**Securing TrustZone: Symbolic Execution for Side-Channel Analysis in ARM-M Binaries**

**Sepideh Pouyanrad, Fritz Alder, Jan Tobias Mühlberg**

**ABSTRACT**

**Keywords:** IoT systems, Timing side-channel attacks, Binary code analysis, Symbolic execution, resource-constrained devices.

**1- INTRODUCTION**

With the rapid proliferation of Internet of Things (IoT) devices in various domains, such as smart homes, healthcare, transportation, and industrial systems, ensuring the security of these interconnected devices has become an utmost concern. IoT systems, consisting of embedded devices and networked components, handle an abundance of sensitive data, making them prime targets for malicious actors seeking to exploit vulnerabilities [1]. Among the multitude of security threats, timing side-channel attacks have emerged as a significant and pervasive challenge, leveraging timing variations to exploit vulnerabilities and compromise the confidentiality and integrity of sensitive data [2, 3, 4].

ARM family processors have emerged as the dominant choice for embedded devices, capturing a substantial market share of over 60% [5]. To enhance security, ARM has incorporated TrustZone [6, 7], a hardware-based security feature, into their processors. TrustZone ensures the isolation of security-critical software and data from the rest of the system, enabling secure execution of critical tasks and protection of sensitive information. It achieves this by dividing the processor into two separate and concurrent security realms or worlds: the 'Normal World' and the 'Secure World.' These worlds operate independently of each other, possessing distinct memory spaces and execution environments.

Here, developers often rely on the presumption that secrets are protected within the secure world due to the processor's isolation guarantees. However, extensive research [30, 32, 7, 8, 9, 10, 11, 12] has revealed the potential vulnerability of the TrustZone secure world to side channel attacks, which can lead to the unintended disclosure of fine-grained secrets. For instance, TruSpy [9] exploits the cache contention between normal world and secure world to implement a timing-based cache side-channel attack and then extract a full 128-bit AES encryption key stored in the trusted environment. The research demonstrated that while the contents of the processor cache are safeguarded by the hardware isolation, the access pattern to these cache lines remains unprotected. Consequently, TrustZone becomes susceptible to cache side-channel attacks, compromising its security measures. Similarly, in [11], researchers targeted Arm TrustZone in a malicious OS scenario. They leveraged the OS's capabilities to invoke interrupts and utilized the Prime+Probe technique to recover a 256-bit private key from Qualcomm's ECDSA algorithm.

Timing side-channel attacks exploit the unintentional leakage of timing information, such as execution time, cache behavior, or branch prediction, to infer sensitive data and breach system security. Early detection of side channel attacks enables proactive mitigation measures to be implemented. Over the years, researchers and practitioners have proposed various approaches to analyze binary code, or source code employing techniques such as symbolic execution [13, 14], type systems [15, 16, 17], and machine learning [18], among others [2]. These approaches aim to identify and mitigate timing side channel vulnerabilities targeting different architectures. However, each approach carries its own limitations and strengths, necessitating a thorough exploration of the existing body of work in this field (refer to Section 3).

In this paper, we present an innovative automated approach utilizing symbolic analysis techniques for the static verification of binaries targeting the ARM Cortex-M23 microcontroller. Our objective is to ensure the absence of timing side channel attacks, interrupt-latency attacks (such as Nemesis [3]), and detect any explicit and implicit information flow, which is roughly equivalent to the concept of storage channels in later literature [19]. This is particularly relevant in the context of applications that are compartmentalized into a security critical application part (such as managing and using cryptographic credentials) and a less critical part (such as sending and receiving network packets) to make use of the ARM TrustZone Trusted Execution Environment (TEE).

The TrustZone technology employed in Armv8-M processors (such as Cortex-M23/ M33/ M35P/ M55 / M85), do not claim to protect against side channel attacks due to secret-dependent control flow with measurable timing differences or secret-dependent memory access patterns [20]. Additionally, it is important to note that TrustZone may not effectively prevent secret leakage stemming from program implementation flaws, which can arise from weaknesses in protocols or algorithms, as well as mistakes made by developers. As an example, let's consider an One-Time Password (OTP) system implemented within the TrustZone environment [22]. In a secure and well-implemented OTP system, once an OTP is utilized, it should immediately become invalid and should not be stored in any accessible location. However, if the OTPs are stored in an insecure manner, such as being logged or stored in plaintext on unprotected memory or external I/O, an unauthorized attacker who gains access to the system or the logs could retrieve the previously used OTPs.

Considering the vulnerabilities discussed above, it is crucial for developers and system designers to implement strong security measures, including secure storage and proper handling of sensitive data, robust encryption algorithms, and appropriate access controls, to complement the security features provided by TrustZone. In this paper, we leverage symbolic execution [21], a widely recognized program analysis technique that computes program behaviors using mathematical constraints based on symbolic inputs, to effectively detect potential data leakages, and provide valuable insights to programmers, enabling them to address and rectify any security vulnerabilities.

This paper introduces SCFARM, an innovative automatic tool named after [17], specifically designed for static verification of ARMv8-M binaries. The primary objective of SCFARM is to track and monitor the flow of secret information between the TrustZone's secure world and the non-secure world, detecting and reporting any potential information leakages. To the best of our knowledge, this tool represents the first of its kind in performing static analysis on ARMv8-M binaries, addressing both timing and storage channels. To validate the effectiveness of our tool, we conduct comprehensive experiments on a collection of 'x' cryptographic libraries employed in ARM Cortex-M23. Through these experiments, we demonstrate the robust capabilities of SCFARM in effectively detecting vulnerabilities and ensuring the absence of any potential information leakage.

***Contributions.*** Our contributions can be summarized as follows:

* *Innovative Approach for Binary-level Information Flow Analysis:* Our research introduces a groundbreaking approach that leverages the capabilities of symbolic execution techniques to perform information flow analysis at the binary level. Specifically designed for applications compartmentalized for ARM TrustZone, a widely adopted security feature found in commercial microcontrollers and mobile devices, aimed at protecting valuable and confidential data.
* *Development of SCFARM Tool:* To automate the process of checking ARMv8-M binaries and identifying potential information leakages, we have implemented our novel approach in a software tool called SCFARM. Written in Python, SCFARM efficiently carries out the analysis and provides detailed reports on identified vulnerabilities. We have made both SCFARM and our benchmark datasets publicly available on the GitHub repository at [https://github.com/sepidehpouyan/’x’](https://github.com/sepidehpouyan/%E2%80%99x%E2%80%99" \t "_new).
* *Detection of Various Security Threats:* We have successfully integrated static analysis techniques into SCFARM, enabling the detection of timing side channel attacks, Nemesis [3] attacks, and undesired direct and indirect information flow to accessible and unprotected locations.
* *Evaluation of SCFARM's Accuracy and Scalability:* To assess the effectiveness and scalability of SCFARM, we conducted a rigorous evaluation by applying it to a set of cryptographic libraries targeting ARM Cortex-M23. Our evaluation encompassed testing numerous scenarios to analyze the accuracy of the tool in identifying information leakages and its ability to handle larger codebases. The results demonstrated the high precision and scalability of SCFARM, validating its utility in real-world security assessments.

***Organization.*** The paper is structured as follows. We start by giving an overview of side channel attacks and explain our threat model in Section 2, followed by a discussion or related work on the detection of side-channel information leakage in programs in Section 3. In Section 4, we elaborate on our approach. We then describe the components of our SCFARM, and their tasks in Section 5. In Section 6, we present the results of our evaluation. Finally, we conclude our work and outline future directions of research.

**2 BACKGROUND AND PROBLEM STATEMENT**

**2.1 Trusted Execution Environment**

Trusted Execution Environments (TEEs) are a fundamental component of modern security architecture, designed to provide a secure execution environment for sensitive and critical computations. TEEs are isolated and tamper-resistant processing environments within a computing system, where the confidentiality and integrity of code and data are safeguarded against a variety of threats [24]. They ensure the authenticity of executed code, verifying that it has not been tampered with. Additionally, TEEs maintain the integrity of the system's runtime states, encompassing vital components such as CPU registers, memory, and sensitive input/output operations. TEEs uphold the confidentiality of code, data, and runtime states, including their secure storage in persistent memory. This comprehensive protection against unauthorized access, code alterations, and data breaches makes TEEs a crucial component in securing sensitive computations and data within a computing system. They enforce strict access control, permitting data access only to code within the same secure execution environment, and code execution is allowed only from predefined entry points to mitigate risks like Return-Oriented Programming attacks [25]. Additionally, TEEs support remote attestation to verify their trustworthiness to third parties [23]. By providing a foundation for secure computing, TEEs have become integral in a variety of domains, from mobile devices to cloud computing. The two main TEE technologies currently available in the market are Intel SGX [26] and ARM TrustZone [7], the latter being the focus of this paper. The effectiveness of TEEs relies on a well-defined attacker model, critical for identifying potential threats and guiding the development of secure TEE-based solutions.

**2.1.1- TrustZone on ARM Cortex-M**

ARM TrustZone is a hardware-based security technology developed by ARM Holdings [7, 27]. TrustZone essentially divides the ARM processor into two distinct execution environments: the "Normal World" and the "Secure World". In fact, this system-wide approach assigns two virtual cores to each physical processor, together with the mechanism to securely switch between both realms. These environments are isolated from each other, and the Normal World is typically where the non-secure, general-purpose operating system and applications run. The Secure World, on the other hand, is a more trusted and isolated area where security-critical operations, cryptographic functions, and sensitive data can be processed and stored.

On ARM application processors (Cortex-A) [28], a separate processor mode known as the secure monitor handles secure context switching between worlds. However, on ARM microcontrollers (Cortex-M) [6, 29] lack a dedicated secure monitor software. Instead, essential mechanisms integrated into the core logic act as gatekeepers, facilitating the transition between secure and non-secure realms. These two worlds are rigidly separated at the hardware level and possess differing levels of privilege. Non-secure software is explicitly restricted from directly accessing resources in the secure world. This paper focuses exclusively on TrustZone features for Cortex-M processors.

TrustZone technology for Armv8-M devices [6, 29] is tailored for ARM microcontrollers, specifically the Cortex-M series. It's been finely tuned for swift context switching and ultra-low power embedded applications. Leveraging specialized hardware integrated into Cortex-M cores along with a dedicated secure instruction set, TrustZone facilitates the establishment of multiple software security domains. These domains enforce strict access controls, allowing trusted software exclusive access to secure memory and I/O, all while maintaining optimal system performance.

**Armv8-M Architecture** [31]typicallyfeatures a set of 32-bit general-purpose registers (R0 to R12, Link Register (LR), Program Counter (PC)) and floating-point register (D0-D15) that are shared between secure and non-secure states. TrustZone-enabled Armv8-M microcontrollers have separate stacks for each security state, with the Stack Pointer (SP) being security-banked, meaning one instance exists in each state. The CONTROL register and some other special-purpose registers are also banked, and the core automatically switches between their instances during state transitions. ARMv8-M architecture introduces a new ISA with additional instructions and features, which enhances code density, reduces interrupt latency, and improves system performance. The architecture includes a two-stage pipeline for instruction execution, providing efficient handling of instructions.

**Memory space** in the Armv8-M architecture is also partitioned into secure and non-secure memory regions. The secure memory space is further divided into two types: secure and non-secure callable (NSC). Secure addresses are exclusively allocated for memory and peripherals that can only be accessed when the core is executing in secure state. The program address, the address of the instruction currently executed, determines the security state of the processor. In contrast, non-secure addresses are designated for memory and peripherals accessible by all software running on the device, including both secure and non-secure components. NSC represents a unique class of secure memory locations that facilitates the transition of software from a non-secure to a secure state, allowing for controlled and secure state changes.

The security state assigned to each memory address are established through either the programmable Secure Attribution Unit (SAU) or by an fixed Implementation Defined Attribution Unit (IDAU). The SAU is always available in Armv8-M cores, while the IDAU is external to the core and the presence depends on the vendors implementation. In cases where both the IDAU and SAU are available within a system, the SAU's attributions take precedence, unless the IDAU specifies a higher security attribute for a particular address.The SAU can only be programmed in the secure state.

In ARM TrustZone-M, the Nested Vectored Interrupt Controller (NVIC) has been enhanced to enable secure and non-secure configuration for each interrupt. The processor seamlessly handles interrupts based on its current security state. Notably, when a non-secure interrupt occurs during secure code execution, the processor securely manages the transition, preserving secure context data and preventing information leakage.

## For Transition between two worlds, three new instructions have been introduced including secure gateway (SG), branch with exchange to non-secure state (BXNS), and branch with link and exchange to non-secure state (BLXNS). The SG instruction is employed for switching from the non-secure to the secure state. It is typically found at the start of a secure entry point's veneer, which consists of an SG instruction followed by a branch to the secure world's function. The veneers are meant to reside in memory regions attributed to the NSC by the linker. The SG instruction serves several functions, such as setting the security level to secure, banking registers, and resetting bit[0] of the LR register to 0, indicating that the return will lead to a transition back from secure to non-secure.To return from the secure world to the non-secure world, as illustrated in Fig. 1, the compiler employs the BXNS instruction. This instruction initiates a branch or return to the non-secure program.

## A diagram of a security system Description automatically generated

## Fig. 1: Secure Function Call

Conversely, secure software can invoke functions in the non-secure world. This action prompts the generation of compiler code that orchestrates the transition. It begins by preserving all registers, including the return address, within the secure stack. Subsequently, the registers are cleared. The BLXNS instruction is used to execute the branch to the non-secure world, where it sets LR to a specific value, FNC\_RETURN (0xFEFFFFFF).

Upon completion of the execution in the non-secure world, a return to the secure world is initiated using BX. When the BX instruction detects the FNC\_RETURN value in LR, it triggers a transition to the secure state. This shift is made possible by restoring all saved registers, including the return address, from the secure stack. It's also important to note that state transitions may also occur due to exceptions and interrupts.

TrustZone is not bullet-proof and has experienced successful attacks across various methods and contexts [7, 30]. The architecture, while designed to provide robust hardware-based security by isolating secure and non-secure worlds, is not immune to microarchitectural side-channel vulnerabilities [7, 9, 10, 30, 31, 32]. These vulnerabilities arise due to the shared resources and memory management between the secure and non-secure domains. Arm [20] has acknowledged that the security extensions for the Armv8-M architecture are not designed to protect against side-channel attacks resulting from control flow or memory access patterns. They argue that such attacks are not exclusive to the Armv8-M architecture and can apply to any code with secret-dependent control flow or memory access patterns. This type of attack can be mitigated by ensuring that the control flow and memory accesses patterns created by the program do not depend on secret state.

**2.2- Covert Storage Channels**

Sensitive information can be unintentionally leaked through covert storage channels [19, 35]. These channels, whether explicit or implicit, provide pathways for the unauthorized transfer of sensitive data, which can compromise system security. Implicit covert storage channels, in particular, introduce challenges, as data can inadvertently traverse indirect or unintended pathways resulting from the program's control structure. This contrasts with explicit information flows, which occur when confidential data is directly assigned to public variables. To illustrate, let us consider a scenario with two security levels: “high” and “low,” denoting varying degrees of confidentiality. We can examine the code presented in Figure 2, which exhibits a data flow from the high variable *h* to the low variable *l*. The insecurity within this code stems from the *l = 1* assignment in a control context conditioned upon the confidential variable *h*.

A white paper with black text

Description automatically generated

**Fig. 2: An implicit information flow**

**2.3- Microarchitectural Side-Channel Attacks**

The security model proposed by TEEs is not foolproof and must be approached with caution regarding side-channel attacks. These attacks aim to uncover secret-dependent information hidden within the shared microarchitectural state during a victim's execution by exploiting observable side effects, notably timing variations. Typically, adversaries begin by initializing the shared microarchitectural elements in a predetermined state. They then proceed to measure state changes during or after the victim's execution, utilizing methods such as transactional memory aborts or performance monitoring counters. However, the most prevalent method for observing microarchitectural state changes is through timing analysis [33]. In cases where microarchitectural optimizations depend on global stateful elements like Translation Lookaside Buffers (TLB), caches, or branch predictors, any modifications to these elements during the victim's execution will result in measurable timing differences in the attacker domain.The analysis of microarchitectural state updates provides valuable insights into the victim's behavior, even in scenarios where attackers are architecturally isolated and have limited interaction with the victim, strictly through defined input and output channels.

Single-purpose embedded processors typically emphasize simplicity, power efficiency, and cost-effectiveness over advanced microarchitectural features like caches, pipelining, and speculative execution. This focus results in predictable instruction timings, reducing the risk of side-channel attacks. However, research [34] has demonstrated that secrets can still be revealed through start-to-end timing side channels, by measuring the overall execution time of secret-dependent branches, even on processors with entirely deterministic instruction timing behavior.

In addition, Nemesis-type interrupt timing attacks [3] can exploit highly precise, instruction- granular timing measurements, which can even compromise secrets from branches with balanced start-to-end timings. These side-channel attacks abuse the CPU's interrupt mechanism to reveal microarchitectural instruction timings within TEEs. The attack leverages the fact that hardware interrupts are only processed upon instruction retirement, afterthe currently executing instruction has completed, resulting in variable CPU cycles for different instruction types and processor states. Consequently, an untrusted operating system can precisely measure interrupt handling time, to retrieve the execution length of interrupted instruction and distinguish between secret-dependent program branches. In essence, for a successful Nemesis attack on processors with constant-time interrupt latency and multi-cycle instruction sets, where each instruction is uninterruptible, an attacker just requires a different execution time for at least one instruction in the if/else branch.

In recent findings, researchers have identified DMA-based side-channel attacks specifically aimed at embedded TEEs [31, 36]. These attacks exploit nuanced timing variations arising from contention between a DMA device and the CPU as they access the shared memory bus. This exploitation enables the attacker to construct a cycle-accurate memory access trace of a victim program. Notably, at the Black Hat Asia conference, Cristiano Rodrigues and team [36] presented a side-channel attack that leverages DMA to target ARM TrustZone. This sophisticated attack effectively bypasses hardware-enforced isolation primitives, providing unauthorized access to trusted applications (TAs) and resulting in the unauthorized leakage of sensitive information.

**2.3.1- BUSted: Software-based Microarchitectural Timing Side-Channels Attacks on TrustZone-M MCUs**

BUSted [36] is a novel class of microarchitectural side-channel attacks that leverage the timing differences exposed in the internal bus interconnect arbitration logic that decides which bus master (e.g., CPU, DMA) controls the internal data bus. It's evident that concurrent access by multiple bus master to a shared bus slave (e.g., memory controller) leads to time delays for at least one, causing subtle variations in timing. By observing the timing drifts on memory transactions, an attacker can extract information regarding the victim’s memory access pattern. Consequently, without breaking security isolation boundaries, a malicious bus master can spy on bus activity and determine when another bus master accessed a specific slave.

ARM adopts a load-store memory access model, restricting memory interaction solely to load/store (LDR and STR) instructions. Thus, an attacker can exploit the MCU's load-store architecture to observe timing variations. Specifically, if the victim code employs secret-dependent control flow where loads/stores execute at different clock offsets on conditional paths, resulting in distinct memory access patterns. An attacker could exploit this to deduce and illicitly obtain a particular secret.

To elaborate further, let's examine a code snippet (compiled for the Arm Cortex-M23) that includes a balanced conditional statement dependent on a secret variable, as visualized in Figure 3. Since both execution paths have an identical execution time of 5 clock cycles, starting from t + 1 after the *cmp* instruction, an attacker wouldn't observe any difference in execution time. Consequently, distinguishing between these execution paths and subsequently extracting the secret becomes unfeasible. Yet, an observer could still detect divergent memory access patterns between these execution paths. When the branch (bne) isn't taken, it completes within a single clock cycle, causing the *str* instruction to occur at clock cycle t + 3. Conversely, if taken, the branch incurs a two-clock-cycle process, resulting in the *str* instruction taking place at clock cycle t + 4. By monitoring either of these specific clock cycles, an attacker can potentially deduce the secret by detecting any contention on the data memory bus.

A screenshot of a computer

Description automatically generated

**Fig. 3: (a) secret-dependent branch, (b) compiled code CFG for Arm Cortex-M23, (c) memory access pattern and monitoring of clock t+3.**

The attack was conducted on a security mechanism employing ARM TrustZone's 'safety deposit box' alongside a trusted keypad to safeguard a user's PIN during unlocking. The successful demonstration of this attack revealed the potential to infer the PIN code without compromising ARM TrustZone's security measures. To put this into perspective, consider a scenario where a smart home operates on an ARM processor. If an attacker gains physical access, they could exploit this side-channel attack, potentially bypassing security measures. This breach could grant them control over the entire system, enabling alterations to thermostat settings, disabling security cameras, or unauthorized unlocking of doors.

Given the nature of this attack, targeting on microarchitectural design issues, comparisons have arisen likening its impact to that of renowned exploits like 'Spectre' and 'Meltdown' which affected more complex architectures in recent history. Embedded projects relying on hardware-assisted privilege separation via TrustZone-M must now factor in the potential for information leakage from secure components operating within the secure world. According to the researchers, there are software-based countermeasures available to mitigate the impact of this microarchitectural design flaw. The crucial consideration in addressing time-based attacks involves minimizing the presence of secret-dependent code in security operation implementations. Essentially, the time required for a security procedure should remain independent of the success or any secret of the operation.

**2.4- Static Code Analysis**

**A. Taint Analysis** [37]stands as a crucial program analysis method, meticulously tracing the flow of data of interest throughout program execution. Taint analysis utilizes 'taint tags’ as markers attached to registers and memory, serving to indicate their taint status. It operates via three integral components:

1. **Taint Sources:** These denote points within the program or memory locations where relevant data is introduced, often focusing on user inputs from local or remote sources.
2. **Taint Propagation:** This involves the transfer of taint tags during program execution, governed by predefined rules aligned with instruction semantics. Consider the instruction ADD src, dst; in this scenario, a taint propagation rule might dictate that the resulting tag of destination (dst) comprises a bitwise OR operation between the tags associated with src and dst.
3. **Taint Sinks:** These specific program instructions serve as checkpoints for taint analysis, assessing the presence of targeted taint tags. They play a critical role in security applications, detecting potential threats like control flow hijacks or information leaks, often associated with control flow transfers or output system calls.

**B. Value Set Analysis** [38] operates as a static program analysis technique, approximating the potential values that each program data object might hold at any given program point. By analyzing individual instructions within a control-flow graph (CFG), value set analysis effectively tracks and estimates the diverse range of values that each data object could hold.

**C. Symbolic Execution** [39, 40] stands as a method to explore the potential paths of program execution by substituting variables with symbolic representations. It systematically traverses the program, executing with symbolic inputs to understand the possible behaviors and identify vulnerabilities or paths that might lead to critical issues.

**3- RELATED WORK**

***Side-channel Attacks on TrustZone-M.***Extensive research has been conducted on microarchitectural timing channels [2], notably introduced by Kocher [42], gaining widespread attention following the disclosure of Spectre [43] and Meltdown [44]. However, exploration into side-channel attacks within TEE context is a relatively recent endeavor. Several authors [8, 9, 10, 11, 30, 41] have raised concerns regarding software side-channel vulnerabilities in higher-end TEEs like Arm TrustZone. Additionally, research efforts on Microcontroller Units (MCUs) [3, 31, 32, 36, 45, 46, 47] have investigated the potential for information leakage through software-based side-channels. For instance, Gnad et al. in [47] capitalized on the correlation between ADC noise and MCU power consumption (STM32 Cortex-M4), utilizing software power consumption traces to extract secret keys from an AES implementation. Similarly, O'Flynn and Alex Dewar in [44] exploited the ADC in a SAM L11 (Cortex-M23) MCU, executing a remote power side-channel attack to bypass TrustZone-M protection and retrieve a secret key. In contrast to power side-channel attacks, Nemesis attack by Van Bulck et al. [3] exploits the CPU's interrupt mechanism to extract instruction timings from MSP430 MCUs. In [36], the authors leverage minor timing variations in unprivileged DMA requests, arising from contention on the shared memory bus within openMSP430 MCUs, to acquire a memory access trace of a victim program. Likewise, BUSted [31] represents a type of side-channel attack utilizing timing discrepancies on the MCU bus interconnect to bypass the security assurances provided by memory protection primitives in Armv8-M MCUs with TrustZone-M.

***Microarchitectural Timing Side Channels Static Analysis.*** There exists substantial literature on timing side channel detection employing ML models [18, 48, 49], dynamic taint analysis [52], fuzzing [53], Abstract interpretation [55, 56], Logical reduction [54], type-based solutions [15, 16, 17, 57-59], and several other methodologies [2, 50, 51]. Our focus will be on approaches that bear direct relevance to our research. Köpf et al. [55] proposed an approach to automatically derive upper bounds on cache leakage within cryptographic executables. Subsequently, CacheAudit [56] expanded upon their research by enhancing abstractions and precision. It uses static analysis for cache side channels to derive formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, based on observing cache states, traces of hits and misses, and execution times. Chen et al. [54] introduced Themis, an innovative end-to-end static analysis tool tailored for Java applications. Themis utilizes Quantitative Cartesian Hoare Logic (QCHL) to verify ϵ-bounded noninterference, enabling the detection of intricate resource-usage side-channel vulnerabilities within real-world Java programs.

FlowTracker [57] offers the capability to statically trace data dependencies, identifying possible timing leaks in LLVM programs. By leveraging the presumption of LLVM code being in Static Single Assignment (SSA) format, the tool computes control dependencies through a sparse analysis method, negating the need to construct the entire Program Dependency Graph. Barthe et al. [16] proposed an assembly-level type system to verify the constant-time policy. Zhang et al. [58] introduced a language-based approach for a basic While-language, aiming to track side-channel leaks. The authors suggested a cooperative model between hardware and software to mitigate covert timing channel. Side Channel Finder (SCF) [59] checks secret-dependent loops and branchings using a type system for static detection of timing channels in Java. In our prior work [17], we proposed a security type system designed for statically analyzing MSP430 binaries. This system ensures the absence of timing leaks, Nemesis-style vulnerabilities, and unintended information flow. To enhance the accuracy of our analysis from previous research and expand our capability to trace information flow across TrustZone-provided protection domains, this study employs a symbolic execution-based approach. This allows for meticulous control over memory operations, refining the precision of our analysis.

***Symbolic Execution.*** Some works [13, 14, 60-66] have, furthermore, focused on detecting microarchitectural side-channel vulnerabilities using symbolic execution. For instance, Bang et al. [60] use symbolic execution, string analysis, and model counting to quantify leakage for a particular type of side channel. Pasareanu et al. [61] proposed a symbolic execution approach for side-channels detection and quantification. They measure side-channel leakage by creating specific public inputs that trigger maximum leakage. This is accomplished through Max-SMT solving applied to the constraints derived from symbolic execution. ENCIDER [62] employs dynamic symbolic execution and taint analysis to uncover timing and cache side-channel vulnerabilities within Intel SGX applications. It decomposes side-channel requirements based on the bounded non-interference property and implements byte-level information flow tracking through API modeling. CoCo-Channel [63] employs taint analysis to detect secret-dependent conditional statements within Java programs. It assigns symbolic cost expressions to various program paths and utilizes symbolic execution to identify and report paths demonstrating secret-dependent timing behavior.

Additionally, various other studies [64, 65, 66] leverage symbolic execution to derive a symbolic cache model and verify that the cache behavior remains independent of sensitive data. Scalability concerns often hinder symbolic execution. Daniel et al. [13] introduced an automatic, efficient binary-level verification method tailored for constant-time analysis. This method conducts both bug identification and bounded verification on practical cryptographic implementations. Employing relational symbolic execution with specialized optimizations in information flow and binary-level analysis, their approach maximizes shared information between executions following the same path. Pitchfork [14] unites symbolic execution and dynamic taint tracking to accurately propagate secret taints across all execution paths, highlighting tainted branch conditions or memory addresses. Notably, Pitchfork can analyze protocol-level code by abstracting the implementation details of primitives through function hooks, allowing separate analysis of these components.

Developing constant-time code presents complexity due to the need for intricate low-level operations that diverge from conventional programming practices. Maintaining this approach proves challenging as compiler optimizations often fail to preserve such implementations. Moreover , the vulnerability revealed in the attack on the constant-time implementation applied to the Curve25519 elliptic curve [67] highlights the error-prone nature of writing such code. This paper explores a purely static method for identifying timing side channels by integrating symbolic execution and taint analysis. Our prototype, SCFARM, was developed to verify ARMv8-M binaries for potential side channel vulnerabilities. To the best of our knowledge, this represents the first static analysis tool capable of automatically detecting timing side channels, Nemesis, BUSted, and covert storage leakage, within ARM-M binaries.

**4- SCOPE AND THREAT MODEL**

*Scope.* Our focus revolves around small-scale embedded systems and IoT devices that operate using MCUs, such as the Arm Cortex-M family. These MCUs operate within strict constraints, characterized by limited computing power and memory. They feature simplified microarchitectures, lacking components like caches and typically operating with 2-3 pipeline stages. Additionally, they do not support virtual memory. Generally, MCUs consist of a single CPU while offering a diverse array of peripherals, including UART, SPI, timers, DMAs, and I2C, among others. Some devices may incorporate MPUs, and the latest iterations of Armv8-M MCUs introduce support for dual security states, namely secure and non-secure worlds (known as TrustZone-M). These MCUs are engineered to ensure highly predictable outcomes, consistently delivering identical outputs for specific inputs within defined timing constraints**.**

*Threat model.* The adversary’s goal is to extract sensitive information from an isolated environment by bypassing the memory isolation security mechanisms in TrustZone. We assume the attacker has access to either the source code or compiled binary code of the victim's program. Additionally, they can monitor the program's execution time and outputs. Furthermore, we consider a more capable attacker who has complete control over the unprotected normal world and its resources. This includes the ability to manage bus masters (e.g., DMA), configure peripherals like timers, and control scheduling decisions. We are not considering side channels arising from cache contention or branch prediction feature, as they fall beyond the scope of this work due to their absence in the targeted MCU architecture.

**5- STATIC INFORMATION FLOW ANALYSIS**

**5.1- A Simple Example**

While TrustZone technology in Armv8-M processors offers robust security measures, it remains susceptible to side channel attacks due to secret-dependent control flow, resulting in observable timing variations or secret-dependent memory access patterns [20]. Furthermore, it's worth noting that TrustZone might not entirely mitigate implicit and explicit information leaks resulting from unintentional developer errors in secure coding practices, system architecture flaws, and non-compliance with stringent security standards and protocols.

To elaborate, consider a secure One-Time Password (OTP) system illustrated in Figure 4. OTP system is integral in verifying mobile users accessing critical web services that require a heightened level of security. An OTP is a dynamically generated numeric or alphanumeric string of characters used to authenticate a user for a single transaction or session with an authentication server. This approach augments the traditional user ID and password authentication by introducing a dynamic password that changes with each authentication attempt. The OTP mechanism operates by generating this code using an internal clock (or counter) and a factory-encoded secret key known as the 'seed.'

To maintain the confidentiality of both the generated OTPs and the seeds, the code and data involved in OTP operations reside within a secure world provided by TrustZone. This creates a distinct, isolated space separate from the regular operating system running in the normal world. TrustZone technology ensures the integrity and confidentiality of OTPs and seeds by restricting access solely to this secure domain.

Nevertheless, if there is an inadvertent mishandling of the seed or OTPs—such as storing them in plaintext on unprotected memory, logging them, or exposed via non-secure I/O —or if an implicit flow is triggered by updating a public variable—an attacker could potentially access the generated OTPs.

A diagram of a computer code

Description automatically generated

**Fig. 4: TrustZone-based Implementation of Secure OTP Generation**

We present an approach that merges taint tracking techniques with symbolic execution of trusted application (TA) binaries to systematically identify the undesired information leakage. Subsequently, we notify developers and designers of these identified leakage points for further action and resolution.

**5.2- Taint Tracking of Function Inputs**

Our method involves an information flow analysis that links each value with a security tag indicating its sensitivity level. We primarily categorize these levels into two labels: 'H' for high sensitivity and 'L' for low sensitivity, where 'H' signifies higher sensitivity than 'L'. Each input (initial state of registers, etc.) and output (final state of registers, etc.) of a program is assigned one of these labels. To effectively track and identify both explicit and implicit data flows of high sensitivity during program execution, our approach incorporates an effective symbolic taint-tracking mechanism.

Specifically, initial register contents and associated memory are transparently substituted with unconstrained symbolic values. We, furthermore, utilize Angr’s annotation system to flag highly sensitive symbolic values *as tainted*.This taint, conservatively propagated throughout symbolic execution, allows for convenient querying during subsequent analysis to identify potential information leakage.

**5.3- Timing-Sensitive** **Information Flow Policy**

An information-flow analysis verifies the absence of undesired information leakage within a program. A timing-sensitive variant of this analysis considers the impact of confidential data on the program's execution time. The intended security assurance is typically defined through information flow policies that prevent secret data from affecting an attacker's observations. This study utilizes a symbolic taint-tracking strategy to monitor potential policy violations during execution.

To this end, taint tags follow Angr's propagation rules during symbolic program execution, aligning with ARMv8-M instruction semantics. For example, in the case of an instruction like *‘ADD dst, src1, src2’*, a taint propagation rule might dictate that the resulting tag of *'dst'* is determined by performing a bit-wise OR operation on the tags of *‘src1’* and *‘src2’.* Specifically, these rules imposes constraints on the sensitivity of data stored in registers and memory cells throughout program execution, affecting their state upon program termination.

**5.3.1- Detection of timing side channels**

To ensure the absence of timing side channels within a program, our approach involves the initial computation of control-dependence regions for each branch instruction dependent on secret data. This computation employs the Safe Over-Approximation Properties (SOAPs) as defined in [15]. In particular, our focus lies in comparing the execution times between the 'then' and 'else' branches, distinguishing two distinct control-dependence regions for jumps influenced by confidential information.

Afterward, we sums up the execution time of all instructions within each region following definition 1. In scenarios where a nested branch *‘br1’* exists within the region of another branch ‘*br0*’, a recursive procedure becomes necessary due to only one path of ‘*br1’* being executed, whereas the positive part of definition 1 accounts for both paths' cumulative execution time. We address this by subtracting the execution time of the 'then' branch of ‘*br1’*.

**Definition 1.** *The function* **branchPathTimer** *is defined recursively as:* branchPathTimer(br) := ∑ execution\_time(ins) – branchPathTimethen(ins)

*where:*

* r ∈ {*then, else*}
* br ∈ {*beq, bne, bgt, blt, bge, ble*}
* ins ∈ regionr(br)

Within ARMv8-M architecture, conditional branch instructions require an extra clock cycle when they are taken. With this fact, if the total time required for executing the 'then' region plus one equals the time taken for jumping and executing the 'else' region in a secret-dependent branch, it becomes impossible to discern the value of secret data by observing the program's overall execution time or its result.

Furthermore, identifying the Nemesis vulnerability in an ARMv8-M binary involves pinpointing a jump instruction depending on secret data. Due to the variable execution times of branch instructions in ARMv8-M architecture when taken or not, an attacker can interrupt a branch instruction to capture the secret information. Consequently, the attacker can even execute a Nemesis attack on balanced paths.

To detect the BUSted vulnerability in a binary, we meticulously traverse every path within a branch dependent on secret information, scrutinizing the execution points of *'str'* or *'ldr'* instructions. If these instructions execute at varying clock offsets along different conditional paths, it exposes a potential vulnerability for attackers to exploit distinct memory access patterns and gain access to sensitive information.

**5.3.2- Augmented Taint Flow Directives**

Secret-dependent branches introduce variations in the program flow based on confidential information. By observing changes in execution time, logical operations, or other side channel information arising from these branches, attackers can deduce details about the secret data, such as its value or structure. Consequently, when a secret-dependent branch involves an operation where a memory cell or register is written within a particular path, it's crucial to mark that register or memory as *tainted*. These elements carry highly sensitive information during program execution, causing variations that attackers can exploit to infer the secret.

We employ Angr’s MemoryMixin extension, which conducts exhaustive validations on every memory access. Specifically, when these accesses occur within secret-dependent branches, we annotate the potentially symbolic values as *tainted* to reflect their elevated sensitivity within that particular context. This approach guarantees the absence of flows from secret information to attacker-observable outputs.

**6- SCFARM: DESIGN AND IMPLEMENTATION**

We have developed the SCFARM specifically for the static analysis of ARMv8-M binaries targeting Cortex-M23, focusing on identifying timing side-channel vulnerabilities. Our approach, centered around symbolic taint-tracking, enables a rapid, cost-effective, and automated assessment of these vulnerabilities. We have implemented an open source[[1]](#footnote-1) version of SCFARM built upon Angr[[2]](#footnote-2) and SCFMSP[17]. Specifically, (1) we use Angr for symbolic execution of binaries and conducting Value Set Analysis (VSA) to determine potential register values or symbolic memory addresses; (2) we expand the functionality of the SCFMSP tool to analyze TrustZone-M targeted binaries and ensure program integrity against a novel DMA-based side-channel attack termed BUSted.

Figure 5 showcases a high-level workflow diagram detailing the components of the SCFARM tool. The dashed box denotes the core elements implementing our proposed side channel evaluation technique. SCFARM is built on approximately 1110 lines of Python code and integrates various established Python libraries, detailed in subsequent sections.

**A diagram of a process

Description automatically generated**

**Fig. 5: Overview of SCFARM**

We'll now provide a concise summary of each element and their practical implementations:

***Parser.*** We employ this component to convert the input binary into assembly language. Initially, we utilize the pyelftools [68] Python library to locate the starting point of instructions within the binary. Subsequently, we leverage the Capstone disassembler[[3]](#footnote-3) to translate the binary into ARMv8-M instructions, generating mnemonic codes and symbolic representations of processor instructions. This disassembly process allows us to extract essential details such as opcodes, addresses, and instruction lengths. Finally, we determine the required clock cycles for executing each instruction based on the specifications outlined in the ARMv8-M Architecture Reference Manual [69].

***CFG Generator.*** This component is responsible to construct an accurate program Control Flow Graph (CFG) using the NetworkX [21] Python library. This component establishes a mutual connection with the subsequent component, enabling the retrieval of the target address for ‘branch and exchange’ instruction *'BX <Rm>*' through VSA in Angr. The resulting graph facilitates the identification of execution point predecessors and successors, aiding in computing control dependence regions for branches and inferring loops within assembly programs.

***Angr.*** Utilizing a JSON-formatted configuration file that lists the starting function's arguments derived from high-level code to determine taint sources (i.e., highly sensitive inputs). This component employs Angr's annotation system to mark registers and memory cells containing confidential data with a taint label. It associates arguments with their corresponding registers following the ARMv8-M calling conventions [69]. These taint tags propagate during the symbolic execution of the input binary. Our taint analyzer tracks both explicit and implicit flows. Upon completion of symbolic execution, the component presents a list of tainted registers and memory cells for subsequent analysis.

***SC Checker.*** This module performs static analysis to identify potential side channel leaks within ARMv8-M assembly programs. Our SCFARM tool currently detects vulnerabilities falling into four distinct classes:

* **Timing leaks:** Detected when a program exhibits varying execution times depending on the secret input. SCFARM detects these leaks by measuring the overall running times of if/else regions within the secret-dependent branches.
* **BUSted vulnerability:** Recognized when a program includes store/load instructions with different time offsets inside the if/else regions of a secret-dependent branch.
* **Nemesis vulnerability:** Identified when a program displays varying execution times for at least one instruction within the if/else regions of a secret-dependent branch.
* **Explicit/Implicit flow leaks:** Flagged if there is an undesired information flow from secret inputs to observable output, such as return registers or non-secure memory.

**7- EVALUATION AND DISCUSSION**

In this section, we present our assessment of SCFARM, emphasizing the accuracy of our static analysis in detecting side channels. Our evaluation of SCFARM included a thorough examination of a carefully selected set of benchmarks, influenced by [71], featuring a diverse range of programs, from those with vulnerabilities to benign ones. These benchmarks were specifically designed to reveal typical and intricate patterns in secret-dependent control flow, providing a comprehensive examination of SCFARM's effectiveness.

The benchmarks consist of C programs compiled using 'arm-none-eabi-gcc,' a cross-compiler toolchain tailored for generating code targeting ARM Cortex-M and Cortex-R processors without an operating system (bare-metal). The compiler is configured with the '-mcpu=cortex-m23' option to generate optimized binary for ARM Cortex-M23 architecture. Here, we provide a brief overview of selected benchmark programs.

* ***busted****:* This program features a vulnerability to the 'busted' attack, primarily due to a secret-dependent branching instruction. The vulnerability arises from the execution of the 'str' instruction at different time offsets within the 'then' and 'else' regions of the branch. However, it is free of start-to-end timing leaks due to having a balanced branch.
* ***busted-free****:* This program introduces dummy instructions, such as 'nop,' strategically placed in the code to delay the execution of the 'str' instruction in the 'else' branch by one clock cycle. This intentional delay ensures that memory access occurs at the same time offset. However, this approach results in an unbalanced branch.
* ***strcmp****:* This code performs a character-by-character comparison of two strings using a loop with an explicit branch. (refer to Fig. 5 (a)). However, because it relies on secret-dependent branches for the comparison process, it introduces potential timing side-channel vulnerabilities.
* ***constant\_time\_strcmp****:*Utilizing a constant-time methodology, we enhance the string comparison code to resist timing side-channel attacks. This involves eliminating branches and employing mathematical formulas within the program (see Figure 5 (b)). The comparison of characters is achieved through bitwise XOR (^) and bitwise OR (|) operations, eliminating the need for explicit branches and effectively mitigating timing side-channel vulnerabilities.

***A close-up of a computer code

Description automatically generated***

**Fig 5: String Comparison**

* ***diamond****:* This program features two branching instructions dependent on secret values and one independent branch.
* ***loop****:* The program contains a loop relies on a condition with public data and includes a secret-dependent branch within the loop body.
* ***secure\_loop****:* In contrast, this code employs bitwise operations to eliminate the need for a conditional branch.
* ***multifork****:* In this program, a switch statement evaluates the value of a secret variable and compares it against multiple cases.
* ***ifthenloop & ifthenloop\_nop\_padded****:* These programs involve a branch dependent on secret data and a loop with a public variable in its condition, executed if the branch is taken. In the *'ifthenloop\_nop\_padded'* version, additional dummy instructions are introduced to ensure a balanced execution across both branches.
* ***ifthenlooplooptail****:* This program introduces nested secret-dependent branches and loops, creating a more intricate control flow to thoroughly test SCFARM.

In Table 1, we present the experimental outcomes obtained by applying SCFARM to the benchmark. Our results showcases that SCFARM effectively analyzes intricate binaries designed for ARM Cortex-M23. The tool not only adeptly identifies a spectrum of side-channel vulnerabilities but also demonstrates its efficiency in swiftly and accurately detecting sensitive data leakage. Keep in mind that utilizing branch balancing or constant-time programming by itself cannot entirely eliminate the risk of unintended information leakage within a program. In such instances, it becomes imperative to take additional measures, such as securely storing program outputs in a designated protected memory space or returning them in an encrypted format. These precautions provide an extra layer of security, effectively mitigating the possibility of sensitive information being unintentionally exposed to attackers, even when employing the mentioned programming techniques.

**Table 1: Evaluation results for SCFARM on the benchmark suite. For each compiled example program we give an indication for C program complexity (LOC = lines of code, CFG Size = number of nodes in the program’s Control Flow Graph), execution time of SCFARM, and list the vulnerabilities found by SCFARM.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Benchmark | LOC | CFG Size | SCFARM exec time | Timing Leak | Nemesis Vulnerable | BUSted Vulnerable | Information Flow Leak |
| busted | 14 | 20 | 3.042s | Ï | P | P | P |
| busted-free | 15 | 21 | 3.725s | P | P | Ï | P |
| strcmp | 10 | 34 | 4.563s | P | P | P | Ï |
| constant\_time\_strcmp | 13 | 39 | 2.564s | Ï | Ï | Ï | Ï |
| diamond | 20 | 35 | 4.762s | P | P | P | P |
| loop | 16 | 49 | 3.330 s | P | P | P | P |
| secure\_loop | 22 | 58 | 4.902s | Ï | Ï | Ï | P |
| fork | 7 | 19 | 3.269s | P | P | P | P |
| fork\_ nop\_padded | 12 | 24 | 3.769s | Ï | P | P | P |
| indirect | 20 | 29 | 3.460s | P | P | P | P |
| multifork | 15 | 34 | 4.013s | P | P | P | P |
| branchless\_multifork | 8 | 42 | 2.713s | Ï | Ï | Ï | P |
| ifcompound | 22 | 33 | 4.480s | P | P | P | P |
| call | 19 | 29 | 2.762s | Ï | Ï | Ï | P |
| ifthenloop | 20 | 43 | 4.476s | P | P | P | Ï |
| ifthenloop\_nop\_padded | 25 | 50 | 4.608s | Ï | P | P | Ï |
| ifthenloopif | 28 | 74 | 6.890s | P | P | P | Ï |
| ifthenlooploop | 28 | 67 | 7.580s | P | P | P | Ï |
| ifthenlooplooptail | 38 | 110 | 8.136s | P | P | P | Ï |
| multiply | 5 | 12 | 3.789s | Ï | Ï | Ï | P |

*Discussion.*

* Path Exploration in symbolic execution
* The time required to execute an instruction on an Cortex-M23 microcontroller, in the absence of advanced architectural features such as paging, caches, or out-of-order instruction pipelining, is completely deterministic. The soundness of our proposed approach relies on this feature which makes an instruction’s execution time accurately predictable.
* Nemesis

**8- CONCLUSIONS AND FUTURE DIRECTIONS**

**REFERENCES**

[1] Papp, D., Ma, Z., Buttyan, L., 2015. Embedded systems security: Threats, vulnerabilities, and attack taxonomy, in: 2015 13th Annual Conference on Privacy, Security and Trust (PST), pp. 145–152. doi:10.1109/PST.2015.7232966.

[2] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. “A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware”. In: Journal of Cryptographic Engineering 8.1 (2018), pp. 1–27.

[3] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In CCS ’18.

[4] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2005. Cache Attacks and Countermeasures: The Case of AES. In Topics in Cryptology - CT-RSA 2006.

[5] “Arm holdings and qualcomm: The winners in mobile.” http://www.forbes.com/sites/darcytravlos/2013/02/28/ arm- holdings- and- qualcomm- the- winners- in- mobile/.

[6] Arm Ltd. 2017. TrustZone technology for ARMv8-M Architecture. Version 2.0.

[7] Pinto, Sandro & Santos, Nuno. (2019). Demystifying Arm TrustZone: A Comprehensive Survey. ACM Computing Surveys. 51. 1-36. 10.1145/3291047.

[8] Z. Kou, W. He, S. Sinha and W. Zhang, "Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict," *2021 58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, 2021, pp. 979-984, doi: 10.1109/DAC18074.2021.9586226.

[9] N.Zhangetal.,“TruSpy: Cache Side-channel Information Leakage from the Secure World on Arm Devices.” Trans. on IACR Cryptol, 2016.

[10] Bukasa, Sebanjila & Lashermes, Ronan & Bouder, Hélène & Lanet, Jean-Louis & Legay, Axel. (2018). How TrustZone Could Be Bypassed: Side-Channel Attacks on a Modern System-on-Chip. 10.1007/978-3-319-93524-9\_6.

[11] K. Ryan, “Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm’s TrustZone,” in Proc. of ACM CCS, 2019.

[12] Saß, Marvin & Mitev, Richard & Sadeghi, Ahmad-Reza. (2023). Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M.

[13] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level, 2019.

[14] Disselkoen, Craig. “Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork.” (2021).

[15] Florian Dewald, Heiko Mantel, and Alexandra Weber. 2017. AVR processors as a platform for language-based security. In Computer Security – ESORICS 2017. Springer, 427–445. https://doi.org/10.1007/978- 3- 319- 66402- 6\_25

[16] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS ’14). ACM, 1267–1279.

[17] Pouyanrad, Sepideh et al. “SCFMSP: static detection of side channels in MSP430 programs.” Proceedings of the 15th International Conference on Availability, Reliability and Security (2020): n. pag.

[18] M. Mushtaq et al., "Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time," 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Bordeaux, France, 2018, pp. 485-488, doi: 10.1109/ICECS.2018.8617994.

[19] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In IEEE Symposium on Security and Privacy, pages 415–429, San Francisco, CA, May 2013.

[20] Arm Developer, “Clarification of Timing Side Channel Attacks on TrustZone enabled Cortex”, <https://developer.arm.com/documentation/ka005578/latest>

[21] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later”, Communications of the ACM, vol. 56, no. 2, 2013.

[22] H. Sun, K. Sun, Y. Wang, and J. Jing. 2015. Trust OTP: Transforming smartphones into secure one-time password tokens. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. ACM, 976–988. DOI:https://doi.org/10.1145/2810103.2813692

[23] Scopelliti, Gianluca & Pouyanrad, Sepideh & Noorman, Job & Alder, Fritz & Baumann, Christoph & Piessens, Frank & Mühlberg, Jan. (2023). End-to-End Security for Distributed Event-Driven Enclave Applications on Heterogeneous TEEs. ACM Transactions on Privacy and Security. 26. 10.1145/3592607.

[24] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. IEEE Transactions on Computers, 67(3):361–374, 2017.

[25] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented program- ming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC), 15(1):1–34, 2012.

[26] V. Costan and S. Devadas. Intel sgx explained. IACR Cryptology ePrint Archive, 2016(086):1–118, 2016.

[27] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. 2016 IEEE 2nd Int. Conf. Collab. and Internet Computing (CIC) (2016), 445–451.

[28] Arm Ltd. 2009. ARM Security Technology: Building a Secure System using TrustZone Technology.

[29] J. Taylor. 2016. Security for the next generation of safe real-time systems. In Proceedings of Embedded World Conference.

[30] Muñoz, Antonio & Rios, Ruben & Roman, Rodrigo & Lopez, Javier. (2023). A survey on the (in)security of Trusted Execution Environments. Computers & Security. 129. 103180. 10.1016/j.cose.2023.103180.

[31] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto, "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect",  2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2024.

[32] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu and Z. Zhao, "Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense," *2023 60th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, 2023, pp. 1-6, doi: 10.1109/DAC56929.2023.10247972.

[33]: Van Bulck, J. (2020). Microarchitectural Side-Channel Attacks for Privileged Software Adversaries (Doctoral dissertation). KU Leuven, Belgium.

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

[35] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," in *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5-19, Jan. 2003, doi: 10.1109/JSAC.2002.806121.

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

[37] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the 12th Annual Network and Distributed Systems Security Symposium, NDSS ’05, 2005.

[38] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In Proceedings of the 2004 International Conference on Compiler Construction, CC ’04, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] Schwartz, Edward & Avgerinos, Thanassis & Brumley, David. (2010). All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). Proceedings - IEEE Symposium on Security and Privacy. 317-331. 10.1109/SP.2010.26.

[40] J. C. King. “Symbolic Execution and Program Testing”. In: Communications of the ACM 19.7 (1976), pp. 385–394.

[41] M. Gross, N. Jacob, A. Zankl, and G. Sigl. Breaking trustzone memory isolation through malicious hardware on a modern fpga-soc. In Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, pages 3–12, 2019.

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

[43] Paul Kocher et al. “Spectre Attacks: Exploiting Spec- ulative Execution”. In: Proc. of S&P. 2019.

[44] Moritz Lipp et al. “Meltdown: Reading Kernel Mem- ory from User Space”. In: Proc. of USENIX Security. 2018.

[45] 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.

[46] Alessandro Barenghi et al. “Exploring Cortex-M Mi- croarchitectural Side Channel Information Leakage”. In: Journ. IEEE Access. 2021.

[47] 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.

[48] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache- based side-channel attacks using hardware performance counters,” Appl. Soft Comput., vol. 49, pp. 1162–1174, Dec. 2016.

[49] Z. Allaf, M. Adda, and A. Gegov, “A comparison study on flush+reload and prime+probe attacks on aes using machine learning approachess,” UK Workshop on Computational Intelligence, pp. 203–213, 2017.

[50] Akram, Ayaz & Mushtaq, Maria & Bhatti, Muhammad & Lapotre, Vianney & Gogniat, Guy. (2020). Meet the Sherlock Holmes’ of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. IEEE Access. PP. 1-1. 10.1109/ACCESS.2020.2980522.

[51] Szefer, Jakub. (2019). Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. Journal of Hardware and Systems Security. 3. 10.1007/s41635-018-0046-1.

[52] Graa, Mariem & Cuppens-Boulahia, Nora & Cuppens, Frédéric & Lanet, Jean-Louis & Moussaileb, Routa. (2017). Detection of Side Channel Attacks Based on Data Tainting in Android Systems. 205-218. 10.1007/978-3-319-58469-0\_14.

[53] Nilizadeh, Shirin & Noller, Yannic & Pasareanu, Corina. (2018). DifFuzz: Differential Fuzzing for Side-Channel Analysis.

[54] Chen, Jia & Feng, Yu & Dillig, Isil. (2017). Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. 875-890. 10.1145/3133956.3134058.

[55] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In International Conference on Computer-aided Verification.

[56] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”, ACM Transactions on Information and System Security, vol. 18, no. 1, 2015.

[57] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-channel Detection. In Proceedings of the 25th International Conference on Compiler Construction (CC 2016). ACM, 110–120.

[58] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’12). ACM, 99–110.

[59] Lux, Alexander & Starostin, Artem. (2011). A Tool for Static Detection of Timing Channels in Java. J. Cryptographic Engineering. 1. 303-313. 10.1007/s13389-011-0021-z.

[60] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). ACM, 193–204.

[61] Corina Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-Run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In Computer Security Foundations Symposium. IEEE.

[62] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. Butler, and D. J. Tian. “ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs”. In: IEEE Transactions on Dependable and Secure Computing (2022).

[63] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu.2018. Symbolic path cost analysis for side-channel detection. In ISSTA.

[64] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In S&P.

[65] S. Chattopadhyay and A. Roychoudhury. 2018. Symbolic Verification of Cache Side-Channel Freedom. Trans. Comput. Aided Des. Integr. Circuits Syst. (2018).

[66] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In ASE.

[67] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, “When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015”, in CANS, 2016.

[68] pyelftools. 2011. pyelftools: Parsing ELF and DWARF in Python. Retrieved 5 Dec 2019 from <https://github.com/eliben/pyelftools>.

[69] Armv8-M Architecture Reference Manual, Arm Limited or its affiliates, 2021.

[70] Aric Hagberg, Pieter Swart, and Daniel S. Chult. 2008. Exploring network structure, dynamics, and function using NetworkX. Technical Report. 11–15 pages.

[71] Hans Winderix, Jan Tobias Mühlberg, frank Piessens, [Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks](https://lirias.kuleuven.be/retrieve/687789), IEEE European Symposium on Security and Privacy, 2021 IEEE European Symposium on Security and Privacy (EuroS&P), pages 667-682, Virtual, September 6-10, 2021.

1. https://github.com/sepidehpouyan/SCF-ARM [↑](#footnote-ref-1)
2. https://angr.io/ [↑](#footnote-ref-2)
3. http://www.capstone-engine.org/ [↑](#footnote-ref-3)