M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M

Cristiano Rodrigues[1], Marton Bognar[2], Sandro Pinto[1], and Jo Van Bulck[2]

[1]*Centro ALGORITMI, Universidade do Minho,* [2]*DistriNet, KU Leuven*

Abstract

Trusted Execution Environments (TEEs) have become a key technology for isolating sensitive *enclave* applications from untrusted operating systems. Extensive research on high-end platforms like Intel SGX and TDX, AMD SEV, and Arm TrustZone-A has exposed their limitations in terms of software-based side-channel analysis, amplified by specialized *single-stepping* attack frameworks that exploit privileged timer interrupts to execute enclaves one instruction at a time. Meanwhile, TEEs are increasingly deployed on resource-constrained IoT devices, with Arm TrustZone-M emerging as a leading solution, which, however, remains largely unexplored for high-resolution, software-based side channels.

This paper presents M-Step, an open and extensible single-stepping attack framework for TrustZone-M. While Cortex-M microcontrollers feature precise timers and deterministic behavior, achieving precise, instruction-level stepping remains challenging due to *(i)* the absence of virtual memory and page tables used in high-end frameworks; and *(ii)* Cortex-M's unique interrupt behavior, where certain multi-cycle instructions are abandoned or paused to reduce latency. To overcome these challenges, we extensively profile interrupt handling CPU behavior and develop a novel approach that uses previously dismissed interrupt-latency leakage to dynamically adjust the timer interrupts. We demonstrate M-Step's improved resolution and practicality by discovering previously unknown vulnerabilities in the latest Arm Mbed TLS library that enable single-trace, deterministic attacks recovering full RSA keys from a TrustZone enclave.

1 Introduction

Trusted execution environments (TEEs) provide strong confidential computing guarantees for hardware-enforced *enclaves* that remain isolated even from a compromised operating system. Widely adopted TEEs such as Intel SGX/TDX, AMD SEV, and Arm TrustZone-A have been deployed across desktops, servers, and mobile platforms to safeguard cryptographic keys, protect user data, and ensure confidentiality and integrity for sensitive workloads. However, despite their strong architectural isolation, TEEs remain vulnerable to software-based microarchitectural side-channel attacks. Over the past decade, a wide range of microarchitectural leakage sources have been uncovered that can leak secret-dependent code or data accesses at varying spatial resolutions, ranging from 4 KiB page-level granularity [15,52,65], to 64-byte cache lines [32,50,66], and even sub-cacheline precision [4, 33, 37, 57].

Complementing this spatial dimension, an orthogonal line of research has focused on *temporal resolution*, investigating how frequently an attacker can sample these side-channel leakage sources. Particularly, leveraging the privileged TEE adversary's control over the untrusted operating system, researchers have demonstrated that it is possible to achieve extremely high-resolution, instruction-level observations by precisely triggering timer interrupts after every executed instruction within the TEE. The seminal SGX-Step [56] framework pioneered this idea and has since been maintained as an active open-source project. Underscoring the impact of open research infrastructure, SGX-Step has to date enabled over 48 high-resolution side-channel attacks on SGX enclaves, many of which were published at top-tier security venues [53, 55]. SGX-Step has also inspired the development of similar single-stepping frameworks for all major high-end TEEs, including SEV-Step [64] for AMD SEV, TDX-Step [27], TDX-down [63], and TDXploit [45] for Intel TDX, and Load-Step [31] and CacheGrab [47] for Arm TrustZone-A. These frameworks have not only enabled powerful new classes of attacks, but have also prompted both academic [16, 18, 51, 60] and production-level mitigations, ranging from extensions to Intel SGX's instruction set architecture (ISA) [19] to modified context-switching behavior in Intel TDX [27].

While much of the prior research has focused on high-end processors with virtual memory support, specialized TEEs are increasingly being deployed on low-power, resource-constrained microcontroller units (MCUs). In this domain, Arm TrustZone-M [43] has emerged as a leading solution for securing embedded applications on the popular Cortex-M

microcontroller family. While TrustZone-M shares its name and dual-world design with the better-known TrustZone-A for Cortex-A processors, its implementation differs fundamentally, being optimized for faster context switching and low-power MCUs [43]. However, despite its growing adoption, side-channel analysis on TrustZone-M remains largely unexplored. While MCUs offer cycle-accurate timers and are highly deterministic—which may facilitate high-resolution and low-noise side-channel analysis—techniques developed for SGX or TrustZone-A do not directly apply to Cortex-M platforms lacking multi-core, virtual memory, and microcode-assist features instrumental for single-stepping frameworks like SGX-Step [19]. More crucially, Cortex-M's interrupt handling mechanism is highly optimized for real-time responsiveness: multi-cycle instructions may be aborted or paused to minimize interrupt latency, making deterministic instruction-level single-stepping a non-trivial challenge.

In this paper, we present *M-Step*, the first single-stepping framework for Arm TrustZone-M. M-Step enables high-resolution, software-based side-channel analysis by leveraging precise control over hardware timers and dynamically adapting to the unique timing behavior of Cortex-M instruction execution. To achieve this, we extensively profile microarchitectural interrupt and pipeline mechanisms, uncovering detailed instruction-level timing characteristics and interrupt edge cases. We then develop a novel methodology to dynamically reconfigure timer intervals in the non-secure world, allowing reliable interrupts at every instruction boundary in the secure world. Underscoring M-Step's practicality and extensibility, we implement debugging and visualization analysis infrastructure along with multiple *plugins* that capture side-channel information, including instruction cache activity, bus contention [46], and interrupt latency [57]—debunking prior claims that Cortex-M platforms are "immune" to interrupt-latency attacks [57]. We demonstrate M-Step's enhanced temporal resolution in real-world attack scenarios, uncovering previously unknown vulnerabilities in the latest version of Arm's widely used Mbed TLS library, as integrated in the Trusted Firmware-M (TF-M) project. Notably, M-Step deterministically extracts full RSA keys from a single trace of Mbed TLS's complex binary extended Euclidean algorithm (BEEA) implementation by accurately probing multiple secret-dependent branches in close succession. More broadly, our results show that high-resolution, software-based side-channel analysis extends beyond high-end platforms and is also feasible on microcontroller-class TEEs, underscoring the need for stronger protections and rigorous code vetting for constant-time behavior. Following our responsible disclosure (cf. §9), Arm issued software patches for Mbed TLS v3.6.5, tracked under CVE-2025-54764.

Contributions. In summary, our main contributions are:

- We present *M-Step*, the first single-stepping side-channel analysis framework for Arm TrustZone-M.
- We profile interrupt handling instruction behavior on our Cortex-M test platform to inform a novel dynamic timer interval configuration algorithm.
- We implement M-Step side-channel plugins for instruction cache activity, bus contention, and interrupt latency.
- We evaluate M-Step through principled microbenchmarks and end-to-end high-resolution attacks, including full key recovery from the latest version of Mbed TLS.

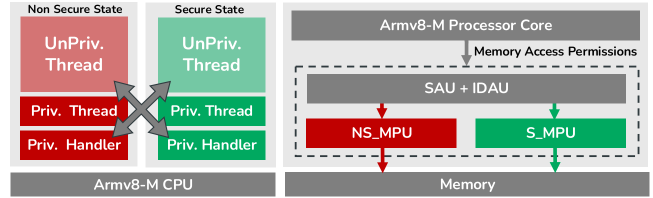


Figure 1: Armv8-M modes and security states (left) and Armv8-M memory protection controllers (right).

M-Step is released as an extensible open-source framework at https://github.com/M-Step-Framework, along with our case-study attacks and evaluation code.

Outline. §2 introduces the necessary background, and §3 defines the threat model and single-stepping objectives. Next, §4 introduces the Cortex-M profiling results and timer configuration algorithm, §5 describes the M-Step framework architecture and plugins, and §6 provides experimental evaluation results. Finally, we discuss mitigation approaches in §7, reflect on limitations in §8, and conclude in §9.

2 Background and Related Work

We first provide background on the architecture of the Arm Cortex-M family, before reviewing the state-of-the-art on software-based side-channel attacks on low-end MCUs.

2.1 Arm Cortex-M Architecture

Complementary to the line of high-performance Cortex-A processors widely deployed in mobile devices, Arm also offers the specialized Cortex-M family, designed for low-end MCUs. Originally introduced for use cases demanding high energy efficiency and deterministic low-latency interrupts, they have been widely deployed in embedded systems. These MCUs are typically rich in peripherals, such as SPI buses, timers or direct memory access (DMA) devices, yet Cortex-M does not include virtual memory support, required to run complex operating systems like Linux. As of 2026, the Arm Cortex-M family comprises ten MCUs [34], spanning three architectural generations: Armv6-M, Armv7-M, and Armv8-M.

TrustZone-M. In the latest generation (Armv8-M), Arm introduced support for TrustZone-M, as illustrated in Fig. 1. Al-

though 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 microcontrollers [43].

Armv8-M extends the earlier Armv6/7-M architecture with two new security states (worlds): Secure (S) and Non Secure (NS). 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). Armv8-M also defines a new memory region called the Non-Secure Callable (NSC) region, which serves as the gateway through which NS code can invoke S-world services. This region is accessible to both worlds, but the NS world can only execute the Secure Gate (SG) instruction within it. Executing SG transitions the processor to the S world and establishes a secure entry point. Any attempt to enter the S world via other means triggers a security fault. In contrast to TrustZone-A, TrustZone-M performs all world transitions entirely in hardware, without any software-based secure monitor in the control path. Similarly, when S-world execution is interrupted by an NS-world interrupt, the processor transitions directly from the secure context to the non-secure interrupt service routine (ISR).

Interrupt Handling. To minimize interrupt latency, Cortex-M implements *automatic stacking*, a hardware mechanism that saves and restores the CPU context on interrupt entry and exit. The stack frame size varies from 8 to 34 registers depending on the execution context. There are four stack frame types, determined by the floating-point unit (FPU) state (enabled/disabled) and the security state (S/NS) of the interrupted context. During cross-world interrupts, the CPU saves the entire integer register file to prevent data leakage. If the FPU is enabled, the floating-point register file is also saved either unconditionally or, if lazy stacking is enabled, only when the ISR executes FPU instructions.

Interruptible Instructions. Cortex-M introduces a microarchitectural feature that allows multi-cycle load and store instructions to be interrupted mid-execution, further reducing interrupt latency. Arm refers to these as Interrupt-Continuable Instructions (ICIs) [35, 36]. ICI instructions exhibit two possible behaviors after an ISR completes: they either resume execution from the point of interruption or restart execution from scratch. Arm refers to all other non-interruptible instructions simply as “instructions”.

2.2 Software-based Side Channels on MCUs

While low-end MCUs have long been the focus of physical side-channel analysis such as electromagnetic leakage, their vulnerability to software-based side channels has only recently gained attention, largely in parallel with the emergence of low-end TEEs for these devices. Compared to the extensive body of work on software-based microarchitectural attacks targeting high-end processors [24], this area remains relatively underexplored. However, there is growing evidence that these type of attacks also pose a non-negligible threat to the security of low-end MCUs [10, 11, 13, 46, 48, 57].

Table 1: Overview of software-based side-channel attacks on Arm Cortex-M, compared by target and leakage source.

			Target			Leakage				
Attack	**Yr**	**MCU**	TZ	RW	ST	Pwr	IRQ	Bus	Cache	Div
Leaky Noise [25]	’19	Cortex-M4	○	●	○	●	○	○	○	○
O’Flynn et al. [41]	’19	Cortex-M23	●	●	○	●	○	○	○	○
Barenghi et al. [9]	’21	Cortex-M4/7	○	◐	○	●	○	○	○	○
BUSted [46]	’24	Cortex-M23/33	●	○	●	○	○	●	○	○
BUSted 2nd stop [8]	’24	Cortex-M4/33	○	◐	○	○	○	●	○	○
KyberSlash [10]	’25	Cortex-M4	○	●	○	○	○	○	○	●
M-Step (this paper)	’25	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.

Table 1 summarizes prior work on software-based side-channel attacks targeting Arm Cortex-M microcontrollers, highlighting that this paper presents the first demonstration of single-stepping and interrupt-latency attacks on Cortex-M, as well as the first single-trace side-channel attack capable of extracting real-world cryptographic keys from TrustZone-M. A first line of work targets onboard analog-to-digital converters present in some Arm Cortex-M devices to recover AES keys via software-based differential power analysis [25, 41]. Barenghi et al. [9] introduced a hybrid approach that combines microarchitectural timing channels to build a leakage model of Cortex-M4 and Cortex-M7, which is then used to perform a power side-channel attack. However, power analysis attacks require numerous traces and specialized interfaces.

A complementary line of work targets subtle timing differences in non-constant-time code. While classical end-to-end timing attacks are well known—also on microcontrollers [26]—Nemesis [57] was the first to show instruction-level microarchitectural leakage from variable interrupt request (IRQ) latencies, originally on a 16-bit openMSP430 softcore and later on real TI MSP430 MCUs [11, 48]. Notably, due to their unique interruptible instructions, Cortex-M devices were considered “immune to IRQ latency timing attacks” [57]. A DMA-based microarchitectural timing channel was first shown on openMSP430 [13, 14], and later realized as a generic bus contention attack in BUSted [46] on Armv8-M. While initial attack demonstrations targeted simplified proof-of-concept scenarios, such as a dummy secure keypad implementation [46, 57], recent work [8, 10] provides preliminary evidence that low-end MCUs can leak sufficient microarchitectural information to compromise cryptographic implementations. Concretely, KyberSlash [10] demonstrated full key recovery on the Kyber post-quantum algorithm by exploiting

secret-dependent timing in integer division and multiplication instructions. This attack, however, was only demonstrated on a Cortex-M4 (Armv7-M) without TrustZone-M, and with victim and attacker sharing privilege and security domains.

3 Problem Statement

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

3.1 Threat Model and Scope

We target small embedded devices powered by state-of-the-art Armv8-M MCUs. Our threat model assumes a software-only adversary whose primary goal is to leak sensitive information from a victim application protected by TrustZone-M through the indirect observation of microarchitectural behavior. Following the standard TrustZone threat model [43], we assume the attacker has full control over the normal world and its resources. In particular, the attacker can configure and use timer peripherals to preempt secure-world execution.

We assume the secure world trusted computing base (TCB) is correct and free of architectural software vulnerabilities [17, 54], but it is assumed to exhibit subtle non-constant-time behavior that leads to secret-dependent control-flow variations. However, the attacker may only get a single opportunity to extract the secret, for example during key generation [38], meaning the attack must succeed using a single trace and cannot rely on accumulating noisy side-channel information across multiple runs. Note that writing general-purpose applications in constant time is notoriously challenging and error-prone. As a case in point, we will show in §6.3 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.

3.2 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 2 summarizes existing single-stepping frameworks, highlighting a growing trend in recent years toward developing such techniques for both high-end TEEs with virtual memory and low-end microcontrollers.

High-End TEEs. SGX-Step [56] 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

Table 2: 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 [56]	'17	Intel SGX	●	●	●	✓ 0-1	◐ [19]	●	●
	CacheGrab [47]	'19	Arm TrustZone-A	●	●	●	✗ >1	○	●	◐
	Load-Step [30]	'21	Arm TrustZone-A	●	●	●	✗ >1	○	○	○
	SEV-Step [64]	'23	AMD SEV-SNP	●	●	●	✓ 0-1	○	●	●
	TDX-Step [27]	'23	Intel TDX	●	●	●	~ ?	● [27]	○	○
	TDXdown [63]	'24	Intel TDX	●	●	●	~ 1/>1	◐ [28]	●	○
	TDXploit [45]	'25	Intel TDX	●	●	●	✓ 1	● [29]	●	○
Low-End	Sancus-Step [20]	'19	openMSP430	●	●	○	✓ 1	◐ [16]	●	◐
	Taking a look [49]	'19	Armv7-M XOM	◐[1]	○	●	~ 1/>1	● [43]	○	○
	RIPencapsulation [48]	'24	TI MSP430/2 IPE	◐[1]	○	●	✓ 1	○	●	○
	M-Step (this paper)	'25	Arm TrustZone-M	●	●	●	✓ 0-1	○	●	●

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

for mounting high-resolution side-channel attacks against SGX [55]. To date, it has been used in over 48 academic publications [53], 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 [19] 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 [19] 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 [64] for AMD SEV, as well as TDX-Step [1, 27] 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 [45, 63], prompting subsequent patches.

While there have been efforts to create single-stepping frameworks for Arm TrustZone-A [30, 47], 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 [47] and Load-Step [30] 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 [20, 48, 49]. Sancus-Step [20] is an experimental unpublished framework for the niche academic Sancus [39] architecture. The other two frameworks [48, 49] 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 [48, 49] 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.

3.3 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.

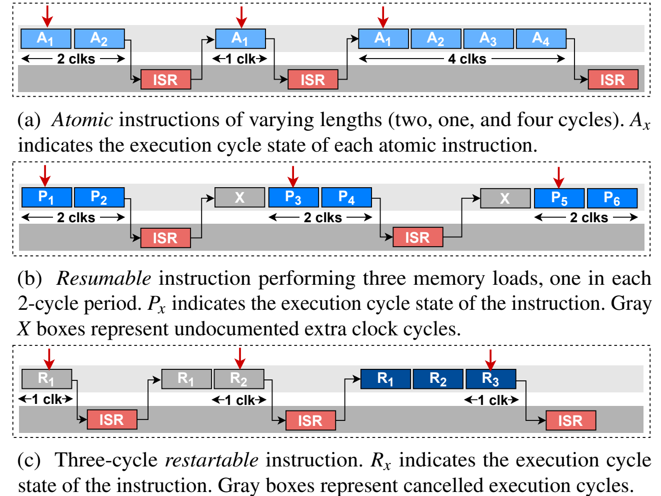
C1 - Heterogeneous instructions. In contrast to prior single-stepping frameworks on Intel, AMD, and Arm Cortex-A, a key challenge on Cortex-M is handling varying interrupt behavior depending on the instruction class, due to the interruptible nature of Arm Cortex-M instructions.

C2 - Observing victim progress. High-end single-stepping frameworks use page faults to advance victim execution and page-table "accessed" bits [19, 57, 64] or performance monitoring counters [21, 64] 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 operating systems 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.



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

4 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 instruction cache (ICache), TrustZone-M, and FPU with lazy stacking enabled.

4.1 Profiling Instruction IRQ Behavior (C1)

Recall from §2.1 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. 2)

[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.

Table 3: Interrupt latencies for memory loads targeting different memory partitions and stack pointer (SP) configurations.

Inst.	Operand	Mem. Access	IRQ Latency	
			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

by systematically triggering interrupts at different clock cycles (red arrows), measuring the resulting IRQ latency, and recording whether the instruction completes, i.e., the program counter (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. 2a). 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 3 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 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. 2b). 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.

Restartable Instructions (Fig. 2c). 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 [36]. 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.

4.2 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 (§4.2.1, C2), followed by a dynamic timer configuration algorithm (§4.2.2, 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.

4.2.1 Distinguishing Victim Progress (C2)

State-of-the-art zero-step detection relies on virtual memory [19, 57, 64], 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.

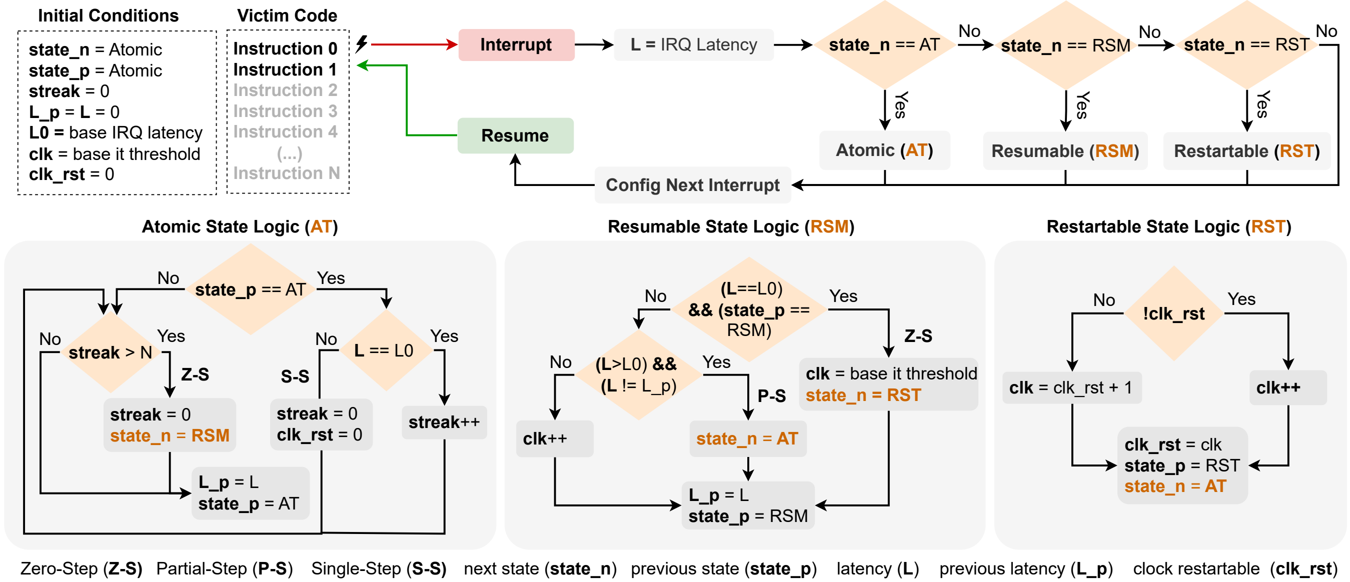


Figure 3: Simplified M-Step algorithm for dynamically adjusting the single-step timer interval based on the interrupt behavior of the currently executing instruction. On each interrupt, the algorithm updates its accumulated state (*state_n*) by combining the observed interrupt latency ($\mathcal{L}$) 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.

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 §4.1. 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. 2b) 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.

4.2.2 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. 3, 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. 2a). 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 §8, we explore possible techniques to avoid these redundant zero-steps if the attacker has partial knowledge of the victim's binary code.

5 M-Step Framework

In §4, 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.

5.1 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 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 §2.1), 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. §4.2.1). 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.

Table 4: 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 [57].	✓	✓
	Mstp-Cache	Reveals cache activity, PRIME+PROBE [40, 42].	✓	✓
	Mstp-BUSted	Reveals memory accesses, BUSted [46].	✓	✓
Arch	**Mstp-Zoom**	Amplifies interrupt-latency leakage §5.2.	✓	✓
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.	–	✓

5.2 Framework Plugins

The M-Step framework is built upon a set of modular plugins, listed in Table 4, 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 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. Appendix A).

Memory-Access Amplification (C4). As shown in Table 3, 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 [46] 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. 4. 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

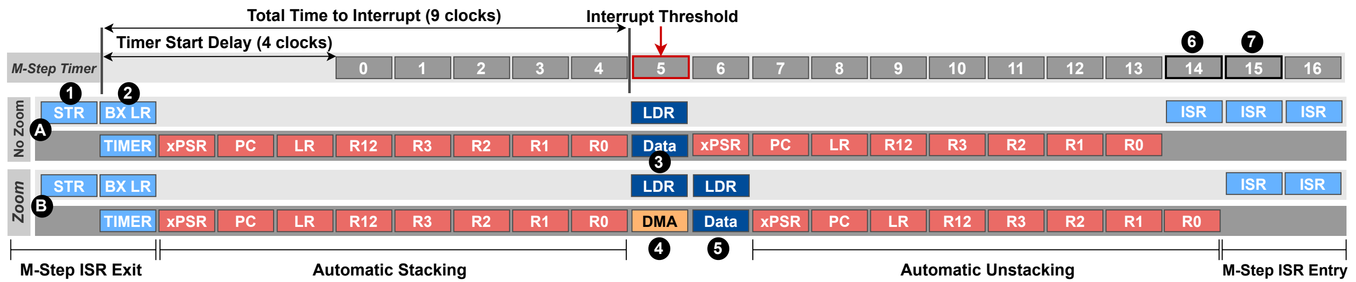


Figure 4: Reconstructed Cortex-M interrupt handling process during an M-Step single-step. The top row shows the timer values for the M-Step interrupt logic. The middle and bottom rows show two cycle-by-cycle views of the CPU pipeline (instructions and interrupt states) for the same LDR instruction, without (A) and with (B) BUSted gadget amplification.

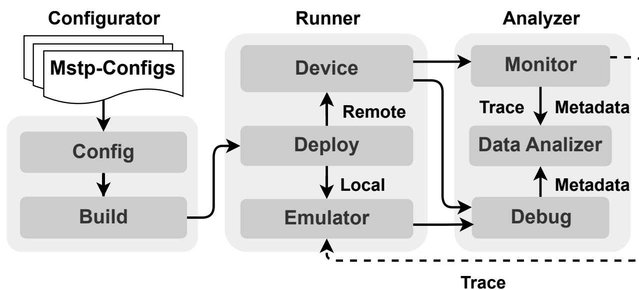


Figure 5: M-Step rapid prototyping framework architecture.

that it arrives in the first clock cycle of the victim instruction. When amplification is enabled (B), a BUSted gadget issues a concurrent DMA access (4) in the same cycle, creating contention on the targeted memory bank. This contention delays the victim instruction's data fetch (3), forcing the load to take an additional cycle (5) and thereby increasing the observed interrupt latency (6 vs. 7). 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 3) would appear indistinguishable from generic single-cycle instructions such as `ADD`. We use Mstp-Zoom in our end-to-end attacks (§6.3) to amplify the interrupt-latency leakage.

5.3 Framework Architecture (C5)

The architecture of the framework (cf. Fig. 5) 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, 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. We utilize this feature in §6 to accelerate exploit development.

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.

Table 5: 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**

6 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 demonstrates real-world, end-to-end RSA key extraction attacks on the latest version of Mbed TLS.

Attack Setup. For all experiments and attacks in this section, we follow the standard TrustZone threat model: an attacker in the NS 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 micro-TEE (uTEE) for Cortex-M. Our attacks target TF-M's default cryptographic library, Mbed TLS (v3.6.3.1).

6.1 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 [55, 64], 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 5 presents the experimental results, including the distribution of step types and the occurrence of instruction folding. Instruction folding is a documented ISA feature in Armv8-M [36], 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 [38].

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. §8). 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.

6.2 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.

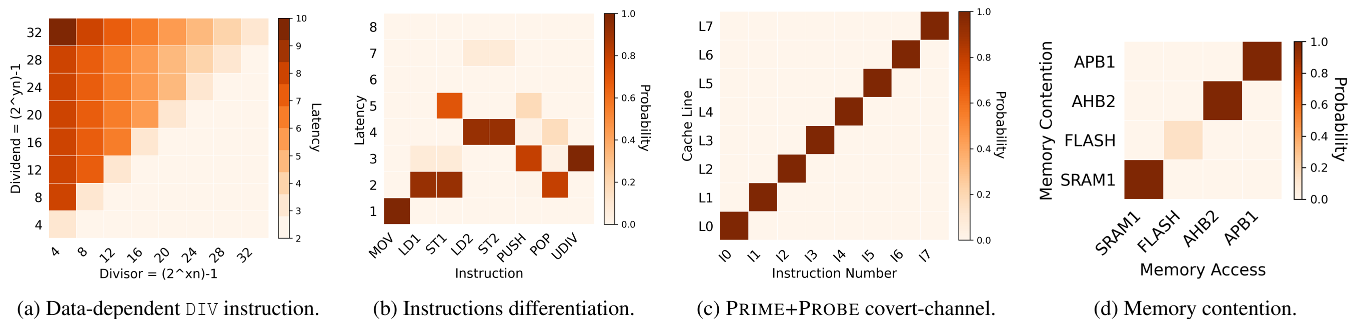
Data-Dependent Divisions. Integer division operations are known to exhibit non-constant-time, operand-dependent timing behavior on both Intel x86 [57, 64] and Arm Cortex-M platforms [10]. We use the Mstp-Nemesis plugin to measure interrupt latency for integer division instructions, sweeping over operand pairs where the dividend ranges from `0x0000000F` to `0xFFFFFFFF` in steps of `0xF`.

The results, shown in Fig. 6a, 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 [10], 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. 6b. 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 periph-



(a) Data-dependent DIV instruction. (b) Instructions differentiation. (c) PRIME+PROBE covert-channel. (d) Memory contention.

Figure 6: 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.

eral regions (LD2/ST2). All measurements in Fig. 6b 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 a 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. 6c 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 [46] channels by delaying memory accesses via targeted contention in specific memory partitions (cf. §5.2). 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. 6d, 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.

6.3 End-to-end Attacks on Mbed TLS

Single-stepping frameworks enable high-resolution, single-trace attacks extracting fine-grained microarchitectural information with minimal noise. M-Step brings such attack capabilities to MCUs, exposing secret-dependent control flow, as subtle as a single instruction. As a real-world end-to-end attack, we leverage M-Step to target Mbed TLS's binary extended Euclidean algorithm (BEEA), which has been a common target for control-flow side-channel attacks [2, 3, 22, 23, 44, 62]. Particularly, we uncover subtle leakage in the latest release of Mbed TLS (v3.6.3.1)[3]. This is very non-trivial leakage 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.

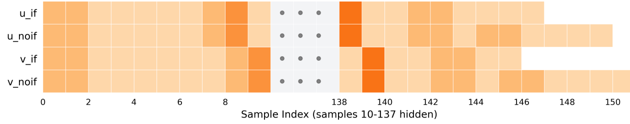
Parsing M-Step Traces. M-Step traces span the entire victim execution, providing one interrupt latency sample per victim instruction. To locate the code of interest within a trace, we exploit the deterministic behavior of MCUs, where a specific code path consistently yields the same trace pattern. M-Step produces identical, clock-cycle-granular traces for a given instruction stream, requiring no statistical analysis over multiple runs (cf. Fig. 7). 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.

6.3.1 Mbed TLS Vulnerabilities

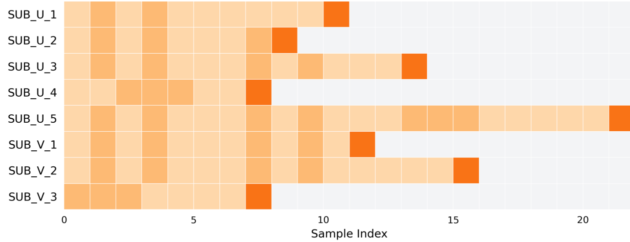
We found two issues in Mbed TLS (v3.6.3.1), which can be exploited using vulnerabilities found in the implementation of the BEEA algorithm. First, we explain the behavior of the BEEA algorithm, then the specific issues found in Mbed TLS.

[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] Mbed TLS guarantees protection against "publicly documented attack techniques", placing local timing side channels explicitly in scope [7]. Upon responsible disclosure, Arm acknowledged our findings and issued a patch and a CVE (cf. §9), confirming that our attack falls within the intended threat model and highlighting its practical relevance.



(a) Shift template matrix. Full patterns are shown in Appendix B.



(b) Subtraction template matrix.

Figure 7: 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).

Mbed TLS BEEA Algorithm. Mbed TLS uses BEEA to compute $X = U^{-1} \pmod{V}$ by iterating over the bits of U and V from right to left, performing non-constant-time conditional subtractions and right-shift operations (see Appendix B for more details). Since the algorithm's final state is known ($U = 0$ and $V = 1$)[4], 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). 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. 7a, 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 identify eight distinct execution paths: five corresponding to $U \geq V$, and 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. 7b. Compared to the shift patterns, these are shorter and exhibit more pronounced leakage characteristics.

[4] For RSA, V holds the greatest common divisor (GCD) of the inputs. Since they are coprime, the final value is $V = 1$.

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 7, Appendix B). 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 Microelectronics (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 [3, 23, 44], 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. 10 in Appendix B). 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. 10). While not as severe as Issue #1, this is still clearly undesirable non-constant-time behavior.

6.3.2 Case-Study Attacks

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 7, Appendix B) 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 6 in Appendix B) are affected by at least one of the vulnerabilities described above. This suggests that these issues likely widely affect real-world, closed-source applications.

7 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 §6, 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 [10, 58]. Therefore, similar to existing single-stepping mitigations deployed in Intel SGX [19] and TDX [27], 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.

7.1 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 [5, 59, 61].

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 trusted applications (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.

7.2 Secure Interrupt Sanitization

Inspired by prior work on real-time TEEs [5, 12, 59, 61], 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 [12, 16] or randomizing interrupt latency, prefetching [19] various microarchitectural states like the instruction cache, or simply limiting the number of closely successive interrupts [27].

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 (§2.1). To address this issue, we propose the introduction of a trampoline mechanism: 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 in NS-accessible memory (i.e., in NS-callable memory), redirecting all NS interrupts to entry points within the S firmware. Upon entry to the S world, the VTOR is updated to point to this trampoline table. For each NS interrupt, control is passed to the S-world firmware, which performs the necessary sanitization before branching to the actual NS handler. Once the S world exits, the original VTOR is restored.

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

8 M-Step Limitations and Discussion

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[5]. 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.

[5] 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).

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 [55] 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 3—the microarchitectural behavior of the three instruction types is architecturally defined by the underlying Arm architecture (Armv6-M, Armv7-M, or Armv8-M). Similarly, for SGX-Step [55], the microarchitectural behavior of Intel CPUs' interrupt logic remains consistent across CPUs, even though specific instruction timings might change. In this paper, 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 §6.3, 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 [14].

9 Conclusion

We introduced M-Step, the first practical framework for precise, interrupt-driven side-channel analysis on TrustZone-M. By overcoming unique Cortex-M microarchitectural challenges, M-Step enables instruction-level single-stepping and exposes new vulnerabilities in the latest Mbed TLS on TF-M, enabling single-trace, deterministic full key recovery. Our results, and the set of countermeasures we propose, underscore the need for improved side-channel protections on MCUs.

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, which is no longer vulnerable to the attacks described in this paper. TF-M subsequently released version v2.2.2 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 on the Mbed TLS website, acknowledging M-Step.

For the platform-specific issues (V#14–V#20 in Table 7), 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.

Ethical Considerations. Compliant with responsible-disclosure practices, we did not publicly disclose the vulnerabilities or release exploit code before appropriate patches became available. As requested by Arm, all details presented in this paper, as well as the associated artifacts, remained under embargo until October 2025, when the relevant fixes were released. During the embargo period, we engaged with Arm engineers to provide feedback on the proposed mitigation approach. All experiments were conducted exclusively on our own hardware in a controlled environment, ensuring no impact on third-party systems or users, including privacy violations or consent issues.

In the broader context of this research, we acknowledge that newly discovered vulnerabilities and attack techniques have the potential to harm users of the affected systems (causing privacy violations, monetary losses, and other damages), in addition to reputational damage to the vendors and the consequences thereof. However, we firmly believe that the benefits of improving the community's understanding and ability to explore and quantify side-channel leakage on real-world Cortex-M platforms outweigh any negative outcomes. A cooperative responsible disclosure process, as followed in this work, effectively mitigates these risks by ensuring that vendors can develop and distribute patches and notify their users before public disclosure of the vulnerabilities. Moreover, open-sourcing advanced research tools like M-Step has a demonstrably positive impact on the broader security community. This is evident from prior frameworks such as SGX-Step [55], which have been widely adopted and enabled follow-up research and concrete security improvements: SGX-Step has been used in over 48 publications to date [53], was recognized with an ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award [55], and ultimately led to the release of new ISA extensions by Intel [19], clearly illustrating the practical value and real-world impact of releasing high-quality, open-source single-stepping frameworks.

Open Science

In compliance with the USENIX Security open-science policy, we provide all artifacts required to reproduce and evaluate our results. This includes: *(i)* the full M-Step framework source code; *(ii)* scripts and configurations for experimental setup and data collection; *(iii)* raw and processed datasets used in our evaluation; and *(iv)* documentation for reproducing all experiments and figures in the paper. All artifacts are archived for long-term access on Zenodo at `https://doi.org/10.5281/zenodo.17910184`.

Additionally, we maintain M-Step as an open-source project at `https://github.com/M-Step-Framework` and are committed to the continuous improvement of its codebase and documentation to foster adoption and ensure long-term maintainability.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and constructive suggestions. Cristiano Rodrigues was supported by FCT grant 2020.08729.BD. This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/2025 - Centro ALGORITMI (ALGORITMI/UM), the European Union's Horizon Europe research and innovation program under grant agreement No 101070537, project CROSSCON (Cross-platform Open Security Stack for Connected Devices), the Research Fund KU Leuven, and the Cybersecurity Research Program Flanders.

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Secu-

rity Review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf, April 2023.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. Cache-timing attacks on rsa key generation. In *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019.

[3] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ecdsa and rsa. In *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *Proc. of S&P*, 2019.

[5] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *28th ACM Conference on Computer and Communications Security (CCS)*, November 2021.

[6] Antmicro. Renode. https://renode.io/. Accessed: 15-06-2025.

[7] Arm. Arm Mbed TLS Security Policy. https://github.com/Mbed-TLS/mbedtls/security/policy. Accessed: 17-12-2025.

[8] André Barbosa, Cristiano Rodrigues, Tiago Gomes, and Sandro Pinto. WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices. In *Proc. of IEEE EWSN*, 2024.

[9] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Exploring cortex-m microarchitectural side channel information leakage. In *Journ. IEEE Access*, 2021.

[10] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. In *Proc. of IACR TCHES*, 2025.

[11] Marton Bognar, Cas Magnus, Frank Piessens, and Jo Van Bulck. Intellectual property exposure: Subverting and securing Intellectual Property Encapsulation in Texas Instruments microcontrollers. In *Proc. of USENIX Sec.*, 2024.

[12] Marton Bognar and Jo Van Bulck. openIPE: An extensible memory isolation framework for microcontrollers. In *10th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2025.

[13] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *Proc. of S&P*, 2022.

[14] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. Microprofiler: Principled side-channel mitigation through microarchitectural profiling. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 651–670, 2023.

[15] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution. In *Proc. of USENIX Security*, 2017.

[16] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *Proc. of IEEE CSF*, 2020.

[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proc. of S&P*, 2020.

[18] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *12th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 7–18, 2017.

[19] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *32nd USENIX Security Symposium*, pages 4051–4068, August 2023.

[20] Sven Cuyt. A security analysis of interrupts in embedded enclaved execution. Master's thesis, KU Leuven, 2019. https://distrinet.cs.kuleuven.be/software/sancus/publications/cuyt19thesis.pdf.

[21] Nicolas Dutly, Friederike Groschupp, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. AEX-NStep: Probabilistic interrupt counting attacks on Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*, 2026.

[22] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *Proc. of USENIX Sec.*, 2017.

[23] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Certified side channels. In *Proc. of USENIX Sec.*, 2020.

[24] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. In *Journ. Cryptogr. Eng.*, 2018.

[25] Dennis R. E. Gnad, Jonas Krautter, and Mehdi B. Tahoori. Leaky noise: New side-channel attack vectors in mixed-signal iot devices. In *Journ. IACR Trans. on Crypto. Hard. and Embed. Syst.*, 2019.

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

[27] 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, April 2023.

[28] Intel. Security Advisory: TDXDown. https://www.intel.com/content/www/us/en/security-center/announcement/intel-security-announcement-2024-10-08-001.html, October 2024.

[29] Intel. 2025.3 IPU, Intel TDX Module Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01312.html, August 2025.

[30] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. 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)*, pages 979–984, 2021.

[31] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict. In *Proc. of DAC*, 2021.

[32] Ben Lapid and Avishai Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. Cryptology ePrint Archive, 2018.

[33] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. of USENIX Security*, 2017.

[34] Arm Ltd. Arm Cortex-M Processors. https://www.arm.com/products/silicon-ip-cpu?families=cortex-m&showall=true. Accessed: 30-07-2025.

[35] Arm Ltd. Arm®v7-M Architecture Reference Manual. Technical report, Feb 2021.

[36] Arm Ltd. Arm®v8-M Architecture Reference Manual. Technical report, Aug 2024.

[37] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. In *Journ. CoRR*, 2017.

[38] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *Proc. of USENIX Security*, 2020.

[39] J. Noorman, J. Van Bulck, J. Tobias Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–33, 2017.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proc. of CT-RSA*, 2006.

[41] Colin O'Flynn and Alex Dewar. On-device power analysis across hardware security domains.: Stop hitting yourself. In *Journ. IACR Trans. on Crypt. Hard. and Embed. Syst.*, 2019.

[42] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan*, 2005.

[43] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. In *Journ. ACM Computing Surveys*, 2019.

[44] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. Frontal attack: Leaking control-flow in sgx via the cpu frontend. In *Proc. of USENIX Sec.*, 2021.

[45] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. Tdxploit: Novel techniques for single-stepping and cache attacks on intel tdx. In *Proc. of USENIX Security*, 2025.

[46] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect. In *Proc. of IEEE S&P*, 2024.

[47] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *26th ACM Conference on Computer and Communications Security (CCS)*, pages 181–194, 2019.

[48] Prakhar Sah and Matthew Hicks. RIPencapsulation: Defeating IP encapsulation on TI MSP devices. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 117–132, 2024.

[49] Marc Schink and Johannes Obermaier. Taking a Look into Execute-Only Memory. In *Proc. of USENIX WOOT*, 2019.

[50] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *14th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[51] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, February 2017.

[52] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 317–328, 2016.

[53] Jo Van Bulck. A Practical Attack Framework for Precise Enclave Execution Control. https://github.com/jovanbulck/sgx-step, 2025.

[54] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proc. of CCS*, 2019.

[55] Jo Van Bulck and Frank 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*, December 2023.

[56] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *2nd Workshop on System Software for Trusted Execution (SysTEX)*, pages 4:1–4:6. ACM, October 2017.

[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *25th ACM Conference on Computer and Communications Security (CCS)*, pages 178–195, October 2018.

[58] Ruben Van Dijck, Marton Bognar, and Jo 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)*, June 2025.

[59] Tom Van Eyck, Hamdi Trimech, Sam Michiels, Danny Hughes, Majid Salehi, Hassan Janjuaa, and Thanh-Liem Ta. Mr-tee: Practical trusted execution of mixed-criticality code. In *Proc. of International Middleware Conference: Industrial Track*, 2023.

[60] Daan Vanoverloop, Andres Sanchez, Flavio Toffalini, Frank Piessens, Mathias Payer, and Jo Van Bulck. TLBlur: Compiler-assisted automated hardening against controlled channels on off-the-shelf Intel SGX platforms. In *34th USENIX Security Symposium (USENIX Security 25)*, August 2025.

[61] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *Proc. of IEEE S&P*, 2022.

[62] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against rsa key generation in intel sgx ssl. In *Proc. of AsiaCCS*, 2018.

[63] Luca Wilke, Florian Sieck, and Thomas 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.

[64] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A single-stepping framework for AMD-SEV. *TCHES*, December 2023.

[65] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. of S&P*, 2015.

[66] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Yiwei Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. In *IACR Cryptol. ePrint Arch.*, 2016.

A M-Step Framework

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 §5, while the latter allows the extraction of interrupt-latency traces [57].

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 4. The plugins are interoperable: for instance, the side-channel plugins will generate microarchitectural information which can be accessed on the visualization tool and

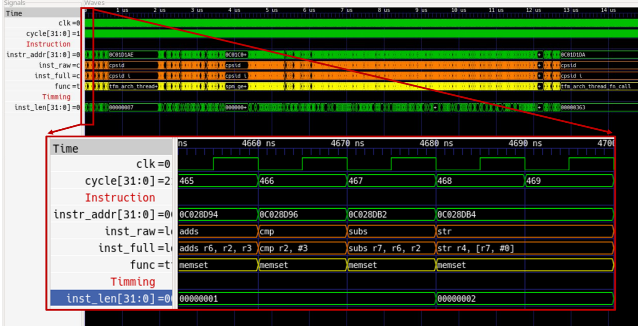


Figure 8: Example of the Mstp-Visualizer depicting the PoC trace of the standard `printf` function from `stdio.h`.

Table 6: 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
rsa_decrypt.c	*mbedtls_rsa_complete*	●	○
rsa_genkey.c	*mbedtls_rsa_gen_key*	●	○
rsa_genkey.c	*mbedtls_rsa_export_crt*	○	●
rsa_sign.c	*mbedtls_rsa_complete*	●	○
key_app.c	*mbedtls_rsa_export_crt*	○	●
key_app_writer.c	*mbedtls_rsa_export_crt*	○	●
dh_server.c	*mbedtls_rsa_complete*	●	○

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. 8.

A.1 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. 8.

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 (§4).

Target Hardware Emulation. The Mstp-Emulator plugin uses Renode [6] 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 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.

B Evaluation

Mbed TLS operates as a TF-M secure service at the highest privilege level. Although not directly exposed, an attacker in the NS world can interact with Mbed TLS through TF-M's cryptographic APIs. The attacker can trigger the vulnerable paths, detailed in Table 7, either directly by requesting a cryptographic service from TF-M or indirectly by invoking a TA that uses these services.

B.1 BEEA Algorithm

Mbed TLS uses the BEEA algorithm, shown in Algorithm 1, 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 $X = A^{-1} \pmod B$ 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 $U = 0$. 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.

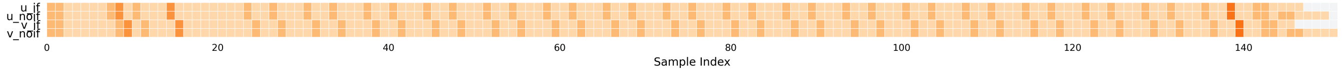


Figure 9: The shift template matrix for the U- and V-loops.

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

---

1. **procedure** MBEDTLS_BEEA(X, A, B)
2. $\quad TU \leftarrow A$, $TV \leftarrow B$;
3. $\quad U1 \leftarrow 1$, $U2 \leftarrow 0$, $V1 \leftarrow 0$, $V2 \leftarrow 1$;
4. $\quad$ **while** $TU \neq 0$ **do**
5. $\quad\quad$ **while** TU *even* **do**
6. $\quad\quad\quad TU \leftarrow TU/2$;
7. $\quad\quad\quad$ **if** $U1$ *or* $U2$ *odd* **then**
8. $\quad\quad\quad\quad U1 \leftarrow U1 + TB$; $U2 \leftarrow U2 - TA$;
9. $\quad\quad\quad U1 \leftarrow U1/2$; $U2 \leftarrow U2/2$;
10. $\quad\quad$ **while** TV *even* **do**
11. $\quad\quad\quad TV \leftarrow TV/2$;
12. $\quad\quad\quad$ **if** $V1$ *or* $V2$ *odd* **then**
13. $\quad\quad\quad\quad V1 \leftarrow V1 + TB$; $V2 \leftarrow V2 - TA$;
14. $\quad\quad\quad V1 \leftarrow V1/2$; $V2 \leftarrow V2/2$;
15. $\quad\quad$ **if** $TU \geq TV$ **then**
16. $\quad\quad\quad TU \leftarrow TU - TV$;
17. $\quad\quad\quad U1 \leftarrow U1 - V1$; $U2 \leftarrow U2 - V2$;
18. $\quad\quad\quad TU \leftarrow TU/2$;
19. $\quad\quad$ **else**
20. $\quad\quad\quad TV \leftarrow TV - TU$;
21. $\quad\quad\quad V1 \leftarrow V1 - U1$; $V2 \leftarrow V2 - U2$;
22. $\quad\quad\quad TV \leftarrow TV/2$;

B.1.1 Issue 2: Non-Constant-Time GCD.

We compiled the code on Fig. 10 for several versions of GCC and found that GCC version < 10 inserts the version which leaks the full number off leading zero bits.

```c
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 10: 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.

Table 7: 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($D,E,$**L**)	$D = E^{-1}$ mod **L**	●	●	●	●	●	code
	V#2	mb_rsa_gen_key	mb_rsa_deduce_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#3	mb_rsa_complete	mb_rsa_deduce_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	○	●	code
	V#4	mb_rsa_complete	mb_rsa_deduce_private_exponent → mb_mpi_inv_mod($D,E,$**K**)	$D = E^{-1}$ mod **K**	○	○	●	●	●	code
	V#5	mb_pk_psa_rsa_sign_ext	mb_rsa_write_key → mb_rsa_export_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#6	rsa_decrypt_wrap	mb_rsa_write_key → mb_rsa_export_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#7	import_pair_into_psa	mb_rsa_write_key → mb_rsa_export_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#8	pk_write_rsa_der	mb_rsa_write_key → mb_rsa_export_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#9	mb_rsa_export_crt	mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
	V#10	mb_psa_rsa_export_key	mb_pk_write_key_der → mb_rsa_export_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	○	●	●	●	code
Mod. Inv. TF-M	V#14	mb_rsa_complete	mb_rsa_deduce_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#15	mb_rsa_parse_key	mb_rsa_complete → mb_rsa_deduce_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
	V#16	mb_psa_rsa_load_representation	mb_rsa_parse_key → mb_rsa_complete → mb_rsa_deduce_crt → mb_mpi_inv_mod($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	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($QP,$**Q**,**P**)	$D = E^{-1}$ mod $(\mathbf{P}-1)(\mathbf{Q}-1)$	○	●	○	●	●	code
GCD	V#21	rsa_gen_key	mb_mpi_gcd($D,E,$**H**)	$D = E^{-1}$ mod **H**	●	●	●	●	●	code
	V#22	rsa_gen_key	mb_mpi_gcd($D,$**P**,**Q**)	**P**, **Q** partial leak following [3]	●	●	●	●	●	code
	V#23	mb_mpi_inv_mod	mb_mpi_gcd($D,$**P**,**Q**)	**P**, **Q** partial leak following [3]	●	●	●	●	●	code
	V#24	mb_rsa_deduce_primes	mb_mpi_gcd($D,$**K**,N)	$P = \mathbf{K}/N$	●	●	●	●	●	code