Wait a Cycle: Eroding Cryptographic Trust in Low-End TEEs via Timing Side Channels

Ruben Van Dijck
DistriNet, KU Leuven
Leuven, Belgium

Marton Bognar
DistriNet, KU Leuven
Leuven, Belgium

Jo Van Bulck
DistriNet, KU Leuven
Leuven, Belgium

***Abstract*—The growing interconnectivity of low-end embedded devices has spurred research into lightweight trusted execution environments (TEEs), which are designed to meet strict power, cost, and real-time constraints. A key focus has been on the development of dedicated frameworks and libraries to ensure message integrity and authenticity through strong, sometimes formally verified cryptography. However, existing security analyses commonly dismiss side channels, assuming that small microcontrollers are less susceptible to timing variations than high-end CPUs and that these variations are easily avoided by good programming practices.**

This paper systematically examines timing side channels in open-source low-end TEEs. We identify subtle vulnerabilities at different levels of the hardware-software stack: (1) the use of non-constant-time C/C++ standard library functions such as `memcmp`; (2) compiler-induced timing leaks for comparing primitive data values; and (3) a hardware-level timing flaw in the cryptographic core of the Sancus TEE. We experimentally validate these timing side channels and build practical exploits to break TEE security guarantees and inject forged messages. Our findings demonstrate that timing side channels pose a critical, yet often overlooked threat to low-end TEEs, underscoring the need for future security models to account for them.

1. Introduction

As embedded devices become increasingly prevalent and interconnected, securing their data and operations is more critical than ever. These devices, which play vital roles in medical equipment, automotive systems, and industrial control, require robust security measures to prevent compromise. However, many 8- and 16-bit embedded microcontrollers used in these settings lack established security mechanisms like virtual memory or CPU privilege levels. To address this gap, specialized low-end trusted execution environments (TEEs) isolating small *enclave* memory regions have emerged as a promising solution, gaining traction both in academic research prototypes [1]–[7] and, to some extent, in commercial microcontrollers [8]–[10]. Given the interconnected nature of these devices, significant effort has been devoted to the development of dedicated frameworks and libraries for *authentic execution*, leveraging strong cryptography to transparently ensure the integrity and authenticity of input and output messages [11]–[16].

While authenticity guarantees for low-end enclaves are well understood and sometimes even formally modeled [2], [15], [17], any information leakage from timing side channels is commonly considered out of scope. Unlike high-end TEEs such as Intel SGX, which have been extensively analyzed for software-exploitable side-channel vulnerabilities [18], embedded TEEs run on small microcontrollers that lack advanced microarchitectural features, making them less susceptible to timing attacks. As Noorman et al. [11] 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 paper, we demonstrate that writing timing-independent code for low-end enclaves is fragile and leakage can be introduced at different layers of the hardware-software stack, breaking otherwise sound cryptographic authenticity and integrity guarantees. First, the standard libraries of C and C++, often used for the development of enclave software, are optimized for performance rather than security. Notably, the `memcmp` and `std::equal` functions halt comparison as soon as they find a difference, creating a clear timing side channel. Concerningly, our analysis of open-source embedded TEE research prototypes reveals uses of plain `memcmp` and `std::equal` [12], [14], and even non-constant-time custom comparison functions [2], [19].

Second, we find that embedded compilers can introduce unexpected timing leaks that may not be immediately apparent even to programmers familiar with known compiler side effects [20]. Specifically, compilers targeting microcontrollers often emit non-linear assembly code when comparing two numbers using C's equality operator (==) for commonly used `uint32_t` or `uint64_t` values. Based on our analysis, this compiler issue affects at least two open-source authentication libraries that rely on primitive data type equality checks for authentication tag comparison [13], [15].

Lastly, even hardware logic itself is not free from timing threats, as we demonstrate by uncovering a subtle flaw in the tag comparison for authenticated encryption in the hardware implementation of the Sancus TEE [1]. To showcase the practical implications of the hardware side channel, we systematically investigate its effects on 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 [12].

From the broader perspective, our findings contribute to the ongoing discourse on the resilience of TEEs and their broader implications for cryptographic trust in embedded systems. Notably, many of the systems we analyze [15], [17], [19] are shown to be vulnerable despite having formal proofs of security. Finally, we also propose and evaluate mitigations in software and hardware for our uncovered vulnerabilities and discuss how to avoid similar problems in the future.

Contributions. In summary, our main contributions are:

- We analyze how subtle timing leakage emerges across different hardware-software layers through insecure library functions, compiler transformations, and hardware finite-state machines.
- We demonstrate the impact of these timing differences on the security guarantees of embedded research TEEs and authentication libraries.
- We design and evaluate software and hardware mitigations with minimal overhead.
- We discuss broader implications and lessons to prevent these types of vulnerabilities in the future.

Open Science. All our code, data, and scripts are available at https://github.com/dnet-tee/wait-a-cycle.

2. Background & Related Work

TEEs are a class of security architectures that provide security guarantees and services for code running on them, including but not limited to isolation, attestation, and data sealing. Intel SGX [21] is a representative example of a commercial TEE running on high-end CPUs. In this paper, we focus on designs aimed at low-end computing systems, such as resource-constrained microcontrollers. Examples in this space include Sancus [1], providing isolation and attestation, and VRASED [2], offering a formally verified attestation primitive. These architectures have also been extended over the years to offer more rich security guarantees [11], [12], [19], [22], [23]. In the following, we provide more details on the Sancus TEE, the main focus of this work. For a more detailed overview and a collection of related publications, we refer to the Sancus website.[1]

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 embedded TEEs [2]–[4], [24] have been developed. These architectures aim to (1) isolate an enclave-like memory region in the single address space; (2) optionally perform attestation of the protected [4], [6], [24] or the unprotected region [2], [3]; (3) offer (automated) facilities for authenticated communication [1], [11], [13]. We briefly overview these primitives below, guided largely by the open-source Sancus architecture as a representative example of a low-end TEE.

Sancus 2.0 [1] extends the original Sancus [24] architecture for networked embedded devices. It is built on top of the openMSP430 processor [25], which is an open-source implementation of the MSP430 microcontroller created by Texas Instruments [26]. The MSP430 is a low-end 16-bit instruction set, featuring peripherals and a flexible clock system that link up using a von Neumann architecture. Sancus implements minimal hardware extensions for isolation, local and remote attestation, and code confidentiality. Sancus features a hardware cryptographic unit that provides integrity and confidentiality to protected enclaves. The cryptographic unit, written in Verilog, is set up as a finite state machine (FSM) and is called through custom assembly instructions. This unit makes use of the SpongeWrap [27] authenticated encryption scheme using SPONGENT [28]. We focus on three primitive operations: `unwrap`, `verify`, and `enable`. In combination with a key derivation function and a key storage mechanism, these primitives are used to provide the security services.

1. https://distrinet.cs.kuleuven.be/software/sancus/research.php

Isolation. Enclaves have a single data and code section in memory. Sancus's custom hardware circuitry for memory access control makes enclave-private code and data sections inaccessible from code located elsewhere in memory. Compared to the original Sancus architecture, Sancus 2.0 also allows for the confidential loading of enclaves, protecting the confidentiality of the enclave code from the untrusted loading mechanism.

Attestation. On platforms supporting multiple enclaves, these enclaves need a way to securely link together and verify the confidentiality and integrity of the enclave they interact with. Potential use cases include a sensor network where a sensor node needs to verify that the data it receives originated from a trusted source.

Secure Communication. Secure communication is required for passing sensitive messages between an enclave and a third party such as the remote software provider. This functionality is supported by offering encryption and decryption primitives in the Sancus hardware. Sancus 1.0 supports data authentication without encryption while Sancus 2.0 supports both. Using these primitives, a software provider can additionally attest the authenticity of a software module remotely, as the module's derived key depends on its contents.

2.2. Authenticity Guarantees

IoT applications, especially event-driven distributed applications, need authentication guarantees to ensure the integrity and authenticity of the data they process. To provide these guarantees, multiple libraries and frameworks have been proposed, which are discussed next.

Authentic Execution. The authentic execution framework [11], [12] can transparently provide authenticity guarantees for distributed, event-driven applications where messages are produced and communicated in a shared, heterogeneous TEE infrastructure. An example usage is secure sensor networks [29] relying on the security guarantees provided by authentic execution. In addition to the implementation, the authors provide a formalization and proof sketch of the security guarantees [17].

Controller Area Network. Several research proposals have investigated transparently retrofitting authentication of security- and safety-critical broadcast messages on low-end Controller Area Network (CAN) buses, widely used in automotive or industrial embedded applications.

VatiCAN [14] proposes a backward-compatible protocol for secure and efficient vehicular communication. The authenticity of messages is ensured through transparently inserting and validating additional messages carrying truncated 64-bit message authentication codes (MACs) on the underlying CAN bus. LEIA [15] similarly relies on truncated MACs to provide lightweight, backward-compatible CAN authentication. Notably, LEIA includes a protocol-level formalization and security proof under the MAC unforgeability assumption.

VulCAN [13] explores the use of lightweight trusted computing technology to further secure CAN authentication, validating MACs and terminating authenticated connections inside enclaves. VulCAN re-implements both the VatiCAN and LEIA protocols on the open-source Sancus TEE architecture and critically relies on Sancus's hardware-level cryptographic unit to provide efficient, real-time-compliant vehicle message authentication, attestation, and enclave isolation.

2.3. Side-Channel Analysis on Low-End MCUs

Side channels are a well-known threat to most computing devices. They are the result of sharing resources between different processes or components. Low-end microcontrollers have fewer shared resources than commercial CPUs, usually lacking components such as caches or branch predictors, and are thus not susceptible to attacks that exploit these features. However, the absence of these shared resources renders CPU execution timings more deterministic. This in turn makes it easier for an attacker to exploit other side channels such as start-to-end timing. In the case of Sancus, any feature shared between enclaves or peripherals can be a potential side channel.

Start-to-End Timing Attacks. After the timing attack by Kocher [30] on cryptographic protocols, many more similar side channels have been discovered. In the case of timing attacks, the variable time needed for executing instructions based on the (secret) input leads to leakage [31]. Once the attacker observes the timing information, they can deduce (part of) the secret input.

Goodspeed [32] reported an example of a start-to-end timing attack on the MSP430. The serial bootstrap loader (BSL) is vulnerable due to unbalanced branches in its password comparison routine. More specifically, 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 correct password byte-by-byte. This reduces the search space and required time significantly. After breaking the BSL password, attackers can read out or flash malicious firmware on the device.

Nemesis. The first microarchitectural side-channel attack on low-end Sancus microcontrollers was Nemesis [33], a timing attack that uses interrupt logic to leak information. Sancus, like most processors, executes instructions over multiple cycles. As interrupts are only triggered after instruction retirement, an attacker can measure the time it takes for the current instruction to finish executing by measuring interrupt latency. Non-linear programs and microarchitecturally unbalanced branches are vulnerable to this attack.

TABLE 1. TIMING LEAKAGE IN THE HARDWARE-SOFTWARE STACK.

System	Library	== operator	Hardware
VRASED+, RATA, ACFA, TRAIN	✖		
VatiCAN	✖		
LEIA		✖	
VulCAN		✖	
Sancus, Authentic Execution	✖		✖

Direct Memory Access. Another microarchitectural side-channel attack on low-end microcontrollers uses direct memory access (DMA) and the shared memory buses to leak information [34], [35]. While DMA accesses cannot directly access the code or data in enclaves, their timing can leak information about the enclave's memory activity. A DMA-capable peripheral can probe the memory bus and detect contention with the enclave by measuring the latency of the DMA requests. Similar to the Nemesis attack, non-linear programs and microarchitecturally unbalanced branches are vulnerable to this attack.

Existing Mitigations. Most existing mitigations against these timing attacks use software modifications. Winderix et al. [36] and Bognar et al. [37] proposed mitigations against the Nemesis and the DMA contention attacks, respectively. These mitigations use compiler-inserted instructions to balance vulnerable branches to exhibit the same leakage, regardless of their outcome. Given that these mitigations balance branches, they also mitigate against simpler start-to-end timing attacks that exploit unbalanced secret-dependent branches such as the BSL attack [32]. An orthogonal proposal masks the Nemesis interrupt-latency leakage directly in hardware [38].

3. Leakage in the Hardware-Software Stack

In the following, we demonstrate that even on low-end microcontrollers, 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. Table 1 summarizes the systems we analyze and the non-constant-time behavior we observed in the hardware-software stack.

Threat Model. The only capability we require for our attacks is running untrusted code on the device, which is in the threat model of all systems we analyze. In addition, (open)MSP430 has a software-accessible cycle-accurate timer, suitable for measuring precise timing differences.

3.1. Standard Library Functions

We start by analyzing the security of the C and C++ standard libraries provided to developers to improve the ease of development. These libraries are understandably optimized for performance rather than security. Low-end embedded devices, often optimized for energy efficiency, rely on these libraries and their performant functions.

```
int secure_memcmp(const uint8_t *s1, const
    uint8_t *s2, int size) {
  int res = 0; int first = 1;
  for (int i = 0; i < size; i++) {
    if (first == 1 && s1[i] > s2[i]) {
      res = 1; first = 0;
    } else if (first == 1 && s1[i] < s2[i]) {
      res = -1; first = 0;
    }
  }
  return res;
}
```

Listing 1. Non-constant-time `secure_memcmp` function used in VRASED+ [2] and derived architectures [19], [22], [23].

Standard library functions like `memcmp` or `std::equal` halt the memory comparison upon finding a difference for performance reasons, creating a clear timing side channel.

Plain `memcmp`. Our analysis of open-source TEE research prototypes reveals widespread use of plain `memcmp` or `std::equal` calls. First, the aforementioned authentic execution framework [12] uses a C++ library [39] that implements the SpongeWrap authenticated encryption with associated data primitive and uses `std::equal` for the comparison of tags. 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 3.3), the non-constant-time C++ library is used in the Intel SGX and Arm TrustZone implementations. Crucially, this would allow reliable exploitation using the SGX-Step [40] single-stepping framework to deterministically count the number of instructions and, hence, the number of correct tag bytes when comparing computed and expected tags. Finally, we investigated the open-source VatiCAN [14] automotive authentication library, which similarly uses the C function `memcmp` to compare the tags in its authentication protocol, making it vulnerable to timing attacks.

Secure `memcmp`. Prior work [34] demonstrated a start-to-end timing leak in the authentication protocol of VRASED+ [2], which used a plain, non-constant-time `memcmp` function. In response, the VRASED+ authentication code [41] was changed to use a custom `secure_memcmp` function that does not terminate early on a byte mismatch, as shown in Listing 1. Crucially, we found that the new implementation is still vulnerable to a timing attack, highlighting the non-triviality of writing constant-time code. The custom `secure_memcmp` implementation even allows attackers to improve linear brute-force attacks using binary search, as detailed in Appendix A (which also includes the compiled assembly code for completeness). Currently, this `secure_memcmp` function is used in the provably secure RATA [19], ACFA [22] and TRAIN [23] hardware-software co-designs, where it may introduce application-specific security or availability concerns.

3.2. Compiler Analysis

Our second finding is that compilers for low-end microcontrollers can introduce timing leakage when comparing values of primitive C data types. Specifically, we found

```
    cmp.w    6(r1), r12
    jne      .L1
    cmp.w    r9, r13
    jne      .L1
```

Listing 2. Primitive data-type comparison of two 64-bit integers compiled using MSP430 `gcc` v14.2.0.

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

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 8- or 16-bit word-sized chucks of the data with early exit jumps.

For instance, consider the code `(uint64_t) a == (uint64_t) b`, which performs a comparison between two 64-bit integers. When compiled using a modern MSP430 `gcc` v14.2.0 with space optimizations (`-Os`), the code will be compiled into non-constant-time assembly code, shown in Listing 2. Other recent versions and optimization levels of the compiler create similar results. Notably, the `sancus-cc` compiler bundled with Sancus, based on an outdated version of LLVM/`clang` v4.0.1, does not exhibit a timing leakage. As shown in Listing 3, this code employs a linear chain of repeated `xor` instructions to perform the comparison, always checking the full 64-bit integer. Appendix B includes compiled assembly code for different compiler versions and optimization levels for popular target architectures, summarized in Table 2. Notably, these results show that even on 32-bit platforms such leakage can occur when comparing 64-bit numbers.

We found that both LEIA [15] and VulCAN [13] use `uint64_t` primitive data types to compare the computed, secret-dependent and the received, attacker-provided MAC values, using simple C equality (==) or inequality (`!=`) operators. This makes enclaves using these automotive authentication libraries vulnerable to the timing attacks described above, depending on the exact compiler version and optimization levels used.

3.3. Hardware Timing Vulnerability

Finally, even at the level of the hardware itself, timing vulnerabilities can be found. In the following, we demonstrate the existence of a subtle early-out comparison flaw in the FSM hardware logic implementing authenticated encryption for Sancus. This analysis is followed by a systematic investigation of the impact on different cryptographic primitives in different versions of Sancus and an end-to-end attack on the authentic execution framework.

Timing Leak. The cryptographic unit of Sancus is built using an FSM in hardware. This FSM controls the cryptographic unit while the SpongeWrap [27] construction is

TABLE 2. 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			✖

TABLE 3. SANCUS VERSIONS AFFECTED BY THE TAG COMPARISON LEAK (C=CONFIDENTIALITY; I=INTEGRITY; A=AVAILABILITY).

Primitive	C	I	A
sancus_unwrap	–	2.0	–
sancus_verify	–	1.0	1.0
sancus_enable	–	2.0	2.0

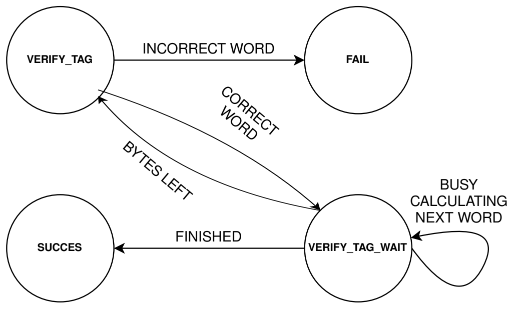

Figure 1. Part of the cryptographic FSM responsible for comparing tags. The first state is VERIFY_TAG on the top left.

used for the actual cryptographic operations. Moreover, the FSM checks 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 1, where comparison is aborted on the first word mismatch.

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 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 [11] 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 and associated plaintext data. Notably, we found that the behavior of the verify primitive for local attestation was changed from using MACs in Sancus 1.0 [24] to hashes in Sancus 2.0 [11], making the observed timing variations for verify in Sancus 2.0 secret-independent. However, since Sancus 2.0 supports the confidential loading of encrypted enclaves, the enable instruction in this version is susceptible to timing leakage. 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.

Proof-of-Concept. We developed an elementary Sancus enclave that takes an encrypted message and a corresponding tag as inputs and uses the unwrap primitive to authenticate the incoming message and decrypt its associated data. By accurately timing the execution of the vulnerable enclave, the 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 from an exponential to a linear effort. 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 garbled messages may be sufficient to carry out a practical attack.

End-to-End Attack. The authentic execution framework [11], [12], discussed in Section 2.2, has an entry point for handling input data. The entry point is called HandleInput (see Listing 8 in Appendix C) and is responsible for decrypting the payload and calling a corresponding callback function to process it. In the framework's Sancus implementation, the cryptographic unit and its vulnerable unwrap primitive are used to decrypt the payload and check the tag.

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, an attacker can construct a payload consisting of a random ciphertext and a tag. As before, the attacker can measure the time it takes to execute the HandleInput function. Based on the timing, the correct tag can be deduced, which allows the attacker to pass the message off as authentic, breaking the main security guarantee of the framework. We successfully reproduced this attack using the open-source artifacts of the authentic execution framework for Sancus.

4. Discussion and Mitigations

Timing attacks have been known for many years. Our findings provide further evidence that low-end TEEs and systems building on them are just as impacted by these vulnerabilities as high-end systems, despite the simpler hardware implementations. We argue further that the observations made in this paper can likely be extended to other low-end TEEs, since time is a shared resource on almost any programmable device.

To understand and mitigate the full impact of the side channels, developers or tools need to be aware of the timing behavior of the whole hardware-software stack. First, care should be taken when using standard libraries since they are optimized for performance and not security. Our examples show that even hand-written library functions such as the investigated secure_memcmp can be vulnerable to subtle timing leaks. Furthermore, we showed that compilers, especially when targeting lower-end platforms, can introduce unexpected timing leaks even when comparing primitive data types. Finally, even the hardware can hide timing leakage that is not detectable just by examining the program code. While in this paper we demonstrated vulnerabilities in these different layers in low-end trusted execution settings, future research could extend the scope to high-end TEEs.

4.1. Using Formal Methods

Formal methods are a powerful tool to increase confidence in the security of a system. However, they tend to defer side-channel-aware modeling for future work. Based on our results, we argue that this side-channel analysis should be conducted either as part of the formalization or as thorough inductive testing [34].

Notably, several of the vulnerabilities we demonstrated impact systems with a formal security model. The timing-dependent `secure_memcmp` function is used in formally verified attestation architectures [2], [19], [22], [23]. Furthermore, the high-level formal model for the LEIA [15] authentication library could not anticipate the concealed low-level timing effect introduced by the compiler. Finally, the hardware-level timing variations we exploit to bypass authenticity guarantees are completely invisible in the application code, the authentic execution compiler framework, and its high-level formal definition [17].

4.2. Software Mitigations

Mitigating the timing leakage introduced by the compiler or insecure libraries should be fixed in software. In all cases, comparing secret-dependent values should always happen in a constant-time fashion, for example using XOR instructions [42], as done by the `sancus-cc` compiler. To validate whether the compiler introduced any timing leakage, binary analysis tools can be used [43].

Mitigating the timing-dependent cryptographic unit in Sancus from software is more difficult, but might be necessary on legacy hardware. A possible solution is to time the execution of the cryptographic unit from the enclave and add a delay if the execution time is too short. This way, the execution time of an entry point will not differ based on the cryptographic operation. However, this mitigation needs to ensure that this software-induced padding cannot be detected and differentiated from hardware-induced delays by other means, such as interrupts [33] or the enclave's memory activity [34].

4.3. Hardware Mitigation

To patch the timing channel in the cryptographic hardware unit, we implement and evaluate two, slightly different, hardware mitigations. To eliminate the leakage, the part of the state machine responsible for the verification needs to be altered in such a way that it always takes the same amount of time to verify the tag. There are two possible ways to achieve this: adding an extra register to keep track of the result of the comparison or adding extra states. A graphical representation of the modified FSMs for the two mitigations is shown in Appendix D.

Extra Register. By adding a register to the cryptographic unit, the state machine can keep track of whether an incorrect word was found during the iterative checks. This way, the state machine can continue with the verification even if an incorrect word is found. Finally, when all words of the tag have been checked, the state machine can indicate whether the verification was successful or not based on this stored register.

TABLE 4. 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		*Crypto 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

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. Extra changes are needed in the original cryptographic unit for this solution, such as increasing the size of the buffer that keeps track of the current state.

Comparison. Both mitigations have advantages and disadvantages. To better evaluate the effects of the changes, we compared the hardware cost of the original implementation and the two mitigations. We obtained these measurements by using Vivado 2024.2 to synthesize the designs for the Kintex UltraScale+ FPGA. The results are shown in Table 4. All architectures are evaluated based on the total number of lookup 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, as changes in the cryptographic unit might have effects on other parts of the design.

Based on these results, we favor the solution of adding an extra register. It requires fewer LUTs compared to the mitigation with extra states, and even fewer than the original implementation, likely thanks 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; this is the FF used to store the result of the comparison.

5. Conclusion

This paper highlighted the importance of considering side-channel attacks in every layer of the hardware-software stack. Our comprehensive side-channel analysis uncovered the use of non-constant-time functions from the C and C++ standard libraries and even non-constant-time custom comparison functions in multiple embedded TEE research prototypes. Moreover, we demonstrated that compilers for low-end microcontrollers can introduce timing vulnerabilities in the assembly code, even for equality checks on primitive data types. We further showed that even the hardware can harbor subtle timing vulnerabilities, as evidenced by a flaw in the cryptographic unit of the Sancus TEE. Our proposed hardware and software mitigations aim to address these vulnerabilities with minimal performance impact. Finally, we argue that formal methods should be combined with thorough side-channel analysis to provide strong security guarantees.

Acknowledgment

This research was partially funded by the ORSHIN project (Horizon Europe grant agreement #101070008), the Research Fund KU Leuven, and the Cybersecurity Research Program Flanders.

References

[1] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Transactions on Privacy and Security*, vol. 20, no. 3, 2017.

[2] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *USENIX Security Symposium*, 2019.

[3] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[4] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: a security architecture for tiny embedded devices," in *EuroSys*, 2014.

[5] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. C. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, 2018.

[6] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Design Automation Conference (DAC)*, 2015.

[7] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, "Secure interrupts on low-end microcontrollers," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014.

[8] M. Bognar, C. Magnus, F. Piessens, and J. Van Bulck, "Intellectual property exposure: Subverting and securing intellectual property encapsulation in Texas Instruments microcontrollers," in *USENIX Security Symposium*, 2024.

[9] Texas Instruments, "MSP code protection features," https://www.ti.com/lit/an/slaa685/slaa685.pdf, 2015.

[10] Microchip, "Codeguard security: Protecting intellectual property in collaborative system designs," http://ww1.microchip.com/downloads/en/DeviceDoc/70179a.pdf, 2006.

[11] J. Noorman, J. T. Mühlberg, and F. Piessens, "Authentic execution of distributed event-driven applications with a small TCB," in *Security and Trust Management - International Workshop (STM)*, 2017, pp. 55–71.

[12] G. Scopelliti, S. Pouyanrad, J. Noorman, F. Alder, C. Baumann, F. Piessens, and J. T. Mühlberg, "End-to-end security for distributed event-driven enclave applications on heterogeneous tees," *ACM Trans. Priv. Secur.*, vol. 26, no. 3, 2023.

[13] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient component authentication and software isolation for automotive control networks," in *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[14] S. Nürnberger and C. Rossow, "vatican - vetted, authenticated CAN bus," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2016, pp. 106–124.

[15] A. Radu and F. D. Garcia, "Leia: A lightweight authentication protocol for CAN," in *European Symposium on Research in Computer Security (ESORICS)*, 2016, pp. 283–300.

[16] S. Vanderhallen, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Robust authentication for automotive control networks through covert channels," *Computer Networks*, vol. 193, 2021.

[17] J. Noorman, J. T. Mühlberg, and F. Piessens, "Authentic execution of distributed event-driven applications with a small TCB [supplementary materials]," Tech. Rep., 2017. [Online]. Available: https://downloads.distrinet-research.be/software/sancus/stm17/secargument.pdf

[18] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel SGX," *CoRR*, vol. abs/2006.13598, 2020.

[19] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[20] L. Simon, D. Chisnall, and R. 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.

[21] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, 2016.

[22] A. Caulfield, N. Rattanavipanon, and I. De Oliveira Nunes, "ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation," in *USENIX Security Symposium*, 2023.

[23] P. Frolikov, Y. Kim, R. T. Prapty, and G. Tsudik, "TOCTOU resilient attestation for iot networks (full version)," *CoRR*, vol. abs/2502.07053, 2025.

[24] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *USENIX Security Symposium*, 2013.

[25] O. Girard, "openmsp430," 2017. [Online]. Available: https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf

[26] Texas Instruments, "MSP430x1xx family: User's guide," https://www.ti.com/lit/ug/slau049f/slau049f.pdf, 2006.

[27] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Duplexing the sponge: Single-pass authenticated encryption and other applications," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, vol. 7118, 2011, pp. 320–337.

[28] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, "SPONGENT: The design space of lightweight cryptographic hashing," *IEEE Trans. Computers*, vol. 62, no. 10, pp. 2041–2053, 2013.

[29] J. Pennekamp, F. Alder, R. Matzutt, J. T. Mühlberg, F. Piessens, and K. Wehrle, "Secure end-to-end sensing in supply chains," in *IEEE Conference on Communications and Network Security (CNS)*, 2020, pp. 1–6.

[30] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology - CRYPTO*, vol. 1109, 1996.

[31] C. Rebeiro, D. Mukhopadhyay, and S. Bhattacharya, *Timing Channels in Cryptography: A Micro-Architectural Perspective*. Springer Publishing Company, Incorporated, 2014.

[32] T. Goodspeed, "Practical attacks against the msp430 bsl," in *Twenty-Fifth Chaos Communications Congress*, 2008.

[33] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[34] M. Bognar, J. Van Bulck, and F. Piessens, "Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[35] C. Rodrigues, D. Oliveira, and S. Pinto, "Busted!!! microarchitectural side-channel attacks on the MCU bus interconnect," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[36] H. Winderix, J. T. Mühlberg, and F. Piessens, "Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[37] M. Bognar, H. Winderix, J. Van Bulck, and F. Piessens, "Microprofiler: Principled side-channel mitigation through microarchitectural profiling," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.

[38] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Securing interruptible enclaved execution on small microprocessors," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 3, 2021.

TABLE 5. EXECUTION TIME OF THE SECURE_MEMCMP FUNCTION FOR A PARTIALLY CORRECT BUFFER SIMULATED ON OPENMSP430.

Correct bytes	0	1	2	3	4	5
Clock cycles	60	76	95	114	133	138

[39] "Authenticexecution/spongent-cpp-rs - github." [Online]. Available: https://github.com/AuthenticExecution/spongent-cpp-rs/blob/main/spongent.cpp

[40] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[41] "Vrased+ - github." [Online]. Available: https://github.com/sprout-uci/vrased-plus/blob/main/vrased/sw-att/wrapper.c

[42] "twisted-pear/mcfd - github." [Online]. Available: https://github.com/twisted-pear/mcfd/blob/master/src/common/crypto_helpers.c

[43] S. Pouyanrad, J. T. Mühlberg, and W. Joosen, "Scf^{msp}: static detection of side channels in MSP430 programs," in *International Conference on Availability, Reliability and Security (ARES)*, 2020.

Appendix A. secure_memcmp Attack

The secure_memcmp function (Listing 1) is used to compare two buffers. However, unlike the name suggests, this function is timing-dependent. In addition, an attacker can exploit this timing dependency to improve linear brute-force attacks using binary search, since a guessed byte that is smaller than the correct byte will take longer to compare than a guessed byte that is bigger than the correct one. Table 5 shows the execution time of the secure_memcmp function for a partially correct buffer, as measured on the openMSP430 core for the compiled assembly code in Listing 4. Note that different compilers and optimization levels generate similarly vulnerable assembly code for the secure_memcmp function of Listing 1.

Appendix B. Compiler Analysis Code

We provide two assembly code snippets showing how comparison code at the C level (Listing 5) can be compiled in timing-dependent and timing-independent ways. The vulnerable code is compiled using MSP430 gcc 14.2.0 (Listing 6) and optimized for space usage (-Os), while the safe code is compiled using sancus-cc (Listing 7).

Appendix C. HandleInput Authentic Execution Code

Listing 8 provides the HandleInput code inserted by the Sancus compiler to transparently handle authentic-execution entry points as discussed in Section 3.3. The vulnerable sancus_unwrap call is on line 29.

Appendix D. Hardware Mitigations FSMs

The modified FSMs for the two hardware mitigations discussed in Section 4.3 are shown in Figures 2 and 3.

```
secure_memcmp:
    push.w  r4
    mov.w   r1, r4
    push.w  r11
    push.w  r10
    push.w  r9
    mov.w   #0, r12
    cmp.w   #1, r13
    jl   .LBB0_9
    mov.w   #1, r11
.LBB0_2:
    cmp.w   #1, r11
    jne .LBB0_8
    mov.b   0(r14), r11
    mov.b   0(r15), r10
    cmp.b   r10, r11
    jhs .LBB0_5
    mov.w   #0, r11
    mov.w   #1, r12
    jmp .LBB0_8
.LBB0_5:
    mov.w   #-1, r9
    cmp.b   r11, r10
    jlo .LBB0_7
    mov.w   r12, r9
.LBB0_7:
    cmp.b   r11, r10
    mov.w   r2, r11
    and.w   #1, r11
    mov.w   r9, r12
.LBB0_8:
    add.w   #1, r14
    add.w   #1, r15
    add.w   #-1, r13
    cmp.w   #0, r13
    jne .LBB0_2
.LBB0_9:
    mov.w   r12, r15
    pop.w   r9
    pop.w   r10
    pop.w   r11
    pop.w   r4
    ret
```

Listing 4. Listing 1 compiled using sancus-cc based on LLVM/clang v4.0.1 and optimized for space usage (-Os).

```
void cmp_secret64(uint64_t a, uint64_t b) {
    if (a == b) helper_func();
}
void cmp_secret32(uint32_t a, uint32_t b) {
    if (a == b) helper_func();
}
void cmp_secret16(uint16_t a, uint16_t b) {
    if (a == b) helper_func();
}
```

Listing 5. Code snippet comparing integers of different sizes.

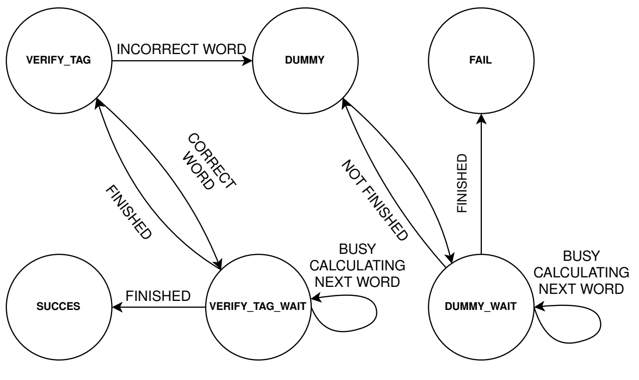

Figure 2. Cryptographic FSM with extra states to delay the final result.

```
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 6. Listing 5 compiled using MSP430 gcc 14.2.0 and optimized for space usage (-Os).

```
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 7. Listing 5 compiled using sancus-cc based on LLVM/clang v4.0.1.

```
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 8. HandleInput entry point generated by Sancus authentic execution framework.

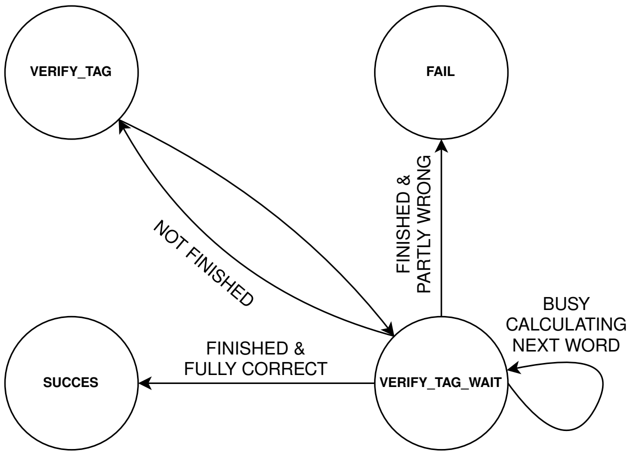

Figure 3. Cryptographic FSM with an extra register to keep track of the result of the comparison.