current instruction completes, the CPU switches context to the attacker domain to execute the ISR logic. The duration of this

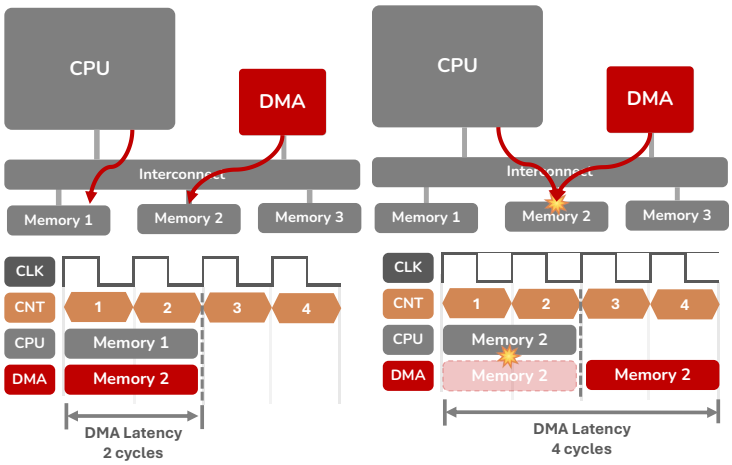


Figure 20: Illustration of the bus contention leakage. The top diagram illustrates an attacker-controlled DMA concurrently accessing memory alongside the CPU to induce contention on a shared memory bank. The bottom diagram details the bus-level effects of these concurrent accesses, where the CPU and DMA rows represent the memory transactions issued by the respective bus mains.

context switch depends on the instruction under execution when the interrupt was fired, which is recorded by the measurement logic inside the ISR. The attacker then repeats this process by reconfiguring the timer and returning execution to the next victim instruction, effectively gathering an interrupt latency reading for every victim instruction. If the victim exhibits secret-dependent control flow, the observed interrupt latency will diverge across execution paths, enabling the attacker to extract fine-grained execution traces that reveal instruction timings and ultimately recover secrets from isolated domains.

3.2.2 Bus Contention Microarchitectural Attacks

A second class of microarchitectural attacks on MCUs exploits contention on shared memory buses. Bognár et al. [135, 193] first demonstrated on the openMSP430 that concurrent accesses to shared memory banks induce observable timing differences, which can be leveraged to extract fine-grained execution traces and reveal secret-dependent patterns in victim code. They leveraged a DMA to perform simultaneous accesses to shared memory and create a microarchitectural timing channel due to contention on a shared bus.

Bus Contention Leakage Overview. We now explain how bus contention can be leveraged to gather information about memory accesses performed by the victim code, which yields control-flow information about their execution. This microarchitectural leakage arises when multiple bus mains, such as the CPU and DMA, issue simultaneous accesses to the same memory bank, forcing the interconnect arbitration logic to serialize transactions and resulting in observable timing differences. As illustrated in Fig. 20, when the CPU and DMA compete for the same resource, the DMA experiences increased memory access latency, for example, four clock cycles instead of two. By analyzing these timing differences, an attacker-controlled bus main can infer whether the victim accessed the same memory bank during a specific time

window.

Attack Flow. The attacker leverages an attacker-controlled DMA to perform concurrent accesses to the same memory bank as the victim code, exploiting the shared interconnect despite the isolated physical memory spaces. To capture the timing variations induced by the interconnect arbitration logic, the attacker configures the DMA to perform continuous memory accesses while the victim executes. By leveraging a high-resolution timer, the attacker precisely measures DMA transfer latency to detect contention on the shared bus. Different instructions exhibit distinct memory access patterns: *(i)* load instructions access data memory, whereas *(ii)* register-to-register instructions do not. Consequently, when the victim executes memory-bound instructions, an attacker-controlled DMA accessing the same memory bank experiences increased latency due to bus contention. If the victim exhibits secret-dependent control flow, the timing and sequence of memory accesses will diverge across execution paths, enabling the attacker to extract fine-grained execution traces that reveal memory access patterns and ultimately recover secrets from isolated domains.

3.2.3 Microarchitectural Attacks on Cryptographic Applications

While cryptographic applications on MCUs have long been targets of physical side-channel attacks [113, 114], their susceptibility to software-based microarchitectural attacks remained largely unexplored. Bus contention and interrupt latency attacks have proven effective in extracting secrets from isolated domains; however, early demonstrations remained confined to simple targets, such as dummy secure keypad implementations [37, 135]. KyberSlash [196] represents the first microarchitectural attack mounted against a real cryptographic algorithm on MCUs. The authors demonstrated full key recovery against Kyber, a post-quantum cryptographic algorithm, by exploiting secret-dependent timing variations in integer division and multiplication instructions. Nevertheless, the attack was demonstrated on a Cortex-M4 (Armv7-M) device without TrustZone-M, with both the attacker and victim executing within the same privilege and security domain, limiting its applicability to threat models that do not involve isolated execution environments.

3.2.4 Other Software-Based Side-Channel Attacks.

While out of scope for this thesis, software-based power side-channel attacks represent an alternative vector for extracting secrets from MCUs. These attacks leverage onboard analog-to-digitals (ADCs) peripherals, which are ubiquitous on Arm Cortex-M devices, to collect power-related traces and recover secret keys via software-based differential power analysis [113, 114]. Gnad et al. [113] exploited the correlation between ADC noise and device power consumption on a Cortex-M4-based platform to extract an AES key from software-based traces. O'Flynn et al. [114] demonstrated a remote power side-channel attack that bypasses TrustZone-M on a SAM L11 (Cortex-M23) and recovers the private key from an AES algorithm running in the TrustZone-M secure world. From a different perspective, Barenghi et al. [199] propose a

hybrid approach in which microarchitectural timing measurements are first used to construct a microarchitectural leakage model of Cortex-M4- and Cortex-M7-powered MCUs, which is then leveraged to augment power side-channel attacks.

3.3 Gap Analysis

The literature reviewed in Section 3.2 reveals three gaps central to this thesis. First, software-based microarchitectural attacks on MCUs remained sparse, leaving unclear whether early demonstrations such as Nemesis [37] reflected an isolated MSP430-specific phenomenon or a broader problem affecting modern MCUs. Second, prior interrupt-latency and bus-contention attacks were largely confined to MSP430 or openMSP430 settings [37, 135, 193–195]. In particular, Arm Cortex-M devices were considered immune to interrupt-latency timing attacks [37], and bus-contention leakage had not been demonstrated beyond the niche openMSP430 platform. Third, prior software-based microarchitectural attacks on MCUs primarily targeted proof-of-concept victims, such as dummy secure keypad implementations [37, 135], leaving open whether these leakage sources could compromise real-world cryptographic software across hardware-enforced isolation boundaries.

While KyberSlash [196] demonstrated the first microarchitectural attack against a real cryptographic implementation by exploiting secret-dependent timing in integer division and multiplication instructions, it remains a solitary example on MCUs, contrasting sharply with the extensive literature on APUs [200]. Moreover, the attack targeted a Cortex-M4 (Armv7-M) without TrustZone-M, with victim and attacker operating in the same privilege and security domain. To the best of our knowledge, as summarized in Table 3, no prior work has demonstrated a software-based microarchitectural attack that simultaneously targets an Arm-based MCU, crosses a TrustZone-M security boundary, and compromises real-world cryptographic software.

3.3.1 Thesis Contributions

In this thesis, as highlighted in Table 3, we advance the state of the art on microarchitectural attacks by demonstrating the first end-to-end attacks on TrustZone-M-enabled MCUs, and provide evidence that such vulnerabilities scale broadly across the MCU spectrum (Chapter 4). When this research began in late 2020, the relevant literature was extremely limited, consisting primarily of two software-based power side-channels [113, 114] and a single microarchitectural side-channel attack, i.e., Nemesis attack [37]. Although Nemesis demonstrated that MCUs could be vulnerable to microarchitectural side-channel attacks, it remained unclear whether this represented an isolated anomaly or early evidence of a broader systemic problem. We address this gap by unveiling: *(i)* BUSted [8], a non-conventional microarchitectural leakage pervasive across the Arm Cortex-M ecosystem (Chapter 5); *(ii)* M-Step [11], the first single-stepping framework for MCUs that extracts instruction-granular traces with high temporal resolution, exposing

Table 3: Overview of software-based power and timing side-channel attacks on MCUs, compared by target and leakage source. Orange highlights contributions associated with this thesis. Light blue denotes concurrent work published since the start of this thesis.

				Target			**Leakage**				
	Attack	**Yr**	**MCU**	TZ	RW	ST	Pwr	IRQ	Bus	Cache	Div
Power	Leaky Noise [113]	'19	Cortex-M4	○	●	○	●	○	○	○	○
	O'Flynn et al. [114]	'19	Cortex-M23	●	●	○	●	○	○	○	○
	Barenghi et al. [199]	'21	Cortex-M4/7	○	◐	○	●	○	○	○	○
Timing	Nemesis [37]	'18	openMSP430	○	○	●	○	●	○	○	○
	Mind the gap [135]	'22	openMSP430	○	○	●	○	◐	◐	○	○
	MicroProfiler [193]	'23	openMSP430	○	○	●	○	○	●	○	○
	RIPencapsulation [194]	'24	TI MSP430/2	○	○	●	○	◐	○	○	○
	IPE Exposure [195]	'24	TI MSP430	○	○	●	○	●	○	●	○
	BUSted [8] (this thesis)	'24	Cortex-M23/33	●	○	●	○	○	●	○	○
	BUSted 2nd stop [9] (co-author)	'24	Cortex-M4/33	○	◐	○	○	○	●	○	○
	KyberSlash [196]	'25	Cortex-M4	○	●	○	○	○	○	○	●
	M-Step [11] (this thesis)	'26	Cortex-M33	●	●	●	○	●	●	●	●

TZ = TrustZone; RW = Real-World Cryptographic Code; ST = Single Trace;
Pwr = Power Traces; IRQ = Interrupt Latency; Bus = Bus Contention;
Cache = Cache Activity; DIV = Division/Multiplication Instructions.

fine-grained execution details of TrustZone-M-protected code (Chapter 6); and *(iii)* an end-to-end microarchitectural attack against TrustZone-M-protected cryptographic software that recovers a full RSA private key from a single execution trace (Chapter 7).

4

Arm Cortex-M Microarchitectural Analysis

MCUs are steadily increasing in complexity and are being deployed in ever more security-critical applications, often under stronger threat models that assume more capable adversaries. At the same time, their ubiquity has made them a cornerstone of modern computational infrastructure. Together, these trends increase their attractiveness as targets and motivate the exploration of attack vectors beyond classic software vulnerabilities, such as memory-safety bugs. This includes side-channel attacks, particularly on state-of-the-art MCUs that integrate security primitives such as TrustZone-M. While there is an extensive body of work on side-channel attacks targeting APUs [12–16], the security community has been comparatively reluctant to study microarchitectural side-channel attacks on MCUs, which are often perceived as too simple to leak. However, as discussed in Section 1.2, we hypothesize that recent advances in MCUs microarchitecture have increased their complexity to the point that: *(i)* microarchitectural components previously exploited on APUs may also be exploitable on MCUs; and *(ii)* MCUs may even expose new microarchitectural channels tied to their unique characteristics.

This chapter revolves around exploring these two hypotheses, providing the first steps toward answering the thesis research question (**RQ**). We will first, in Section 4.1, perform a comprehensive analysis of the microarchitecture of Arm's full Cortex-M portfolio to gain insights into the microarchitectural components present in Arm MCUs that might be susceptible to microarchitectural leakage. Then, in Section 4.2, based on this analysis and existing literature on side-channel attacks, we will select three microarchitectural components and use them to mount covert-channels on multiple MCUs, providing the first evidence of widespread microarchitectural leakage on these devices. While this microarchitectural leakage steers us in the right direction, it does not yet conclusively validate the hypotheses stated above. To substantiate our findings and move from initial evidence of microarchitectural leakage to side-channel attacks on MCUs, we end this chapter by presenting three toy attacks, in Section 4.3, which leverage three distinct microarchitectural components to leak secrets from synthetic victims deployed in the TrustZone-M secure world. This chapter will set the stage for the microarchitectural exploration presented in the remainder of this thesis.

Related Contributions and Artifacts. The research carried out in this chapter has been partially published in two scientific articles (**P1** and **P2**). Moreover, as part of our broader open-science effort, we

have also open-sourced (**A1**) all the source code, data sets and scripts used in the experiments of this chapter (i.e., covert-channel on Section 4.2 and toy attacks on Section 4.3).

P1 - Publication: C. Rodrigues, D. Oliveira, and S. Pinto, *"BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect"*, In: IEEE Symposium on Security and Privacy (S&P), 2024. *Link*.

P2 - Publication: C. Rodrigues, M. Bognár, S. Pinto, and J. Van Bulck *"M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M"*, (To appear) In: 35th USENIX Security Symposium, 2026. *Link*.

A1 - Artifacts: https://github.com/ESCristiano/uarch-attacks-mcus.

4.1 Deep Dive on Cortex-M

In this section, we survey Arm Cortex-M-powered MCUs to identify microarchitectural components likely to leak at both the CPU and system level. Based on this survey, we formulate hypotheses on potential microarchitectural leakage in MCUs, which we then empirically validate in Section 4.2.

4.1.1 Arm-based MCUs Microarchitectural Landscape

Building on the extensive body of work on side-channel attacks affecting APUs (cf. Chapter 3), we surveyed the microarchitecture of a broad set of Arm-based MCUs to identify components most susceptible to leakage. Our survey encompasses both Arm Cortex-M processor cores and vendor-specific system-level hardware blocks integrated within MCUs, enabling us to identify candidate components at both the CPU and system level. Table 4 summarizes the results of our survey across a diverse set of MCUs spanning multiple vendors (Ambiq, ST, NXP, Silicon Labs, Microchip, GigaDevice) and processor families (Armv6-M through Armv8-M). The survey covers MCUs with varying interconnect arbitration policies, hardware isolation support, and cache strategies, enabling us to characterize the microarchitectural landscape and identify components susceptible to leakage.

Best Candidates for Microarchitectural Leakage Analysis. From the microarchitectural components identified in our survey (Table 4), we selected three candidates for deeper leakage analysis: the bus interconnect, the interrupt logic, and the cache. We applied two criteria to guide this selection: *(i)* first, we select microarchitectural components already exploited in APUs and *(ii)* second, we focus on components pervasive across the MCU spectrum, which are shared between multiple MCU software entities.

The cache satisfies the first criterion. Cache-based attacks have a long history on APUs [118, 136–138], making caches an obvious candidate for microarchitectural leakage analysis on MCUs. The interrupt logic satisfies both criteria. It is a core microarchitectural feature common to modern MCUs, and prior work had already shown that it can extract side-channel information on AVR-based MCUs [37]. However,

Table 4: Survey of microarchitectural characteristics from a distinct pool of MCUs. Architectures: Von Neumann (VM) or Harvard (H). Vendors: Ambiq (AM), ST, NXP, Silicon Labs (SL), Microchip (MC) and Gigadevice (GD). CPUs: Cortex-Mx (Mx). ISAs: Armv6-M (v6M), Armv7-M & Armv7E-M (v7M), Armv8-M (v8M). Interconnect arbitration policy: Priority-based (P), Round-robin (R) and Priority-based or Round-robin (PR). MCUs with any hardware isolation primitive (–). Caches: MCU with CPU-level native caches (●), MCU with system-level vendor-specific caches (◐), and MCUs without any cache (○). Feature present (✓) or absent (✗).

	MCU					CPU					Memory Subsystem			
MCU	Vendor	CPU	MHz	ISA	Arch	Dual-Issue	Bran. Pred.	Spec. Reads	Spec. Exec.	Int. Ctrl	Isolation	ICache	DCache	Bus Arb. P.
CY8CKIT-046	CY	M0	48	v6M	VM	✗	✗	✗	✗	✓	–	○	○	PR
NK-M032KI	NU	M0	72	v6M	VM	✗	✗	✗	✗	✓	–	◐	○	R
STM32L0/G0	ST	M0+	32	v6M	VM	✗	✗	✗	✗	✓	MPU	○	○	R
EFM32HG	SL	M0+	24	v6M	VM	✗	✗	✗	✗	✓	MPU	◐	○	R
SAM D21	MC	M0	48	v6M	VM	✗	✗	✗	✗	✓	–	◐	○	P
LPC84X EVAL	NXP	M0+	30	v6M	VM	✗	✗	✗	✗	✓	MPU	○	○	PR
CY8C5868LTI	CY	M3	67	v7M	H	✗	✗	✗	✗	✓	MPU	◐	○	PR
EFM32GG	SL	M3	48	v7M	H	✗	✗	✗	✗	✓	MPU	◐	○	R
SAM L10/L11	MC	M23	32	v8M	VM	✗	✗	✗	✗	✓	TZ	◐	◐	PR
NK-M2351	NU	M23	64	v8M	VM	✗	✗	✗	✗	✓	TZ	◐	◐	PR
STM32L5	ST	M33	110	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	R
STM32U5	ST	M33	250	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	◐	R
MuscaB1	Arm	M33	160	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	R
LPC5500	NXP	M33	150	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	P
RA4M3	RE	M33	100	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	PR
nRF9160	NO	M33	64	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	P
EFM32PG22	SL	M33	76	v8M	H	✓ [1]	✗	✗	✗	✓	TZ	◐	○	R
STM32F4	ST	M4	180	v7M	H	✗	✗	✗	✗	✓	MPU	◐	◐	R
SAM E54	MC	M4	120	v7M	H	✗	✗	✗	✗	✓	MPU	◐	◐	P
FRDM-KV31F	NXP	M4	120	v7M	H	✗	✗	✗	✗	✓	MPU	◐	◐	PR
RA6M3	RE	M4	120	v7M	H	✗	✗	✗	✗	✓	MPU	◐	○	PR
EFM32PG12	SL	M4	40	v7M	H	✗	✗	✗	✗	✓	MPU	◐	○	R
MSP432E401Y	TI	M4	120	v7M	H	✗	✗	✗	✗	✓	MPU	○	○	PR
STM32F7/H7	ST	M7	216	v7M	H	✓	✓	✓	✗	✓	MPU	●	●	R
i.MX-RT1020	NXP	M7	500	v7M	H	✓	✓	✓	✗	✓	MPU	●	●	PR
SAM E70	MC	M7	300	v7M	H	✓	✓	✓	✗	✓	MPU	●	●	P
GD32H7	GD	M7	600	v7M	H	✓	✓	✓	✗	✓	MPU	●	●	R
M55M1	NU	M55	220	v8M	H	✓ [1]	✓	✓	✗	✓	TZ	●	●	PR
Apollo510	AM	M55	250	v8M	H	✓ [1]	✓	✓	✗	✓	TZ	●	●	PR
STM32N6	ST	M55	800	v8M	H	✓ [1]	✓	✓	✗	✓	TZ	●	●	PR
STM32V8	ST	M85	800	v8M	H	✓	✓	✓	✗	✓	TZ	●	●	PR
RA8P1	RE	M85	1000	v8M	H	✓	✓	✓	✗	✓	TZ	●	●	PR

[1] Limited to some Thumb 16-bit instruction pairs.

it was previously considered non-exploitable on Arm Cortex-M, an assumption we revisit here. The bus interconnect satisfies the second criterion. It is pervasive across modern MCUs and constitutes a highly shared resource. In contrast to the cache and the interrupt logic, no known exploits had directly targeted the bus interconnect at the time of our initial analysis.[1] This gap in the literature made the bus interconnect a particularly interesting target for microarchitectural leakage analysis on MCUs.

Lastly, while some Arm Cortex-M processors (e.g., Cortex-M7, Cortex-M55, Cortex-M85, and Cortex-M52) feature branch predictors, we excluded this microarchitectural component from our analysis. First, branch predictors are not pervasive across all Cortex-M CPUs. Second, Arm Cortex-M CPUs do not support speculative execution, which precludes the exploitation of transient execution attacks on these devices [14] (cf. Section 3.1).

[1] Subsequent work [135], published roughly concurrently with our findings [7, 8], independently identified a similar vulnerability on AVR-based MCUs.

4.1.2 Cache

As discussed in Section 2.2 (cf. Table 2), most Arm Cortex-M CPUs lack native caches. One might therefore conclude that cache-based microarchitectural channels are limited to a small set of high-performance MCUs. However, Table 2 (Section 2.2) does not depict the full MCU microarchitectural picture. As is common practice, MCU vendors integrate a standard Cortex-M CPU core with vendor-specific hardware blocks, such as timers, DMAs, and other peripherals, within a single SoC (recall Section 2.1). One such hardware block is a system-level cache, which vendors often add precisely when the selected CPU core lacks a native cache.

Pervasiveness of Cache Structures on MCUs. We surveyed MCUs from multiple vendors (summarized in Table 4) and found that cache-like structures are common, even on simple devices, at least along the instruction path. For instance, SAM L10/11 [201], STM32L5 [202], and FRDM-KV31F [203] all integrate a system-level ICache, despite being powered by cacheless Arm Cortex-M processors (Cortex-M23, Cortex-M33, and Cortex-M4, respectively). The data path further influences how caches are used. For example, on STM32U5 [204], the data cache is used only when data resides in external memory, which is common given the limited size of internal MCUs memories. As a result, an attacker may observe not only cache-access patterns but also which type of memory the victim is using, potentially revealing additional side-channel information.

Our Hypothesis. Although most Arm Cortex-M family CPUs do not provide a native cache (cf. Table 2 Section 2.2), MCUs integrating these cores are frequently equipped with vendor-specific, system-level cache-like structures. We therefore hypothesize that cache-based mechanisms may leak useful information not only on Cortex-M cores with built-in caches, but also across a broader range of MCUs whose CPUs lack internal caches. On APUs, cache-based side-channel attacks are typically noisy due to both the complex microarchitecture of these devices and their large software stacks. MCUs, however, may offer a more favorable setting, since they feature simpler microarchitecture and software stacks. As a result, an attacker may be able to extract useful information even from a weaker signal. We therefore expect cache structures on MCUs to leak microarchitectural information with a favorable signal-to-noise ratio.

4.1.3 Interrupt Logic

The interrupt logic has been leveraged either to obtain high-resolution samples of other microarchitectural components, which became known as single-stepping frameworks [37, 58–62, 65], or as a direct source of side-channel information through timing differences in interrupt latency [37]. In both cases, the attacker generates a stream of interrupts spaced a few clock cycles apart, aiming to interrupt individual victim instructions. In the first case, the attacker uses the interrupt window to gather instruction-granular information about other microarchitectural components; in the second, the attacker exploits interrupt latency itself as the leakage source. Exploiting this microarchitectural component relies on a simple premise: the CPU services an interrupt only after the currently executing instruction has been dispatched. Because

instruction latency varies across instructions and is further affected by multiple microarchitectural events (e.g., cache misses and page walks), the interrupt latency is naturally tied to the underlying microarchitectural context (i.e., the instruction under execution and the microarchitectural state), allowing an attacker to gather instruction-granular side-channel information about the victim code being executed.

This vulnerability was known to affect a broad range of the computing spectrum, including high-end Intel [37, 58–61, 65], AMD [62], and Arm processors [63, 64], as well as low-end, simple MSP430 MCUs [37, 135, 193, 195]. However, Arm-based MCUs were considered immune to this vulnerability [37], since Arm Cortex-M processors abandon multi-cycle instructions when interrupted. Nevertheless, instruction latency is affected by several factors outside the CPU's control, such as which bus is used for a memory fetch, whether the access results in a cache hit or miss, and the latency of the target memory. Thus, while Cortex-M processors incorporate microarchitectural mechanisms to improve determinism and reduce interrupt latency (cf. Section 2.2), for instance by making some multi-cycle instructions interruptible, this property does not hold uniformly across the ISA. According to the Arm Cortex-M family architectural manuals [2, 4], only a subset of multi-cycle instructions (e.g., multiword load/store, multiplication/division instructions) can be interrupted. Moreover, on several Cortex-M CPUs (e.g., Cortex-M0+, Cortex-M23, and Cortex-M4), single load/store instructions take at least two cycles to execute, whereas simple register-to-register operations typically take a single cycle. Importantly, neither of these instructions is interruptible; hence, an interrupt that arrives during their execution must be serviced only after their completion, which affects the time to attend the interrupt differently.

Our Hypothesis. While Arm Cortex-M processors include mechanisms intended to make IRQ handling more deterministic, we argue that, in some cases, the instruction under execution may still affect the total IRQ latency. Assuming that the IRQ context switch is constant-time (as Arm claims for the majority of Arm Cortex-M family processors), and that both `mov` and `ldr` are non-interruptible [2, 4], but `mov` typically takes 1 cycle whereas `ldr` takes at least 2 cycles, an IRQ arriving during these two instructions should experience different IRQ latencies. While the Nemesis authors [37] deemed Arm Cortex-M immune to interrupt-latency attacks, we hypothesize that the interrupt logic also leaks microarchitectural information on Arm-based MCUs.

4.1.4 Bus Interconnect

While the interrupt logic and cache were already known sources of microarchitectural leakage in other computational units [37, 118, 136–138], the bus interconnect, despite being a pervasive element across the computing spectrum in both high- and low-end systems, had remained largely unexplored prior to this thesis. The closest work on APUs used the interconnect as a vehicle to exploit coherence mechanisms in cache-based side-channel attacks [132–134], but did not target leakage on the interconnect itself. Nonetheless, the bus interconnect is a centralized microarchitectural component that connects all main

and secondary entities in the system, making it a highly shared resource. When a bus main[2] communicates with a memory-mapped peripheral or memory region, the interconnect establishes a communication path between the originating main port and the appropriate secondary. This behavior is fundamental to central interconnects across the computing spectrum, whether in APUs or MCUs. However, interconnects in APUs are substantially more sophisticated. They typically implement advanced arbitration algorithms and quality of service (QoS) policies that regulate bandwidth allocation per bus main (BM), which seems to make side-channel analysis not feasible on them. In contrast, interconnects used in MCUs typically rely on simple arbitration logic with no or only very rudimentary QoS mechanisms.

Arbitration Policies. We surveyed MCUs from different vendors and found that there are two main arbitration policies: *(i) priority-based* or *(ii) round-robin*. 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. Most of the surveyed MCUs either lack explicit QoS mechanisms altogether or only provide very rudimentary ones (cf. Table 4), which may further exacerbate contention-induced leakage. For example, the SAML11 MCU implements a basic QoS scheme in which, at the strongest setting, the CPU is assigned the highest system priority. In such a configuration, any other BM will experience systematic delays whenever the CPU accesses the shared memory bank. Consequently, an adversarial BM can reliably detect CPU activity on the shared resource, since it consistently observes an access latency increase relative to its baseline whenever contention occurs.

Our Hypothesis. There are no known attacks on the bus interconnect on APUs, most likely due to both the complexity of the interconnect and the APU. However, from insights gathered surveying the 32 MCUs arbitration policies, we found the interconnects to exhibit behavior likely to create microarchitectural leakage. When there is contention on a MCU interconnect port, i.e., two bus mains concurrently targeting the same bus secondary, the interconnect handles the contention by granting or denying access to one bus main at the expense of the other. The bus main whose memory access was delayed will experience different memory latencies depending on whether contention occurred. We speculate that if a CPU is involved in this contention, it may leak secret-dependent memory accesses of the executing code to other bus mains.

4.2 First Evidence of Microarchitectural Leakage

In this section, we empirically investigate whether the microarchitectural components selected in Section 4.1 can be used to establish microarchitectural channels and leak information across otherwise isolated domains. To this end, we developed three PoCs covert-channels, one for each selected microarchitectural component (cf. Section 4.1). Covert-channels leverage shared microarchitectural elements to

[2]We replaced *master* and *slave* terminology used in technical documentation and manuals with the keywords *main* and *secondary*.

establish unauthorized communication between isolated domains, encoding and transmitting information from a trojan to a spy. There is a common assumption that when there is a covert-channel, the system is vulnerable to side-channel attacks [205]. For each covert-channel, we try to establish an 8-bit[3] unauthorized microarchitectural channel and transmit a secret from an isolated domain (where a trojan executes) to the spy domain, thereby bypassing the system's isolation boundaries. To assess the pervasiveness of microarchitectural leakage on MCUs, we deployed these covert-channels across multiple platforms from five different vendors. All covert-channels target the strongest hardware isolation primitive available on each platform, such as world isolation in TrustZone-enabled MCUs or memory isolation in MCUs with a simple MPUs.

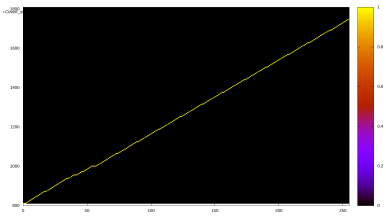
4.2.1 Cache Covert-Channel

As we saw in Section 4.1, while native caches are available only on Cortex-M7, Cortex-M55, and Cortex-M85, vendors often integrate proprietary hardware blocks with cache-like structures to improve performance and reduce power consumption. This makes caches in MCUs more prevalent than one would expect for such resource-constrained computing units. Nevertheless, these caches are substantially smaller than those available in APUs, typically ranging from hundreds of bytes to a few kilobytes. To assess whether cache can be leveraged on MCUs to establish microarchitectural channels, we mounted a covert-channel on six representative platforms from five different vendors. We evaluated whether differences in cache topology (built-in versus system-level caches), MCU complexity (e.g., Cortex-M23 versus Cortex-M7), and cache size affect the feasibility of mounting an cache microarchitectural channel.

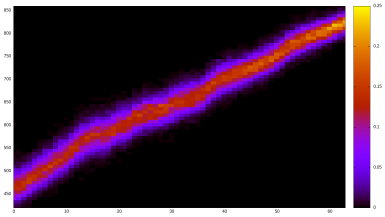
Covert-channel Design. Since ICache is the most pervasive cache-like structure on MCUs (cf. Table 4), we mounted an ICache-based covert-channel on all evaluated MCUs. We implemented the ICache covert-channel following the Prime+Probe exploitation technique [101, 104]. This technique operates in three phases: *(i)* the Prime phase establishes a known microarchitectural state by filling the target cache with spy-controlled code; *(ii)* the Victim phase invokes the trojan, which evicts some of the spy's cache lines through its execution; and *(iii)* the Probe phase re-executes the buffer and measures access time. Cache lines touched by the victim incur cache misses (fetched from memory, higher latency), while untouched lines hit in cache (lower latency). The cache covert-channel proceeds as follows: *(i)* at compile time, the attacker embeds a statically defined ICache buffer containing a long stream of branch instructions[4] in the spy's binary; *(ii)* at runtime, the spy primes the ICache by executing jumps through all cache lines; then *(iii)* the spy invokes the trojan in the victim domain, which executes a specific number of instructions encoding the secret value; and lastly, *(iv)* the spy probes the ICache by re-executing the buffer and measuring total access time. The measured time correlates with the number of instructions executed by the trojan, thereby encoding the transmitted value.

[3] There is no specific reason to cap the channel at 8 bits; this value was chosen arbitrarily for evaluation purposes and is not motivated by any particular microarchitectural constraint.

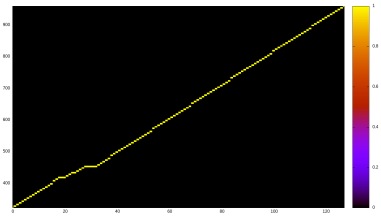
[4] The number of jumps is defined by the ratio $N_\text{Jumps} = \frac{\text{ICache_Size}}{\text{ICache_Line_Size}}$.



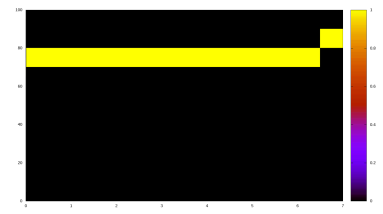
(a) Nuvoton M2351 with a Cortex-M23 (7.22/8 bits).



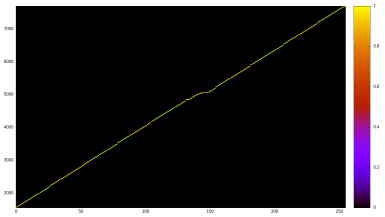
(b) Arm Musca B1 with a Cortex-M33 (2.27/8 bits)



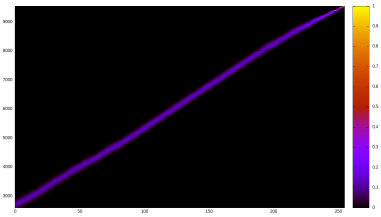
(c) Nordic nRF9160 with a Cortex-M33 (6.83/8 bits)



(d) NXP LPC55S69 with a Cortex-M33 (0/8 bits)



(e) STM32L552 with a Cortex-M33 (7.77/8 bits)



(f) STM32F767 with a Cortex-M7 (4.87/8 bits)

Figure 21: Channel matrix of ICache-based covert-channels across six representative MCU platforms. The x-axis encodes the number of cache sets accessed, while the y-axis represents aggregate access time. Color intensity indicates the probability of observing a specific time for a given cache set count.

Results. We successfully established channels on five of the six evaluated platforms. Figure 21 summarizes the results and reports corresponding channel capacities[5]. We encountered implementation challenges on MCUs with system-level caches, primarily due to undocumented vendor-specific design choices, such as lack of information about cache replacement policy, cache line size, and the number of ways. However, these challenges affected engineering effort only, not feasibility. The only MCU on which we could not establish a channel was the NXP LPC55S69 (Fig. 21d), owing to its extremely small 128-byte cache.

We also observed two outliers: *(i)* the Arm Musca B1 (Fig. 21b) and *(ii)* STM32F767 (Fig. 21f). The Arm Musca B1, a prototype board for early Cortex-M33 silicon, exhibited high inter-sample variability, likely stemming from an immature system-cache implementation which is most likely specific to this board. The STM32F767 (Cortex-M7) exhibited noise levels comparable to microarchitectural channels on APUs [205], which we attribute to the more complex Cortex-M7 microarchitecture, including superscalar execution and more advanced prediction logic.

Overall, these results corroborate the leakage expectations stated in Section 4.1.2. First, we successfully mounted microarchitectural channels exploiting cache on MCUs. Second, despite cache sizes substantially smaller than those typical in APUs (e.g., STM32L5: 4 KByte 2-way ICache [202] versus 32–64 KByte L1 ICache in Cortex-A or Intel x86 CPUs), we were able to establish microarchitectural channels with very low noise. This low-noise environment may amplify the exploitability of cache-based attacks on MCUs, relative to their more complex counterparts.

[5]The channel capacity was measured with the leakiest tool [206], which is based on the concepts of information theory presented in [207, 208].

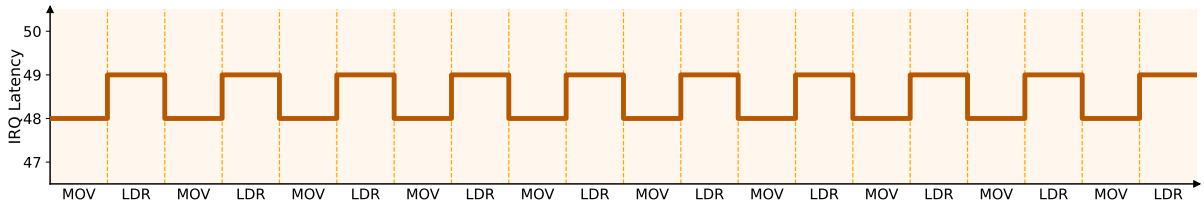


Figure 22: Long stream of interleaved MOV and LDR instructions (x-axis) and the respective interrupt request (IRQ) latency for each interrupted instruction (y-axis).

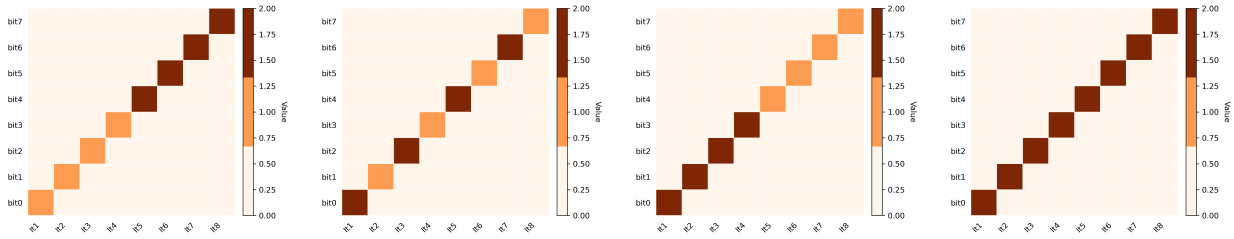
4.2.2 Interrupt Logic Covert-Channel

In contrast to bus interconnect and cache, interrupt logic is architecturally defined by Arm and is present across all Arm Cortex-M family processors. We therefore mounted the covert-channel on a single representative MCU and extrapolated the effect across the broad Arm Cortex-M family spectrum. This does not imply that all MCUs exhibit the same degree of leakage. Different MCUs will show different leakage profiles; nonetheless, the ability to observe distinct IRQ timings across different instructions should remain consistent across Arm Cortex-M CPUs. This behavior mirrors what has been observed on APUs. For instance, SGX-Step [58, 65] is a generic framework for interrupt-driven side-channel analysis on Intel-based CPUs, which exploits a similar interrupt logic vulnerability affecting Intel processors broadly. Nevertheless, each Intel CPU exhibits its own unique microarchitectural leakage profile.

Preliminary Leakage Analysis. To assess whether Cortex-M exhibits any microarchitectural leakage, we first performed a simple test to validate our hypothesis that MOV and LDR instructions have different impacts on IRQ latency, as discussed in Section 4.1.3. We created a long stream of interleaved MOV and LDR instructions and interrupted each instruction in sequence. Figure 22 illustrates the trace gathered during this basic test, where the x-axis represents the instructions being interrupted, and the y-axis represents the IRQ latency per interrupt. We would expect no horizontal timing variation (y-axis) across instructions (x-axis) if there were no leakage. However, we observed the opposite: a precise and consistent latency difference between interrupted MOV and LDR instructions. We leverage this leakage primitive to mount the interrupt logic covert-channel, using the different MOV and LDR instruction latencies to encode 1 bit of information: a "0" corresponds to MOV latency, whereas a "1" corresponds to LDR latency.

Covert-channel Design. The interrupt logic covert-channel has the following steps: *(i)* the spy triggers a timer configured to generate interrupts on each victim instruction[6] and adds logic to the interrupt handler to measure the time elapsed from interrupt until the CPU switches to the ISR; *(ii)* it then invokes the trojan, which executes a sequence of 8 instructions, i.e., a mix of MOV and LDR instructions, to encode an 8-bit secret value. For instance, if the trojan wants to encode the value "0x0F", it executes four MOV instructions followed by four LDR instructions, yielding the bit pattern "00001111".

[6]We are oversimplifying the exploitation of this process to ease reader understanding. This is much more challenging than what we are now conveying, and is one of the reasons the Nemesis authors [37] deemed Cortex-M immune to this microarchitectural leakage in the first place. We will delve deep into these challenges and how we overcame them in Chapter 6.



(a) 0x0F (b) 0x55 (c) 0xF0 (d) 0xFF

Figure 23: Illustration of four runs in which a trojan transmits four different values (0x0f, 0x55, 0xf0, and 0xff) over the 8-bit interrupt logic microarchitectural channel, as observed by the spy.

Results. As suggested by the experiment in Fig. 22, we were able to establish the target 8-bit microarchitectural channel. We instructed the trojan to encode all 256 possible values and the spy successfully recovered them all, without observable noise. Fig. 23 illustrates four of those transmitted values in four different runs of the trojan, values: *(i)* 0x0F, *(ii)* 0xF0, *(iii)* 0x55, and *(iv)* 0xFF. This covert-channel is particularly powerful because it requires only a small amount of trojan code to encode the secret, yet enables a noiseless microarchitectural communication channel, which may allow a trojan to transmit a packet of data at an high bit rate.

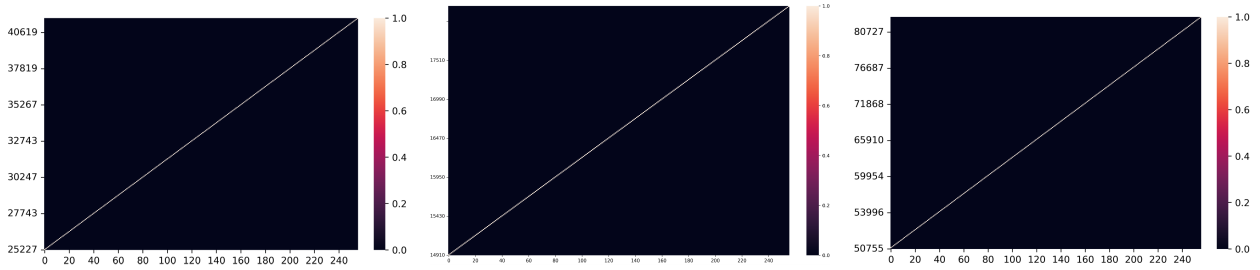
4.2.3 Bus Interconnect Covert-Channel

Similar to cache, bus interconnect is a microarchitectural component whose properties vary across MCUs. However, unlike cache, bus interconnect is present in all modern MCUs, deployed at the system level, and exhibits vendor-specific characteristics. To assess whether different microarchitectural implementations affect leakage, we mounted a covert-channel on twelve platforms from five different vendors, spanning MCUs with five distinct ISAs[7].

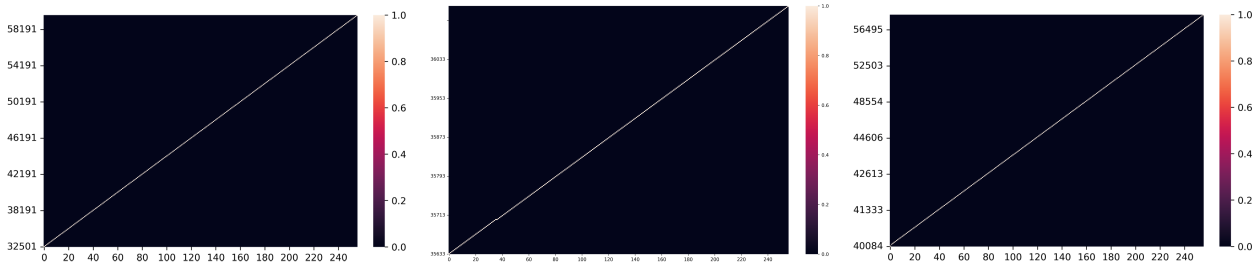
To establish this microarchitectural channel, the spy must create contention on an interconnect secondary port while the trojan accesses it, requiring control of a BM. Since MCUs typically feature a single core CPU, the spy cannot mount a cross-core microarchitectural channel using the CPU. However, other BMs on MCUs, such as DMAs, can access memory concurrently with the CPU without its intervention. We assume an attacker controlling a DMA, whose memory accesses are confined to their own protected memory space (consistent with in-place memory protections against direct memory access attacks). However, while the attacker cannot access other protected memory spaces, they may share a memory bank with the trojan code. With control over a BM, the attacker creates contention on a target memory bank shared with the trojan and correlates the total DMA transfer time with the secret transmitted by the trojan, i.e., the trojan encodes information by generating more or fewer memory accesses to the shared resource.

Covert-channel Design. The bus interconnect covert-channel proceeds as follows: *(i)* the spy triggers

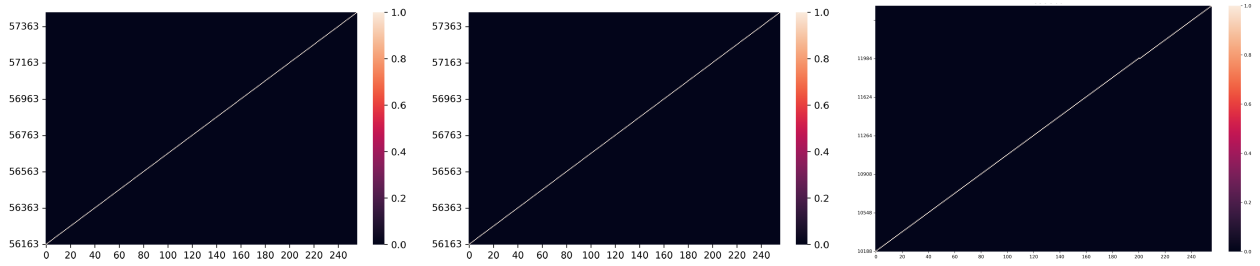
[7]Since bus interconnect is a fundamental component of modern MCUs, we extended our analysis to include non-Arm platforms: *(i)* a PIC-based and *(ii)* a RISC-V-based MCU.



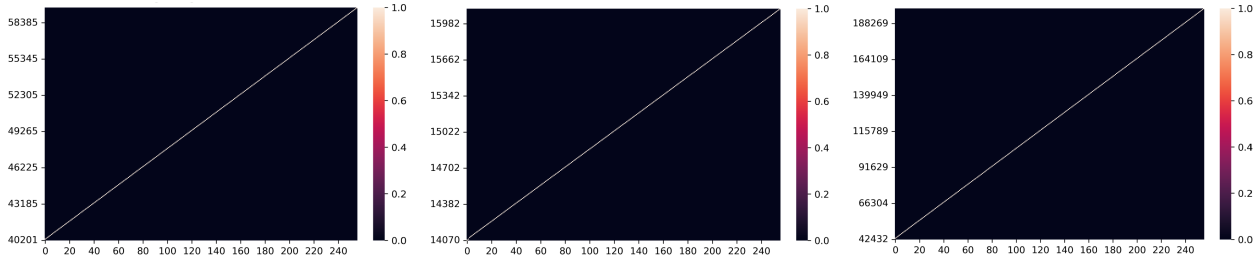
(a) PIC18F16Q41 (PIC18). (b) GD32VF103 (RV32). (c) STM32L073 (CM0+).



(d) SAMD21 (CM0+). (e) EFM32GG (CM3). (f) SAML11 (CM23).



(g) STM32L5 (CM33). (h) STM32U5 (CM33). (i) LPC5500 (CM33).



(j) STM32F407 (CM4). (k) SAME54 (CM4). (l) STM32F767 (CM7).

Figure 24: Channel matrix of all bus interconnect covert-channels across twelve representative MCU platforms. The x-axis encodes the secret value (correlated to the number of memory accesses), while the y-axis represents the aggregate access time. Color intensity indicates the probability of observing a specific access time for a given secret input.

a DMA to initiate the transfer of N bytes of data from a shared memory bank and starts a timer; *(ii)* it then invokes the trojan, 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 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. Across the twelve evaluated platforms, we were able to establish a channel on all of them. In contrast to the cache covert-channel, we found no additional challenges related to differences in arbitration policy across MCUs or to vendor-specific interconnect implementations (cf. Table 4). Figure 24 summarizes the results and highlights the prevalence of this covert-channel across the full MCU spectrum (even outside Arm MCU portfolio). All established microarchitectural channels exhibit the same qualitative behavior across the evaluated MCUs and appear orthogonal to CPU complexity (16-bit PIC versus superscalar Cortex-M7), interconnect arbitration policy (round-robin versus priority-based), and the ISA implemented by the MCUs.

4.3 Toying with Microarchitectural Leakage

While Section 4.2 provides the first evidence of microarchitectural leakage, it does not, by itself, show that useful information can be extracted from it. In this section, we therefore assess the signal quality of this leakage and the extent to which it can reveal victim information. To this end, we devise a simple toy attack in which a victim with secret-dependent control flow, i.e., a basic if-then-else statement, branches on a secret value. We then carefully time the victim's execution and record the resulting side-channel trace. To show that all microarchitectural components analyzed in Section 4.2 can yield exploitable information, we mounted the same toy attack in three scenarios: *(i)* one exploiting a cache-based channel, *(ii)* another an interrupt-based channel, and *(iii)* the last a bus contention-based channel.

Since we already showed that cache and bus interconnect leakage are pervasive across MCUs, and that interrupt logic is common to Arm Cortex-M processors through Arm Cortex-M family, we focus our implementation effort here on a single representative platform, an STM32L5 MCU powered by a Cortex-M33.

4.3.1 Toy Attack Setup

Following the TrustZone-M threat model from Section 1.3, we mount a toy attack in which malicious code in the non-secure world gathers side-channel information to infer the control flow of a victim application in the secure world. We deploy a synthetic application as a secure world secure service that executes upon request from the non-secure world. Its code structure is intentionally designed to leak information if there is an exploitable microarchitectural channel. For testing, we use a balanced conditional statement that branches based on a secret but has a constant execution time to prevent leakage through end-to-end timing attacks.

For the bus interconnect and interrupt logic toy attacks, we devise a simple victim with a basic if-then-else structure, as shown in Fig. 25a. Depending on whether the secret value is "1" or "0", the code executes the corresponding branch and assigns that value to a variable. While the victim is overly simplistic, we strive to make the victim as lean as possible to show that even tiny differences across execution paths can be exploited using both microarchitectural components. Our goal at this stage is

```c
void victim(){
  int var = 0;
  if(secret==1)
      var = 1;
  else
      var = 0;
}
```

(a) Toy Victim.

```c
void victim(){
    int var = 0;
    if(secret==1)
        process_secret(secret);
    else
        dummy_operation(secret);
}
```

```c
int process_secret(int secret){
    store_secret = secret;
}
```

```c
int dummy_operation(int secret){
    dummy_secret = secret;
}
```

(b) Toy Victim modified for the cache toy attack.

Figure 25: Code of the synthetic victims with a secret-dependent conditional branch used in the toy attacks. Left, victim used in the interrupt logic and bus interconnect toy attacks, which simply assigns a value to a variable. Right, victim used in the cache toy attack, which calls two different functions, i.e., touches two different cache lines.

not to perform a full-fledged attack against real-world code (we will do that in Chapter 5, Chapter 6, and Chapter 7). Rather, we aim to demonstrate that if secret-dependent code executes in the secure world, the microarchitectural leakage observed in Section 4.2 can reveal useful information about the victim's control flow and, ultimately, the secret.

For the cache toy attack, we anticipated that we would not be able to distinguish the execution paths of the synthetic victim used for the bus interconnect and interrupt logic toy attacks (cf. Fig. 25a). The granularity of the microarchitectural leakage is limited to the cache line size (16 bytes), and the whole if-then-else statement fits in a single cache line. Therefore, we slightly modified the toy victim for this attack, as depicted in Fig. 25b, so that each branch intentionally touches two different cache lines, which is equivalent to having more than 16 bytes of code in each branch.

4.3.2 Toying with Cache

Our ICache toy attack leverages the timing differences that arise from accessing different levels of the memory hierarchy (see Section 2.1). Recall from Section 3.1 that a cache-based attack relies on the principle that code not present in the cache takes longer to execute than code already cached. We follow a Prime+Probe approach similar to the one used in Section 4.2. The attacker first primes the entire ICache to a known state, then invokes the victim, and finally probes each cache line to identify which ones the victim accessed. The probe phase relies on assembly code gadgets that, by executing a specific instruction sequence, carefully touch one ICache line at a time. The attacker then measures the execution time of each probe gadget. Longer execution time indicates a cache miss, revealing that the corresponding cache line was touched by the victim. By detecting this footprint on the ICache, the attacker can infer the victim's control flow and derive the secret, provided that each execution path touches different cache lines. To minimize noise from the spy code itself, we configured all non-secure code as non-cacheable, except for the buffer used for the priming and probing operations.

Results. We traced the target toy example for the two possible secret values that trigger the two execution

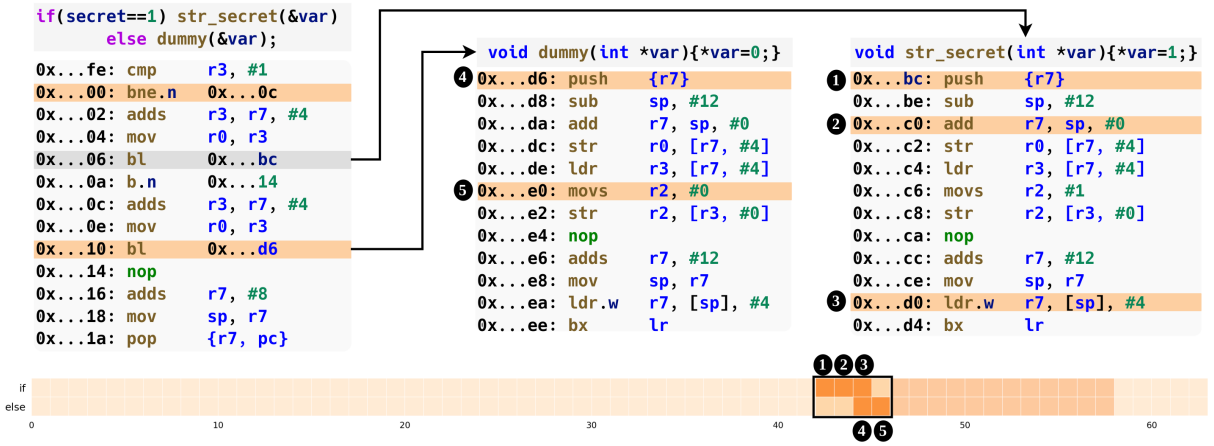


Figure 26: Illustration of the microarchitectural leakage exploited in the cache toy attack. Top, control-flow of the victim conditional branch, depending on the secret value. Orange lines highlight the instructions which experienced a cache miss. Bottom, Prime+Probe ICache-based trace of both victim conditional branches (if and else). The highlighted window (black square) corresponds to calls to `str_secret` (first row) and `dummy` (second row). Light orange indicates cache hits; dark orange indicates cache misses. For brevity, we show only 64 of 256 cache lines.

paths and compared the resulting changes in the microarchitectural state of each ICache cache line. The resulting traces, shown in Fig. 26, exhibit distinct access patterns across execution paths. The cache lines touched by the *if-path* (❶, ❷, and ❸) differ from those touched by the *else-path* (❹ and ❺). These differences allow an external observer to infer the victim's control flow and unequivocally determine the outcome of the conditional branch, thereby revealing the secret.

4.3.3 Toying with Interrupt Logic

Our interrupt logic toy attack leverages the timing variation that different instructions introduce into the interrupt-handling process. These variations allow an attacker to correlate the observed interrupt latencies with the victim's execution. By precisely interrupting the victim after each instruction, the attacker can obtain a clock-cycle-accurate trace. This trace reveals fine-grained details about each instruction executed by the victim, thereby exposing its control flow. This toy attack uses a timer configured to interrupt the victim's execution, starting from its first instruction and continuing for each subsequent one. The attacker uses this timer as both the interrupt source and a high-resolution wall-clock. When an interrupt is generated, the processor context-switches from the victim to the attacker's ISR, while the timer continues to run until it is read inside the ISR. This process measures the time the CPU takes to handle each IRQ, creating an interrupt-latency trace that captures the subtle timing effects that each victim instruction has on the interrupt logic. Since the timer can resolve differences as small as a single clock cycle, this method can reveal even very small microarchitectural leakage introduced by individual instructions.

Results. We traced the target toy victim (Fig. 25a) for the two possible secret values that trigger the two different execution paths, as shown in Fig. 27. Both paths exhibit unique interrupt-latency patterns,

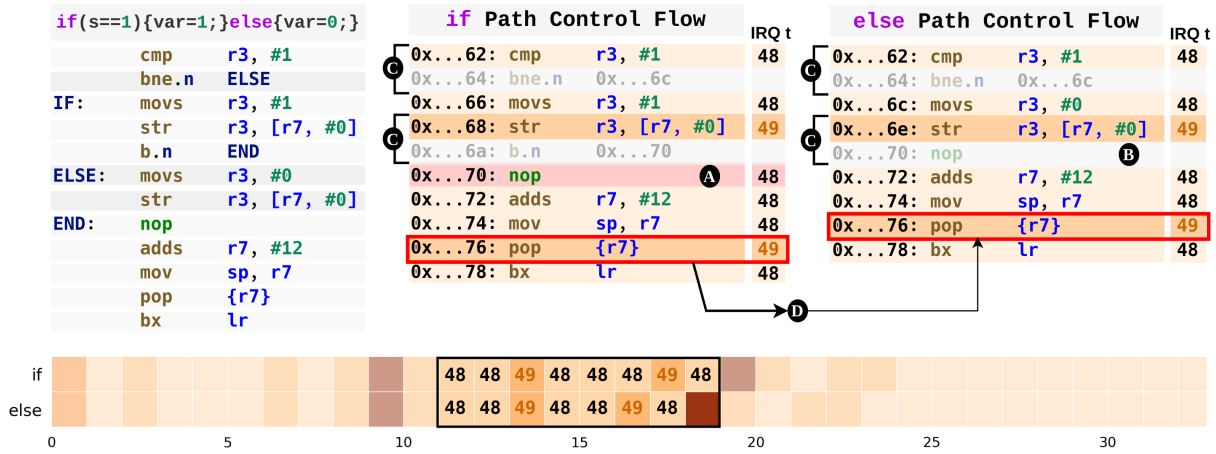


Figure 27: Illustration of the microarchitectural leakage exploited in the interrupt logic toy attack. Top, control flow of the victim conditional branch, showing all executed instructions and their effect on the interrupt-latency trace. Grey lines mark dual-issued instructions (C). Light orange indicates 48-cycle and dark orange 49-cycle IRQ latencies. Bottom, instruction-granular interrupt-latency trace. The highlighted region (black square) corresponds to the conditional-branch assembly code.

but with a subtle difference. This difference appears due to a single instruction (i.e., NOP in A), which is executed only in the *if-path*. To investigate the root cause of this difference, we profiled the victim code by sampling its program counter (PC) and obtaining a trace of each executed instruction per CPU clock cycle. We observed that certain instructions are executed together (C), i.e., in a single cycle. Upon further investigation of Arm documentation [4], we found that the Arm Cortex-M33 supports a limited form of dual-issue behavior, referred to by Arm as instruction folding [4] (see Section 2.2.2). This instruction folding feature allows certain instruction pairs to be issued and executed within a single clock cycle. In our case, this means that the NOP in the *if-path* (A) is effectively single-issued, as it cannot be folded with the preceding branch instruction (B). However, in the *else-path* (B), the corresponding NOP is folded with the preceding STR instruction, and both are dual-issued in a single cycle. Although the difference between the two execution paths is as subtle as a single instruction, this microarchitectural effect (i.e., instruction folding) is visible in the interrupt-latency trace and can be exploited to recover the secret. For instance, an external observer could leverage the shift in the interrupt latency of the POP instruction (D) to detect the outcome of the conditional branch.

4.3.4 Toying with Bus Interconnect

Our bus interconnect toy attack leverages the timing differences exposed by the arbitration logic of the MCU bus interconnect. In particular, the attacker exploits bus-arbitration side effects to recover the victim's memory-access pattern. First, the spy triggers the DMA to perform memory accesses and then invokes the victim. Using an external wall clock, the spy measures the BM transfer latency and can therefore determine whether contention occurred. Using this methodology, the spy detects secret-dependent differences in the victim's accesses to the monitored memory. A spy can observe those differences by leveraging the MCUs

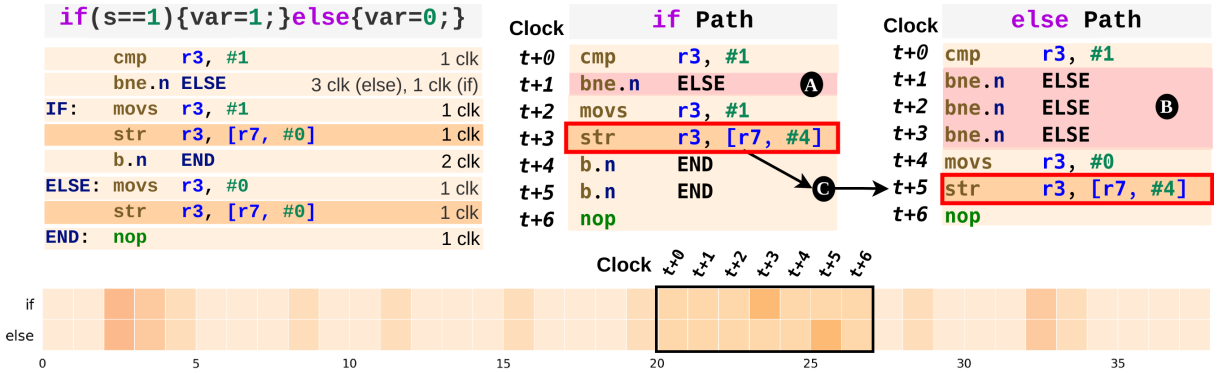


Figure 28: Illustration of the microarchitectural leakage exploited in the bus interconnect toy attack. Top, control flow of the victim's conditional branch, showing the instructions executed on each CPU clock cycle. Dark orange indicates memory contention; light orange indicates no contention. Bottom, per-cycle memory-contention trace. The highlighted region (black square) corresponds to the memory contention observed during the clock cycles in which the executed *if-* and *else-path* instructions access memory.

load-store architecture, which has two types of instructions: *(i)* instructions that interface with memory, i.e., loads and stores; and *(ii)* ALU operations. If the victim code has secret-dependent control flow, loads and stores may occur at different clock offsets across conditional paths, producing different memory-access patterns. The spy can leverage these differences to infer the provessed secret.

Results. By tracing the target code under both possible branch outcomes, we obtained the two memory-contention traces shown in Fig. 28. These traces present clear differences that allow an attacker to correlate them with the value of the secret used by the victim code. In this case, the notable difference between the two execution paths is due to the branch instruction. If the branch (BNE) is not taken, it last only one clock cycle (Ⓐ); if it is taken, it elapses three clock cycles (Ⓑ). Overall, this shifts the position of the STR instruction (Ⓒ) relative to the BNE 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 one of these two clock cycles can infer the secret by observing whether contention occurs on the shared data memory bank.

4.4 Discussion

In this chapter, we show for the first time that microarchitectural leakage on MCUs, contrary to prior belief, is widespread even on very rudimentary low-end devices. We start from two hypotheses: first, *(i)* that some microarchitectural leakage previously studied on APUs may also apply to MCUs; and second, *(ii)* that new leakage sources may emerge from MCU-specific characteristics. Our results validate both hypotheses and provide empirical evidence that microarchitectural leakage is pervasive across the MCU computing spectrum and can be leveraged to leak secret information. Our experiments also show that the perceived simplicity of MCUs is not a limitation for side-channel analysis; on the contrary, it can be a key enabler. MCU-specific characteristics expose leakage sources that would be impractical to exploit on APUs. For

instance, the bus interconnect leakage we explore in this chapter would be very difficult, if not impossible, to exploit on an APU, given the complexity of modern CCIs.

We also observe that the microarchitectural channels from our experiments exhibit both (i) a high signal-to-noise ratio and (ii) highly deterministic traces. In particular, all bus interconnect microarchitectural channels show a perfect 45-degree channel (cf. Fig. 24), meaning that each encoded secret maps to a single, unique timing value. We observe the same behavior for the interrupt logic microarchitectural channel, which also yields a perfect encoding without ambiguity. These two channels are also remarkably deterministic. Across repetitions, we obtain exactly the same wall-clock readings, without even a one-cycle deviation, for all samples. Since our experiments span multiple MCUs, this appears to be a pervasive property rather than a peculiarity of a single device. Given that MCUs are optimized for deterministic behavior, in hindsight this is a logical observation; nonetheless, it is a surprising finding in the context of side-channel attack. More importantly, it substantially simplifies the analysis of the collected side-channel traces, a property that we explicitly leverage to mount more advanced attacks in Chapter 5, Chapter 6, and Chapter 7.

Regarding the exploitation of the three analyzed microarchitectural channels, the interrupt logic and the bus interconnect yield very fine-grained traces that capture subtle differences in the victim's control flow. The same is not true for cache. While we were still able to leverage the ICache to infer coarse, secret-dependent information, this may not generalize to real-world scenarios. Carefully written code can be balanced in terms of cache-line accesses and may therefore avoid visible timing differences to an external observer. Recall that we had to modify our toy victim to expose visible differences in the traces of the cache toy attack (Section 4.3.2). While we could trace the victim's control flow in this toy setting, the resulting granularity may not be sufficient to extract useful information from real-world victims. For these reasons, we decide not to explore this microarchitectural leakage further[8], and instead focus on the other two (bus interconnect and interrupt logic), which offer more pristine side-channel information, producing high-precision, fine-grained traces.

This chapter demonstrates that MCUs are not immune to microarchitectural leakage, and that such leakage can, in principle, enable an attacker to retrieve secret information from an isolated victim domain. However, it still leaves open the question of whether these effects remain exploitable in real-world scenarios. In the next chapters (Chapter 5, Chapter 6, and Chapter 7), we continue this investigation to provide a definitive answer.

[8]Nonetheless, in Chapter 6 we show that, by combining cache with other microarchitectural components and by leveraging more advanced leakage techniques, it is possible to obtain a per-instruction view of the cache's microarchitectural state, and to exploit secret-dependent structures with more subtle leakage (e.g., toy victim 1 in Fig. 25a).

5

BUSted: Non-Convention Microarchitectural Leakage on MCUs

In Chapter 4, we have challenged the common assumption that MCUs are too simple to exhibit exploitable microarchitectural leakage. We demonstrated on multiple MCUs that not only do they leak through their microarchitectural components, but that attackers can extract secret information by observing subtle timing leakages. However, despite this promising early evidence, the question remains whether the microarchitectural leakage we observed can be leveraged to attack more generic applications, and whether the toy attacks we explored can translate to real-world side-channel attacks.

In this chapter, we contribute towards answering this question by unveiling *BUSted*. BUSted is a novel, non-conventional side-channel attack that explores the side effects of the MCU bus interconnect arbitration logic to bypass security guarantees enforced by memory protection primitives, such as the Arm MPU and TrustZone-M. This attack builds upon the insights gathered during the comprehensive microarchitectural leakage analysis we performed in Chapter 4. We have chosen to explore first the bus interconnect, instead of the interrupt logic (which we will explore next in Chapter 6), due to *(i)* its pervasiveness on MCUs (cf. Section 4.2), *(ii)* its noiseless, high-resolution microarchitectural leakage, which allows the attacker to detect differences as subtle as a single clock cycle (cf. Section 4.3), and *(iii)* because it is a fairly unexplored microarchitectural component, which its leakage seems to be specific to MCUs due to their lower-complexity characteristics. However, evolving from the microarchitectural leakage observation (cf. Chapter 4) to a full-fledged attack has proven challenging, mainly due to MCU limitations. First, MCUs typically feature a single-core CPU, hindering the concurrent execution of traditional spy and victim logic. Second, the bus interconnect is stateless; therefore, timing differences must be assessed on the fly, since they cannot be observable after victim execution.

In Section 5.1, we present our attack methodology and demonstrate how to apply known side-channel techniques to exploit this leakage. Next, in Section 5.2, we introduce *hardware gadgets*, leveraging them to overcome MCU attack limitations, such as the single-core architecture and the stateless nature of the bus interconnect. These hardware gadgets allow an attacker to use untrusted hardware blocks on MCUs as spy logic, sampling the target microarchitectural state without CPU intervention to enable concurrent

side-channel attacks on single-core devices. Notably, the highly deterministic microarchitecture of MCUs allows the attacker to synchronize the spy and victim at the clock-cycle level. Section 5.3 details the design and implementation of the BUSted attack. Finally, in Section 5.4, we evaluate BUSted on two different MCUs, demonstrating how the attacker can bypass TrustZone-M to extract a 4-digit secret PIN from a security-critical application deployed as a secure service in the TF-M uTEE.

Related Contributions, Artifacts and Dissemination.

The research presented in this chapter was first disclosed in a talk at BlackHat Asia 2023 (**T3**) and in a white paper (**W1**), prepared as part of our responsible disclosure process to communicate the vulnerability to affected parties (cf. Section 5.7.1). We made **W1** publicly available after the embargo period ended. Following our responsible disclosure process, Arm published a security advisory (**S1**) acknowledging the attack. The wolfSSL Product Security Incident Response Team (PSIRT) team also published a security advisory (**S2**), highlighting the stealthiness of our attack but stating that wolfSSL products remain unaffected. This was a proactive disclosure by wolfSSL, as we did not engage in a formal disclosure process with them because our research did not target wolfSSL software. The BUSted attack and its associated contributions were published in **P1**.

During the deployment of the BUSted attack on the SAML11 MCU, we uncovered an orthogonal vulnerability that allows a secure partition, controlled by an untrusted application, to bypass uTEE isolation boundaries and access other secure partitions or trusted kernel assets. We responsibly disclosed this issue to the affected parties and presented it at BlackHat Asia 2024 (**T2**). Furthermore, the findings presented in this chapter have motivated follow-up work that we co-authored. In **P5**, the authors adapt the techniques presented here to more complex targets, such as cryptographic applications; and in **P6**, the authors propose a formal verification approach to identify and mitigate the class of vulnerabilities discussed in this chapter. Finally, as part of our open-science effort, we have open-sourced the artifacts for this chapter in **A2**.

Summary of Related Contributions:

T1 - Talk: C. Rodrigues and S. Pinto, *"Hand Me Your SECRET, MCU! Microarchitectural Timing Attacks on Microcontrollers Are Practical"*. In: BlackHat ASIA, 2023. *Link*. *Video*. *Slide deck*.

W1 - White paper: C. Rodrigues, D. Oliveira, and S. Pinto, *"White Paper: Software-based Microarchitectural Timing Side-Channel Attacks on TrustZone-M MCUs"*, In: www.bustedattack.com, 2023. *Link*.

S1 - Security advisory: Arm. *"Clarification of Timing Side Channel Attacks on TrustZone enabled Cortex-M based systems"*, 2023. *Link*.

S2 - Security advisory: wolfSSL. *"BUSted: Side-channel attacks to TrustZone-M separation"*, 2023. *Link*.

T3 - Talk: C. Rodrigues and S. Pinto, *"What the TrustZone-M Doesn't See, the MCU Does Grieve Over: Lessons Learned from Assessing a Microcontroller TEE"*. In: BlackHat ASIA, 2024. *Link*. *Video*. *Slide deck*.

P1 - Publication: C. Rodrigues, D. Oliveira, and S. Pinto, *"BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect"*, In: IEEE Symposium on Security and Privacy (S&P), 2024. *Link*.

P5 - Publication: A. Barbosa, **C. Rodrigues**, T. Gomes, and S. Pinto, *"WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices"*. In: International Conference on Embedded Wireless Systems and Networks, 2024. *Link*.

P6 - Publication: J. Müller, A. Antón, L. Deutschmann, D. Mehmedagić, **C. Rodrigues**, D. Oliveira, M. Fadiheh, K. Devarajegowda, S. Pinto, D. Stoffel, and W. Kunz. *"MCU-Wide Timing Side Channels and Their Detection"*. In: Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24), 2024. *Link*.

A2 - Artifacts: https://github.com/ESCristiano/BUSted.

5.1 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 [209]. 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 [117, 210, 211], 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 5.2.2). 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 Fig. 29). 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.

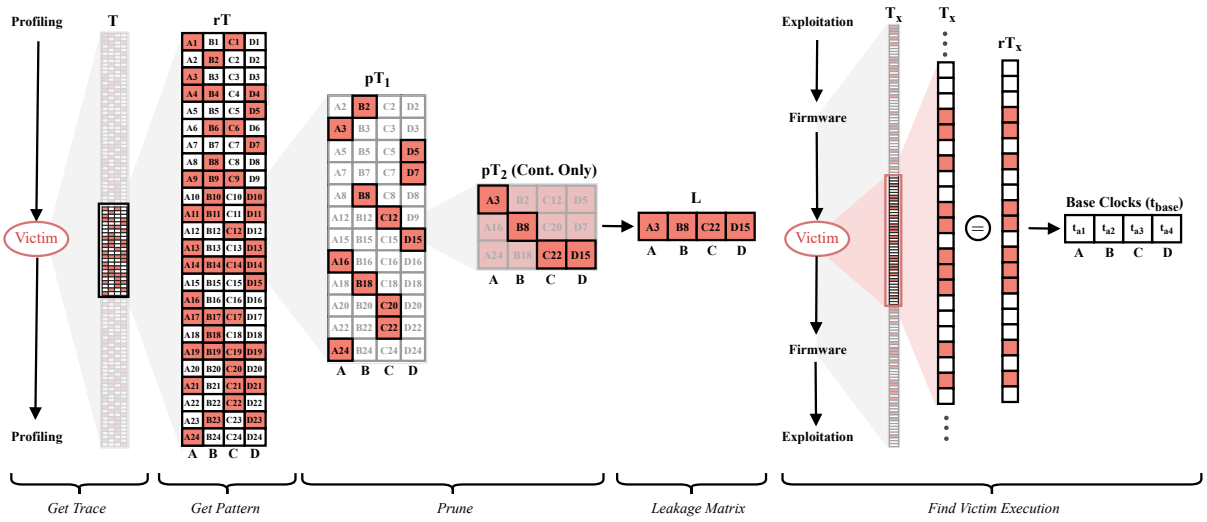


Figure 29: Attack methodology: core steps to identify "vulnerable"clock cycles. First, the attacker profiles the victim (represented with four execution paths A, B, C, and D) to obtain the trace T. Next, the attacker identifies the pattern rT for the victim's vC. The rT matrix is then pruned to yield 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.

5.1.1 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 5.1.2); or (ii) pruned 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 Fig. 29) 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

Algorithm 1: Profiling Phase.

input : Victim Binary, vB
output: Template Matrices, rT and pT

```
1 begin Profiling vB
2     Identify the vC and their execution paths E⃗
3     foreach Execution path x in E⃗ do
4         foreach Victim clock cycle t do
5             Generate contention in clock cycle t
6             Call victim to execute execution path x
7             T_t ← Record contention in clock cycle t
8         rT⃗_x ← Get vC pattern in trace T⃗
9     pT ← Prune raw template matrix rT
```

instance, in the conditional statement depicted in Fig. 25 of the bus interconnect toy attack (Section 4.3), there are two possible execution paths, i.e., $S = 1$ 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 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 5.2.2).

Get Pattern $r\vec{T}_x$. For each x execution path in $\vec{E}$, the attacker searches manually 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. Notably, 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.

5.1.2 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.

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

```
1  begin Find Execution Pattern
2      T⃗ ← Invoke victim and get trace
3      t⃗_base ← Find pattern rT⃗_x in T⃗
4  begin Attack
5      L, t⃗_offset ← Select pT entries to monitor
6      Launch the attack and invoke the victim
7      while Victim Executes do Simultaneously
8          foreach t_base in t⃗_base do
9              foreach t_offset in t⃗_offset do
10                 if t = t_base + t_offset then
11                     Generate contention in t
12                     C_t ← Check contention in t
13     foreach Pattern L⃗_x in L do
14         if C⃗ = L⃗_x then
15             S ← x
```

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 5.2.2) 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 and (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 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.

5.2 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.

5.2.1 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, have not been addressed in prior work.

***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 (see Section 2.3.3). 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[1], 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 [37, 212] 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.

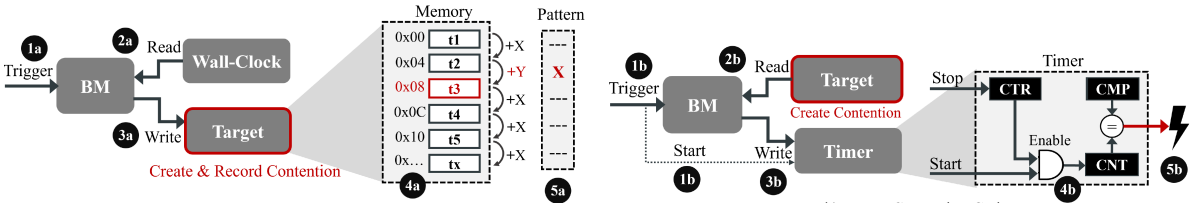
5.2.2 Hardware Gadgets

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.

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 5.3): the *record contention gadget* (Fig. 30a) and the *detect contention gadget* (Fig. 30b). 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*, Fig. 30a, 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: *(i)* a BM for data transfer, *(ii)* a free-running timer

[1]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.



(a) Record contention gadget.

(b) Detect contention gadget.

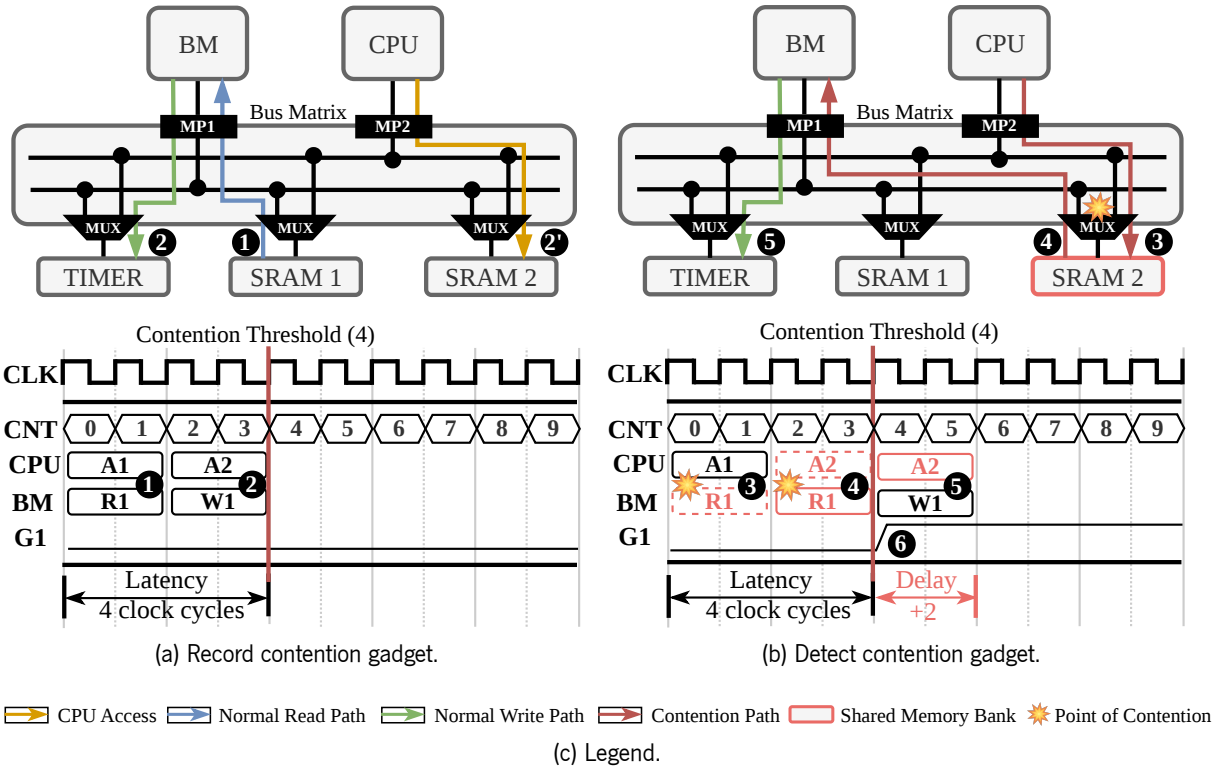
Figure 30: Illustration of two gadgets used to record contention and profile each victim clock-cycle over multiple runs (left), and to detect contention on-the-fly in a single victim run (right).

acting as a wall clock, and *(iii)* 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 Fig. 28. This gadget enables attackers to perform the profiling phase of the single-run variant or to mount the multiple-run variant of the attack.

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. Fig. 30b depicts the gadget operation and the timer peripheral. This gadget also consists of three peripherals: *(i)* the BM for data transfer, *(ii)* the timer for detecting contention, and *(iii)* 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 Fig. 28) and detect contention on-the-fly without the CPU's intervention.

5.2.3 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



(a) Record contention gadget.

(b) Detect contention gadget.

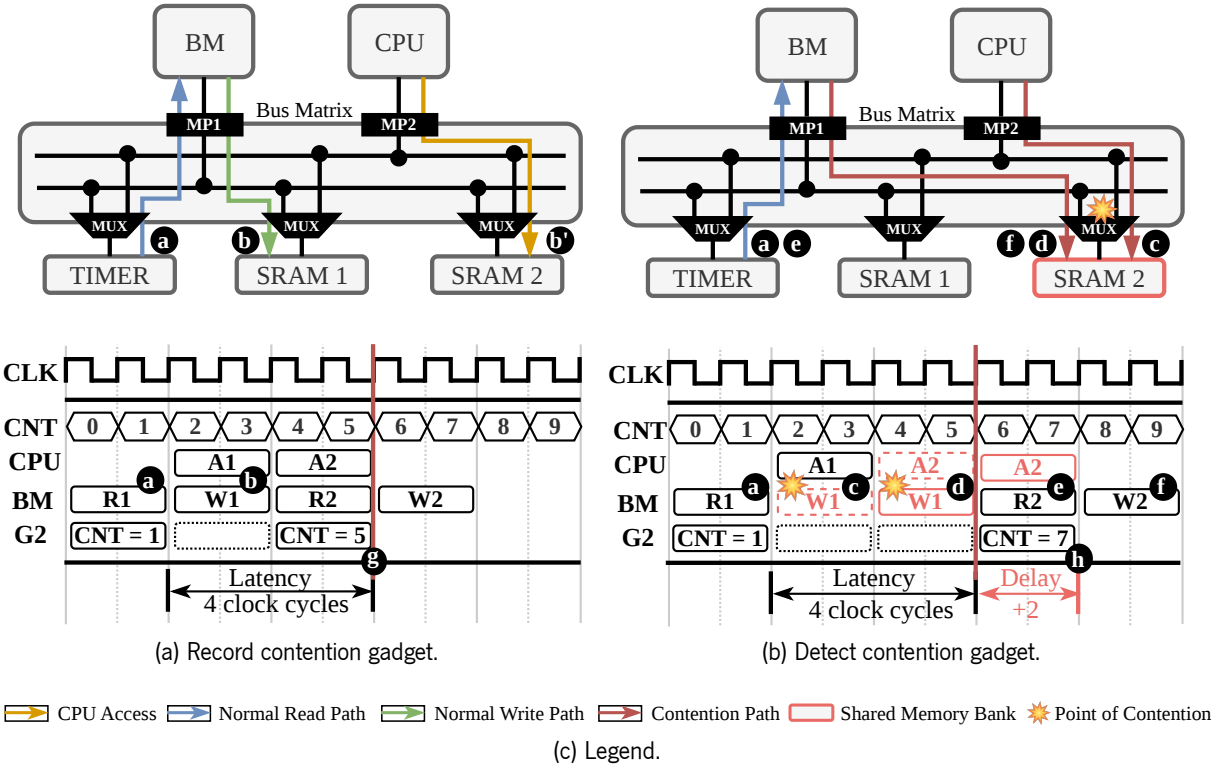
(c) Legend.

Figure 31: Illustration of the `read` stage contention, depicted without (left) and with (right) contention at the system-level, with the MCU diagram, and at bus-level, with the timing diagram.

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.

The bus view of a `read` and `write` stage contention are depicted in Fig. 31 and Fig. 32 respectively. 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 Fig. 31 and Fig. 32) 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 the figures, the bus quantum (❸) and (ⓒ) are assigned to the CPU's A1 access, and the BM transfers are postponed to the next bus quantum (❹) and (ⓓ). A contention happens between the second CPU memory access, A2, and the delayed BM operation, R1 (❹) and W1 (ⓓ). 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 (❺) and (ⓔ).

Read Stage Contention. Fig. 31 illustrates the timing and bus level interactions at the `read` stage. In a memory transaction without contention (Fig. 31a), the BM reads from SRAM 1 (❶) and then writes the read value to the timer control register (❷). At the same time, the CPU does two consecutive accesses to SRAM 2 (❷'). In this case, there is no contention because there is no dispute for the SRAM access. When CPU and BM issue concurrent accesses (Fig. 31b), the bus matrix round-robin arbitration process



(a) Record contention gadget.

(b) Detect contention gadget.

(c) Legend.

Figure 32: Illustration of the `write` stage contention, depicted without (left) and with (right) contention at the system-level, with the MCU diagram, and at bus-level, with the timing diagram.

the CPU A1 and delays the BM R1 read (❸). Therefore, R1 is delayed by one bus quantum (❹), and thus the W1 write operation (❺). 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 (Fig. 31a), the timer starts counting upon the BM transfer A1 (❶) and is immediately stopped by the BM write W1 (❷), 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 (Fig. 31b), the BM write W1 is delayed (❺), and the timer interrupt is triggered (❻).

Write Stage Contention. Fig. 32 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 (Fig. 32a), 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 (Fig. 32b), 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 (Fig. 32a), the BM will read a value of 1 and 5, with

a transfer latency of 4 (g). If there is contention (Fig. 32b), the BM will read a value of 1 and 7, with a transfer latency of 6 (h).

5.3 BUSted Attack

In this section, we introduce the BUSted attack. Like other side-channel attacks [35, 37, 64, 68, 147, 154], 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.

5.3.1 Off-the-Shelf Smart Lock Application

The selected smart lock application is a component of Vulcan [213], a vehicular authentication system that adheres to a reference implementation of a trusted keypad[2] provided by Texas Instrument [214]. This very same application was used in the Nemesis attack [37]. There are also popular open-source firmwares such as QMK [218] and TMK [219] that use a similar logic to interface with keyboards, i.e., larger keypads.

We embedded this off-the-shelf logic in a TA deployed in the secure domain of a TustZone-enabled MCU. It receives service requests from the main 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 Fig. 33.

To steal the PIN, the spy monitors the victim's control flow and checks whether the victim executes the secret-dependent path (req square in Fig. 33). 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 of our BUSted attack paper [8]) 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 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.

[2]Texas Instruments offers several products based on the trusted keypad reference implementation [214], e.g., the Low-Power Hex Keypad [215], the BOOST-IR BoosterPack Plug-in Module [216], and a USB Keyboard [217].

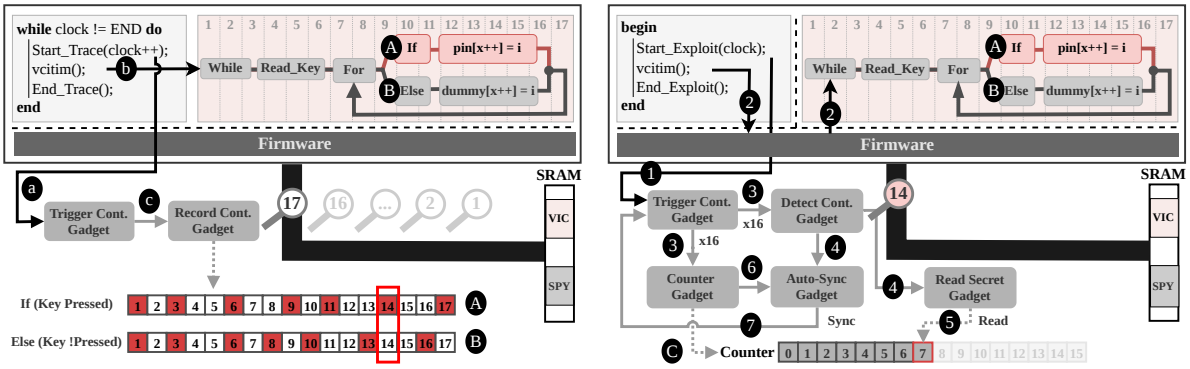
```
signed int read_keypad(void){
    int is_pressed, mask = 0x1;
    int new_key_state = get_keypad_state();                    read key
    for (int key = 0; key < KYPD_NB_KEYS; key++){
        is_pressed = (new_key_state & mask) & ~(key_state & mask);
        if (is_pressed)                      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

Figure 33: 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`.

5.3.2 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 the attack methodology (see Section 5.1), we resort to two SGNs: *(i)* one for the profiling phase and *(ii)* another for the exploitation phase.

The *profiling SGN* (pSGN), Fig. 34a, 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), Fig. 34b, 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.



(a) Profiling phase.

(b) Exploitation phase.

Figure 34: Profiling (left) and Exploitation (right) 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.

5.3.3 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. Fig. 34 shows an overview of the attack.

Offline Profiling Phase. In this phase, Fig. 34a, 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[3] (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 Fig. 34a), 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: *(i)* one when a key is pressed (`if` path Ⓐ) and *(ii)* another when there is no key press (`else` path Ⓑ).

Online Exploitation Phase. In the exploitation phase, Fig. 34b, 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 (❷) 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

[3]We mimic this behavior by instrumenting the keypad with a shunt resistor.

(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 (❶) the eSGN to generate contention in the selected clock cycle and invokes the victim (❷). Then, the *trigger gadget* creates 16 triggers (❸), 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 (❸) is fed into the *counter gadget*, which will keep track of the number of triggers, from 0 to 15 (❻). Upon reaching the count of 15, the *counter gadget* signals the completion of the `for` loop (❻).

(**Step 3**) 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* (❻) to reveal the key (secret).

(**Step 4**) 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 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.

5.4 Evaluation

We implemented BUSted on the NUCLEO-L552ZE-Q [202] and ATSAML11E16A [201] (Armv8-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 [46]. Although there are other TEE kernel options, we either were not granted an evaluation license (e.g., ProvenCore-M [84]) or we decided to exclude them due to limited adoption (e.g., mTower [85], and uTango [86]). 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 [220], Zephyr [221]), we focused on security-oriented TEE hardware technologies for Arm MCUs (TrustZone-M).

5.4.1 TF-M: PSA Isolation Level 2

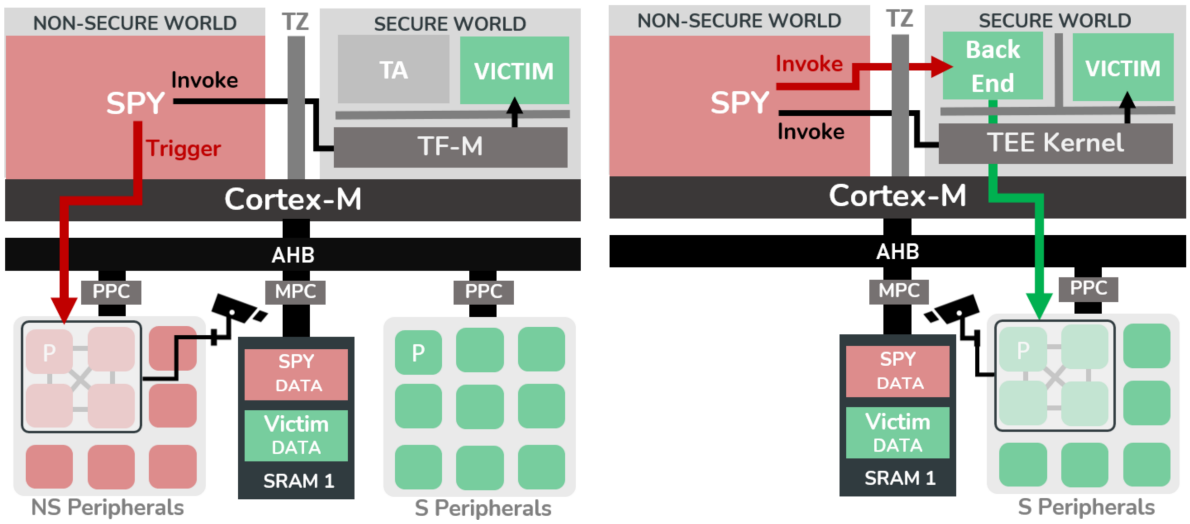
Attack Setup. We leveraged TrustZone-M to create a trusted and an untrusted domain (secure and normal worlds, respectively). As depicted in Fig. 35a, the spy operates in the normal world, while the victim (smart lock application) is encapsulated in a trusted application (TA). The TA runs in secure unprivileged thread mode on top of the privileged trusted kernel (TF-M). TF-M [46] was configured at PSA isolation level 2 [48], which provides isolation between the secure and non-secure world (TrustZone), as well as between trusted kernel and TAs (via 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: *(i)* one with 192KB (where the NS data is located) and *(ii)* 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.

5.4.2 PSA Isolation Level 3

Attack Setup. Fig. 35b 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. Thus, the TEE kernel should be configured with PSA isolation level 3 [48], 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 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



(a) TF-M: PSA Isolation Level 2.

(b) PSA Isolation Level 3.

Figure 35: Illustration of the BUSted attack setup.

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 CPU holds the highest priority in the bus arbitration logic, which grants access to the CPU first and only services remaining BMs when the CPU is idle. Consequently, all other BMs experience contention.

5.4.3 Platform Constraints on SGNs

Table 5 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.

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 Section 5.2) do not have the counter register synchronized

Table 5: 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 (–).

Gadgets	**Timers**		**DMAs**		**DMA MUX**		**Event System**		**LUTs**	
	SAM	*ST*	*SAM*	*ST*	*SAM*	*ST*	*SAM*	*ST*	*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**		**Bus**		**Memory**		**I-Cache**		**N Peri.**	
	CPU	Freq.	Arch.	Arb.	SRAM	FLASH	Size	Status	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

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.

5.4.4 Real Victim trace

Fig. 36 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 (Fig. 33), 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 Section 5.1.1), we obtain the pT matrix with all the unique contention

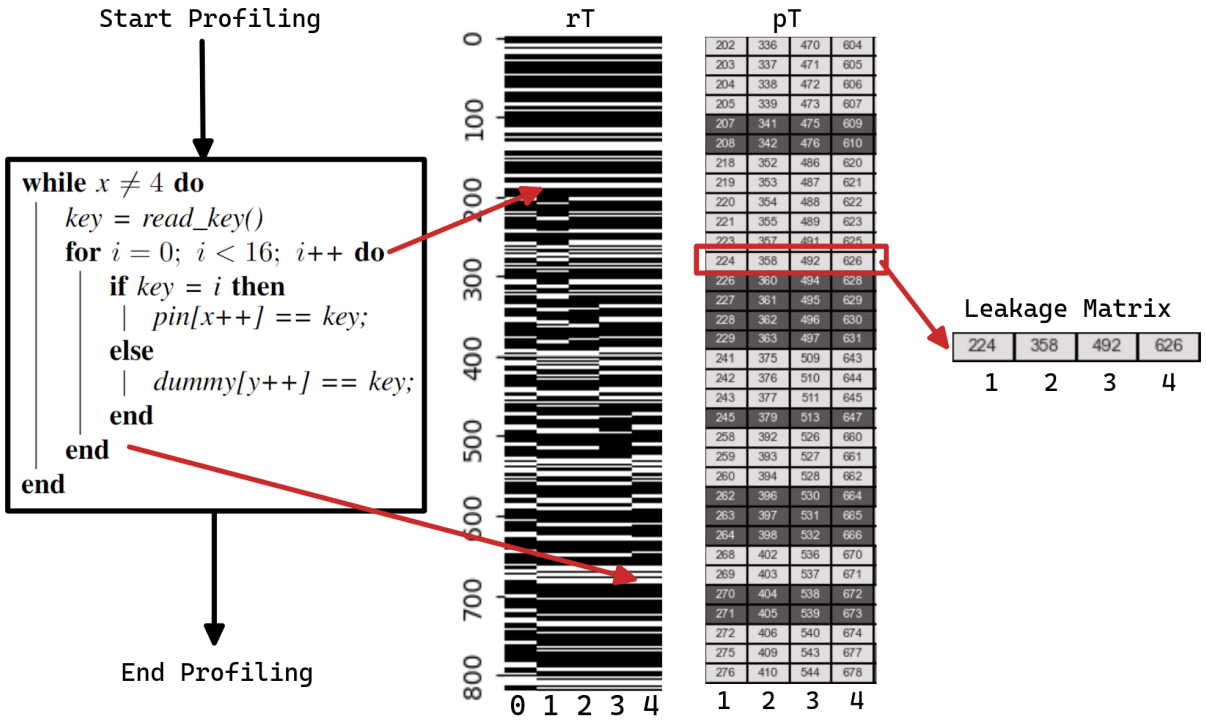


Figure 36: Smart lock application trace recorded in SAM. Both boards generate similar traces. We limit the trace to 4 execution paths, i.e., 4 keys.

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.

5.5 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 [222]. 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 [223]. 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 §5.4.2), 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.

5.6 BUSted Limitations.

BUSted Generalization to Other Applications. We acknowledge that BUSted is highly tailored to the target application; however, we argue that our attack methodology (see Section 5.1) 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 [37] and SGX-Step [58] (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 (cf. Section 4.2.3), 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 Section 4.2. 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 5.4).

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.

5.7 Summary

In this chapter, we presented a new class of software-based microarchitectural timing side-channel attacks affecting MCUs that exhibits characteristics fundamentally different from prior exploits. Our findings provide empirical evidence challenging the common assumption that MCUs are too simple to be vulnerable

to microarchitectural timing side-channel attacks. In fact, our attack methodology operates on the opposite principle, i.e., the more deterministic and simple the microarchitecture, the better. Microarchitectural components, such as caches and complex predictors, and performance features, such as out-of-order or speculative execution, are not required for our attack to succeed. Instead, we leverage a microarchitectural structure that is pervasive across modern MCUs, including the most basic 16-bit AVR MCUs (cf. Section 4.2.3). We demonstrated that the microarchitectural leakage first explored in Chapter 4 can be leveraged to mount end-to-end attacks against real-world applications. This constitutes the first evidence that MCUs not only exhibit microarchitectural leakage, but that such leakage can be exploited by an attacker to extract secrets from code executing in an isolated domain, i.e., TrustZone-M. Moreover, we showed that intrinsic MCU characteristics (notably, their deterministic behavior and the wide availability of peripherals) create new exploitation opportunities specific to this class of systems, for instance, by enabling hardware gadgets. These findings demonstrate that the system security community must revise the prevailing narrative around side-channel attacks on MCUs. Rather than being *too simple to be exploitable*, MCUs are *simple, yet exploitable*.

5.7.1 Ethical Considerations

Responsible Disclosure. We first shared our findings with the Trusted Firmware (TF) security team and the ST Microelectronics (ST) PSIRT. Both teams acknowledged the exploit. The TF security team argues that no vulnerability exists in the TF-M code (TF-M is simply a convenient framework for demonstrating the exploit), and that the vulnerability lies 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 ST's product but rather related to Arm's TrustZone-M (Armv8-M), we do not necessarily agree with their view. We also reported our findings to Arm, which acknowledged the attack and released a public statement on its website [224]. We have also reported our findings to other silicon manufacturers where we measured the microarchitectural leakage in Section 4.2.3. We had discussions with Microchip, NXP, and Silicon Labs, which did not yield any outcome. As for GigaDevice, we have not received any response.

6

M-Step: Single-Step Framework for TrustZone-M.

In Chapter 5, we demonstrated that microarchitectural leakage on MCUs can be exploited in real-world scenarios. However, while BUSted established the bus interconnect as an exploitable microarchitectural channel, we have yet to determine whether side-channel leakage on MCUs is, as on APUs, a pervasive phenomenon spanning multiple components. Concretely, in this chapter, we investigate the interrupt logic, the second microarchitectural component we identified as a highly promising target (cf. Section 4.4), and revisit the leakage previously demonstrated in Section 4.3. We analyze the main challenges of mounting these attacks on the interrupt logic in Section 6.1 and provide a root-cause analysis by profiling interrupt-handling behavior on our Cortex-M test platform in Section 6.2.

Building on these insights, we present *M-Step*, the first single-stepping framework for side-channel analysis on Arm TrustZone-M (Section 6.3). Single-stepping frameworks enable an attacker to sample microarchitectural state with instruction-level granularity across multiple components by emulating single-stepping on devices configured with debugging disabled. These frameworks exploit interrupt logic to generate a tightly controlled stream of interrupts such that each victim instruction is followed by an interrupt, forcing the CPU to advance execution by exactly one instruction at a time. While this technique has been exploited repeatedly on APUs to obtain high-resolution leakage from multiple microarchitectural structures, Arm Cortex-M-specific challenges (cf. Section 6.1) have prevented its deployment on MCUs. In this chapter, we show that similar capabilities can be achieved on Arm Cortex-M.

We evaluate M-Step (Section 6.4) through microbenchmarks, showing that it enables reliable single-stepping of generic victim code and allows an attacker to sample microarchitectural state with high temporal resolution across multiple microarchitectural components. We deploy M-Step across multiple attack scenarios and mount four covert-channel that each leverage a distinct microarchitectural structure. In doing so, we show that microarchitectural leakage on MCUs is, as on APUs, a transversal phenomenon that spans multiple components. To conclude the chapter, we mount the same proof-of-concept attack previously studied by Nemesis [37] and BUSted [8] to extract a 4-digit PIN from a smart lock application that interfaces with a trusted keypad.

Related Contributions, Artifacts and Dissemination

The results presented in this chapter have been published in the **P2** scientific article. To support follow-up research and enable reproducibility, we release all artifacts (**A3**) required to reproduce and evaluate our findings. We submitted these artifacts to the USENIX Security Artifact Evaluation, where three reviewers independently reproduced and validated our results. As part of this process, we were awarded all three USENIX Security Artifact Evaluation badges (Artifacts Available, Functional, and Reproduced), confirming that the artifacts are publicly accessible, exercisable, and sufficiently mature to be reused by other researchers. We maintain M-Step as an open-source project at `https://github.com/M-Step-Framework` and commit to improving the codebase and documentation to support adoption and long-term maintainability.

Summary of Related Contributions

P2 - **Publication: C. Rodrigues**, M. Bognár, S. Pinto, and J. Van Bulck *"M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M"*, (To appear) In: 35th USENIX Security Symposium, 2026. *Link*.

A3 - **Artifacts:** https://github.com/M-Step-Framework.

6.1 Problem Statement

This section defines the attack-specific assumptions, reviews prior work on single-stepping frameworks, and outlines the challenges for precise single-stepping on Arm Cortex-M.

6.1.1 Attack-Specific Assumptions

We assume the attacker may only get a single opportunity to extract the secret, for example during key generation [68], meaning the attack must succeed using a single trace and cannot rely on accumulating noisy side-channel information across multiple runs, like we did in the BUSted attack (cf. Chapter 5). Note that writing general-purpose applications in constant time is notoriously challenging and error-prone. As a case in point, we will show in Chapter 7 that even Arm's vetted Mbed TLS and TF-M libraries exhibit subtle non-constant-time behavior that can be exploited in practice using our high-resolution M-Step attack framework.

6.1.2 Single-Stepping Frameworks

Since debug features are commonly disabled in production TEEs, prior work has developed specialized "single-stepping" attack frameworks that leverage privileged timer interrupts to advance the victim TEE one instruction at a time, enabling side-channel observations with maximal instruction-level temporal resolution. Table 6 summarizes existing single-stepping frameworks, highlighting a growing trend in recent

Table 6: M-Step is the first true single-stepping framework for production TEEs on low-end MCUs without virtual memory. Comparison in terms of target TEE: memory isolation *(Mem)*, secure interrupts *(IRQ)*, production vs. academic *(Prod)*; and features: maximum instruction temporal resolution *(Resol)*, presence of single-stepping mitigations *(Fix)*, open-source availability *(Open)*, extensible framework design *(Ext)*.

				Target			Single-Step			
	Framework	**Yr**	**Architecture**	Mem	IRQ	Prod	Resol	Fix	Open	Ext
High-End	SGX-Step [58]	'17	Intel SGX	●	●	●	✓ 0-1	◐ [225]	●	●
	CacheGrab [63]	'19	Arm TrustZone-A	●	●	●	✗ >1	○	●	◐
	Load-Step [64]	'21	Arm TrustZone-A	●	●	●	✗ >1	○	○	○
	SEV-Step [62]	'23	AMD SEV-SNP	●	●	●	✓ 0-1	○	●	●
	TDX-Step [59]	'23	Intel TDX	●	●	●	~ ?	● [59]	○	○
	TDXdown [60]	'24	Intel TDX	●	●	●	~ 1/>1	◐ [226]	●	○
	TDXploit [61]	'25	Intel TDX	●	●	●	✓ 1	● [227]	●	○
Low-End	Sancus-Step [228]	'19	openMSP430	●	●	○	✓ 1	◐ [229]	●	◐
	Taking a look [230]	'19	Armv7-M XOM	◐[1]	○	●	~ 1/>1	● [6]	○	○
	RIPencapsulation [194]	'24	TI MSP430/2 IPE	◐[1]	○	●	✓ 1	○	●	○
	M-Step (this chapter)	'25	Arm TrustZone-M	●	●	●	✓ 0-1	○	●	●

[1] No or limited support for controlled entry points.

years toward developing such techniques for both high-end TEEs with virtual memory and low-end microcontrollers.

High-End TEEs. SGX-Step [58] was the first framework of its kind, enabling an attacker to deterministically single-step Intel SGX enclaves one instruction at a time. Its open-source availability and extensible Linux-based framework design rapidly established SGX-Step as the de facto standard for mounting high-resolution side-channel attacks against SGX [65]. To date, it has been used in over 48 academic publications [231], supporting a wide range of attacks ranging from interrupt latency and instruction counting to high-resolution cache and branch predictor probing, as well as transient-execution and memory-safety exploits. A detailed root-cause analysis [225] later revealed that SGX-Step critically depends on virtual memory CPU features to delay enclave instructions by leveraging microcode-assisted page table walks in high-end Intel processors. This insight led to the AEX-Notify [225] hardware-software co-design deployed in modern Intel SGX platforms to mitigate single-stepping by prefetching the next instruction's page on interrupt.

Motivated by the success and utility of SGX-Step, researchers developed similar frameworks for more recent virtual-machine-based TEEs. This includes SEV-Step [62] for AMD SEV, as well as TDX-Step [59, 232] developed by Google and Intel researchers for Intel TDX. For the latter, Intel introduced a mitigation in the TDX security monitor that detects high interrupt rates and adds randomized delays to disrupt single-stepping attacks; however, this mitigation has suffered from subtle flaws [60, 61], prompting subsequent patches.

While there have been efforts to create single-stepping frameworks for Arm TrustZone-A [63, 64], both the threat model and architectural design of TrustZone-A have posed significant barriers. First, unlike in

Intel SGX/TDX and AMD SEV, the attacker in TrustZone-A does not control the trusted application's memory or its (extended) page tables, which are managed by the EL1 trusted OS. Second, all NS interrupts during S world execution are routed through the EL3 secure monitor before being delivered to the NS world, leaving the attacker with no direct control over interrupt handling. CacheGrab [63] and Load-Step [64] are the only frameworks that have used interrupts to control victim execution on TrustZone-A, both relying on inter-processor interrupts rather than timers. However, neither was able to achieve true instruction-level single-stepping on arbitrary instructions.

Low-End TEEs. To the best of our knowledge, only three prior works have explored single-stepping on MCUs [194, 228, 230]. Sancus-Step [228] is an experimental unpublished framework for the niche academic Sancus [198] architecture. The other two frameworks [194, 230] use single-stepping to recover code from proprietary execute-only memory (XOM) implementations by exploiting residual data in registers after interrupts. However, such attacks would be ineffective against production TEEs like Arm TrustZone-M, which clear registers on non-secure interrupts. Moreover, the variants [194, 230] targeting Arm MCUs fail to account for the heterogeneous nature of Arm Cortex-M instructions, presenting a major limitation of their single-step primitives. In summary, existing MCU-based single-stepping frameworks do not target production TEEs, fail to address key Cortex-M architectural challenges, and only apply to weaker XOM-based isolation mechanisms.

6.1.3 Single-Stepping Challenges on Arm MCUs

Before presenting the design of M-Step in the next section, we first summarize the key challenges that a successful single-stepping framework must overcome on Cortex-M, rooted in its microarchitectural and architectural features. While certain MCU characteristics such as low-noise timers, deterministic execution, and simple microarchitectures simplify some aspects of single-stepping relative to high-end frameworks, the interrupt behavior and the microarchitectural heterogeneity of Cortex-M MCUs introduce distinct challenges that require new techniques to achieve reliable single-stepping.

C1 - Heterogeneous instructions. In contrast to prior single-stepping frameworks on Intel, AMD, and Arm Cortex-A, a key challenge on Cortex-M is handling varying interrupt behavior depending on the instruction class, due to the interruptible nature of Arm Cortex-M instructions.

C2 - Observing victim progress. High-end single-stepping frameworks use page faults to advance victim execution and page-table "accessed" bits [37, 62, 225] or performance monitoring counters [62, 233] to detect zero-steps, where the timer fires too early and the TEE fails to make progress. On Arm Cortex-M, such features are unavailable due to the lack of virtual memory and advanced CPU features.

C3 - Generic instruction streams. A practical framework must support single-stepping arbitrary instruction streams in the victim TEE, without hardcoding expected instruction latencies and timer intervals for specific attack scenarios.

C4 - Heterogeneous memory. Unlike high-end processors, which typically feature a unified memory architecture, MCUs commonly have highly heterogeneous memory architectures, potentially leading to varying timing behaviors for instructions accessing different memory banks.

C5 - Resource constraints. Unlike high-end frameworks that run on feature-rich OSes with abundant resources, MCUs are memory-constrained and operate in cross-development environments. As a result, leakage traces must be lightweight and typically have to be offloaded to the host environment for visualization and post-processing.

6.2 Single-Stepping on Cortex-M

In this section, we present key insights from profiling Cortex-M's interrupt handling, revealing distinct instruction behaviors. Building on these findings, we develop a generic algorithm to enable single-stepping over arbitrary instruction streams, a technique we dubbed M-Step.

All experiments were performed on an Arm Cortex-M33 CPU implementing Armv8-M Mainline, the most feature-rich Cortex-M architecture.[1] We selected the STM32L5 MCU, which features the target CPU (Cortex-M33) clocked at 110 MHz with ICache, TrustZone-M, and FPU with lazy stacking enabled.

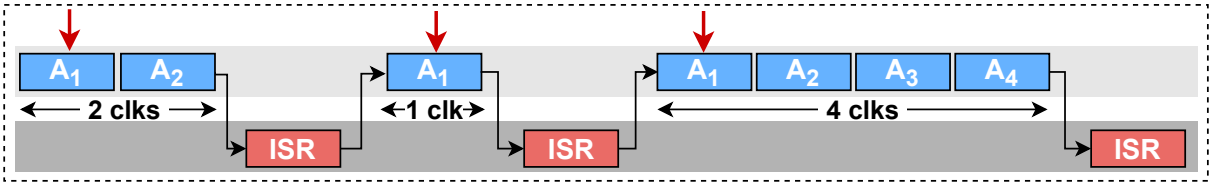
6.2.1 Profiling Instruction IRQ Behavior (C1)

Recall from Section 2.2 that certain Arm Cortex-M instructions can be interrupted mid-execution to reduce interrupt latency. To accurately classify the full range of microarchitectural behaviors exhibited during interrupts, we first profile the interrupt behavior of individual instructions (visualized in Fig. 37) by systematically triggering interrupts at different clock cycles (red arrows), measuring the resulting IRQ latency, and recording whether the instruction completes, i.e., the PC advances. This allows to distinguish three instruction classes; we classify all non-ICI instructions as *atomic*, and divide ICIs into two subgroups: *resumable* and *restartable* instructions.

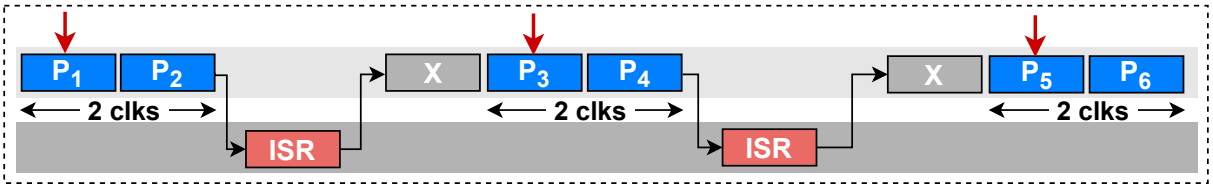
Atomic Instructions (Fig. 37a). These instructions cannot be interrupted during execution: the CPU defers handling any pending interrupt request until the instruction has fully completed. Atomic instructions include register-to-register operations (`MOV`, `ADD`) and single-word memory accesses (`LDR`, `STR`).

Notably, while most atomic instructions incur only a single-cycle delay upon interrupt, we found that some, such as `LDR`, and `STR`, can inflict two or more cycles. Further examination attributed this variation to different memory backends on our target platform, which includes separate SRAM1, SRAM2, and flash partitions (cf. C4). Table 7 summarizes our profiling results, showing that accessing SRAM1 takes 1 cycle and accessing SRAM2 or flash takes 2 cycles, with an additional cycle of IRQ latency added if the stack pointer points to a different partition than the accessed memory. This extra delay is likely due to the

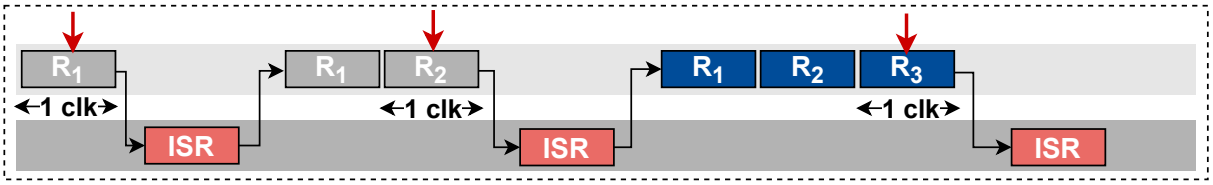
[1] Armv8-M Baseline and Armv7/6-M are subsets of Armv8-M Mainline. Armv8-M Baseline and Armv6-M do not support resumable instructions.



(a) *Atomic* instructions of varying lengths (two, one, and four cycles). A_x indicates the execution cycle state of each atomic instruction.



(b) *Resumable* instruction performing three memory loads, one in each 2-cycle period. P_x indicates the execution cycle state of the instruction. Gray *X* boxes represent undocumented extra clock cycles.



(c) Three-cycle *restartable* instruction. R_x indicates the execution cycle state of the instruction. Gray boxes represent cancelled execution cycles.

Figure 37: Cortex-M interrupt behavior for different instruction types and interrupt arrival times (red arrows).

hardware's automated interrupt stacking requiring a bank switch. In the default configuration of our target board, the secure world stack pointer is always in SRAM1. Thus, interrupting identical instructions (e.g., LDR) may exhibit different timings depending on the accessed memory region: 1 cycle for SRAM1, 3 for SRAM2, and 2 for literal pools. Though platform-specific, this timing behavior introduces an additional source of side-channel leakage.

> **Single-stepping goal:** IRQs can arrive anywhere during the execution of atomic instructions, ideally in the first cycle after the context switch to accurately measure latency.

Resumable Instructions (Fig. 37b). These instructions support an interrupt-followed-by-resume behavior: when an interrupt occurs during their execution, the CPU saves the execution context, services the interrupt, and then resumes the instruction from the point it was interrupted. Examples include PUSH and POP with multiple registers. We observed undocumented behavior for these instructions: when resuming after an interrupt, an extra clock cycle is needed to restore CPU context before the instruction can proceed and make progress. Furthermore, these instructions cannot be abandoned at *any* cycle, as individual load and store operations within the larger resumable instruction are handled atomically and will be completed before handling a pending interrupt request.

> **Single-stepping goal:** Resumable instruction can be interrupted during their execution, but subsequent IRQs for the same instruction should arrive from the second cycle following the context switch to make progress.

Table 7: Interrupt latencies for memory loads targeting different memory partitions and stack pointer (SP) configurations.

			IRQ Latency	
Inst.	**Operand**	**Mem. Access**	SP (SRAM1)	SP (SRAM2)
LDR	$\{rx, sram1\}$	$rx \leftarrow$ [sram1]	1	2
LDR	$\{rx, sram2\}$	$rx \leftarrow$ [sram2]	3	2
LDR	$\{rx, flash\}$	$rx \leftarrow$ [flash]	2	2

Restartable Instructions (Fig. 37c). These instructions are restarted when interrupted: the CPU abandons partial execution the cycle after IRQ arrival and reissues the instruction after the ISR completes. Documented examples include multi-word memory accesses like LDRD and STRD. However, we found that several other instructions also follow this restart behavior. In particular, multiply and divide instructions consistently restart on interrupt, contradicting Arm's definition of ICIs as strictly "multicycle load and store" instructions [4]. Moreover, we observed that some instructions documented as resumable (e.g., POP) actually adopt a restart strategy when the PC is included in the register list.

Single-stepping goal: Restartable instructions only make progress if the IRQ occurs during their final execution cycle.

6.2.2 Single-Stepping Instruction Streams

The key challenge for any single-stepping framework is configuring the timer to interrupt TEE execution at precisely the intended point. If the timer fires too early (i.e., during context switch, in the first cycle of a previously interrupted resumable instruction, or in a non-final cycle of a restartable instruction) the secure world makes no progress, resulting in a *zero-step*. If it fires too late, multiple secure-world instructions may execute before the interrupt, yielding a *multi-step*. Unlike prior single-stepping frameworks, M-Step must also handle an additional case: the *partial-step*, where an interrupt occurs during a resumable instruction, causing internal progress without advancing the program counter.

The following subsections first present an approach for distinguishing zero- and partial-steps using interrupt latency (Section 6.2.2.1, C2), followed by a dynamic timer configuration algorithm (Section 6.2.2.2, C3) that ensures progress without executing more than one victim instruction by conservatively underestimating the timer interval and gradually increasing it upon detecting repeated zero-steps.

6.2.2.1 Distinguishing Victim Progress (C2)

State-of-the-art zero-step detection relies on virtual memory [37, 62, 225], which MCUs lack. We, therefore, propose to leverage interrupt latency as a deterministic channel to distinguish victim progress in terms of zero- and partial-steps.

Detecting Zero-Steps. We leverage two key observations to detect zero-steps: *(i)* when M-Step is zero-stepping, it will keep observing the same interrupt latency over and over; and *(ii)* code is typically a mixed set of single- and multi-cycle instructions. Thus, it is very unlikely that we will observe a long trace of identical interrupt latencies, unless we are zero-stepping. As a heuristic, we consider to be stuck in zero-stepping when we observe N same-cycle interrupt latencies in a row, where N is a configurable threshold which we call the *streak limit*.

Based on the consistently observed interrupt latency during zero-step streaks, we can further distinguish two distinct cases: *(i)* if the interrupt occurs before any secure-world instruction runs, the latency is $\mathcal{L}_0$, i.e., the fixed context-switching time; whereas *(ii)* if it hits during a restartable instruction or during the first cycle of a previously interrupted resumable instruction, we observe $\mathcal{L}_1 > \mathcal{L}_0$. Thus, once the context-switching latency has been established and the base timer interval is correctly configured, zero-steps should only occur for resumable or restartable instructions.

Detecting Partial-Steps. To detect partial-steps, which can only occur for resumable instructions, we leverage the microarchitectural insight from the instruction profiling presented in Section 6.2.1. Conceptually, interrupting resumable instructions appears similar to a sequence of atomic load or store instructions. However, to continue resumable execution from where it was interrupted, the CPU requires an extra clock cycle to restore the context (cf. Fig. 37b) Thus, partial-steps will require a higher timer configuration threshold, otherwise resulting in zero-steps. This makes partial-steps unequivocally distinguishable from all other events and instruction types.

6.2.2.2 Dynamic Timer Interval Adjustment (C3)

To address the remaining challenge of autonomously single-stepping full programs (C3), we introduce a generic timer-configuration algorithm that operates without any prior knowledge of the victim's instruction stream. The algorithm is realized as a state machine, summarized in Fig. 38, which iteratively schedules timer interrupts to infer the currently executing instruction and the timer interval to single-step it.

The key idea is to increase the timer interrupt threshold iteratively yet *conservatively*, enabling forward progress while strictly preventing multi-stepping in the victim program. The algorithm always begins with the minimum single-step interval sufficient for the simplest case of interrupting atomic instructions (cf. Fig. 37a). Only after reaching the *streak limit* of N consecutive zero-steps does the algorithm attempt to allocate an additional cycle to handle a resumable instruction. Similarly, additional cycles for restartable instructions are granted only if execution remains stuck in zero-stepping; in that case, the interrupt threshold is increased incrementally one cycle at a time, and only after the previous threshold has again reached the streak limit.

This conservative strategy guarantees that no instructions are skipped, at the cost of potentially many zero-steps, which may slow down single-step trace extraction. In Section 6.6, we explore possible techniques to avoid these redundant zero-steps if the attacker has partial knowledge of the victim's binary code.

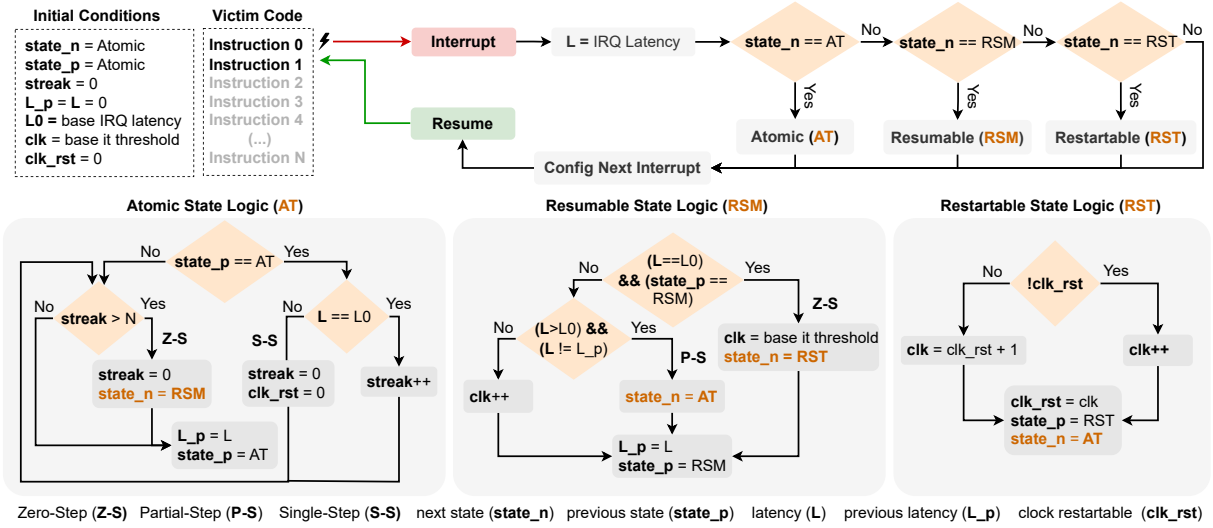


Figure 38: Simplified M-Step algorithm for dynamically adjusting the single-step timer interval based on the interrupt behavior of the currently executing instruction. On each interrupt, the algorithm updates its accumulated state ($state_n$) by combining the observed interrupt latency ($\mathcal{L}$) with the information carried over from the previous interrupt ($state_p$, $\mathcal{L}_p$, $streak$, clk_rst) to adjust the single-step timer interval (clk). Orange statements define state transitions and determine the action of the next iteration. The bottom three diagrams show the internal logic of the Atomic, Resumable, and Restartable states from the top flowchart.

6.3 M-Step Framework

In Section 6.2, we presented the M-Step core mechanisms which enable single-stepping on Arm Cortex-M. In this section, we introduce a framework (addressing challenge C5) which facilitates the use of M-Step and aids the *post-mortem* analysis of M-Step traces.

6.3.1 Implementation Aspects

Timer. We implemented an interrupt mechanism based on a single hardware timer, used to generate interrupts and measure interrupt latency. This timer is dynamically configured to trigger an interrupt when a programmed amount of cycles has passed, which varies to account for different instruction types and stack frame sizes in the secure world. After the interrupt is raised, the timer continues counting upwards in free-running mode. The timer is started at the end of each ISR just before re-entering the secure world, and stopped at the beginning of the next ISR immediately after exiting the secure world. Thus, the timer value observed at the start of the ISR corresponds to the interrupt latency, which yields crucial side-channel information about the interrupted instruction.

To eliminate compiler-induced variability, the ISR logic responsible for starting and stopping the timer is hand-written in assembly. The remaining non-critical M-Step ISR logic is implemented in C for maintainability and ease of development. To further reduce noise, we configure all M-Step memory regions as non-cacheable using the non-secure MPU. Additionally, we avoid floating-point operations within the ISR

Table 8: List of current M-Step plugins. Available in production (P) or only in debug mode (D).

	Plugin	Description	P	D
SC	**Mstp-Nemesis**	Reveals interrupt-latencies, Nemesis [37].	✓	✓
	Mstp-Cache	Reveals cache activity, Prime+Probe [101, 104].	✓	✓
	Mstp-BUSted	Reveals memory accesses, BUSted [8].	✓	✓
Arch	**Mstp-Zoom**	Amplifies interrupt-latency leakage Section 6.3.2.	✓	✓
Framework	**Mstp-Production**	Single-step for production code.	✓	–
	Mstp-Debug	Single-step with debug information.	–	✓
	Mstp-Metrics	Single-step performance metrics.	–	✓
	Mstp-Emulator	MCU emulator with side-channel information.	–	✓
	Mstp-Visualizer	Interactive interface to visualize M-Step traces.	✓	✓
	Mstp-opDecoder	Runtime library to decode Armv8-M opcodes.	–	✓
	Mstp-Test	Evaluation and regression testing framework.	–	✓

to prevent triggering lazy stacking, which may otherwise increase automatic stack save latency.

Fine-Tuning. Due to the heterogeneous nature of MCUs and the possibility of operating under different conditions (e.g., four distinct stack frames, see Section 2.2), M-Step must adapt its interrupt mechanism to the specific stack frame in use (handled automatically) and be fine-tuned for each target MCU (performed manually). The manual tuning is required only once per MCU and is used to establish the base interrupt threshold, which depends on the context-switching latency $\mathcal{L}_0$ (cf. Section 6.2.2.1). This parameter can be initially calibrated using M-Step's base stack frame, i.e., the NS stack with the FPU disabled, which serves as the reference for adjusting parameters to support other stack frame configurations.

To automatically adjust the timer interval, M-Step must identify the current stack frame in use. This information is encoded in the link register via a special value called `EXC_RETURN`, which is automatically loaded by the hardware upon exception entry, replacing the usual return address. The `EXC_RETURN` value contains key metadata, including the security state of the interrupted context and the type of stack frame (i.e., integer-only or including floating-point context). Using this information, M-Step can determine the active stack frame and dynamically adapt the interrupt mechanism to ensure reliable operation across all configurations.

6.3.2 Framework Plugins

The M-Step framework is built upon a set of modular plugins, listed in Table 8, which define the core M-Step mechanisms and provide optional functionalities. These plugins are grouped into three categories: *(i)* side-channel (SC) plugins, which specify the microarchitectural channels to monitor; *(ii)* architectural (Arch) plugins, which augment M-Step's base capabilities; and *(iii)* framework plugins, which offer supporting functionalities such as debugging and trace visualization. This modular design reduces the engineering effort required when adapting to new MCUs or use cases—a common challenge with monolithic

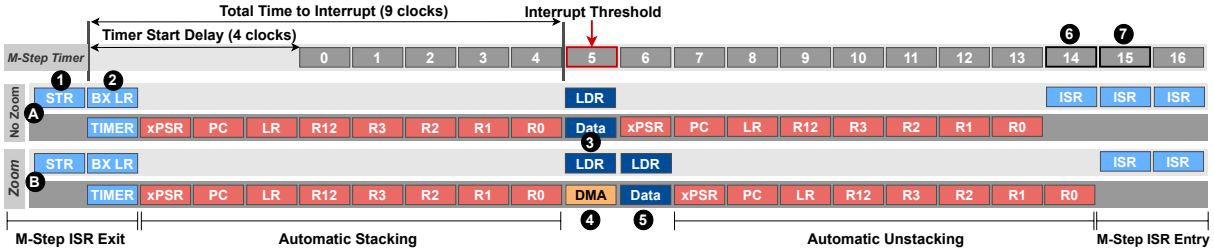


Figure 39: Reconstructed Cortex-M interrupt handling process during an M-Step single-step. The top row shows the timer values for the M-Step interrupt logic. The middle and bottom rows show two cycle-by-cycle views of the CPU pipeline (instructions and interrupt states) for the same LDR instruction, without (Ⓐ) and with (Ⓑ) BUSted gadget amplification.

frameworks—and grants users the flexibility to tailor the framework to their specific needs. M-Step allows saving arbitrary combinations of plugins/configuration into convenient "profiles" (cf. Section 6.3.4).

Memory-Access Amplification (C4). As shown in Table 7, certain MCU settings can generate loads with single-cycle interrupt latencies. This increases ambiguity, making it harder for an adversary to distinguish between instructions based on IRQ latency traces. To address this, we introduce an optional architectural plugin that amplifies interrupt-latency leakage using BUSted [8] gadgets. These gadgets exploit on-chip DMA peripherals, operating independently of the CPU, to create contention on a specific memory bank precisely during the clock cycle of interest.

We apply this technique to selectively amplify the interrupt-latency leakage of single-cycle, memory-accessing instructions, as illustrated in Fig. 39. Upon a M-Step ISR exit, the M-Step timer is armed (❶) and the exception return instruction is executed (❷), triggering automatic stacking. Once stacking completes, the next M-Step interrupt is asserted such that it arrives in the first clock cycle of the victim instruction. When amplification is enabled (Ⓑ), a BUSted gadget issues a concurrent DMA access (❹) in the same cycle, creating contention on the targeted memory bank. This contention delays the victim instruction's data fetch (❸), forcing the load to take an additional cycle (❺) and thereby increasing the observed interrupt latency (❻ vs. ❼). Importantly, only instructions that access the targeted memory region are affected, while others retain their original execution timing.

We refer to this amplification method as Mstp-Zoom. It enables fine-grained, single-instruction step tracing on MCUs with weaker interrupt-latency leakage profiles. For instance, without Mstp-Zoom, instructions like LDR on the STM32L5 (Table 7) would appear indistinguishable from generic single-cycle instructions such as ADD.

6.3.3 Framework Architecture (C5)

The architecture of the framework (cf. Fig. 40) is split into three main blocks representing different phases of the framework's operation: *(i)* configurator; *(ii)* runner; and *(iii)* analyzer. Each block is composed by a set of configurable modules which can be tailored to e.g., the target MCU or attack conditions.

Configurator. The configurator takes a set of M-Step configurations, such as fine-tuning parameters,

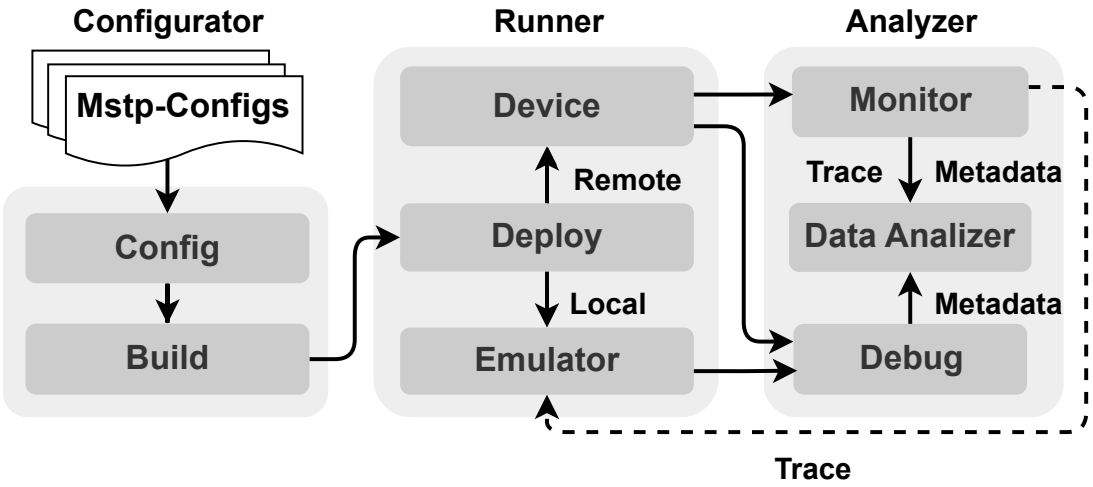


Figure 40: M-Step rapid prototyping framework architecture.

the MCU leakage profile, and M-Step profiles, which collectively define the framework's operation. The configurator consists of two modules: *Config* and *Build*. The former sets up the development environment according to the provided configurations, while the latter compiles the code for the target device. This configurability is paramount, allowing the framework to adapt to the target's software stack.

Runner. The runner takes the compiled binaries from the configurator and deploys them either to the target device via a remote connection or to an MCU emulator on the host. This module standardizes code deployment, allowing for rapid switching between vendor-specific cross-development tools to program the target device. However, relying solely on a hardware-in-the-loop setup poses a significant challenge to rapid prototyping. MCUs operate at low frequencies, and the wired connection for transferring binaries is often slow (e.g., the STM32L5 connection is clocked at 4 MHz). To mitigate this, the framework can be configured to use an emulator-in-the-loop, which runs at the host's speed with near-instantaneous deployment. Although the victim must first run on the device to gather traces, subsequent runs can leverage the emulator to analyze leakage patterns, debug M-Step mechanisms, and develop exploits using the collected traces.

Analyzer. The analyzer gathers traces and provides tools for processing, visualization, and debugging, leveraging both hardware- and emulator-in-the-loop approaches. It consists of three modules: *Monitor*, *Data Analyzer*, and *Debug*. The Monitor module establishes a remote connection to the device, logging its output to extract M-Step metadata (e.g., performance metrics) and the side-channel traces. The Data Analyzer facilitates the analysis of M-Step metadata and the *post-mortem* processing and visualization of side-channel traces. The Debug module standardizes debugging across MCU targets, offering a verbose operational context. This context is then used by the Data Analyzer to provide further insights for M-Step development or exploitation.

Framework Interface. We provide a command-line interface (CLI), which allows users to interact with the M-Step framework and orchestrate all its operations.

6.3.4 M-Step Profiles

M-Step profiles are meant to describe a M-Step specification defined by a set of M-Step plugins. There is no limit for the number of profiles. The user can create as many as they see fit by combining plugins in a different set to suit their use case. We define two reference profiles, base and debug.

Base Profile. The minimal set of plugins that enable single-stepping in the most restrictive scenario, i.e., production mode without any access to debug primitives. This profile only uses Mstp-Production and Mstp-Nemesis. The former enables the base M-Step features defined in Table 8, while the latter allows the extraction of interrupt-latency traces [37].

Debug Profile. Full-fledged M-Step specification which enables single-stepping in debug mode with all the M-Step features available. This profile uses Mstp-Debug to define the core M-Step features and includes all plugins described in Table 8. The plugins are interoperable: for instance, the side-channel plugins will generate microarchitectural information which can be accessed on the visualization tool and the emulator. Among others, this profile provides advanced debugging primitives (e.g., Mstp-Emulator), as well as access to multiple microarchitectural channels (Mstp-Nemesis, Mstp-Cache, and Mstp-BUSted), all of which can be visualized in a single compact interface (Mstp-Visualizer); see Fig. 41.

6.3.5 Rapid Prototyping & Trace Analysis

Trace Visualization. We demonstrate rapid prototyping and trace analysis with a simple proof-of-concept: tracing the standard `printf` function from `stdio.h` that prints "M-Step". The Mstp-Debug plugin extends Mstp-Production by adding debug primitives, including access to the PC of each single-stepped instruction. As a result, traces collected with Mstp-Debug include both side-channel data and the corresponding PC for each sample. The analyzer uses this information to extract symbol lists and decode instructions from the *victim.elf* file, correlating leakage with the exact instructions producing it. The Mstp-Emulator enables live debugging by replaying victim code execution on the host using the same traces. After processing, the analyzer generates a standard value change dump (VCD) file, which can be visualized with GTKWave or any other VCD-compatible tool, as shown in Fig. 41.

Debugging and Evaluation. Mstp-Debug extends Mstp-Production by adding debug primitives, such as access to the PC of the current instruction during interrupts. With the PC, M-Step can decode the corresponding opcode (via the Mstp-opDecoder plugin) and identify the exact instruction being interrupted. Mstp-Metrics uses this information to provide accurate statistics on the number of zero-, partial-, single-, and multi-steps produced by M-Step. This enables effective debugging and evaluation of the M-Step mechanism (Section 6.2).

Target Hardware Emulation. The Mstp-Emulator plugin uses Renode [234] to emulate the full MCU (CPU and peripherals) for fast, host-based side-channel analysis. Since side-channel leakage cannot be emulated, we first collect traces on real hardware using Mstp-Debug, which records the PC for each

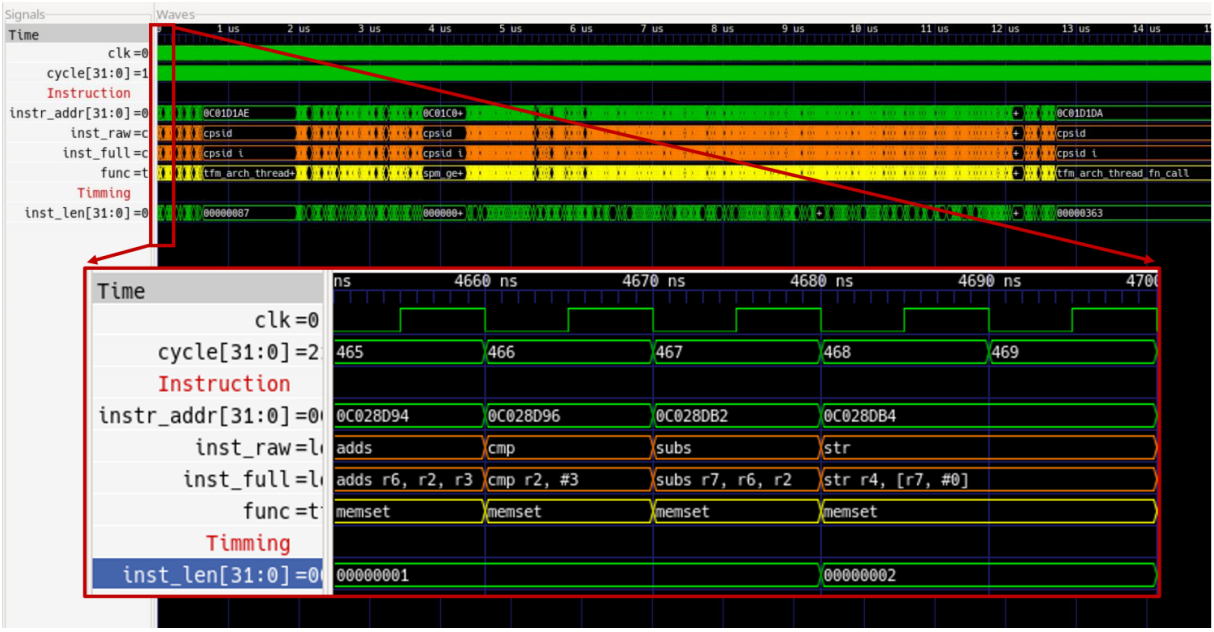


Figure 41: Example of the Mstp-Visualizer depicting the PoC trace of the standard `printf` function from `stdio.h`.

sample. These traces are then imported into the emulator, enabling correlation between the emulated code execution and the recorded side-channel data by matching the current emulated PC to the corresponding trace sample.

6.4 Evaluation

This section first evaluates M-Step's single-stepping accuracy through controlled microbenchmark experiments, then examines leakage from various microarchitectural components using covert channel setups, and finally demonstrate a real-world PoC attack on TrustZone-M.

Attack Setup. For all experiments and attacks in this section, we follow the standard TrustZone threat model: an attacker in the non-secure world invokes a target secure service in the S world. The S world runs the latest version of Arm TF-M (v2.2.0), the reference uTEE for Cortex-M.

6.4.1 Single-Step Reliability Microbenchmarks

To assess M-Step's accuracy in achieving reliable single-stepping while avoiding unintended multi-steps, we follow the methodology of prior work [62, 65], single-stepping a secure application that executes a tight benchmark stream of 2,000 consecutive instructions. To obtain ground truth, we run the secure application in debug mode and record the secure world's PC after each interrupt. Based on the PC we classify the interrupt as single-step if the PC advances by 4 bytes (32-bit instructions) or 2 bytes (16-bit instructions); multi-step if larger; and zero-/partial-step if it does not advance. Table 9 presents the

Table 9: M-Step's single-stepping reliability metrics across various instruction streams, averaged over 10 runs. *Error Rate* reports the number of multi-steps per million interrupts.

		Synthetic Code				**Generic Code**		
	M-Step Metrics	*atomic*	*resumable*	*restartable*	*Mix*	*TF-M*	*printf*	*RSA*
Step Type	Zero	0	29,337	3,719	98,996	6,700	223	7.35M
	Partial	0	2,334	0	0	548	104	601K
	Single	2,000	1,996	3	2,000	965	1,305	1.39M
	Multi	**0**	**0**	**146**	**0**	**0**	**0**	**7**
Statistics	Folding	0	0	0	0	46	3	98,011
	Total Steps	2,000	4,330	3	2,000	1,513	1,409	1.99M
	Interrupts	2,000	33,667	3,868	100,996	8,213	1,633	8.74M
	Error Rate	**0**	**0**	**38K**	**0**	**0**	**0**	**0.8**

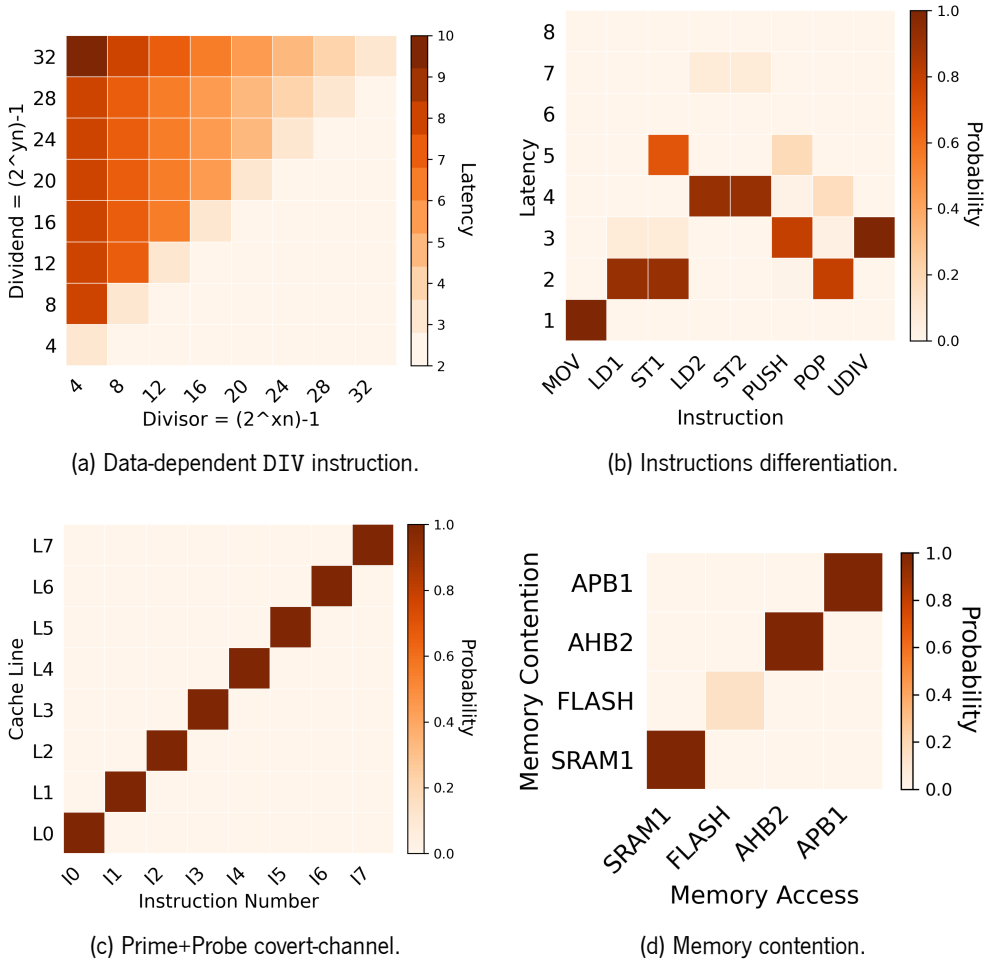
experimental results, including the distribution of step types and the occurrence of instruction folding. Instruction folding is a documented ISA feature in Armv8-M [4], where two instructions are merged into a single micro-operation, causing the PC to advance by more than one instruction. We classify such cases as single-steps, as they result from ISA-level behavior rather than stepping inaccuracies. This phenomenon is analogous to macro-op fusion in the x86 ISA, which has also been observed in prior single-stepping frameworks like SGX-Step [68].

We begin by evaluating M-Step using three synthetic benchmarks, each consisting of 2,000 instructions of atomic, resumable, or restartable types. Atomic and resumable sequences produced no multi-steps, while restartable yielded some, as our algorithm cannot distinguish consecutive restartable boundaries. As long consecutive restartable streams are rare, we treat this as a minor limitation of M-Step (cf. Section 6.5). A fourth benchmark (Mix), interleaving restartable and atomic instructions, indeed shows that restartable instructions can be perfectly single-stepped when combined with other types.

Next, we evaluate M-Step on generic, compiler-generated instruction streams: the TF-M secure-world entry code, the `printf` standard library function, and Mbed TLS's RSA implementation. Single-stepping through TF-M and `printf` code yielded no multi-steps, while a full RSA decryption produced only 7 multi-steps out of 8.74 million interrupts. Across 10 runs, all experiments yielded exactly the same traces. These results confirm that M-Step achieves precise and reliable single-stepping by producing either zero- or single-steps with fully deterministic behavior on real-world code.

6.4.2 Covert-Channel Leakage Analysis

We evaluate M-Step's leakage capabilities through controlled covert-channel experiments targeting interrupt latency, cache activity, and bus contention via their respective plugins.



(a) Data-dependent DIV instruction.

(b) Instructions differentiation.

(c) Prime+Probe covert-channel.

(d) Memory contention.

Figure 42: M-Step leakage analysis experiments. Each subfigure shows a heatmap of interrupt latency or memory access probability for a specific microarchitectural leakage source, measured while single-stepping synthetic benchmarks on TrustZone-M.

Data-Dependent Divisions. Integer division operations are known to exhibit non-constant-time, operand-dependent timing behavior on both Intel x86 [37, 62] and Arm Cortex-M platforms [196]. We use the Mstp-Nemesis plugin to measure interrupt latency for integer division instructions, sweeping over operand pairs where the dividend ranges from `0x0000000F` to `0xFFFFFFFF` in steps of `0xF`. The results, shown in Fig. 42a, reveal clear operand-dependent timing leakage: the divisor (x-axis) has a strong impact on latency, while the dividend (y-axis) shows a slightly weaker effect. Notably, when the divisor exceeds the dividend (below the diagonal), the leakage reflects only their relative magnitude. This non-constant-time behavior was recently exploited by the KyberSlash attack [196], which targets compiler-inserted divisions on Arm Cortex-M4 by analyzing full execution time. In contrast, M-Step exposes this leakage at much higher temporal resolution—revealing operand-dependent timing behavior at the level of individual `DIV` instructions.

Differentiating Instructions. We further used the Mstp-Nemesis plugin to measure interrupt latency

for eight representative instructions spanning all three categories, as shown in Fig. 42b. The set includes five atomic instructions (`MOV`, `LD1`, `ST1`, `LD2`, `ST2`), two resumable instructions (`PUSH`, `POP`), and one restartable instruction (`UDIV` with fixed operands). The y-axis in the figure shows the range of observed latencies, with color gradients representing the frequency of each value. The results reveal not only clear timing differences across instruction types but also variations within the same type. For example, memory accesses to SRAM1 (`LD1/ST1`) exhibit different latencies compared to accesses to peripheral regions (`LD2/ST2`). All measurements in Fig. 42b were collected with the Mstp-Zoom plugin enabled.

Cache Activity. Beyond interrupt latency, M-Step can monitor fine-grained cache activity using the Mstp-Cache plugin, revealing microarchitectural details that are inaccessible to a Nemesis-only attacker. While Mstp-Nemesis can infer cache misses indirectly through increased instruction latency, the Mstp-Cache plugin additionally provides precise information about which cache lines are accessed. We implemented an ICache-based[2] covert channel in which a Trojan transmits an 8-bit secret in a single execution. This is achieved by executing 8 instructions, each deliberately mapped to a distinct cache line. M-Step accurately detects every cache access with minimal noise. Fig. 42c illustrates the accessed cache lines (y-axis) corresponding to each single-stepped Trojan instruction (x-axis).

Memory Contention. M-Step can also extract information from BUSted-style [8] channels by delaying memory accesses via targeted contention in specific memory partitions (cf. Section 6.3.2). To evaluate this capability, we benchmark load and store operations across different memory regions, with the victim performing 1000 memory accesses to each. We then single-step through the target application four times, each time focusing on a different memory region. As shown in Fig. 42d, M-Step accurately tracks memory accesses in all regions except flash memory, where caching behavior obscures most events. Nevertheless, this limitation itself reveals useful information, offering an indirect signal of cache activity.

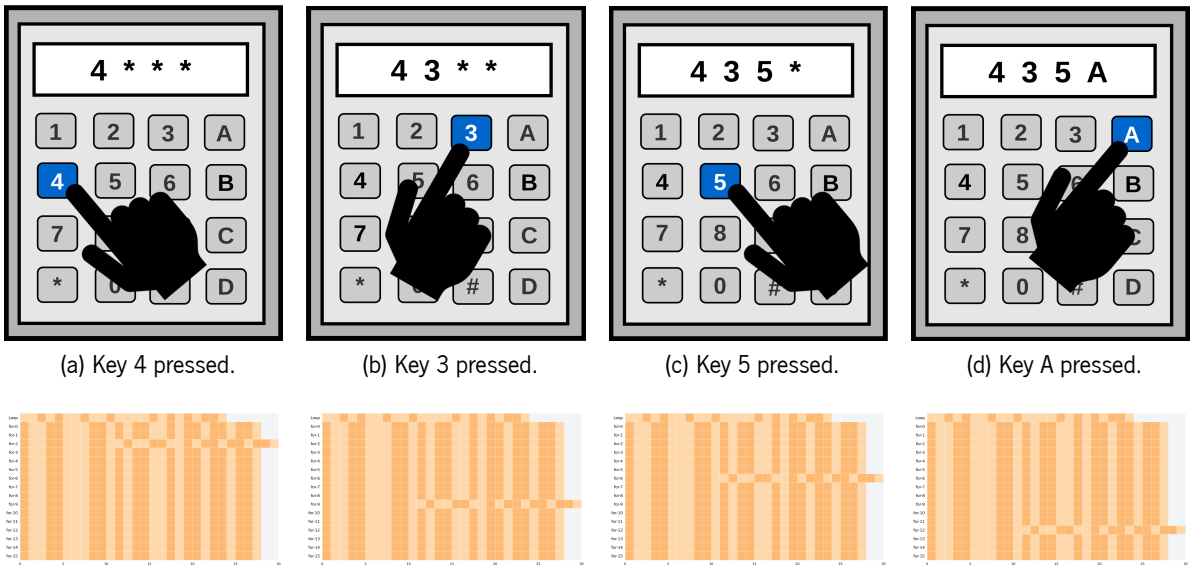
6.4.3 Off-the-Shelf Smart Lock Application

To demonstrate the practical impact of M-Step, we conduct a case study on an off-the-shelf smart-lock application interfacing with a trusted keypad, the same target previously studied by of Nemesis [37] and BUSted [8]. While this application is known to be vulnerable to interrupt-latency attacks [37], such Nemesis-style attacks were long considered ineffective on Arm-based MCUs. By revisiting this victim[3], we (i) show that interrupt-latency leakage can indeed be exploited on Armv8-M-based MCUs and (ii) illustrate M-Step's ability to trace and analyze real-world secure-critical services at instruction-level granularity. For background on the application and our setup, we refer the reader to Section 5.3.1, where we target the same code under the same threat model.

Recovering the 4-digit Secret PIN. The target code, shown in Fig. 33 (Section 4.3), continuously scans the keypad by looping over all keys and checking whether there was a keypress. To avoid end-to-end timing

[2]We target the ICache as our evaluation platform (STM32L5) lacks a data cache (DCache). The Mstp-Cache plugin currently only has support for ICaches, but it can be easily extended to support DCaches on other MCUs.

[3]In Chapter 7, we will demonstrate end-to-end attacks on more complex victims.



(a) Key 4 pressed. (b) Key 3 pressed. (c) Key 5 pressed. (d) Key A pressed.

(e) Trace Key 4 (loop index 2). (f) Trace Key 3 (loop index 9). (g) Trace Key 5 (loop index 6). (h) Trace Key A (loop index 12).

Figure 43: Top: the key pressed by the user. Bottom: the corresponding interrupt-latency fingerprint for each key press. Although shown as four panels, each fingerprint is part of a single continuous trace captured during one execution; we extracted only the segments for each key press and discarded the remaining samples. Each fingerprint is rendered as a 17×30 matrix: rows correspond to the keypad scan patterns (1×*loop*, 15×*for-no-pressed*, 1×*for-pressed*) and columns encode the measured interrupt latency per executed path. Light orange denotes 1-cycle instructions and dark orange denotes 2-cycle instructions. The index of the loop does not map directly to the key. Each key is encoded with a different index.

leakage, the code always iterates over all keys, even if a key press is detected early on. Nevertheless, each iteration performs a conditional check whose microarchitectural behavior differs depending on whether the currently tested key is pressed. Although the loop is balanced and executes dummy operations when no key is pressed, M-Step still exposes instruction-level control-flow differences between the two paths, similar to the toy example in Section 4.3.3.

By single-stepping the entire keypad TA with M-Step, we obtain deterministic traces that reveal which key is pressed in each loop iteration (if any). Fig. 43 shows the interrupt-latency traces for a single run of the target keypad code when the user enters the 4-digit PIN *435A*. Each keypress produces a distinct and repeatable fingerprint that uniquely identifies the pressed key. Concretely, we observe three recurring latency patterns: *(i)* a pattern, we denote as *loop*, marking the beginning of a new scan round (top row in each trace); *(ii)* a pattern, we denote as *for-pressed*, corresponding to the single iteration in which the currently tested key is pressed (the odd-one-out row in each trace); and *(iii)* a pattern, we denote as *for-no-pressed*, corresponding to the dummy operations performed when no key is pressed. Using these patterns, the attacker segments the trace into the 16 iterations (one per keypad key) and locates the unique iteration that corresponds to the pressed key, i.e., where the *for-pressed* pattern is observed. The index of this iteration directly reveals the pressed key. Repeating this process for all four keypresses yields

the full 4-digit PIN.

We simulate user key presses and repeat the experiment for multiple randomly generated PINs. In all cases, we recover the complete PIN from a single trace, a crucial property in this setting, since the attacker typically has only one opportunity to capture the victim's PIN. Overall, this PoC demonstrates the practical impact of M-Step-enabled interrupt-latency attacks on real-world targets and, by revisiting the same victim as Nemesis [37], confirms that interrupt logic-based microarchitectural leakage on Arm Cortex-M can be observed and exploited in practice.

6.5 M-Step Limitations

High Zero-Step Count. Our M-Step approach adapts on demand to different instruction streams and enables blind single-stepping through generic code. However, this comes at the cost of a high number of zero-steps. This increases the time required to profile applications and leak secrets, since many generated interrupts do not yield useful interrupt-latency information, making traces take longer to complete. Nevertheless, one could trade generality for speed. In side-channel attacks, it is common to assume that the attacker has access to the victim code (e.g., open-source cryptographic libraries). For compute-intensive applications with open-source codebases, M-Step could be configured to leverage knowledge of the victim code instead of relying solely on interrupt latencies to identify the instruction type under execution and configure the required timer interval. With this relaxed assumption, the number of zero-steps could be greatly reduced—potentially down to zero[4]. Such an approach, however, would be inherently fragile and highly target-specific, requiring manual effort and careful handling of (secret-dependent) branches to correctly configure the timer interval for the next victim instruction.

Consecutive Restartable Instructions. M-Step currently cannot reliably single-step through long sequences of consecutive restartable instructions. While our results show that such instruction patterns are rare and have a negligible impact on the evaluated codebases, we acknowledge that some applications may make more extensive use of restartable instructions, potentially increasing the occurrence of restartable-induced multi-steps in the scenario where the attacker has no knowledge of the victim code. It is possible that future work could address this limitation by developing more sophisticated techniques, which could be easily integrated into the M-Step framework as additional plugins.

Portability. The current M-Step functionality depends on the leakage profile of the target MCUs to detect M-Step events and instruction types, requiring a one-time fine-tuning for each new MCU—a standard practice for single-stepping frameworks. For example, SGX-Step [65] must also be manually configured for each new Intel CPU to enable effective single-stepping. While different MCUs (or even different runtime configurations on the same MCU) can influence instruction timings—such as by using faster or slower memories, as demonstrated in our STM32L5 experiments in Table 7—the microarchitectural behavior of

[4] Unlike on high-end platforms where zero-steps stem from noisy timers that trigger interrupts too early, M-Step's zero-steps are a byproduct of blindly single-stepping through resumable or restartable instructions streams (cf. C1).

the three instruction types is architecturally defined by the underlying Arm architecture (Armv6-M, Armv7-M, or Armv8-M). Similarly, for SGX-Step [65], the microarchitectural behavior of Intel CPUs' interrupt logic remains consistent across CPUs, even though specific instruction timings might change. In this chapter, we evaluated M-Step on a single target platform. Future work could extend the M-Step framework to additional Cortex-M-class MCUs and develop new plugins to capture and exploit leakage sources beyond interrupt latency, such as DCache or more sophisticated predictors.

Extensions. When the attacker does not have access to the target code to single-step, but lower trace times are still desired, a more sophisticated fine-tuning approach could be employed by extending the framework with additional plugins. We envision that an attacker with no knowledge of the victim source code could initially use M-Step in generic blind-stepping mode, and then, by analyzing the collected traces, gradually infer the types of instructions executed by the victim. This process would resemble the profiling demonstrated in Section 6.4.3, where the target is run multiple times to create template matrices. At runtime, this information could then be used by M-Step to infer the control-flow path and provide coarse-grained predictions about the expected instruction type at each step. While not perfect, this approach would reduce the number of unnecessary zero-steps while still addressing challenge C3 . Alternatively, a hybrid approach could be adopted: blindly single-stepping through unknown code (e.g., a proprietary uTEE kernel), while applying more informed single-stepping to third-party applications with known code. To aid developers in avoiding leakage through side channels, our framework could be extended with compiler-based mitigation or binary analysis tools by systematically profiling the microarchitectural leakage of all instructions [193].

6.6 Summary

In this chapter, we introduced M-Step, the first practical framework for precise, interrupt-driven side-channel analysis on TrustZone-M, and challenged the prevailing assumption that Arm Cortex-M devices are resilient to interrupt-based timing leakage. By overcoming unique Cortex-M microarchitectural challenges, M-Step enables instruction-level single-stepping unlocking a new powerful threat vector against TrustZone-M.

A particularly noteworthy observation is that we once again encounter the same behavior observed in Chapter 5, i.e., an exceptionally deterministic trace-acquisition process, where repeated measurements of the same code yield identical traces. In our experiments, this determinism persists even for cryptographic victims producing traces with around nine million samples. Across repeated runs (with the same code and dataset), every single sample out of those nine million matches perfectly, with not even a single sample differing by a single clock. These results further corroborate our earlier conclusion that the simplicity of MCUs can often benefit the attacker, rather than constituting a limitation that precludes practical side-channel attacks.

Another noteworthy observation is that by combining multiple microarchitectural components within

a single setup (e.g., interrupt logic and bus interconnect) we were able to amplify the observable side-channel signal and induce microarchitectural leakage that would otherwise remain undetectable when manipulating only a single microarchitectural component (as shown in Fig. 39). This, coupled with M-Step's capabilities to sample multiple microarchitectural components (as shown in Section 6.4.2), allows an attacker to have multiple perspectives on the MCUs's microarchitectural context, which might enable the exploitation of code structures that would otherwise appear secure when considering only a single source of leakage.

We will not explore this possibility further and leave it for future work; however, based on the evidence shown in this chapter (mainly in Section 6.4.2), we suspect that combining M-Step trace capabilities (Mstp-Busted, Mstp-Cache, and Mstp-Nemesis) within a single trace analysis could yield new insights into the MCU's microarchitectural context that would not be available from a single source of microarchitectural leakage. For instance, the Mstp-Busted plugin can be used to detect whether the CPU performed a memory access in a specific clock cycle. By combining it with Mstp-Cache, that access can be pinpointed to a cache set, thus narrowing down the accessed memory space. In addition, by incorporating Mstp-Nemesis microarchitectural information, it may be possible to identify the set of instructions that could have performed that access, for example a `LDR` vs. `PUSH` (recall Fig. 42b and the different instruction leakage profiles). Ultimately, by fusing information from the three plugins, one could potentially determine the precise cycle where the memory access occurred (Mstp-Busted), which cache line it mapped to (Mstp-Cache), and a set of instructions that may have caused it (Mstp-Nemesis). This would allow an attacker to profile the victim execution in greater detail and potentially enable more stealthy side-channel attacks.

7

Breaking Cryptographic Applications on MCUs

In Chapters 4 to 6, we demonstrate that microarchitectural leakage is pervasive across MCUs, enabling attackers to leverage this leakage to bypass Arm TrustZone-M. However, our prior proof-of-concept attacks targeted simple applications [8, 37]; the question of whether these techniques scale to real-world cryptographic applications remained open. In this chapter, we answer this question by targeting cryptographic implementations, the gold standard for evaluating a system's resistance to side-channel leakage. Cryptographic code is particularly compelling as an attack surface: *(i)* its implementations are openly available for detailed analysis and *(ii)* the assets they protect justify the attacker effort to mount such complex attacks. Moreover, by mounting end-to-end side-channel attacks against cryptographic software on state-of-the-art TrustZone-M-based MCUs, we expose the concrete real-world impact of microarchitectural attacks on these devices.

In this chapter, we present the first single-trace side-channel attack that extracts real-world cryptographic keys from TrustZone-M. As a concrete case study, we target Arm Mbed TLS v3.6.3.1[1], the default cryptographic library used by TF-M. In Section 7.1, we profile the Mbed TLS implementation of the BEEA algorithm using M-Step (Chapter 6) and construct template matrices that reveal its control flow with instruction-level precision. Next, in Section 7.2, we expose 24 distinct vulnerable function calls that enable full leakage of RSA private keys. We then mount two end-to-end attacks against Mbed TLS's RSA key generation and key importation routines, recovering full RSA private keys in a single trace. Finally, in Section 7.3, we discuss countermeasures that provide actionable short-term guidance, alongside a principled mitigation that addresses the underlying root cause: the non-secure world's ability to arbitrarily interrupt the secure world.

Related Contributions, Artifacts and Dissemination.

The findings in this chapter have been published in the **P2** scientific article. As part of our responsible disclosure process, Arm published a security advisory (**S3**) following a fix that replaces the vulnerable BEEA algorithm with a constant-time alternative in the affected functions. This vulnerability was assigned

[1] The most up-to-date version at the time of our research.

CVE-2025-54764 (**S4**). All code, datasets, and scripts used in the reported attacks are available in **A3**. We have also submitted a talk to BlackHat USA 2026 that unveils the vulnerability and M-Step framework capabilities to the security community (**T2**).

Summary of Related Contributions:

S3 - Security advisory: Arm Mbed TLS. *"Side channel in RSA key generation and operations (SS-Bleed, M-Step)"*, 2025. *Link*.

S4 - Security advisory: CVE Program. *"CVE-2025-54764"*, In: National Vulnerability Database, 2025. *CVE-2025-54764*.

P2 - Publication: C. Rodrigues, M. Bognár, S. Pinto, and J. Van Bulck *"M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M"*, (To appear) In: 35th USENIX Security Symposium, 2026. *Link*.

T2 - Talk: C. Rodrigues, M. Bognár, S. Pinto, and J. Van Bulck *"Tiny Chips, Big Leaks: Breaking TrustZone-M with Single-Stepping Attacks"*. (To appear) In: BlackHat USA, 2026. *Link*.

A3 - Artifacts: https://github.com/M-Step-Framework.

7.1 Unleashing M-Step on Mbed TLS

Single-stepping frameworks enable high-resolution, single-trace attacks that extract fine-grained microarchitectural information with minimal noise. M-Step brings such attack capabilities to MCUs, exposing secret-dependent control flow, down to a single instruction. As a real-world end-to-end attack, we leverage M-Step to target Mbed TLS's BEEA. The BEEA algorithm is known to be vulnerable to control-flow side-channel attacks and has been exploited in various libraries, including OpenSSL [68, 235–238], WolfSSL [68], Libgcrypt [68], and Mbed TLS [238–240]. These exploits have targeted high-end platforms, such as Intel SGX, using different leakage primitives such as FLUSH+RELOAD [235, 237, 238], page faults [236, 239], and interrupt counting [68]. Despite these known issues, we found that the latest version of Mbed TLS (v3.6.3.1)[2] still has multiple vulnerabilities rooted on the non-constant-time nature of the BEEA algorithm. This leakage is non-trivial to exploit without M-Step, as it requires distinguishing a sequence of closely spaced, secret-dependent branches, often several in a row, at instruction-level granularity. Traditional side-channel techniques (e.g., end-to-end timing or cache attacks) cannot reliably resolve such fine-grained control flow, making these vulnerabilities only exploitable with a single-stepping framework like M-Step.

Parsing M-Step Traces. M-Step traces span the entire victim execution, providing one interrupt latency sample per victim instruction. To locate the code of interest within a trace, we exploit the deterministic

[2]Mbed TLS guarantees protection against "publicly documented attack techniques", placing local timing side channels explicitly in scope [241]. Upon responsible disclosure, Arm acknowledged our findings and issued a patch and a CVE (cf. Section 7.4), confirming that our attack falls within the intended threat model and highlighting its practical relevance.

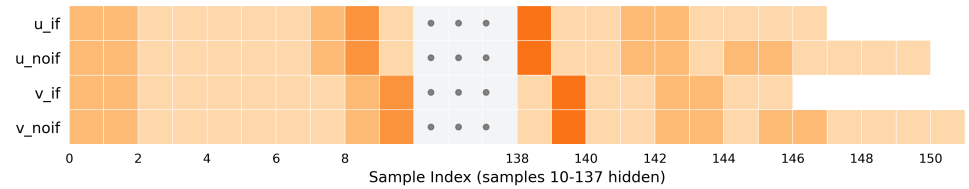
Algorithm 3: Mbed TLS's BEEA algorithm to compute $X = A^{-1} \pmod{B}$. Light blue modular inversion-specific steps and light orange greatest common divisor (GCD)-specific steps.

```
1  procedure Mbedtls_BEEA(X, A, B)
2      TU ← A, TV ← B;
3      U1 ← 1, U2 ← 0, V1 ← 0, V2 ← 1;
4      while TU ≠ 0 do
5          while TU even do
6              TU ← TU/2;
7              if U1 or U2 odd then
8                  U1 ← U1 + TB; U2 ← U2 − TA;
9              U1 ← U1/2; U2 ← U2/2;
10         while TV even do
11             TV ← TV/2;
12             if V1 or V2 odd then
13                 V1 ← V1 + TB; V2 ← V2 − TA;
14             V1 ← V1/2; V2 ← V2/2;
15         if TU ≥ TV then
16             TU ← TU − TV;
17             U1 ← U1 − V1; U2 ← U2 − V2;
18             TU ← TU/2;
19         else
20             TV ← TV − TU;
21             V1 ← V1 − U1; V2 ← V2 − U2;
22             TV ← TV/2;
```

behavior of MCUs, where a specific code path consistently yields the same trace pattern. M-Step produces identical, clock-cycle-granular traces for a given instruction stream, requiring no statistical analysis over multiple runs (cf. Fig. 44). This property enables an attacker to profile the target code in advance and construct a template, i.e., a set of side-channel patterns that deterministically and uniquely identify secret-dependent control flows. During exploitation, the attacker correlates the live trace of a single run with the precomputed template to determine the exact execution path taken by the victim.

7.1.1 Recovering Mbed TLS BEEA Control-Flow

Mbed TLS uses the BEEA algorithm, shown in Algorithm 3, to perform both modular inversion and GCD computation. The core structure of the algorithm is shared between both operations; however, for modular inversion, the algorithm tracks the Bézout coefficients ($V1$, $V2$, $U1$, and $U2$), while for GCD computation, these are omitted. The differences are highlighted in light blue for modular inversion-specific steps and in light orange for GCD-specific steps. The BEEA algorithm computes the modular inverse $X = U^{-1}$



(a) Shift template matrix. Full patterns are shown in [11].



(b) Subtraction template matrix.

Figure 44: Template matrices for detecting U and V shifts and subtraction type. Color gradient indicates interrupt latency for the instruction at the given index (darker: more cycles).

$(\mathrm{mod}\ V)$ by iteratively reducing two variables, U and V, using conditional subtractions and right-shift operations. The algorithm initializes U and V with the inputs A and B, respectively, and then repeatedly performs right-shifts until they become odd. When both U and V are odd, the algorithm compares them and subtracts the smaller from the larger ($U \leftarrow U-V$ or $V \leftarrow V-U$). This process repeats until $U = 0$. Since the algorithm's final state is known ($U = 0$ and $V = 1$)[3], an attacker who can observe the sequence of subtractions and right shifts can recover the initial secret inputs by applying the reverse operations (left shifts and additions). Note that the algorithm's control flow depends on the bit patterns of its inputs. An attacker observing this control flow with M-Step can reconstruct the exact sequence of subtractions and shifts, and consequently recover the initial secret inputs A and B. In essence, the attacker must learn the number of U- and V-shifts, as well as the outcomes of the conditional subtraction to recover the secrets.

Recovering the Number of Shifts. Mbed TLS's BEEA implements right-shifts using two while loops (one for U and one for V), each containing an if-statement, resulting in four unique execution patterns that we denote as *u_if*, *u_noif*, *v_if*, and *v_noif*. To construct the shifts' template matrix shown in Fig. 44a, we profiled the target code and selected the shortest pattern that unequivocally identifies each execution path. Although these patterns are very similar, M-Step can distinguish subtle differences between them, e.g., *u_if* and *v_if* differ by only a single instruction.

Recovering the Conditional Subtraction. The algorithm performs subtractions based on the relative sizes of U and V, determined by the outcome of the non-constant-time bignum comparison $U \geq V$. We

[3]For RSA, V holds the GCD of the inputs. Since they are coprime, the final value is $V = 1$.

identify eight distinct execution paths: *(i)* five corresponding to $U \geq V$ and *(ii)* three to $U < V$. Similar to the shift templates, we profile the target code and select patterns that uniquely identify each path, as illustrated in Fig. 44b. Compared to the shift patterns, these are shorter and exhibit more pronounced leakage characteristics.

7.2 End-to-end Attacks on Mbed TLS.

We identified two distinct issues in Mbed TLS (v3.6.3.1) that can be exploited through vulnerabilities in its implementation of the BEEA algorithm. The template matrices derived from profiling Mbed TLS's BEEA (cf. Fig. 44) enable exploitation of 24 distinct Mbed TLS function calls. Overall, we uncover subtle non-constant-time behavior in the latest release of Mbed TLS (v3.6.3.1) that can be leveraged in end-to-end attacks against real-world applications.

7.2.1 Mbed TLS Vulnerabilities.

Issue #1: Modular Inversion with Unmasked Inputs. The Mbed TLS modular inversion uses the non-constant-time BEEA algorithm described above. While Mbed TLS typically masks inputs by multiplying them with a random value, we identified 13 vulnerable RSA calls where inputs remain unmasked (see V#1–V#13 in Table 10). These vulnerabilities affect multiple RSA configurations, with at least one exploitable call always present. Some calls are only vulnerable when chinese remainder theorem (CRT) parameters are enabled (e.g., V#2), whereas others are only exploitable when they are disabled (e.g., V#4).

Extended Issue #1.1: TF-M-Specific Vulnerabilities. TF-M introduces 7 additional vulnerable calls (V#14–V#20) beyond the 13 already discussed. In upstream Mbed TLS, these calls are only executed if CRT parameters are missing. However, we found that on ST boards with cryptographic accelerators, TF-M always executes these calls (V#14–V#20), regardless of whether CRT parameters are present. This unconditional execution defeats Mbed TLS's countermeasure against prior attacks [238–240], which was intended to prevent exploitation via these vulnerable calls when CRT parameters are missing.

Issue #2: Non-Constant-Time GCD. The Mbed TLS GCD function also uses a non-constant time algorithm similar to BEEA, producing 4 vulnerable calls (V#21–V#24). Unlike modular inversion, the right-shift leakage profile depends on the version of the compiler. We found that with older compilers (e.g., GCC < 10), code associated with the right-shift have secret-dependent control flow that leaks the exact number of shifts performed (cf. code path *B* in Fig. 45). Newer compilers from GCC v10 onwards (released in 2020) produce code that only leaks whether an entire limb (32 bits on Arm Cortex-M) is zero or not (cf. code path *A* in Fig. 45). While not as severe as Issue #1, this is still clearly undesirable non-constant-time behavior.

Table 10: List of vulnerable functions. The "mbedtls_"prefix, was shortened to "mb_"for brevity. Always exploitable (A). Exploitable if CRT enabled (C). Exploitable if CRT disabled (!C). Missing CRT parameters (P). Exploitable if D is missing (D).

	Vuln.	Function	Call Tree (vulnerable path)	Recover Private Key	A	C	!C	P	D	Ref.
Mod. Inv.	V#1	mb_rsa_gen_key	$mb_mpi_inv_mod(D, E, \mathbf{L})$	$D = E^{-1} \bmod \mathbf{L}$	●	●	●	●	●	code
	V#2	mb_rsa_deduce_crt	mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#3	mb_rsa_complete	mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	○	●	code
	V#4	mb_rsa_complete	mb_rsa_deduce_private_exponent → $mb_mpi_inv_mod(D, E, \mathbf{K})$	$D = E^{-1} \bmod \mathbf{K}$	○	○	●	●	●	code
	V#5	mb_pk_psa_rsa_sign_ext	mb_rsa_write_key → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#6	rsa_decrypt_wrap	mb_rsa_write_key → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#7	import_pair_into_psa	mb_rsa_write_key → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#8	pk_write_rsa_der	mb_rsa_write_key → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#9	mb_rsa_export_crt	$mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#10	mb_psa_rsa_export_key	mb_pk_write_key_der → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#11	mb_psa_rsa_export_public_key	mb_psa_rsa_export_key → mb_pk_write_key_der → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#12	mb_psa_rsa_import_key	mb_psa_rsa_export_key → mb_pk_write_key_der → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#13	mb_psa_rsa_generate_key	mb_psa_rsa_export_key → mb_pk_write_key_der → mb_rsa_export_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
Mod. Inv. TF-M	V#14	mb_rsa_complete	mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#15	mb_rsa_parse_key	mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#16	mb_psa_rsa_load_representation	mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#17	mb_psa_rsa_sign_hash	mb_psa_rsa_load_representation → mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#18	mb_psa_rsa_verify_hash	mb_psa_rsa_load_representation → mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#19	mb_psa_asymmetric_encrypt	mb_psa_rsa_load_representation → mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#20	mb_psa_asymmetric_decrypt	mb_psa_rsa_load_representation → mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → $mb_mpi_inv_mod(QP, \mathbf{Q}, \mathbf{P})$	$D = E^{-1} \bmod (\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
GCD	V#21	rsa_gen_key	$mb_mpi_gcd(D, E, \mathbf{H})$	$D = E^{-1} \bmod \mathbf{H}$	●	●	●	●	●	code
	V#22	rsa_gen_key	$mb_mpi_gcd(D, \mathbf{P}, \mathbf{Q})$	**P**, **Q** partial leak following [239]	●	●	●	●	●	code
	V#23	mb_mpi_inv_mod	$mb_mpi_gcd(D, \mathbf{P}, \mathbf{Q})$	**P**, **Q** partial leak following [239]	●	●	●	●	●	code
	V#24	mb_rsa_deduce_primes	$mb_mpi_gcd(D, \mathbf{K}, N)$	$P = \mathbf{K}/N$	●	●	●	●	●	code

```
size_t mbedtls_mpi_lsb(const mbedtls_mpi *X)
{
    size_t i;

#if defined(__has_builtin)
#if (MBEDTLS_MPI_UINT_MAX == UINT_MAX) && __has_builtin(__builtin_ctz)
    #define mbedtls_mpi_uint_ctz __builtin_ctz
#elif (MBEDTLS_MPI_UINT_MAX == ULONG_MAX) && __has_builtin(__builtin_ctzl)
    #define mbedtls_mpi_uint_ctz __builtin_ctzl
#elif (MBEDTLS_MPI_UINT_MAX == ULLONG_MAX) && __has_builtin(__builtin_ctzll)
    #define mbedtls_mpi_uint_ctz __builtin_ctzll
#endif
#endif

#if defined(mbedtls_mpi_uint_ctz)
A ==> for (i = 0; i < X->n; i++) {
        if (X->p[i] != 0) {
            return i * biL + mbedtls_mpi_uint_ctz(X->p[i]);
        }
    }
#else
    size_t count = 0;
B ==> for (i = 0; i < X->n; i++) {
        for (size_t j = 0; j < biL; j++, count++) {
            if (((X->p[i] >> j) & 1) != 0) {
                return count;
            }
        }
    }
#endif

    return 0;
}
```

Figure 45: Mbed TLS `mbedtls_mpi_gcd` vulnerable code to count the leading zeros of variable X. Code **A** is inserted in newer compiler versions and leaks the number of zero bits with 32-bit granularity. Code **B** is inserted in older versions and leaks with bit granularity the number of leading zeros.

7.2.2 Case-Study Attacks.

We now present two end-to-end PoCs attacks against Mbed TLS that recover an RSA private key in two distinct scenarios: *(i)* during RSA key generation (Case study #1) and *(ii)* when importing a pre-existing key for decryption (Case study #2). All attacks succeed in a single execution of the victim and exploit Mbed TLS's modular inversion vulnerabilities above reported in Section 7.2.2.

Attack setup. Mbed TLS operates as a TF-M secure service at the highest privilege level. Although not directly exposed, an attacker in the non-secure world can interact with Mbed TLS through TF-M's cryptographic APIs. The attacker can trigger the vulnerable paths, detailed in Table 10, either directly by requesting a cryptographic service from TF-M (Case study #1) or indirectly by invoking a TA that uses these services (Case study #2). To collect side-channel traces, the attacker configures M-Step in production mode with the Mstp-Nemesis plugin and invokes the victim through a service request. Our attacks use the default TF-M configuration, which enables the CRT and sets the public exponent to the standard value of $E = 2^{16} + 1$. We use the default TF-M RSA key size, of 2048 bits.

Case study #1: Exploiting RSA Key Generation. Our first case-study attack targets the RSA key generation process, which an attacker can trigger by requesting a secure service public key from TF-M. This scenario presents a significant challenge, as key generation occurs only once, making it a true single-trace attack. Mbed TLS's key generation function contains a vulnerable call (V#2 in Table 10), that leaks the secret prime factors P and Q. With these, the attacker can fully reconstruct the RSA private key by computing $D = E^{-1} \pmod{(\mathbf{P}-1)(\mathbf{Q}-1)}$. We repeated the attack 10 times on different RSA keys and achieved a deterministic success rate of 100%.

Case study #2: Exploiting RSA Key Importing. Our second case-study attack targets RSA key importing, a common operation in applications using persistent keys. We attack the Mbed TLS RSA decryption reference application, which imports an RSA key without CRT parameters, thereby triggering vulnerable call V#3. This vulnerability leaks the secret prime factors P and Q, enabling full recovery of the private key. The issue is especially striking with TF-M on ST boards equipped with cryptographic accelerators (cf. extended issue #1.1): although the decryption itself is performed in secure hardware, a software-based side-channel attack still allows full key extraction. We repeated the attack 10 times with different RSA keys and again achieved a deterministic 100% success rate.

Vulnerabilities in the Wild. All official Mbed TLS RSA reference applications (listed in Table 11) are affected by at least one of the vulnerabilities described above. The pervasiveness of these vulnerabilities across the official Mbed TLS RSA reference applications suggests that they likely also affect real-world, closed-source applications.

Table 11: Reference Mbed TLS applications using RSA with vulnerable code paths, as found in the official repository. Exploitable when: CRT enabled (C) or CRT disabled (!C).

Example Program	Vulnerable Function	C	!C	Ref.
rsa_decrypt.c	*mbedtls_rsa_complete*	●	○	**code**
rsa_genkey.c	*mbedtls_rsa_gen_key*	●	○	**code**
rsa_genkey.c	*mbedtls_rsa_export_crt*	○	●	**code**
rsa_sign.c	*mbedtls_rsa_complete*	●	○	**code**
key_app.c	*mbedtls_rsa_export_crt*	○	●	**code**
key_app_writer.c	*mbedtls_rsa_export_crt*	○	●	**code**
dh_server.c	*mbedtls_rsa_complete*	●	○	**code**

7.3 Countermeasures

The most effective countermeasure for timing-based leakage is to ensure all security-critical code runs in constant time. However, achieving truly constant-time code is notoriously challenging. As demonstrated in Section 6.4, even established cryptographic libraries continue to be vulnerable to side-channel attacks. Moreover, constant-time source code does not guarantee the generation of constant-time binaries, as compilers may inadvertently introduce non-constant-time code [196, 242]. Therefore, similar to existing single-stepping mitigations deployed in Intel SGX [225] and TDX [59], we argue for defense-in-depth mitigations to hinder exploitation with M-Step. In this section, we outline potential mitigation strategies that can be integrated with existing hardware and are deployable on current Cortex-M platforms.

7.3.1 Naive Approaches

Disabling Interrupts. A straightforward countermeasure is to disable NS interrupts upon S-world entry and re-enable them on exit. While this prevents interrupt-based attacks, it undermines a core MCU feature, namely deterministic, low-latency interrupts, making it unsuitable for safety-critical, real-time applications [243–245].

A more targeted approach is to disable interrupts only during security-critical code. TrustZone-M allows the S world to assign all NS interrupts a lower priority than S-world interrupts, ensuring that NS interrupts cannot preempt S-world execution. In principle, this feature could enable uninterruptable execution of sensitive operations in "handler mode" in the S world. However, only privileged code (e.g., the uTEE kernel or Mbed TLS, which is part of a TF-M privileged secure service) can use handler mode, whereas TAs cannot. Although still not guaranteeing real-time deadlines, this solution would prevent the NS world from being completely starved of interrupts while presenting an effective countermeasure against our attacks on Mbed TLS.

Mediating Peripherals through the Secure World. Another potential solution is to assign all peripherals to the Secure world and provide an interface for the NS world to access them. In this approach, all interrupts would be handled in the S world, allowing the S world uTEE to sanitize them and prevent

single-stepping attacks. Peripherals could then be accessed upon NS request, with the uTEE mediating their use. However, this may introduce unnecessary overhead. Moreover, this approach would significantly increase the S world TCB, potentially reducing the overall security of the system by expanding the attack surface for malicious third-party TAs or even benign TAs through memory safety vulnerabilities.

7.3.2 Secure Interrupt Sanitization

Inspired by prior work on real-time TEEs [243–246], we propose a principled countermeasure to protect applications in the TrustZone-M secure world from interrupt-driven attacks while preserving essential MCU features, including peripheral access and interrupt-driven operation for low-power and real-time applications. The core concept involves introducing a trusted sanitization software layer that executes on every NS interrupt and clears the microarchitectural state prior to invoking the untrusted NS handler. This sanitization process may encompass techniques such as padding [229, 246] or randomizing interrupt latency, prefetching [225] various microarchitectural states like the instruction cache, or simply limiting the number of closely successive interrupts [59].

Trapping NS Interrupts. A key challenge in implementing this sanitization mechanism is how to effectively trap NS interrupts to perform sanitization without creating any opportunities for the NS world to execute malicious code. Unlike Cortex-A, Cortex-M lacks a software monitor, and NS interrupts switch directly to the NS application, effectively bypassing the S-world firmware (Section 2.2). To address this issue, we propose the introduction of a trampoline mechanism (cf. Fig. 46): on each NS interrupt, the S firmware is invoked first by redirecting the NS vector table via the vector table register (VTOR) to S-controlled code. The S world constructs a trampoline vector table (**B**) in NS-callable memory, redirecting all NS interrupts to entry points within the S firmware. Upon entry to the S world (❶), the VTOR is updated to point to this trampoline table. For each NS interrupt, control is passed to the S-world firmware (❷), which performs the necessary sanitization (❸) before branching to the actual NS handler (❹). Once the S world exits (❺), the original VTOR is restored (**A**).

Preliminary Validation. We implemented a proof of concept of the proposed mechanism in a bare-metal setting, where the S world is managed by a vanilla S-world firmware providing only minimum functionality to boot the board. We extended its NS-callable entry points to construct the trampoline vector table (**B**), which redirects NS interrupts to S-controlled entry points. In the S world, we implemented a basic sanitization layer that introduces a small pseudo-random delay of fewer than 10 cycles for each interrupt. In the NS world, we configured a timer, similar to the M-Step timer described in Section 6.3.1, to extract IRQ latency traces. With this setup, we qualitatively observed that the relationship between IRQ latency and the interrupted instruction became substantially less pronounced, consistent with the intended effect of injecting noise into the latency traces and reducing their correlation with the interrupted instruction stream. While these initial results are encouraging, they are intended as an initial sanity check of the proposed countermeasure rather than a full implementation. We leave to future work a systematic study of

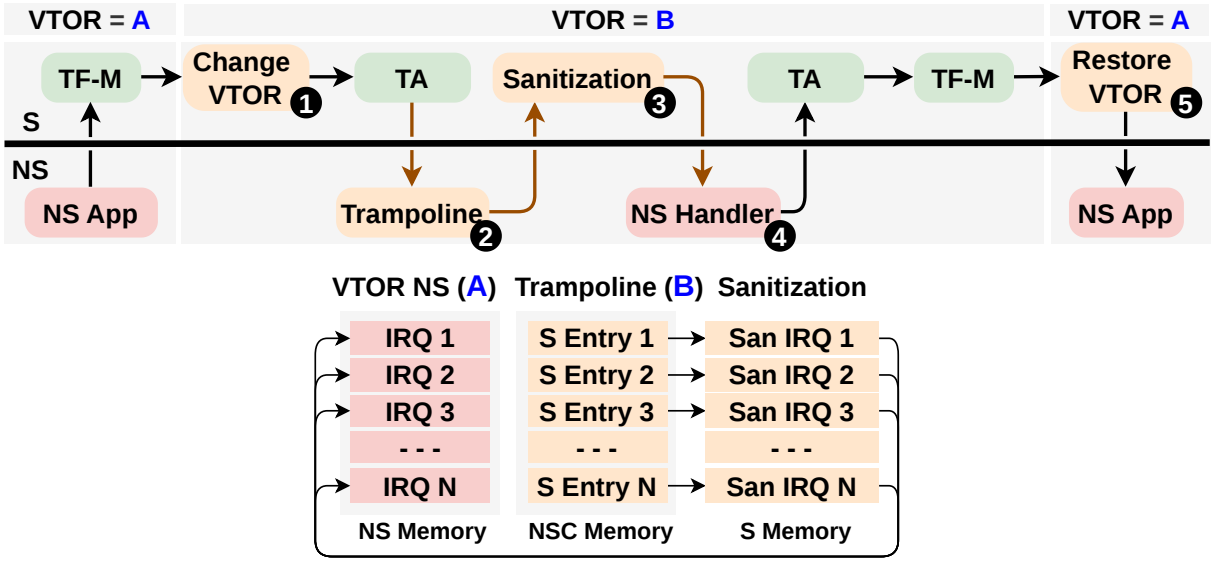


Figure 46: Illustration of the secure interrupt sanitization mechanism. Green components represent the baseline S world code, red the NS code, and orange the countermeasure-specific components.

cost-effective sanitization strategies across different microarchitectural channels (e.g., Nemesis, BUSted, M-Step, and cache channels), their impact on key MCU metrics (e.g., jitter, determinism, and power consumption), and the engineering challenges of integrating such mechanisms into TF-M.

7.4 Summary

In this chapter, we provided conclusive evidence that microarchitectural side-channel attacks on MCUs constitute a practical threat. We leveraged M-Step's enhanced temporal resolution to uncover previously unknown vulnerabilities in the latest version of Arm's widely used Mbed TLS library, the default cryptographic library of TF-M. Notably, M-Step deterministically extracts full RSA keys from a single trace of Mbed TLS's complex BEEA implementation by accurately probing multiple secret-dependent branches in close succession. Our results show that M-Step exposes subtle microarchitectural leakage across multiple Mbed TLS execution paths that differ by as little as a single instruction.

We also found that this leakage persists on platforms equipped with cryptographic accelerators, challenging the assumption that offloading operations to dedicated hardware ensures security against side-channel attacks. This finding reveals that accelerator-backed implementations can still expose exploitable leakage when secret-dependent control flow appears in code that does not perform the secret operations themselves but merely manipulates the key, e.g., the key-import routine we demonstrated.

More broadly, our results demonstrate that high-resolution, software-based side-channel analysis extends beyond high-end platforms and is feasible on microcontroller-class TEEs. This underscores the need for stronger protections, rigorous validation of constant-time behavior, and principled mitigation strategies. To address these concerns, we propose *Secure Interrupt Sanitization*, a long-term mitigation strategy for interrupt-driven side-channel attacks on MCUs. Crucially, it preserves TrustZone-M's interrupt awareness,

thereby avoiding disruptions to the deployment of real-time features on TrustZone-M-enabled devices in security-oriented applications.

7.4.1 Ethical Considerations

Responsible Disclosure. We followed responsible disclosure best practices by reporting all vulnerabilities shortly after their discovery to Arm, Trusted Firmware (TF-M), and Mbed TLS on June 28, 2025. Mbed TLS acknowledged our report and developed a fix that replaces the vulnerable BEEA algorithm with a constant-time alternative in the affected functions. This mitigation was publicly released on October 15, 2025 as part of Mbed TLS v3.6.5 (link), which is no longer vulnerable to the attacks described in this chapter. TF-M subsequently released version v2.2.2 (link) on November 27, 2025, which incorporates the patched Mbed TLS fixing the vulnerabilities in its own codebase. Arm has also assigned CVE-2025-54764 to track the vulnerability and published a security advisory (link) on the Mbed TLS website, acknowledging M-Step and their end-to-end attacks.

For the platform-specific issues (V#14–V#20 in Table 10), the TF-M PSIRT team redirected us to ST, whom we contacted on August 5, 2025. On October 30, 2025, ST responded that they do not plan to publish a dedicated security advisory for our attack, referring to prior guidance (SB0032, SB0033) that recommend their users to update the affected ST packages to the latest Mbed TLS version.

8

Conclusion and Future Work

In this thesis, we have demonstrated that software-based microarchitectural attacks are not confined to complex, high-end processors. Contrary to prior belief, we have shown that these attacks can be successfully mounted on resource-constrained computing units, namely MCUs. In this final chapter, we summarize our core findings in Section 8.1 and outline future research directions in Section 8.2.

8.1 Conclusions

While software-based microarchitectural attacks have been a well-established threat on APUs, their applicability to MCUs remained largely unknown prior to this thesis. In light of our initial research question (**RQ**), we have shown that both MCU-specific and previously exploited microarchitectural components can be leveraged to mount real-world exploits. From our three key contributions (cf. Section 1.4), we have drawn three main takeways to the field of microarchitectural attacks.

MCU-Specific Properties as a Key Enabler. First, we have demonstrated that the unique characteristics of MCUs, such as their simplistic microarchitecture, do not preclude microarchitectural attacks on these devices. Instead, the attacker can leverage MCU-specific properties, notably their highly deterministic architecture and rich set of peripherals, to gather deterministic clock-cycle-granular traces. While MCUs present unique challenges, such as the lack of concurrent execution and the absence of microarchitectural buffers to preserve state after victim execution, we have shown that the attacker can overcome these limitations. Concretely, we have shown that the attacker can craft hardware gadgets, such as a set of interconnected untrusted peripherals, to bypass the need for concurrent execution while maintaining highly predictable behavior. Moreover, the simplistic nature of these devices directly augments the attacker's capabilities, as the observed side-channel patterns are highly deterministic and exhibit an exceptionally high signal-to-noise ratio. Building on these MCU-specific exploitation techniques, we have uncovered a non-conventional side-channel attack that leverages the MCU interconnect arbitration logic, demonstrating that microarchitectural components unique to MCUs can be leveraged to mount real-world exploits. While the bus interconnect is a rudimentary microarchitectural component, its presence is pervasive across the MCU spectrum. Notably, much like cache-based side-channel attacks on APUs, this interconnect

leakage provides evidence that widespread microarchitectural attacks on MCUs are a practical reality. Based on this evidence, we argue that it is highly likely that other unique MCU-specific microarchitectural components and characteristics can be leveraged in a similar fashion to mount novel software-based microarchitectural attacks in the future.

Takeway 1: The unique characteristics of MCUs are not a limitation; rather, they serve as a key enabler for microarchitectural attacks on Arm-based MCUs.

APU's Exploitation Techniques Can Span to MCUs. Second, we have demonstrated that both microarchitectural leakage and exploitation techniques previously leveraged on APUs can translate to MCUs. Prior work on APUs has developed specialized single-stepping attack frameworks that use timer interrupts to advance the victim execution one instruction at a time, leveraging the resulting interrupt latency to gather instruction-granular timing side-channel information. While interrupt-latency attacks were assumed to be not possible on Arm-based MCUs [37], our findings debunk this assumption. Concretely, we have presented M-Step, the first single-stepping framework for Arm-based MCUs. We we have brought the fine-grained control of high-end exploitation frameworks, such as SGX-Step [58], to Arm-based MCUs, enabling an attacker to sample distinct MCU microarchitectural structures with instruction-level precision. While interrupt-latency attacks and single-stepping on MCUs posed non-trivial engineering challenges, we have leveraged MCU-specific characteristics to overcome them. First, in contrast to prior single-stepping frameworks on Intel, AMD, and Cortex-A processors, the microarchitectural behavior on Cortex-M is highly heterogeneous, yet also highly deterministic. Second, MCUs lack virtual memory and advanced CPU features, such as page faults and page-table accessed bits, which high-end single-stepping frameworks use as a core primitive to advance victim execution and detect zero-steps. Nevertheless, we have shown that the deterministic nature of MCUs can be used as a single-step oracle. This deterministic behavior allows an attacker to leverage the interrupt latency to steer the single-stepping mechanism, accurately detecting victim progress and zero-steps without relying on virtual memory-related mechanisms. Based on this evidence, we argue that other exploitation techniques and microarchitectural components previously leveraged on APUs can similarly be adapted to mount novel software-based microarchitectural attacks on MCUs in the future.

Takeaway 2: Microarchitectural leakage and exploitation techniques previously leveraged on APUs can also be effective on Arm-based MCUs.

Different Devices, Same Threats. Third, we have demonstrated that software-based side-channel attacks against real-world production software are not confined to complex, APU-based high-end platforms. We have shown that these attacks can also subvert the security guarantees of MCU-class TEEs and can be leveraged to exploit heavily scrutinized cryptographic code. Concretely, we have leveraged the attack primitives introduced in this thesis, i.e., M-Step and hardware gadgets, to mount the first end-to-end attack on a cryptographic library protected by TrustZone-M. Using M-Step, we have exposed 24 vulnerable function

calls in the latest version of Mbed TLS[1]. Notably, we have shown that even Mbed TLS instances running on platforms with cryptographic accelerators remain vulnerable, when the private key is still manipulated in software by scaffolding code. Moreover, because Mbed TLS is integrated into the TF-M kernel, the reference Arm TEE implementation for TrustZone-M, these vulnerabilities were likely to be present in the wild[2]. Based on this evidence, we argue that the pervasive lack of awareness regarding microarchitectural side-channels on MCUs might leave a wide range of applications vulnerable to similar attacks or variants of the attacks unveiled in this thesis.

Takeaway 3: Microarchitectural attacks against real-world production software extend beyond high-end platforms and pose a real threat to Arm-based MCUs.

8.1.1 Concluding Remarks.

In summary, our findings provide empirical evidence that challenge the prevailing assumption that MCUs are too simple to be vulnerable to microarchitectural attacks. We have demonstrated that an attacker can reliably extract secret information from an isolated victim domain on a TrustZone-M-enabled MCU. Contrary to prior belief, the perceived simplicity of MCUs is not a limitation; but rather, a key enabler of side-channel attacks on MCUs. In fact, we have shown that the highly deterministic and simple microarchitecture of these devices yields significant advantages for an attacker which experience: *(i)* an improved signal-to-noise ratio and *(ii)* highly deterministic traces. Moreover, intrinsic MCU characteristics, notably their deterministic behavior and rich set of peripherals, create novel exploitation opportunities that enable the construction of hardware gadgets. Ultimately, our findings suggest that the system security community must revise its prevailing narrative around side-channel attacks on MCUs. Rather than being dismissed as too simple to be exploitable, MCUs must be recognized as simple, yet still exploitable.

8.2 Future Work

While this thesis explored several novel attack vectors on well-established, widely deployed MCUs, next-generation devices will introduce new opportunities for more sophisticated microarchitectural exploitation. MCUs are increasingly being released with advanced microarchitectural features and are being integrated into more complex systems, naturally expanding the microarchitectural attack surface of these platforms. We identify three trends that are likely to increase the likelihood of microarchitectural leakage on MCUs: *(i)* trend #1, Cortex-M CPU microarchitectures are becoming more complex; *(ii)* trend #2, MCUs are integrating specialized processing units (xPUs), including neural processing units (NPUs) and graphics processing units (GPUs); and *(iii)* trend #3, emerging SoCs designs colocate MCUs and APUs on a single die. As these trends unfold, the microarchitectural attack surface on MCUs will expand, creating new opportunities for sophisticated side-channel exploitation.

[1] The latest version at the time of our research, Mbed TLS v3.6.3.1.

[2] On versions prior to TF-M V2.2.2, before we performed responsible disclosure and Arm implemented patches.

Table 12: Representative survey of MCUs integrating specialized processing units (NPUs and GPUs).

			CPU			NPU				GPU		
MCU	**Year**	**Vendor**	N	CPU	MHZ	N	NPU	MHZ	GOPS	N	GPU	MHZ
Ensemble E3 [247]	'21	Alif	2	Cortex-M55	400	2	Ethos-U55	400	250	1	D/AVE 2D	400
MCX N Series [248]	'22	NXP	2	Cortex-M33	150	1	eIQ Neutron	150	4.8	0	N.A.	–
STM32N6 [249]	'24	ST	1	Cortex-M55	800	1	Neural-ART	1000	600	1	Neo-Chrom 2.5D	800
RA8P1 [250]	'24	Renesas	1	Cortex-M85	1000	1	Ethos-U55	500	250	1	2D Drawing Engine	1000
PSOC Edge E8x [251]	'25	Infineon	2	Cortex-M33/55	400	1	Ethos-U55	400	250	1	2.5D GPU	400
i.MX RT700 [252]	'25	NXP	2	Cortex-M33	325	1	eIQ Neutron	325	41.6	1	2.5D GPU	325
NuMicro M55M1 [253]	'25	Nuvoton	1	Cortex-M55	200	1	Ethos-U55	200	128	0	N.A.	–
Ensemble E4 [247]	'25	Alif	2	Cortex-M55	400	3	Ethos-U55/U85	400	454	1	D/AVE 2D	400
AM13Ex Series [254]	'26	Texas	1	Cortex-M33	200	1	TinyEngine	200	6.4	0	N.A.	–

Trend #1 - Increasing Microarchitectural Complexity in Cortex-M CPUs. While traditional MCUs were constrained to simple in-order execution, modern Cortex-M CPUs increasingly incorporate performance-oriented microarchitectural features. Notably, recent state-of-the-art Arm Cortex-M CPUs, such as the Cortex-M55 and Cortex-M85, introduce branch predictors, superscalar pipelines, and built-in caches (cf. Table 2, Section 2.2), substantially increasing microarchitectural complexity. This evolution naturally widens the attack surface for microarchitectural side-channels on MCUs. In this thesis, we demonstrated that even rudimentary microarchitectural components can be leveraged to mount powerful attacks. More complex microarchitectural features are therefore likely to further exacerbate this threat. Moreover, while these CPUs are being developed to deliver higher performance, they retain the same underlying design goal (recall Section 2.1): high determinism and predictable real-time behavior. Complex microarchitectural components operating in this low-noise deterministic environment may therefore improve an attacker's ability to extract information from advanced microarchitectural structures.

We expect microarchitectural attacks on MCUs to mirror the trend observed over the past three decades on APUs. Early attacks exploited rudimentary end-to-end timing variations [96, 101–104]. As microarchitectural complexity increased, attacks evolved to exploit more powerful microarchitectural primitives such as caches [61, 67, 101, 104–112, 115, 117–119, 124, 136–146], branch predictors [34–36, 127–129], and speculative execution [14, 171, 184]. We expect this pattern to extend to MCUs as their microarchitectural complexity increases. We identify this as a compelling direction for future research on novel microarchitectural attacks against MCUs.

Trend #2 - Integration of Specialized Processing Units into MCUs. To support more compute intensive workloads such as ML, artificial intelligence (AI), and applications with graphical user interfaces (GUIs), vendors are progressively integrating specialized processing units, including NPUs and GPUs, into MCUs, as summarized in Table 12. This integration substantially increases MCU complexity and, because these xPUs share common hardware resources at the system level, introduces the risk of microarchitectural leakage between the CPU and competing processing units. These xPUs provide sophisticated code-execution capabilities that extend far beyond conventional peripherals. This widens the attack surface. In this thesis, we demonstrated that complex attacks can already be mounted using rudimentary peripherals, such as DMAs and timers; an attacker controlling a more capable unit, such as a GPU or

Table 13: Representative survey of heterogeneous multiprocessing platforms integrating an APU and an MCU on the same SoC.

			APU			MCU		
Platform	**Year**	**Vendor**	N	CPU	MHZ	N	CPU	MHZ
i.MX 8ULP [255]	'21	NXP	2	Cortex-A35	1000	1	Cortex-M33	216
RZ/G3S [256]	'21	Renesas	1	Cortex-A55	1100	2	Cortex-M33	250
Sitara AM62x [257]	'22	Texas	4	Cortex-A53	1400	1	Cortex-M4	400
i.MX 93 [258]	'23	NXP	2	Cortex-A55	1700	1	Cortex-M33	250
STM32MP2 [259]	'24	ST	2	Cortex-A35	1500	2	Cortex-M0+/33	400
Ensemble E8 [247]	'25	Alif	2	Cortex-A32	800	2	Cortex-M55	400

NPU, could therefore craft more advanced hardware gadgets. Such xPUs may allow attackers to extract high-resolution information from the CPU and mount concurrent-execution microarchitectural attacks by running malicious code alongside victim xPUs on shared hardware.

Among existing xPUs, NPUs present a particularly compelling threat model. NPUs are expected to be shared across mutually distrusting entities, fundamentally expanding the threat model of MCUs. When deploying AI at the edge, models from different providers, which may represent thousands of dollars in development cost, must safely coexist in an inherently hostile environment. This constitutes the first realistic use case for multi-tenancy on MCUs, where AI models from distinct third parties execute on shared hardware. Notably, some platforms already reflect this threat model. For instance, the STM32N6 [249] features an NPU designed to be time-shared across multiple AI models and implements security mechanisms for model confidentiality, such as on-the-fly encryption and decryption of model weights, preventing a compromised CPU from leaking them via direct readout.

While this architectural isolation prevents direct model leakage, it may still leave room for microarchitectural side-channel attacks. We identify two attack scenarios in which AI models running on the NPU may be vulnerable to microarchitectural attacks: *(i)* xPU-to-NPU attacks, where a CPU, GPU, or other xPU running concurrently with the NPU exploits contention on shared system resources to extract information from executing AI models; *(ii)* NPU-to-NPU attacks, where competing AI models time-share the NPU and a malicious model extracts information from microarchitectural state left by prior models executions. Furthermore, the NPU may also be used to exploit the rest of the system (NPU-to-xPU attack), where an attacker deploys a malicious AI model on the NPU and launches microarchitectural attacks against other processing units.

The proliferation of capable processing units on modern MCUs, coupled with increasingly strong threat models, substantially expands the microarchitectural attack surface of next-generation MCUs. We identify this as a compelling direction for future research, particularly as computation continues to shift toward the deep edge.

Trend #3 - Integration of MCUs and APUs on the Same SoC. While this thesis focuses on single-core MCUs, which remain the dominant segment of the MCU market, multi-core MCUs are beginning to emerge, such as the Ensemble E3 [247] and PSoC Edge E8x [251] (cf. Table 12). More significantly,

an emerging architectural trend integrates MCUs into heterogeneous multiprocessing SoCs, where both MCUs and APUs are colocated on the same die, as summarized in Table 13. These integrated designs share a common interconnect, system-level peripherals, and some memory banks, expanding the microarchitectural attack surface. Concretely, an attacker controlling the APU, which typically operates at much higher clock frequencies, may be able to sample the MCU's microarchitectural state with high temporal resolution. Moreover, the MCU's inherent determinism, combined with the APU's access to system-level peripherals, may enable low-noise side-channel measurements and sophisticated APU-to-MCU attacks. In this thesis, we explored only intra-core microarchitectural attacks. These emerging architectures may instead enable high-resolution cross-core attacks that exploit contention on shared system resources between asymmetric processing units.

The integration of Arm Cortex-M CPUs alongside APUs within larger SoCs is already reflected in the TF-M [46] architecture, where the MCU serves as a secure processing environment and the APU operates as the untrusted domain. An attacker controlling the APU is thus incentivized to target the MCU, which executes security-critical operations. As computation demands and emerging use cases continue to drive deeper integration of MCUs into complex systems, heterogeneous multiprocessing is likely to introduce novel microarchitectural attack vectors between asymmetric processing units. We identify this as a compelling direction for future research on cross-core microarchitectural attacks in SoCs featuring MCUs.

Bibliography

[1] M. T. Inc. *PIC18F04/05/14/15Q41 Data Sheet*. Tech. rep. Microchip Technology Inc, 2021 (cit. on p. 1).

[2] A. Ltd. *Arm®v7-M Architecture Reference Manual*. Tech. rep. 2021 (cit. on pp. 1, 25, 28, 52).

[3] RISC-V. *Riscv-Worlds Offical Github*. `https://github.com/riscv/riscv-worlds`. Accessed: 17-03-2026 (cit. on p. 1).

[4] A. Ltd. *Arm®v8-M Architecture Reference Manual*. Tech. rep. 2024 (cit. on pp. 1, 25, 28, 52, 62, 93, 101).

[5] Arm. *Arm Cortex-M Processor Comparison Table*. Tech. rep. 2023 (cit. on pp. 1, 24, 25).

[6] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *Journ. ACM Computing Surveys*. 2019 (cit. on pp. 1, 4, 5, 23, 24, 28, 89).

[7] C. Rodrigues and S. Pinto. "Talk: Hand Me Your SECRET, MCU! Microarchitectural Timing Attacks on Microcontrollers are Practical". In: *BlackHat Asia*. `https://blackhat.com/asia-23/briefings/schedule/#hand-me-your-secret-mcu-microarchitectural-timing-attacks-on-microcontrollers-are-practical-30579`. 2023 (cit. on pp. 1, 50).

[8] C. Rodrigues, D. Oliveira, and S. Pinto. "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect". In: *Proc. of IEEE S&P*. 2024 (cit. on pp. 1, 6, 7, 9, 40, 46, 47, 50, 76, 87, 96, 97, 103, 108).

[9] A. Barbosa et al. "WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices". In: *Proc. of IEEE EWSN*. 2024 (cit. on pp. 1, 47).

[10] J. Müller et al. "MCU-Wide Timing Side Channels and Their Detection". In: *Proc. of DAC*. 2024 (cit. on p. 1).

[11] C. Rodrigues et al. "M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M". In: *Proc. of USENIX Security*. 2026 (cit. on pp. 1, 33, 40, 46, 47, 111).

[12] Q. Ge et al. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journ. Cryptogr. Eng.* 2018 (cit. on pp. 2, 3, 31, 33, 42, 48).

[13] H. Naghibijouybari, E. M. Koruyeh, and N. Abu-Ghazaleh. “Microarchitectural Attacks in Heterogeneous Systems: A Survey”. In: *ACM Computing Surveys* (2022) (cit. on pp. 2, 3, 10, 31, 33, 48).

[14] C. Canella et al. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *Proc. of USENIX Security*. 2019 (cit. on pp. 2, 3, 10, 31, 33, 35, 40, 42, 48, 50, 122).

[15] M. Mushtaq et al. “Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA”. In: *Information Systems*. 2020 (cit. on pp. 2, 3, 33, 48).

[16] W. Xiong and J. Szefer. “Survey of Transient Execution Attacks and Their Mitigations”. In: *Journ. ACM Comput. Surv.* 2022 (cit. on pp. 2, 3, 33, 35, 42, 48).

[17] L. Zhuang, F. Zhou, and J. D. Tygar. “Keyboard Acoustic Emanations Revisited”. In: *Proc. of CCS*. 2005 (cit. on pp. 2, 3, 30, 31).

[18] D. Genkin, A. Shamir, and E. Tromer. “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis”. In: *Proc. of CRYPTO*. 2014 (cit. on pp. 2, 3, 30, 31).

[19] D. Genkin et al. “Synesthesia: Detecting Screen Content via Remote Acoustic Side Channels”. In: *Proc. of S&P*. 2018 (cit. on pp. 2, 3, 30).

[20] J.-J. Quisquater and D. Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards”. In: *Proc. of E-SMART*. 2001 (cit. on pp. 2, 3).

[21] D. Genkin et al. “ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels”. In: *Proc. of CCS*. 2016 (cit. on pp. 2, 3).

[22] G. Camurati et al. “Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers”. In: *Proc. of CCS*. 2018 (cit. on pp. 2, 3).

[23] K. Gandolfi, C. Mourtel, and F. Olivier. “Electromagnetic Analysis: Concrete Results”. In: *Proc. of CHES*. 2001 (cit. on pp. 2, 3).

[24] P. Kocher, J. Jaffe, and B. Jun. “Differential Power Analysis”. In: *Proc, of CRYPTO*. 1999 (cit. on pp. 2, 3, 31).

[25] T. Messerges, E. Dabbish, and R. Sloan. “Examining smart-card security under the threat of power analysis attacks”. In: 2002 (cit. on pp. 2, 3).

[26] L. Goubin. “A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems”. In: *Proc. of PKC*. 2002 (cit. on pp. 2, 3).

[27] M. Lipp et al. “PLATYPUS: Software-based Power Side-Channel Attacks on x86”. In: *Proc. of S&P*. 2021 (cit. on pp. 2, 3, 31, 32).

[28] P. C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Proc. of CRYPTO*. 1996 (cit. on pp. 2, 3, 31, 32, 35, 40).

[29] R. Spreitzer et al. “Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices”. In: *IEEE Communications Surveys & Tutorials* (2018) (cit. on pp. 2, 31).

[30] Y. Abdelrahman et al. “Stay Cool! Understanding Thermal Attacks on Mobile-based User Authentication”. In: *Proc. of CHI*. 2018 (cit. on pp. 3, 30).

[31] B. Gras et al. “ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures”. In: *Proc. of NDSS*. 2020 (cit. on pp. 3, 33).

[32] A. C. Aldaya et al. “Port Contention for Fun and Profit”. In: *Proc. of S&P*. 2019 (cit. on pp. 3, 33).

[33] A. Bhattacharyya et al. “SMoTherSpectre: Exploiting Speculative Execution Through Port Contention”. In: *Proc. of CCS*. 2019 (cit. on pp. 3, 33, 42).

[34] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *Proc. of ASIACCS*. 2007 (cit. on pp. 3, 32, 33, 122).

[35] S. Lee et al. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *Proc. of USENIX Security*. 2017 (cit. on pp. 3, 32, 33, 76, 122).

[36] D. Evtyushkin et al. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *Proc. of ASPLOS*. 2018 (cit. on pp. 3, 32, 33, 122).

[37] J. Van Bulck, F. Piessens, and R. Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *25th ACM Conference on Computer and Communications Security (CCS)*. 2018-10, pp. 178–195 (cit. on pp. 3, 6, 42, 43, 45–47, 49, 51, 52, 56, 72, 76, 84, 87, 90, 93, 96, 99, 102, 103, 105, 108, 120).

[38] Statista. *DIGITAL & TRENDS Microcontrollers*. Tech. rep. Statista, 2023 (cit. on pp. 3, 24).

[39] yolegroup. *Microcontroller Quarterly Market Monitor Q1 2022*. Tech. rep. 2022 (cit. on pp. 3, 24).

[40] M. Costa et al. “Shifting Capsule Networks from the Cloud to the Deep Edge”. In: *ACM Trans. Intell. Syst. Technol.* 2022 (cit. on p. 4).

[41] M. Costa and S. Pinto. “David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge”. In: *Proc. of EuroS&P*. 2024 (cit. on p. 4).

[42] M. Costa, T. Gomes, and S. Pinto. “SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems”. In: *IEEE Internet of Things Journal*. 2025 (cit. on p. 4).

[43] 360 Research Reports. “ARM Microcontrollers Market Growth, Size & Trend”. In: *360 Research Reports*. 2026 (cit. on pp. 4, 24).

[44] Grand View Research. “Microcontroller Market Size & Share | Industry Report, 2033”. In: *Grand View Research*. 2026 (cit. on pp. 4, 24).

[45] Mordor Intelligence. “Microcontroller Market Size, Share & Growth Trends 2031”. In: *Mordor Intelligence*. 2026 (cit. on pp. 4, 24).

[46] Arm. *Arm Trusted Firmware*. `www.trustedfirmware.org`. Accessed: 2026-03-22 (cit. on pp. 4, 29, 79, 80, 124).

[47] Arm. *Arm® Platform Security Architecture Firmware Framework 1.0*. Tech. rep. 2019 (cit. on pp. 4, 29).

[48] Arm. *Arm® Platform Security Architecture Trusted Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M 2.0*. Tech. rep. 2019 (cit. on pp. 4, 7, 29, 80).

[49] V. Costan and S. Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive*. 2016 (cit. on pp. 4, 23).

[50] Intel. *Intel Trust Domain Extensions (Intel TDX)*. `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html`. Accessed: 17-03-2026 (cit. on pp. 4, 23).

[51] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. `https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more`. 2020 (cit. on pp. 4, 23).

[52] D. Cerdeira et al. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: *Proc. of S&P*. 2020 (cit. on p. 5).

[53] J. Van Bulck et al. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *Proc. of CCS*. 2019 (cit. on pp. 5, 38, 39).

[54] M. SaB, R. Mitev, and A.-R. Sadeghi. “Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M”. In: *arXiv*. 2023 (cit. on p. 5).

[55] J. Obermaier, M. Schink, and K. Moczek. “One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers”. In: *Proc. of USENIX WOOT*. 2020 (cit. on p. 5).

[56] C. Shepherd et al. “Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis”. In: *Journ. Comp. & Sec.* 2021 (cit. on p. 5).

[57] C. H. Kim and J.-J. Quisquater. “Faults, Injection Methods, and Fault Attacks”. In: *Journ. IEEE Design & Test of Comp.* 2007 (cit. on p. 5).

[58] J. Van Bulck, F. Piessens, and R. Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, 2017-10, 4:1–4:6 (cit. on pp. 6, 39, 51, 52, 56, 84, 89, 120).

[59] Intel. *Intel Trust Domain Extension Research and Assurance*. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/tdx-security-research-and-assurance.html`. 2023-04 (cit. on pp. 6, 39, 51, 52, 89, 115, 116).

[60] L. Wilke, F. Sieck, and T. Eisenbarth. “TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX”. In: *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2024 (cit. on pp. 6, 39, 51, 52, 89).

[61] F. Rauscher et al. “TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX”. In: *Proc. of USENIX Security*. 2025 (cit. on pp. 6, 32, 35, 39, 51, 52, 89, 122).

[62] L. Wilke et al. “SEV-Step: A Single-Stepping Framework for AMD-SEV”. In: *TCHES* (2023-12) (cit. on pp. 6, 39, 51, 52, 89, 90, 93, 100, 102).

[63] K. Ryan. “Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm’s TrustZone”. In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019, pp. 181–194 (cit. on pp. 6, 52, 89, 90).

[64] Z. Kou et al. “Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+ evict”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 979–984 (cit. on pp. 6, 52, 76, 89, 90).

[65] J. Van Bulck and F. Piessens. “SGX-Step: An Open-Source Framework for Precise Dissection and Practical Exploitation of Intel SGX Enclaves”. In: *ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award Finalist Short Paper*. 2023-12 (cit. on pp. 6, 39, 51, 52, 56, 89, 100, 105, 106).

[66] Trustonic. *Trustonic Kinibi-M*. `www.trustonic.com/technology/`. Accessed: 2023-02-09 (cit. on pp. 7, 29).

[67] K. Ryan. “Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm’s TrustZone”. In: *Proc. of ACM CCS*. 2019 (cit. on pp. 10, 35, 40, 122).

[68] D. Moghimi et al. “COPYCAT: Controlled Instruction-Level Attacks on Enclaves”. In: *Proc. of USENIX Security*. 2020 (cit. on pp. 10, 38, 39, 76, 88, 101, 109).

[69] S. van Schaik et al. “CacheOut: Leaking Data on Intel CPUs via Cache Evictions”. In: *Proc. of S&P*. 2021 (cit. on pp. 10, 40, 42).

[70] AMD. *AMD Ryzen AI 400 Series Processors: Leadership Performance to Power Your Potential*. `https://www.amd.com/content/dam/amd/en/documents/partner-hub/ryzen/ryzen-ai-400-series-qrg.pdf`. 2025 (cit. on p. 16).

[71] S. Microelectronics. *Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 256KB flash, 64KB SRAM, USB FS, LCD, ext. SMPS*. Tech. rep. 2024 (cit. on p. 16).

[72] J. Ahn et al. “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing”. In: *Proc. of ISCA*. 2015 (cit. on p. 19).

[73] O. Mutlu et al. “A Modern Primer on Processing in Memory”. In: *arXiv preprint arXiv:2012.03112*. 2020 (cit. on p. 19).

[74] D. Patterson et al. “A Case for Intelligent RAM”. In: *IEEE Micro*. 1997 (cit. on p. 19).

[75] D. Oliveira et al. “Shared Resource Contention in MCUs: A Reality Check and the Quest for Timeliness”. In: *Proc. of ECRTS*. 2024 (cit. on p. 20).

[76] D. Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proc. of EuroSys*. 2020 (cit. on p. 23).

[77] X. Li et al. “Design and Verification of the Arm Confidential Compute Architecture”. In: *Proc. of OSDI*. 2022 (cit. on p. 23).

[78] R. Sahita et al. “CoVE: Towards Confidential Computing on RISC-V Platforms”. In: *Proc. of Computing Frontiers*. 2023 (cit. on p. 23).

[79] P. Sparks. *The route to a trillion devices: The outlook for IoT investment to 2035*. Tech. rep. 2017 (cit. on p. 24).

[80] A. Ltd. *Arm Cortex-M Processors*. `https://www.arm.com/products/silicon-ip-cpu?families=cortex-m&showall=true`. Accessed: 30-07-2025 (cit. on p. 24).

[81] A. Ltd. *Arm®v6-M Architecture Reference Manual*. Tech. rep. 2018 (cit. on p. 25).

[82] Arm. *Arm CPU Security Bulletin: Spectre/Meltdown*. `https://developer.arm.com/documentation/110280/latest/`. Accessed: 23-03-2026 (cit. on p. 25).

[83] J. Yiu. “The Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors”. In: *Newnes*. 2020 (cit. on p. 26).

[84] ProvenRun. *ProvenCore-M: A secure Real Time Operating System*. `https://provenrun.com/products/provencore-m/`. Accessed: 2023-02-09 (cit. on pp. 29, 79).

[85] T. A. Drozdovskyi and O. S. Moliavko. “mTower: Trusted Execution Environment for MCU-based devices”. In: *Journ. Open Sour. Soft.* 2019 (cit. on pp. 29, 79).

[86] D. Oliveira, T. Gomes, and S. Pinto. “uTango: An Open-Source TEE for IoT Devices”. In: *Journ. IEEE Access*. 2022 (cit. on pp. 29, 79).

[87] MCUBoot. *MCUboot: Secure boot for 32-bit Microcontrollers!* `https://docs.mcuboot.com/`. Accessed: 23-03-2026 (cit. on p. 29).

[88] A. Barisani and D. Bianco. “Sniffing Keystrokes With Lasers and Voltmeters”. In: *Proc. of Black Hat USA*. 2009 (cit. on p. 30).

[89] A. Davis et al. “The Visual Microphone: Passive Recovery of Sound from Video”. In: *ACM Transactions on Graphics (TOG)*. 2014 (cit. on p. 30).

[90] B. Nassi et al. “Lamphone: Passive Sound Recovery from a Desk Lamp’s Light Bulb Vibrations”. In: *Proc. of USENIX Security*. 2022 (cit. on pp. 30, 31).

[91] M. Backes et al. “Acoustic Side-Channel Attacks on Printers”. In: *Proc. of USENIX Security*. 2010 (cit. on p. 31).

[92] D. Agrawal et al. “The EM Side-Channel(s)”. In: *Proc. of CHES*. 2003 (cit. on p. 31).

[93] M. Hutter and J.-M. Schmidt. “The Temperature Side Channel and Heating Fault Attacks”. In: *Proc. of CARDIS*. 2013 (cit. on p. 31).

[94] M. Randolph and W. Diehl. “Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman”. In: *Cryptography*. 2020 (cit. on p. 31).

[95] S. Picek et al. “SoK: Deep Learning-based Physical Side-channel Analysis”. In: *ACM Comput. Surv.* 2023 (cit. on p. 31).

[96] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *Proc. of IEEE S&P*. 2013 (cit. on pp. 31, 32, 38, 42, 122).

[97] M. C. Martínez-Rodríguez, I. M. Delgado-Lozano, and B. B. Brumley. “SoK: Remote Power Analysis”. In: *Proc. of ARES*. 2021 (cit. on p. 31).

[98] C. O’Flynn and Z. Chen. “ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research”. In: *Proc. of COSADE*. 2014 (cit. on p. 31).

[99] T. Roche et al. “A Side Journey To Titan”. In: *Proc. of USENIX Security*. 2021 (cit. on pp. 31, 42).

[100] T. Roche et al. “EUCLEAK Side-Channel Attack on the YubiKey 5 Series”. In: *Proc. of IEEE S&P*. 2025 (cit. on pp. 31, 42).

[101] C. Percival. “Cache Missing for Fun and Profit”. In: *Proc. of BSDCan*. 2005 (cit. on pp. 32, 35–38, 42, 54, 96, 122).

[102] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *Proc. of CT-RSA*. 2018 (cit. on pp. 32, 33, 38, 42, 122).

[103] E. Tromer, D. A. Osvik, and A. Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. In: *Journal of Cryptology*. 2010 (cit. on pp. 32, 37, 38, 42, 122).

[104] D. A. Osvik, A. Shamir, and E. Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *Proc. of CT-RSA*. 2006 (cit. on pp. 32, 33, 35–38, 42, 54, 96, 122).

[105] L. Giner, S. R. Neela, and D. Gruss. “Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2025 (cit. on pp. 32, 35, 38, 122).

[106] Y. Jin et al. “GhostCache: Timer- and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips”. In: *Proc. of CCS*. 2025 (cit. on pp. 32, 35, 122).

[107] S. Son, D. Moghimi, and B. Gulmezoglu. “SMaCk: Efficient Instruction Cache Attacks via Self-Modifying Code Conflicts”. In: *Proc. of ASPLOS*. 2025 (cit. on pp. 32, 35, 122).

[108] D.-o. Kim, J. Song, and I. Yun. “CROSS-X: Generalized and Stable Cross-Cache Attack on the Linux Kernel”. In: *Proc. of CCS*. 2025 (cit. on pp. 32, 35, 122).

[109] L.-C. Chiang and S.-W. Li. “Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV”. In: *Proc. of ASPLOS*. 2025 (cit. on pp. 32, 35, 38, 122).

[110] F. Rauscher et al. “A Systematic Evaluation of Novel and Existing Cache Side Channels”. In: *Proc. of NDSS*. 2025 (cit. on pp. 32, 35, 36, 38, 122).

[111] S. R. Neela et al. “Eviction Notice: Reviving and Advancing Page Cache Attacks”. In: *Proc. of NDSS*. 2026 (cit. on pp. 32, 35, 122).

[112] H. Weissteiner et al. “Continuous User Behavior Monitoring using DNS Cache Timing Attacks”. In: *Proc. of NDSS*. 2026 (cit. on pp. 32, 35, 122).

[113] D. R. E. Gnad, J. Krautter, and M. 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 (cit. on pp. 32, 45–47).

[114] C. O’Flynn and A. Dewar. “On-Device Power Analysis Across Hardware Security Domains.: Stop Hitting Yourself”. In: *Journ. IACR Trans. on Crypt. Hard. and Embed. Syst.* 2019 (cit. on pp. 32, 45–47).

[115] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proc. of USENIX Security*. 2014 (cit. on pp. 32, 33, 35–37, 122).

[116] D. Gruss et al. “Flush+ Flush: A Fast and Stealthy Cache Attack”. In: *Proc. of DIMVA*. 2016 (cit. on pp. 32, 33, 38).

[117] D. Gruss, R. Spreitzer, and S. Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *Proc. of USENIX Security*. 2015 (cit. on pp. 32, 33, 35–38, 67, 122).

[118] F. Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *Proc. of IEEE S&P*. 2015 (cit. on pp. 32, 33, 35, 37, 49, 52, 122).

[119] M. Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *Proc. of USENIX Security*. 2016 (cit. on pp. 32, 33, 35–38, 122).

[120] N. Zhang et al. “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices”. In: *IACR Cryptol. ePrint Arch.* 2016 (cit. on pp. 32, 33).

[121] C. Canella et al. “Fallout: Leaking Data on Meltdown-Resistant CPUs”. In: *Proc. of ACM CCS*. 2019 (cit. on pp. 32, 33, 40, 42).

[122] B. Gras et al. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *Proc. of USENIX Security*. 2018 (cit. on pp. 32, 33).

[123] S. Shinde et al. “Preventing Page Faults from Telling Your Secrets”. In: *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2016, pp. 317–328 (cit. on pp. 32, 33).

[124] B. Gras et al. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *Proc. of NDSS*. 2017 (cit. on pp. 32, 33, 35, 122).

[125] S. van Schaik et al. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think”. In: *Proc. of USENIX Security*. 2018 (cit. on pp. 32, 33).

[126] J. Koschel et al. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs”. In: *Proc. of EuroS&P*. 2020 (cit. on pp. 32, 33).

[127] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Understanding and Mitigating Covert Channels Through Branch Predictors”. In: *Journ. ACM Trans. Archit. Code Optim.* 2016 (cit. on pp. 32, 33, 122).

[128] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. “Predicting Secret Keys Via Branch Prediction”. In: *Proc. of CT-RSA*. 2007 (cit. on pp. 32, 33, 122).

[129] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *Proc. of IEEE/ACM MICRO*. 2016 (cit. on pp. 32, 33, 122).

[130] M. Schwarz and D. Gruss. “How Trusted Execution Environments Fuel Research on Microarchitectural Attacks”. In: *IEEE Security & Privacy*. 2020 (cit. on p. 32).

[131] Y. Yarom, D. Genkin, and N. Heninger. “CacheBleed: A Timing Attack on OpenSSL Constant Time RSA”. In: *Proc. of CHES*. 2016 (cit. on p. 33).

[132] R. Paccagnella, L. Luo, and C. 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 (cit. on pp. 33, 52).

[133] J. Wan et al. “MeshUp: Stateless Cache Side-channel Attack on CPU Mesh”. In: *Proc. of S&P*. 2022 (cit. on pp. 33, 52).

[134] Z. Wu, Z. Xu, and H. Wang. “Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud”. In: *Proc. of USENIX Security*. 2012 (cit. on pp. 33, 52).

[135] M. Bognar, J. V. Bulck, and F. Piessens. “Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures”. In: *Proc. of S&P*. 2022 (cit. on pp. 33, 42, 44–47, 50, 52).

[136] O. Aciiçmez. “Yet another MicroArchitectural Attack:: exploiting I-Cache”. In: *Proc. of CSAW*. 2007 (cit. on pp. 35, 37, 49, 52, 122).

[137] D. Gullasch, E. Bangerter, and S. Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *Proc. of IEEE S&P*. 2011 (cit. on pp. 35, 37, 49, 52, 122).

[138] G. Irazoqui, T. Eisenbarth, and B. Sunar. “S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing–and Its Application to AES”. In: *Proc. of IEEE S&P*. 2015 (cit. on pp. 35, 37, 49, 52, 122).

[139] M. Green et al. “AutoLock: Why Cache Attacks on ARM Are Harder Than You Think”. In: *Proc. of USENIX Security*. 2017 (cit. on pp. 35, 122).

[140] P. Cronin et al. “An Exploration of ARM System-Level Cache and GPU Side Channels”. In: *Proc. of ACSAC*. 2021 (cit. on pp. 35, 122).

[141] J. Wang et al. “Cache-in-the-Middle (CITM) Attacks: Manipulating Sensitive Data in Isolated Execution Environments”. In: *Proc. of ACM CCS*. 2020 (cit. on pp. 35, 122).

[142] G. Haas, S. Potluri, and A. Aysu. “iTimed: Cache Attacks on the Apple A10 Fusion SoC”. In: *Proc. of IEEE HOST*. 2021 (cit. on pp. 35, 40, 122).

[143] A. Shusterman et al. “Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses”. In: *Proc. of USENIX Security*. 2021 (cit. on pp. 35, 122).

[144] X. Zhang, Y. Xiao, and Y. Zhang. “Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices”. In: *Proc. of ACM CCS*. 2016 (cit. on pp. 35, 122).

[145] B. Lapid and A. Wool. *Cache-Attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis*. Cryptology ePrint Archive. 2018 (cit. on pp. 35, 122).

[146] D. Katzman et al. “The Gates of Time: Improving Cache Attacks with Transient Execution”. In: *Proc. of USENIX Security*. 2023 (cit. on pp. 35, 40, 122).

[147] Y. Xu, W. Cui, and M. Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *Proc. of S&P*. 2015 (cit. on pp. 35, 38, 39, 76).

[148] O. Aciicmez, W. Schindler, and C. K. Koc. “A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL”. In: *Proc. of CT-RSA*. 2007 (cit. on p. 37).

[149] M. S. Inci et al. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *Proc. of USENIX Security*. 2016 (cit. on p. 37).

[150] M. Schwarz et al. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *14th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2017 (cit. on p. 37).

[151] F. Brasser et al. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *Proc. of WOOT*. 2017 (cit. on p. 37).

[152] D. Genkin et al. “Drive-by Key-Extraction Cache Attacks from Portable Code”. In: *Proc. of IEEE S&P*. 2018 (cit. on p. 37).

[153] M. Neve and J.-P. Seifert. “Advances on Access-Driven Cache Attacks on AES”. In: *Proc. of SAC*. 2006 (cit. on p. 37).

[154] M. Kayaalp et al. “A High-Resolution Side-Channel Attack on Last-Level Cache”. In: *Proc. of DAC*. 2016 (cit. on pp. 37, 76).

[155] M. Fernández et al. “Cache Misses and the Recovery of the Full AES 256 Key”. In: *Proc. of Applied Sciences (MDPI)*. 2019 (cit. on p. 37).

[156] J. Götzfried et al. “Cache Attacks on Intel SGX”. In: *Proc. of EUROSEC*. 2017 (cit. on p. 37).

[157] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *Proc. of CHES*. 2017 (cit. on p. 37).

[158] Y. Zhang et al. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *Proc. of CCS*. 2012 (cit. on p. 37).

[159] M. Hähnel, W. Cui, and M. Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: *Proc. of USENIX ATC*. 2017 (cit. on p. 37).

[160] X. Wang et al. “PAPP: Prefetcher-Aware Prime and Probe Side-Channel Attack”. In: *Proc. of DAC*. 2019 (cit. on p. 37).

[161] G. Irazoqui et al. “Wait a Minute! A Fast, Cross-VM Attack on AES”. In: *Proc. of RAID*. 2014 (cit. on p. 37).

[162] N. Benger et al. ““Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way”. In: *Proc. of CHES*. 2014 (cit. on p. 37).

[163] J. van de Pol, N. P. Smart, and Y. Yarom. “Just a Little Bit More”. In: *Proc. of CT-RSA*. 2015 (cit. on p. 37).

[164] H. Cho et al. “Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone”. In: *Proc. of ACSAC*. 2018 (cit. on p. 38).

[165] C. Disselkoen et al. “Prime+ Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *Proc. of USENIX Security*. 2017 (cit. on p. 38).

[166] S. Briongos et al. “RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks”. In: *Proc. of USENIX Security*. 2020 (cit. on p. 38).

[167] J. Van Bulck et al. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *Proc. of USENIX Security*. 2017 (cit. on pp. 38, 39).

[168] W. Wang et al. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *Proc. of CCS*. 2017 (cit. on pp. 38, 39).

[169] J. Gyselinck et al. “Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution”. In: *Proc. of ESSoS*. 2018 (cit. on pp. 38, 39).

[170] W. Lee et al. “T-Time: A Fine-Grained Timing-Based Controlled-Channel Attack Against Intel TDX”. In: *Proc. of ESORICS*. 2025 (cit. on p. 39).

[171] P. Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proc. of S&P*. 2019 (cit. on pp. 40–42, 122).

[172] M. Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Proc. of USENIX Security*. 2018 (cit. on pp. 40, 42).

[173] M. Schwarz et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *Proc.s of ACM CCS*. 2019 (cit. on pp. 40, 42).

[174] S. van Schaik et al. “RIDL: Rogue In-Flight Data Load”. In: *Proc. of S&P*. 2019 (cit. on pp. 40, 42).

[175] J. Van Bulck et al. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *Proc. of S&P*. 2020 (cit. on pp. 40, 42).

[176] M. Lipp, D. Gruss, and M. Schwarz. “AMD Prefetch Attacks through Power and Time”. In: *Proc. of USENIX Security*. 2022 (cit. on p. 40).

[177] J. Ravichandran et al. “PACMAN: Attacking ARM Pointer Authentication with Speculative Execution”. In: *Proc. of ACM ISCA*. 2022 (cit. on p. 40).

[178] J. R. S. Vicarte et al. “Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest”. In: *Proc. of S&P*. 2022 (cit. on p. 40).

[179] VUSec. *MDS: Microarchitectural Data Sampling*. `https://mdsattacks.com/`. 2020 (cit. on p. 42).

[180] S. van Schaik et al. “CrossTalk: Speculative Data Leaks Across Cores Are Real”. In: *Proc. of IEEE S&P*. 2020 (cit. on p. 42).

[181] D. H. Lutas and A. V. Lutas. “Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction”. In: *Proc. of Black Hat*. 2019 (cit. on p. 42).

[182] M. H. I. Chowdhury and F. Yao. “BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions”. In: *Proc. of ICCD*. 2020 (cit. on p. 42).

[183] M. Schwarz et al. “NetSpectre: Read Arbitrary Memory Over Network”. In: *Proc. of ESORICS*. 2019 (cit. on p. 42).

[184] C. Trippel, D. Lustig, and M. Martonosi. “MeltdownPrime and SpectrePrime: Automatically Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols”. In: *IEEE Computer Architecture Letters*. 2018 (cit. on pp. 42, 122).

[185] V. Kiriansky and C. Waldspurger. “Speculative Buffer Overflows: Attacks and Defenses”. In: *arXiv preprint arXiv:1807.03757*. 2018 (cit. on p. 42).

[186] G. Chen et al. “SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution”. In: *Proc. of IEEE S&P*. 2019 (cit. on p. 42).

[187] E. M. Koruyeh et al. “Spectre Returns! Speculation Attacks Using the Return Stack Buffer”. In: *Proc. of WOOT*. 2018 (cit. on p. 42).

[188] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers”. In: *Proc. of CCS*. 2018 (cit. on p. 42).

[189] C. Canella et al. “KASLR: Break It, Fix It, Repeat”. In: *Proc. of AsiaCCS*. 2020 (cit. on p. 42).

[190] J. Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proc. of USENIX Security*. 2018 (cit. on p. 42).

[191] O. Weisse et al. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical Report*. 2018 (cit. on p. 42).

[192] J. Stecklina and T. Prescher. “LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels”. In: *arXiv preprint arXiv:1806.07480*. 2018 (cit. on p. 42).

[193] M. Bognar et al. “MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2023, pp. 651–670. doi: 10.1109/EUROSP57164.2023.00045 (cit. on pp. 42, 44, 46, 47, 52, 106).

[194] P. Sah and M. Hicks. “RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices”. In: *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. 2024, pp. 117–132 (cit. on pp. 42, 46, 47, 89, 90).

[195] M. Bognar et al. “Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers”. In: *Proc. of USENIX Sec.* 2024 (cit. on pp. 42, 46, 47, 52).

[196] D. J. Bernstein et al. “KyberSlash: Exploiting Secret-Dependent Division Timings in Kyber Implementations”. In: *Proc. of IACR TCHES*. 2025 (cit. on pp. 42, 45–47, 102, 115).

[197] T. Goodspeed. “Practical Attacks Against the MSP430 BSL”. In: *25th Chaos Communications Congress*. 2008 (cit. on p. 42).

[198] J. Noorman et al. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), pp. 1–33 (cit. on pp. 42, 90).

[199] A. Barenghi et al. “Exploring Cortex-M Microarchitectural Side Channel Information Leakage”. In: *Journ. IEEE Access*. 2021 (cit. on pp. 45, 47).

[200] X. Lou et al. “A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography”. In: *ACM Comput. Surv.* 2021 (cit. on p. 46).

[201] Microchip. *SAM L10/L11 Family Data Sheet*. Tech. rep. Microchip, 2020 (cit. on pp. 51, 79).

[202] STMicroelectronics. *RM0438 Reference manual: STM32L552xx and STM32L562xx advanced Arm-based 32-bit MCUs*. Tech. rep. STMicroelectronics, 2020 (cit. on pp. 51, 55, 79).

[203] NXP. *KV31F Sub-Family Reference Manual*. Tech. rep. NXP, 2020 (cit. on p. 51).

[204] STMicroelectronics. *RM0456 Reference manual: STM32U575/585 Arm-based 32-bit MCUs*. Tech. rep. STMicroelectronics, 2021 (cit. on p. 51).

[205] Q. Ge et al. “Your processor leaks information — and there’s nothing you can do about it”. In: *arXiv: Cryptography and Security*. 2016 (cit. on pp. 54, 55).

[206] T. Chothia, Y. Kawamoto, and C. Novakovic. “A Tool for Estimating Information Leakage”. In: *Proc of. Comp. Aid. Verific.* 2013 (cit. on p. 55).

[207] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell Sys. Tech. Jour.* 1948 (cit. on p. 55).

[208] J. K. Millen. “Covert Channel Capacity”. In: *Proc. of S&P*. 1987 (cit. on p. 55).

[209] R. P. Chari Suresh Rao Josyula R. “Template Attacks”. In: *Proc. of CHES*. 2002 (cit. on p. 67).

[210] B. B. Brumley and R. M. Hakala. “Cache-Timing Template Attacks”. In: *Proc. of ASIACRYPT*. 2009 (cit. on p. 67).

[211] M. Schwarz, F. Lackner, and D. Gruss. “JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits”. In: *Proc. of NDSS*. 2019 (cit. on p. 67).

[212] M. Busi et al. “Securing Interruptible Enclaved Execution on Small Microprocessors”. In: *Journ. ACM Trans. Program. Lang. Syst.* 2021 (cit. on p. 72).

[213] J. Van Bulck, J. T. Mühlberg, and F. Piessens. “VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks”. In: *Proc. of ACSAC*. 2017 (cit. on p. 76).

[214] T. Instruments. *Implementing An Ultra-Low-Power Keypad Interface With MSP430™ MCUs*. Tech. rep. Texas Instruments, 2002/18 (cit. on p. 76).

[215] T. Instruments. *Low-Power Hex Keypad Using MSP430™ MCUs*. Tech. rep. Texas Instruments (cit. on p. 76).

[216] T. Instruments. *BOOST-IR Infrared (IR) BoosterPack™ Plug-in Module*. Tech. rep. Texas Instruments (cit. on p. 76).

[217] T. Instruments. *USB Keyboard Using MSP430™ Microcontrollers*. Tech. rep. Texas Instruments (cit. on p. 76).

[218] QMK. *QMK Firmware*. `https://qmk.fm/` (cit. on p. 76).

[219] TMK. *tmk_keyboard*. `https://github.com/tmk/tmk_keyboard` (cit. on p. 76).

[220] S. Pinto and C. Garlati. “Multi Zone Security for Arm Cortex-M Devices”. In: *Proc. of Embed. World Conf.* 2020 (cit. on p. 79).

[221] T. L. Foundation. *Zephyr Project*. `www.zephyrproject.org`. Accessed: 2022-12-01 (cit. on p. 79).

[222] H. Jayakumar, A. Raha, and V. Raghunathan. “Hypnos: An Ultra-Low Power Sleep Mode with SRAM Data Retention for Embedded Microcontrollers”. In: *Proc. of ACM CODES*. 2014 (cit. on p. 83).

[223] H. Winderix, J. T. Mühlberg, and F. Piessens. “Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks”. In: *Proc. of EuroS&P*. 2021 (cit. on p. 83).

[224] 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 (cit. on p. 86).

[225] S. Constable et al. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". In: *32nd USENIX Security Symposium*. 2023-08, pp. 4051–4068 (cit. on pp. 89, 90, 93, 115, 116).

[226] Intel. *Security Advisory: TDXDown*. https://www.intel.com/content/www/us/en/security-center/announcement/intel-security-announcement-2024-10-08-001.html. 2024-10 (cit. on p. 89).

[227] Intel. *2025.3 IPU, Intel TDX Module Advisory*. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01312.html. 2025-08 (cit. on p. 89).

[228] S. Cuyt. "A Security Analysis of Interrupts in Embedded Enclaved Execution". https://distrinet.cs.kuleuven.be/software/sancus/publications/cuyt19thesis.pdf. MA thesis. KU Leuven, 2019 (cit. on pp. 89, 90).

[229] M. Busi et al. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors". In: *Proc. of IEEE CSF*. 2020 (cit. on pp. 89, 116).

[230] M. Schink and J. Obermaier. "Taking a Look into Execute-Only Memory". In: *Proc. of USENIX WOOT*. 2019 (cit. on pp. 89, 90).

[231] J. Van Bulck. *A Practical Attack Framework for Precise Enclave Execution Control*. https://github.com/jovanbulck/sgx-step. 2025 (cit. on p. 89).

[232] E. Aktas et al. *Intel Trust Domain Extensions (TDX) Security Review*. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf. 2023-04 (cit. on p. 89).

[233] N. Dutly et al. "AEX-NStep: Probabilistic Interrupt Counting Attacks on Intel SGX". In: *IEEE Symposium on Security and Privacy (S&P)*. 2026 (cit. on p. 90).

[234] Antmicro. *Renode*. https://renode.io/. Accessed: 15-06-2025 (cit. on p. 99).

[235] C. P. García and B. B. Brumley. "Constant-Time Callees with Variable-Time Callers". In: *Proc. of USENIX Sec.* 2017 (cit. on p. 109).

[236] S. Weiser, R. Spreitzer, and L. Bodner. "Single Trace Attack Against RSA Key Generation in Intel SGX SSL". In: *Proc. of AsiaCCS*. 2018 (cit. on p. 109).

[237] A. C. Aldaya and B. B. Brumley. "Cache-Timing Attacks on RSA Key Generation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019 (cit. on p. 109).

[238] C. P. García et al. "Certified Side Channels". In: *Proc. of USENIX Sec.* 2020 (cit. on pp. 109, 112).

[239] A. C. Aldaya and B. B. Brumley. “When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020 (cit. on pp. 109, 112, 113).

[240] I. Puddu et al. “Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend”. In: *Proc. of USENIX Sec.* 2021 (cit. on pp. 109, 112).

[241] Arm. *Arm Mbed TLS Security Policy*. `https://github.com/Mbed-TLS/mbedtls/security/policy`. Accessed: 17-12-2025 (cit. on p. 109).

[242] R. Van Dijck, M. Bognar, and J. Van Bulck. “Wait a Cycle: Eroding Cryptographic Trust in Low-End TEEs via Timing Side Channels”. In: *8th Workshop on System Software for Trusted Execution (SysTEX)*. 2025-06 (cit. on p. 115).

[243] F. Alder et al. “Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves”. In: *28th ACM Conference on Computer and Communications Security (CCS)*. 2021-11 (cit. on pp. 115, 116).

[244] T. Van Eyck et al. “Mr-TEE: Practical Trusted Execution of Mixed-Criticality Code”. In: *Proc. of International Middleware Conference: Industrial Track*. 2023 (cit. on pp. 115, 116).

[245] J. Wang et al. “RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone”. In: *Proc. of IEEE S&P*. 2022 (cit. on pp. 115, 116).

[246] M. Bognar and J. Van Bulck. “openIPE: An Extensible Memory Isolation Framework for Microcontrollers”. In: *10th IEEE European Symposium on Security and Privacy (EuroS&P)*. 2025 (cit. on p. 116).

[247] Alif Semiconductor. *ENSEMBLE*. Tech. rep. 2021 (cit. on pp. 122, 123).

[248] NXP Semiconductors. *MCX N Series Microcontrollers*. Tech. rep. 2022 (cit. on p. 122).

[249] STMicroelectronics. *STM32N6 Series*. Tech. rep. 2024 (cit. on pp. 122, 123).

[250] Renesas. *RA8P1: 1GHz Arm Cortex-M85 and Ethos-U55 NPU Based AI Microcontroller*. Tech. rep. 2024 (cit. on p. 122).

[251] Infineon Technologies. *PSOC Edge E8x*. Tech. rep. 2024 (cit. on pp. 122, 123).

[252] NXP. *i.MX RT700 Crossover MCU with Arm Cortex-M33, NPU, DSP and GPU Cores*. Tech. rep. 2024 (cit. on p. 122).

[253] Nuvoton Technology Corporation. *NuMicro M55M1 Series*. Tech. rep. 2024 (cit. on p. 122).

[254] Texas Instruments. *AM13x Sitara Processors*. Tech. rep. 2026 (cit. on p. 122).

[255] NXP. *i.MX 8ULP Applications Processor Family*. Tech. rep. 2021 (cit. on p. 123).

[256] Renesas Electronics. *RZ/G3S Group User’s Manual: Hardware*. Tech. rep. 2023 (cit. on p. 123).

[257] Texas Instruments. *AM62x Sitara Processors Technical Reference Manual*. Tech. rep. 2022 (cit. on p. 123).

[258] NXP. *i.MX 93 Applications Processor Family – Arm Cortex-A55, ML Acceleration, Power Efficient MPU*. Tech. rep. 2021 (cit. on p. 123).

[259] STMicroelectronics. *STM32MP25xx Reference Manual*. Tech. rep. 2024 (cit. on p. 123).