Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures

Marton Bognar
marton.bognar@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Jo Van Bulck
jo.vanbulck@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Frank Piessens
frank.piessens@kuleuven.be
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

***Abstract*—The security claims of a system can be supported or refuted by different kinds of evidence. On the one hand, *attack research* uses empirical, experimental, inductive methods to refute security claims. If motivated and competent attackers do not succeed in breaking a specific security property, this provides some support (but no definite proof) that the system is secure.**

On the other hand, *formal methods* use mathematical, deductive methods that can prove the security of a *model* of the system. The process of constructing a proof can uncover vulnerabilities that can then be fixed. The use of formal methods can be very powerful and is attractive because it seems to provide irrefutable evidence of security. However, that evidence applies only to the mathematical model, not to any actual system, and, hence, it is important to understand the gap between the model and the real-world system.

In this paper, we present a case study that examines this gap for two embedded security architectures that use formal methods to prove their security properties. Despite strong formal evidence for security, we discover numerous attacks against the implementations, all of which falsify proven security properties. These attacks range from exploiting simple programming errors to a novel DMA-based side-channel attack. The simple attacks demonstrate that the construction of systems and proofs is error-prone, while some of the more sophisticated attacks serve as examples to show that formal methods alone can never guarantee the security of a real-world system.

From our case study, we also distill actionable guidelines on how to provide stronger evidence for the security of a system.

I. Introduction

How can we be sure that a computer system satisfies some security property of interest? How can potential users be convinced that the system's security can be relied upon? How can we provide scientific evidence for the security of a system?

These questions were already asked 35 years ago [1] and are still at the heart of security research and practice today [2]–[4]. Still, to date, there is no consensus on what the best approach is to provide such evidence of security.

Providing scientific evidence can be done using *inductive* or *deductive* methods [3], [4]. *Induction* uses experiments and observations on a real system to provide empirical evidence for system properties. *Deduction* uses mathematical methods to establish definite truths about a formal system model. Both inductive and deductive methods can be applied with varying levels of *rigor*. For instance, experiment design and methodology impact the rigor of inductive methods. Deduction can use less rigorous informal arguments, or very rigorous (even machine-checked) formal proofs.

Attack research in security is an *inductive* method: it involves producing attacks (breaking the security properties) on real-world systems. If even after extensive effort, no successful attacks are found, we may increase our belief that the system is secure. Penetration testing and bug bounty programs can help ensure that sufficient effort is invested. The strengths of attack research include that it is applicable to complex systems, even in a black-box manner, and that it can provide strong, indisputable evidence for the *insecurity* of a particular system. In fact, over the past decades, attack research papers have regularly uncovered severe vulnerabilities in complex, widely used systems, including processors [5] or popular web applications [6]. Attack research can provide deep insight into the root cause of underlying vulnerabilities and the effort and conditions required to successfully exploit them [4], [7]. As a result, attack research has proven indispensable for guiding effective countermeasure design, as evident by the ongoing arms race in – among others – the memory-safety domain [8].

An important weakness, however, is that attack research remains an inductive method, and can, hence, fundamentally not *guarantee* the security of a system. That is, even if no attacks have been found after extensive analysis, one can never state with certainty that attacks will not be found in the future. It is also hard to quantify the security assurance obtained from the observation that no attacks have been found within a specific time window by a specific team of attack researchers.

Formal security proof, on the other hand, is a *deductive* method: its strength lies in using mathematical methods to provide strong evidence that a formal system *model* satisfies specific security properties. This model can be abstract and constructed separately from the system, but it is also possible to have a stronger connection between the two: source code in a programming language or hardware description language can be seen as a system model by defining an operational semantics, and, hence, formal methods can *prove* the absence of vulnerabilities in such code. Under the assumption that the real system executes the code as defined in the semantics, this rules out entire classes of attacks on the real-world system.

But important weaknesses of the formal approach include limited scalability, and the need for simplifying assumptions to keep system models analyzable and understandable. Furthermore, whether a real-world system satisfies certain assumptions remains a claim that cannot be proven by deductive

methods alone. In other words, formal methods make it possible to better focus attack efforts (focus on invalidating assumptions made by the formal model), but can never render attack effort unnecessary.

Some subfields of the broad security research field have developed certain maturity in providing evidence of security. In cryptography, competitions leading to standardized algorithms, such as TLS 1.3 [9] and post-quantum cryptography [10], are examples where both inductive and deductive techniques have been used to achieve sufficient security. Other subfields, systems security in particular, are still far from a consensus on how to provide evidence. Recent papers appearing in the literature take widely different approaches for producing their security arguments.

Case-study approach: The main objective of this paper is to showcase the gap in systems security between formal, deductive-only approaches and real-world security. To fit this within the scope of a single paper, we follow a case study-driven approach. Many different architectures could be selected for such a case study; for our case study, we selected small embedded systems that *(i)* have been published at recognized security conferences, *(ii)* support trusted-execution related functionality, *(iii)* are accompanied by formal (deductive) evidence proving their security properties, and *(iv)* have an open-source implementation.[1] We chose small embedded systems to make an in-depth systematic analysis at our scale possible, while trusted execution was chosen for being an emerging security paradigm that has also received substantial attention from a formal perspective [11]–[15].

Several systems satisfying these criteria have been published over the past years. The two systems we examine are both part of mature research projects with multiple collaborators and a rich publication history, both by the original designers [14]–[22] and by outside researchers [23], [24]. One is an extension [14] to the Sancus system [16] that offers secure interruptible enclaves; and one is VRASED [15], a remote attestation framework with a number of extensions [19]–[22] building on its security properties.

Contributions: In our case study, we provide the following novel contributions:

- We present a significant number of attacks that directly falsify formally proven security claims in recently published, peer-reviewed papers, which have not been shown insecure before. These attacks are implemented and validated to work on the provided implementations with no modifications (unless explicitly mentioned).
- We describe a novel DMA-based side-channel attack that is effective against these systems and is interesting in its own right.
- As a more indirect contribution, we provide evidence for the value of combining inductive and deductive methods (attack research and formal proofs) in systems security, share the lessons learned from our case study, and provide guidelines to strengthen the security claims of a system, supported by examples in the paper.

We want to emphasize two important points right from the start: first, our attacks range from simple programming errors to more advanced side-channel attacks. It can be tempting to dismiss the more straightforward attacks as simple implementation oversights. However, if a paper claims to avoid implementation bugs, e.g., because it "*uses a verified cryptographic software implementation and combines it with a verified hardware design to guarantee correct implementation of RA security properties*" [15], then even simple oversights falsify the claims in the paper. Perhaps more importantly, the more advanced attacks demonstrate that formal methods alone can *never* guarantee the complete security of a system.

Second, due to the nature of our paper, we have to be critical of the systems in the case study. This should not be mistaken as questioning the value or quality of these papers. In fact, we consider these papers to be comparable or better than others in the literature. The papers were selected based on the positive qualities of providing open-source implementations, precise security claims, and detailed security proofs.

Reproducibility and open-source artifacts: To ensure the reproducibility of our findings and to encourage further research, we made all of our experiments open-source at https://github.com/martonbognar/gap-attacks. This repository, furthermore, includes a continuous integration framework that provides a fully reproducible build environment and reference output for all our attacks, executed via a cycle-accurate `iverilog` simulation of the systems' respective openMSP430 designs.

II. Background and Selected Architectures

A. $Sancus_V$: Provably secure interruptible enclaves

Sancus [16] is a lightweight trusted execution environment (TEE) [25] for embedded devices with a zero-software trusted computing base. More specifically, the open-source Sancus research prototype extends the openMSP430 processor architecture [26] with a hardware-level, program-counter-based memory access control mechanism that isolates protected software modules, called *enclaves*, against all other software on the platform, including the operating system and other enclaves. Sancus has seen a line of follow-up work [17], [18], [24] that modifies or extends the functionality offered by the base architecture. In this paper, we focus on $Sancus_V$ [14], the only such system that offers formal security guarantees. The journal version [27] of the $Sancus_V$ paper provides a detailed outline of the formal model and security proof.

1) Interrupt latency attacks: Following similar embedded TEE designs [28]–[30], recent upstream versions [31] of Sancus support interruptible enclaves, where the processor's interrupt logic is extended to securely save CPU registers within the enclave before vectoring to the untrusted operating system. However, while architecturally sound, this scheme has been shown to enable subtle microarchitectural side-channel timing leakage through the Nemesis attack [32]. Concretely, on openMSP430 – as in most other processor architectures –

[1] In this paper, the implementation of a system refers to its source code, not a physical realization.

interrupt requests are only handled after the current instruction has finished executing. This means that by precisely measuring the time it takes for an interrupt to be handled, a Nemesis attacker can retrieve the execution length (number of clock cycles) of the interrupted enclave instruction.

Interestingly, while start-to-end timing attacks have long been known to enable the leakage of secret information [33], [34], Nemesis attackers can exploit much more fine-grained, instruction-granular timing measurements that can even leak secrets from branches with balanced start-to-end timings. Listing 1 shows a minimal example of a password comparison routine [32], [34], that is carefully padded with `nop` instructions to exhibit balanced start-to-end execution times, yet remains vulnerable to an advanced Nemesis interrupt latency attacker.

```
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 1. Password comparison enclave (excerpt, based on MSP430 BSL).

Because execution lengths are different for *individual* instructions in the two branches, the attacker can determine which branch was executing at the time of the interrupt. Figure 1 displays the observed latencies when consecutively interrupting every instruction for the two branches of the enclave in Listing 1. Based on the observed interrupt latency traces, the attacker can determine whether the comparison for an individual password byte succeeded (thereby reducing a brute-force attack from an exponential to a linear effort).

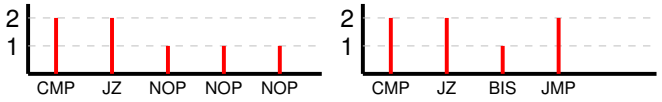

Fig. 1. Interrupt latency traces for the two branches of Listing 1.

2) Interrupt latency padding defense: A recent extension [14] to Sancus, referred to as Sancus$_V$ from here on, implements secure interruptible enclaves. It also formally proves that this modification does not introduce any new information leakage, including from side channels.

Sancus$_V$ defends enclaves against Nemesis attackers by implementing a carefully crafted, double padding mechanism during interrupt handling. At a high level, the hardware-level padding defense first makes sure that the observable number of clock cycles between issuing an interrupt and the execution of the interrupt service routine (ISR) is always the same. To this end, while in enclave mode, the processor will start an internal counter once an interrupt request arrives in cycle t_1. As soon as the interrupt request is ready to be handled, i.e., after the currently executing enclave instruction has finished in cycle t_2, the processor will delay the execution of the ISR until the internal counter register reaches a specified value T. The amount of padding cycles added can, hence, be expressed as $p_1 = T - (t_2 - t_1)$. Crucially, when T is carefully chosen to be larger than or equal to the maximal execution length of an openMSP430 instruction, an attacker will always observe a constant interrupt latency of T cycles.

A secondary type of padding is, furthermore, required to protect against advanced resume-to-end attackers that measure the remainder of the interrupted enclave execution time (which has now been shortened with the length of the interrupted instruction). This complementary amount of padding cycles can be expressed as $p_2 = (t_2 - t_1)$ and is automatically added when resuming a previously interrupted enclave via the `reti` (return from interrupt) instruction.

3) Formalization outline: We provide a schematic of Sancus$_V$'s system and attacker model in Figure 2. The processor core is trusted, and its behavior is fully modeled as a small-step operational semantics. A peripheral device, under the control of the attacker, is connected to the system. In the model, the functionality of the peripheral is abstract: it can measure time with a clock cycle granularity and issue cycle-accurate interrupts, and it can be configured through specially modeled `IN/OUT` instructions. On the software level, there is one trusted, but unmodeled and unverified enclave. This is the enclave whose isolation the system is protecting. The attacker is assumed to have control over all other software running on the platform, including the operating system and ISRs.

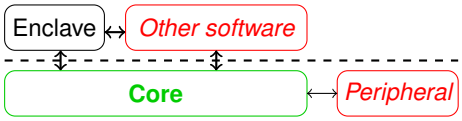

Fig. 2. Sancus$_V$ overview with components that are trusted and verified (green, bold), trusted but unverified (black), and untrusted (red, italic).

For its security definition, Sancus$_V$ uses the notion of *contextual equivalence*. The *context* of an enclave contains everything that the attacker controls: passed parameters, contents of unprotected memory, the peripheral, etc. Two enclaves are contextually equivalent if there exists no context that allows the attacker to distinguish them. The security claim of the system is that two enclaves are contextually equivalent on the version of Sancus without interrupts if and only if they are contextually equivalent on Sancus$_V$. Intuitively, this means that two enclaves that cannot be distinguished without interrupts also cannot be distinguished after interrupts are introduced. This property is shown to hold using a pen-and-paper style proof. An unverified prototype implementation of the model is provided as an extension to the original Sancus architecture [16], based on the openMSP430 core.

B. VRASED: Verifiable remote attestation

VRASED [15] is a remote attestation (RA) [35], [36] framework that can calculate cryptographically strong evidence about the integrity of untrusted software running on the system. The open-source VRASED research prototype is also based on the openMSP430 core. We limit the description here to the original architecture but introduce multiple recently published derived architectures [19]–[22] in Appendix A that

strongly rely on VRASED's security arguments as the basis of their own. Furthermore, recent work [23] has reimplemented VRASED's security monitor on a bare-metal microprocessor. However, this result makes extensive modifications to the VRASED implementation and proofs, and is not open-source, so it is not further analyzed in this paper.

1) Remote attestation architecture: The security-related functionality of the system is implemented by two separate components, one at the hardware and one at the software level.

The trusted software component, SW-Att, is responsible for calculating an HMAC signature of the untrusted software using a secret key. SW-Att is stored in immutable ROM and is partially verified: it includes the HMAC function from the formally verified HACL* library [37]. SW-Att is trusted software with access to an exclusive stack XS and secret key K, which are isolated by the processor from all other software.

The fully verified external hardware module, HW-Mod, is connected to selected signals from the openMSP430 core and it monitors security violations. Particularly, HW-Mod monitors *(i)* the program counter, *(ii)* memory addresses read or written by the core, *(iii)* interrupts, and *(iv)* direct memory access (DMA) requests by untrusted peripherals. At a high level, HW-Mod enforces security invariants, including (P1) access control of the key, (P2) no key leakage through memory, and (P3) secure reset that cleans secrets. Whenever any of the monitored signals indicate a deviation from these security invariants, HW-Mod resets the system.

2) Formalization outline: We provide a schematic view of the VRASED architecture in Figure 3. At the software level, the trusted SW-Att component consists of an unverified wrapper, manually written in C, which invokes the formally verified HACL* library [37] to compute an HMAC over the desired memory region using the secret key. VRASED relies on the existing HACL* proofs [37] for functional correctness, memory safety, secret-independent timing behavior, and deterministic stack memory usage of the cryptographic HMAC primitive. HACL* is implemented and verified in the F* programming language, which is subsequently translated into readable C code with a proof that the translation preserves correctness [38]. To finally obtain executable assembly code, VRASED relies on the standard and unverified `msp430-gcc` compiler [39], which is explicitly trusted to *(i)* preserve semantics, and *(ii)* clean all registers before exiting a function.

At the hardware level, only HW-Mod has been formally verified to preserve certain security invariants. To provide the actual computation infrastructure for SW-Att, VRASED builds on a slightly modified openMSP430 core, which is trusted to adhere to several assumptions (cf. Appendix B), but its precise function is not modeled or verified. The attacker has, furthermore, complete control over a peripheral device that is capable of issuing DMA requests to the core.

The designers show that VRASED's remote attestation is *(i)* sound, i.e., the HMAC is calculated from the memory contents and a challenge; and *(ii)* secure, i.e., an attacker can only forge an HMAC output that does not correspond to the memory contents with a low probability. Verification is

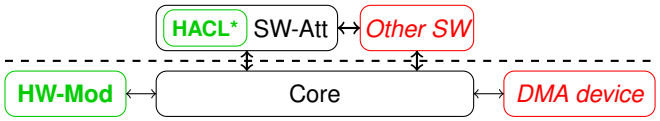

Fig. 3. VRASED overview with components that are trusted and verified (green, bold), trusted but unverified (black), and untrusted (red, italic).

carried out "for all trusted components, including hardware, software, and the composition of both, all the way up to end-to-end notions for RA soundness and security" [15]. The state machine model of HW-Mod is directly derived from the hardware implementation's Verilog files, and this model is proven to satisfy the needed security invariants specified as linear temporal logic (LTL) rules. The properties of SW-Att are manually modeled based on the security properties of the underlying HACL* library.

III. Methodology and Attack Techniques

A. System and attacker model

To visualize the different system components that are involved in the studied security arguments, Figure 4 presents an abstract model that is general enough to cover both architectures, yet detailed enough to offer a systematic overview of the different components and their interactions.

Fig. 4. Abstract machine model with components that are trusted and verified (green, bold), trusted but unverified (black), and untrusted (red, italic).

Trusted components are indicated in black and are at least partially verified (green, bold). Adversary interactions are captured by showing which components of the system are assumed to be under direct attacker control (red, italic). In line with the attacker models of the systems we studied, we assume a powerful adversary who can execute arbitrary software on the device and may additionally interact with untrusted peripherals that can be connected to the core. Physical hardware attacks, e.g., electromagnetic probes, remain out of scope.

B. Scope of our analysis

Both systems we analyze mainly rely on deductive claims to provide evidence for their security. To a large extent, the deductive arguments are high quality (sometimes even machine-checked), and we do not spend effort finding flaws in rigorous mathematical arguments. However, we do look for occurrences of less rigorous deductive reasoning. For instance, rigorous proofs about separate parts of the system are sometimes combined very informally to claim a property of the full system.

In addition, as explained in the introduction, even a perfect deductive argument can never guarantee the absence of attacks on the real system. Hence, the main effort of our analysis

systematically tries to find attacks that falsify the claimed security properties on a real-world instantiation of the systems. We focus our efforts on trying to invalidate assumptions regarding the system or the attacker in the deductive security arguments. Any behavior of the real system that breaks an assumption of the formal model is a potential vulnerability.

Finally, we assume that the desired security properties have been stated correctly. It is certainly possible to have errors or oversights in the statement of a security property (e.g., it should have included availability, not only confidentiality or integrity), but these are out of scope for our analysis.

C. Research methodology

1) Identifying falsifiable assumptions about system behavior: Both papers build their deductive arguments on assumptions about the system. In the case of VRASED [15], the authors formulate a set of 7 explicit assumptions that reportedly encapsulates all the assumptions placed on the functionality of the trusted processor and compiler (cf. Appendix B). In the case of Sancus_V [14], the pen-and-paper definition of the operational semantics introduces the main formal assumption: that the real system complies with that definition. The Sancus_V paper decomposes this complex assumption into simpler ones that are easier to falsify by explicitly documenting the main simplifications coded into the operational semantics throughout the text. It is, furthermore, explicitly stated that violating these assumptions in an implementation voids the proof.

The first step in our methodology is to collect these assumptions. We go through the papers to collect assumptions mentioned explicitly, and we check the deductive arguments to see if they rely on additional implicit or hidden assumptions. At this step, our objective is to compose a list of possibly falsifiable assumptions that can later be empirically tested.

To identify assumptions that are likely falsifiable, we manually scrutinize the formal model and the corresponding openMSP430-based real-world implementation. This step is enabled by the fact that both the model and the source code for Sancus_V and VRASED are publicly available.

Following standard security analysis best practices [40], we focuse our analysis on the interfaces between different trust domains and the assumptions that components make about interactions over these interfaces. The conceived machine model (Figure 4) makes this task easier, as it already shows the different interactions we have to focus on: how the peripheral device communicates with the core, how untrusted software interacts with trusted software, or whether the core's defenses can be bypassed by executing malicious untrusted code.

The final outcome of this step is a list of assumptions to validate. In this paper, we only report the assumptions that we managed to falsify in the next step, as only these represent potential vulnerabilities.

2) Validating the implementation: For each assumption identified, we then *(i)* validate whether the assumption holds in the real system, and *(ii)* if not, determine whether the resulting mismatch between the model and the real system can be exploited (i.e., whether and how we can use it to break the claimed security properties of the system).

Note that this is inductive research by nature: we try to falsify assumptions, and if we succeed, we have evidence that the assumption is flawed. However, even if we do not succeed in falsifying an assumption, we can never be sure that it holds: more effort might still lead to a counterexample.

This step involves a systematic code review of the relevant parts of the Verilog code of the Sancus_V and VRASED hardware implementations, the source code of the software (in case of VRASED), as well as extensive empirical testing with carefully chosen attacker code or input values.

The step from breaking an assumption to breaking a security property relies both on attack expertise and experience, as well as on the analysis of the deductive argument (i.e., how does the security proof rely on the – broken – assumption). We mark a broken assumption as exploitable once we have a working proof-of-concept attack running on the real system.

The outcome of this step is a list of falsified assumptions, an indication of their exploitability, and if exploitable, a proof-of-concept attack. Note that, in line with the setup used by Sancus_V and VRASED, we conducted all attack experiments via a cycle-accurate `iverilog` simulation of the openMSP430 core.

3) Exploiting missing attacker capabilities: In a third step, we focus on the assumptions made about the attacker. These are significantly harder to validate: assumptions about the behavior of the system itself can be validated by *running* the system since a full implementation is available. But assumptions about the attacker cannot be validated this way: essentially, we must determine whether the formal attacker model used in the paper adequately captures the informal attacker model (i.e., remote code execution on the device, including control over untrusted DMA peripherals, but no physical access, cf. Section III-A). For any attack we find against the real system that is not captured by the formal attacker model, one should ask: is the attack *intentionally* out of scope, or is it a shortcoming of the formal attacker model? Our approach in this paper is to err on the side of security: in case of doubt, we report the attack and consider the formal attacker model incomplete. Of course, attacks that are *clearly* out of scope (e.g., physical attacks that require the attacker to open the device) are not reported or even investigated.

For this step, we use domain expertise about openMSP430 and known attacks from the literature to identify potential attacks abusing features that are not modeled.

This is also inductive research: we try to find attacks that are missed, and if we succeed, this is evidence that the formal attacker model is incomplete. However, even if we do not succeed, there might still be attacks that we overlooked.

The outcome of this step includes both the identification of unmodeled attack capabilities and the development of proof-of-concept attacks using these capabilities. The attack techniques used can originate from the literature or can be novel in themselves (cf. Section IV).

4) Exposing deductive errors: Finally, the less rigorous parts of the deductive arguments in both papers are reviewed, and reasoning errors are investigated to see if they can invalidate the security properties of the system.

The outcome of this step is a list of errors in the formal proof. Since these are errors in a strictly deductive argument, in principle, no empirical validation is required to show the error. However, we still illustrate potential attacks to show that the deductive error leads to an actual violation of security.

D. Attack classification

We classify the attacks found using the above methodology into the following three categories:

1) **Implementation/model mismatches:** Attacks that are successful on the implementation and can be represented in the model, but fail there. This implies a disconnect between the formal model and the implementation, either of which might be considered incorrect depending on the informal description of the system.
2) **Missing attacker capabilities:** Attacks that are successful on the implementation but cannot be represented in the formal model due to missing features or components.
3) **Deductive errors:** Attacks that can be represented and are successful within the formal model. This implies a mistake in the formalization itself, i.e., a flaw in the proof.

IV. A novel DMA contention side channel

In this section, we introduce a novel DMA-based side-channel attack effective on the openMSP430 platform, on which both Sancus_V and VRASED are based. This attack requires the attacker to have control over a DMA-capable peripheral connected to the system. More precisely, the attacker needs to be able to read and write the signals that are exposed to the peripheral from the core. This could happen either by plugging in a custom untrusted peripheral; or by compromising the firmware of an already connected sophisticated peripheral, constituting a fully remote attack. In this section, we briefly introduce the idea behind the attack, whereas we will expand on its impact on the studied systems in the following sections.

A. Security concerns with DMA

On systems with no security measures, DMA requests can access the entire memory space, thus interfering with sensitive processes running on the system [41]. Other researchers have abused DMA to bypass improperly configured I/O memory management unit protection and access protected memory regions [42]–[44]. Even without direct access to secrets, DMA has been used as a side channel to facilitate other attacks, e.g., analyzing write access patterns using memory snapshots [45] or sampling analog-digital converter data [46] to reconstruct CPU activity.

In security architectures, DMA is usually more restricted. On VRASED [15] and the upstream version of Sancus [31], DMA requests are not allowed to access any memory that belongs to protected software. The policy is the same on high-end security architectures, for instance on Intel SGX [47].

B. Attack idea

The key idea of the attack is to measure subtle timing delays arising from contention between an untrusted DMA device and the trusted CPU when accessing the shared memory bus.

Transmitting or exfiltrating data through different components of the main memory unit of an architecture is a lively research field [48]–[51], but to the best of our knowledge, no previous attacks utilized side-channel timing differences of DMA requests to reconstruct the memory accesses of a protected program running on the CPU.

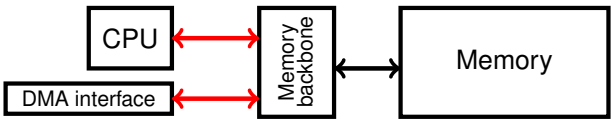

Fig. 5. Memory bus contention between CPU and DMA on openMSP430.

On openMSP430, the CPU and DMA-enabled devices are connected to the same memory bus through the memory backbone, as shown in Figure 5. One memory request can be served per clock cycle. In case of concurrent memory accesses within the same cycle, by default the openMSP430 memory backbone gives priority to the CPU, delaying any outstanding DMA requests. This resource contention can be used to infer the exact, cycle-accurate timing of memory accesses by the CPU. More specifically, by issuing a DMA request to unprotected memory and measuring if the request takes longer than one cycle to complete, a malicious peripheral can infer whether the memory bus was used by the CPU in that specific cycle.

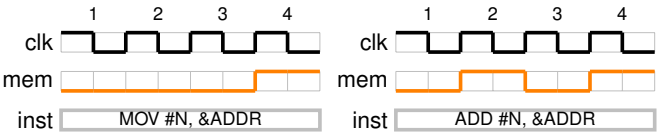

Fig. 6. Memory traces of `mov` and `add`.

To provide a quick glance into the details of the attack, consider the extracted memory traces presented in Figure 6. This figure shows the execution of the `mov` and `add` instructions with the same parameters. The execution length of both these instructions is 4 cycles[2], making them indistinguishable to an instruction-granular Nemesis attacker (cf. Section II-A1). Our DMA side channel, however, can capture the memory accesses of the CPU at a cycle-level granularity *within* the execution of both instructions. Hence, we can distinguish the instructions in Figure 6, based on whether a CPU memory access is detected (i.e., the DMA request is delayed) in the second cycle or not.

V. Security analysis of Sancus_V

A. Identifying falsifiable assumptions

As explained above, the security argument for Sancus_V [14] relies on the main assumption that the implementation follows

[2]The execution length of a given instruction also depends on its operands. Later, we will see `mov` and `add` instructions with different execution lengths.

the operational semantics defined in the pen-and-paper model. Because this assumption is very complex and difficult to falsify, we decomposed it into multiple sub-assumptions. We list the ones that we falsified and found to be exploitable in the top part of Table I. The bottom part lists exploitable features that were not modeled in Sancus$_V$. Importantly, we did not find any deductive errors in the Sancus$_V$ proof.

TABLE I. List of falsified and exploitable assumptions found in Sancus$_V$. IM = Implementation-model mismatch; MA = Missing attacker capability.

IM	V-B1	Instruction execution time does not depend on the context.
	V-B2	The maximum instruction execution time is $T = 6$.
	V-B3	Interrupted enclaves can only be resumed once with `reti`.
	V-B4	Interrupted enclaves cannot be restarted from the ISR.
	V-B5	The system only supports a single enclave.
	V-B6	Enclave software cannot access unprotected memory.
	V-B7	Enclave software cannot manipulate interrupt functionality.
MA	V-C1	Untrusted DMA peripherals are not modeled.
	V-C2	Interrupts from the watchdog timer are not modeled.

B. Validating the implementation

1) Variable instruction length following `reti`: In the Sancus$_V$ model, instructions always take the same number of clock cycles to execute, independent of the previously executed instruction.

a) Broken assumption: We found that, in the real-world openMSP430 implementation, non-jump instructions executing after a `reti` instruction take an extra cycle to execute. To make matters worse, this extra cycle is also added to the interrupt handling logic if it follows a `reti` instruction.

b) Attack: This subtle microarchitectural effect can clearly be abused to differentiate two otherwise contextually equivalent enclaves: $E_1 = \{$`add; nop; nop`$\}$ and $E_2 = \{$`nop; nop; jmp`$\}$, where `nop` normally takes 1 cycle, and `add` and `jmp` take 2 cycles. When interrupting both enclaves after two clock cycles, the first `nop` in E_1 gets an extra cycle, while `jmp` in E_2 does not. Hence, E_1's resume-to-end execution time will be one cycle longer compared to E_2.

Beyond this specific example, we experimentally showed that *any* two instructions with different execution lengths can be differentiated, even when E_1 and E_2 do not contain `jmp` instructions. This attack scheme is demonstrated in Figure 7, where we can detect whether the very first executed instruction was a two-cycle `add` in E_1 or a one-cycle `nop` in E_2.

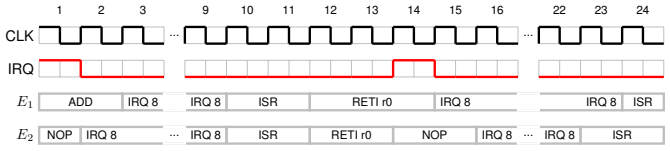

Fig. 7. Second interrupt handler is delayed.

The trick is to cleverly issue two consecutive interrupts. The first interrupt arrives while the target enclave instruction is executing (cycle 1). Next, Sancus$_V$'s hardware-level double padding defense, described in Section II-A2, balances interrupt latency to make sure the ISR executes at the same time (cycle 10) in both cases. Furthermore, to ensure constant resume-to-end timings, the next `reti` will be padded with the number of cycles the initial interrupt processing was delayed (1 cycle in the case of E_1, zero for E_2). We now schedule the second interrupt when we expect the longer `reti` to finish (cycle 14). In the case of E_1, the interrupt handling logic will immediately follow `reti` and will gain the extra cycle. In the case of E_2, however, the next regular instruction will have already started executing by the time the interrupt comes in, and the following interrupt handling logic will have the correct number of clock cycles (as it does not follow the `reti` directly). Hence, the attacker-controlled ISR will execute at a different time, i.e., cycle 23 vs. 24, depending on the initially interrupted instruction. This means a complete bypass of the defense.

c) Mitigation: This issue highlights the risks of a separate implementation and pen-and-paper model. Patching it requires a thorough analysis of the openMSP430 two-stage pipeline to identify the precise cause of the added delay and either eliminating it or making sure it applies to every instruction and the interrupt handling logic itself.

2) Instructions with execution time $T > 6$: To correctly calculate the required padding, Sancus$_V$ needs to know the maximum instruction execution length. This length is an explicit parameter in the Sancus$_V$ model, and is defined to be $T = 6$ cycles, as this is the longest length listed in the openMSP430 documentation [26].

a) Broken assumption: Our analysis revealed two cases where Sancus-specific instructions exceed the assumed limit. First, the real-world Sancus implementation extends the original openMSP430 core to also allow writes to program memory. We found that such writes induce an extra penalty cycle, such that instructions of the form `mov &ram, &rom` will take 7 cycles. Second, Sancus$_V$ does not model the cryptographic instructions added by Sancus. These are executed atomically, as with any other openMSP430 instruction, but can take several thousands of cycles, depending on the passed parameters [16].

b) Attack: The real-world implementation continuously decrements a 3-bit padding counter (starting from the value 6) after the interrupt arrives and before the current instruction finishes. We experimentally confirmed that when this counter underflows, an incorrect length is calculated for the subsequent interrupt handling logic. This breaks the padding scheme and enables attackers to distinguish enclaves that use instructions exceeding 6 cycles.

c) Mitigation: The implementation should be correctly parameterized with the real maximum execution bound T. This is relatively straightforward in the case of 7 cycles for program memory writes, but less so for unmodeled cryptographic instructions that can take thousands of cycles (with an impractical upper bound in the order of 2^{16} when hashing the entire 16-bit address space). Options for cryptographic operations may include adopting an abandon-restart interrupt policy [18], or disallowing them altogether inside enclaves, which would, however, break crucial attestation functionality.

Ideally, to avoid further implementation-model mismatches, T should be determined from the Verilog code, e.g., using

static analysis to determine the highest possible number of clock cycles an instruction might spend in execution.

3) Resuming an enclave with `reti` multiple times: The Sancus$_V$ model includes a separate shadow register file, referred to as the "backup", to securely save and restore the interrupted enclave's secret register values. The model explicitly dictates that the CPU should only fill the backup when an interrupt arrives in enclave mode, and, upon the next `reti` instruction, check whether the backup is non-empty and, if so, restore the original values from the backup, mark it as empty, and return control to the interrupted enclave.

a) Broken assumption: Our audit of the real-world implementation revealed a serious bug, where, after first correctly restoring the shadow register contents, `reti` incorrectly does not clear the hardware flag that indicates a non-empty backup. This means that once an enclave is successfully interrupted, any subsequent untrusted `reti` instruction will also incorrectly restore the register values from the backup, which still contains the secret values from when the enclave was interrupted. Since this includes the program counter – as well as all other register values – the attacker is practically capable of setting a *checkpoint* in the enclave via an interrupt and later jumping back to it.

b) Attack: To demonstrate the attack, we use the following minimal enclave that increases a zero-initialized credit counter by one, but stores a private `is_modified` flag to prevent multiple increases. While this enclave should be contextually equivalent with one that always returns 1 in `r5`, we experimentally validated that interrupts break this equivalence by allowing the value to increase over 1.

```
1      cmp #0x0, &is_modified  ; if (is_modified == 0)
2      jnz 1f                  ; {
3      add #0x1, &user_credit  ;     user_credit += 1
4      mov #0x1, &is_modified  ;     is_modified = 1
5  1:  mov &user_credit, r5    ; } return user_credit
```

Listing 2. Credit management enclave (all variables are initialized to zero).

Sancus [16] only allows enclaves to start executing from their entry points (the first instruction in this case): jumps to the middle of the code section are blocked in hardware. However, if we first interrupt the enclave at the start of line 3, right before the `add` instruction, but, importantly, after the check on `is_modified`; we can subsequently use `reti` from outside the enclave to jump back to that point and increase our credit an arbitrary number of times.

c) Mitigation: The Verilog implementation should be corrected to adhere to the pen-and-paper model by clearing the hardware flag indicating a non-empty backup upon `reti`.

4) Restarting enclaves from the ISR: The Sancus$_V$ model imposes that interrupted enclaves can only be resumed via `reti`, and cannot be reentered from the start.

a) Broken assumption: Our audit revealed that this behavior is not enforced in the real-world Verilog implementation. Reentering an interrupted enclave from the start, instead of resuming it properly via `reti`, may allow an attacker to manipulate enclave values in an unintended way.

b) Attack: Reconsider the credit management enclave of Listing 2. This time, however, we interrupt at the start of line 4, right after the `add` instruction. At this point, the enclave has incremented the credit balance, but has not yet set the private `is_modified` flag to block further updates. Instead of executing `reti`, the attacker now simply reenters the enclave again from the start. We experimentally validated that the second start-to-end run of the enclave breaks contextual equivalence by once again incrementing the credit before finally setting `is_modified` and returning 2 (instead of the expected value 1) in `r5`.

c) Mitigation: The Verilog implementation should be corrected to adhere to the pen-and-paper model by disallowing jumps to an interrupted enclave's entry point before `reti`.

5) Multiple enclaves: The Sancus$_V$ paper explicitly documents that only a single enclave is modeled.

a) Broken assumption: The real-world openMSP430-based Sancus$_V$ implementation can be parameterized with any number of hardware-enforced enclaves. Our audit revealed that the implementation uses the default number of 4 enclaves.

b) Attack: Consider the credit management enclave E_c of Listing 2, which is again interrupted at the start of line 4. In this attack, however, instead of directly resuming or reentering E_c, the ISR jumps to another enclave, E_a. The purpose of this attacker-controlled second enclave is to set the program counter to the entry point of E_c, just before getting interrupted itself (which is also scheduled by the attacker). Upon the second interrupt, the single backup register file – originally containing the E_c register values – is overwritten with the attacker-controlled E_a values, including the modified program counter pointing to the start of E_c. Hence, the original progress in E_c is lost, and we experimentally validated that, when the ISR finally calls `reti`, control will be incorrectly transferred to the start of E_c.

c) Mitigation: The Verilog implementation should override the default number of supported enclaves to one. If support for multiple enclaves is desired, the Sancus$_V$ model and proof should be extended to rule out any additional attacks, such as – but not necessarily limited to – the one above.

6) Enclave accessing unprotected memory: The Sancus$_V$ paper highlights that an important condition is that enclaves cannot access unprotected memory outside of the enclave.

a) Broken assumption: Our security audit revealed that the real-world implementation still allows enclaves to read from and write to unprotected shared memory locations.

b) Attack: This oversight can clearly be abused to differentiate two otherwise contextually equivalent enclaves: $E_1 = \{$`mov #1, &addr; mov #0, &addr`$\}$ and $E_2 = \{$`mov #2, &addr; mov #0, &addr`$\}$, where `addr` lies outside the enclave. While without interrupts the value at `addr` always ends up being zero, we experimentally validated that E_2 can be trivially distinguished by interrupting after the first instruction and inspecting the value at `addr`.

c) Mitigation: This important model assumption should be properly enforced in the Verilog implementation. The most straightforward solution would be to check target memory

addresses in the openMSP430 memory backbone and only allow unprotected accesses based on whether the enclave is executing in the given cycle.

7) Manipulating interrupt behavior from the enclave: The Sancus_V model specifies that interrupt-related functionality cannot be influenced from within the enclave itself. Concretely, enclaves cannot manipulate *(i)* the interrupt-enable bit (`GIE`) in the status register, *(ii)* the interrupt vector table (IVT) containing ISR addresses, and *(iii)* the timer peripheral itself.

a) Broken assumption: We experimentally validated that enclaves are currently allowed all 3 of the above forbidden behaviors in the real-world Verilog implementation.

b) Attack: (i) Consider $E_1 = \{$`nop; dint`$\}$ and $E_2 = \{$`dint; nop`$\}$, where interrupts are disabled from the instruction following `dint`. While contextually equivalent without interrupts, the enclaves can be trivially distinguished by interrupting during the second instruction and observing whether the ISR executes (E_1) or not (E_2).

(ii) Another way to break contextual equivalence is to map part of the enclave's data section over the fixed IVT location. This allows to trivially distinguish two enclaves that write different unprotected ISR addresses to their private data memory: the attacker just needs to schedule an interrupt and observe which ISR executes.

(iii) Finally, we found that contextual equivalence can also be broken by mapping part of the enclave's private data section over the memory-mapped I/O (MMIO) registers of the timer peripheral. This allows the enclave to directly trigger interrupts. Clearly, attackers controlling the ISR can distinguish enclaves that schedule two consecutive interrupts vs. only one.

c) Mitigation: While in enclave mode, updates to `GIE` should be blocked in the Verilog code of the core. Furthermore, the core can easily be extended to disallow the creation of enclaves that map over the fixed IVT location.

However, disallowing enclave timer configuration is less straightforward and highlights the consequences of simplifying a model. Sancus_V only models a single abstract peripheral that is configured through simplified `IN/OUT` instructions, whereas the real-world implementation may include several peripherals that can reside at different MMIO address ranges (cf. Section V-C2). One option would be to disallow enclaves to map over *any part* of the MMIO range, but this would break the important use case of secure enclave drivers [16]

C. Exploiting missing attacker capabilities

1) DMA side-channel leakage: In line with the original Sancus 2.0 architecture [16], the Sancus_V implementation is based on an older version of the openMSP430 core without DMA capabilities. The Sancus_V formalization, hence, does not model attackers with DMA capabilities.

a) Unmodeled capability: Although DMA is currently not part of the formal model nor the implementation of Sancus_V, we still consider it an interesting attack vector against this system, as both more recent versions of openMSP430 [26] and the upstream version of Sancus [31] support DMA. Moreover, this attack demonstrates how an extension not directly related to interrupts can still undermine security properties related to them, showing valuable insight.

b) Attack: An enclave can easily be constructed to demonstrate how a DMA attacker with access to cycle-accurate memory traces can obtain information that is hidden to a Nemesis attacker (cf. Section IV). However, to directly break the security guarantees of Sancus_V, we need to show an example where the introduction of interrupts provides additional leakage compared to the DMA side channel alone. Listing 3 provides an example enclave with conditional branches that are carefully balanced to be DMA side channel resistant.

```
1       mov #0x42, r5
2       cmp r6, &password
3       jnz 1f
4       mov #0x42, r6   ; 2 cycles, 2 accesses
5       jmp 2f          ; 2 cycles, 2 accesses
6   1:  mov r5, r6      ; 1 cycle,  1 access
7       mov r5, r6      ; 1 cycle,  1 access
8       jmp 2f          ; 2 cycles, 2 accesses
9   2:  mov #0x0, r7
```

Listing 3. Memory-balanced branches.

Both branches have an execution time of 4 cycles, during which the CPU continuously accesses program memory (fetching the instructions and the constants). Hence, these branches are indistinguishable both to start-to-end timing and to DMA side-channel adversaries. Since the instructions in the two branches have different individual execution lengths, the Sancus_V padding defense is still needed to protect against a Nemesis attacker. However, even with correctly implemented interrupt handling and padding, the DMA attacker can distinguish between the two branches if they are interrupted. Figure 8 shows the program memory accesses when interrupting the first cycle after the branch, i.e., at the start of line 4 or 6.

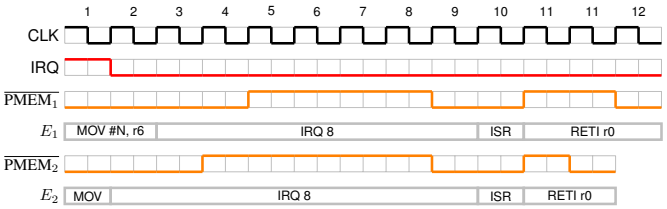

Fig. 8. Program memory accesses for the two flows in Listing 3.

The $\overline{\text{PMEM}}$ signal is high when the memory bus is free, these are the cycles when the attacker's DMA requests would be served. We can see that the additional padding cycles (5-8 and 4-8) added to the interrupt handling logic do not access the memory. Using DMA requests, an attacker can count how many cycles have no memory access, i.e., how many padding cycles were added. The number of padding cycles is directly correlated with the interrupted instruction's execution length, which can be reconstructed this way, completely bypassing the Nemesis defense.

c) Mitigation: In case DMA support would be added to the current Sancus_V implementation, a straightforward defense may be to disable DMA completely during enclave execution. However, care needs to be taken, since even unfinished DMA

TABLE II. List of falsified and exploitable assumptions found in VRASED. IM = Implementation-model mismatch; MA = Missing attacker capability; DE = Deductive error.

IM	VI-B1	The `dma_addr` bus contains the full address being accessed.
	VI-B2	All components use a consistent key size.
MA	VI-C1	Shared peripheral bus is not modeled.
	VI-C2	Secure stack initialization in SW-Att is not modeled.
	VI-C3	Timing attacks on SW-Att outside of HACL* are not modeled.
	VI-C4	Interrupt latency timing attacks are not modeled.
	VI-C5	DMA timing attacks are not modeled.
DE	VI-D	Missing assumptions about the core.

requests may leak information (cf. Section VI-C5). More sophisticated defenses that preserve some of the performance gains offered by DMA are out of scope for this paper.

2) Scheduling interrupts with the watchdog timer: The openMSP430 architecture has multiple hardware components that can generate interrupts. The model of Sancus$_V$, however, only has the notion of a single, abstract peripheral timer device that can issue interrupts in any attacker-chosen clock cycle.

a) Unmodeled capability: The implementation supports two timer peripherals: `TIMER_A` and the watchdog timer (WDT). The number of padding cycles is calculated based on the `irq_arrived` signal. This signal is raised for external interrupts coming from `TIMER_A`, but not for the integrated, on-chip WDT, which has its dedicated `wdt_irq` signal.

b) Attack: Although configuration options are more limited, the WDT can still schedule cycle-accurate interrupts. We experimentally validated that, for WDT interrupts, no padding is added, thus completely and trivially breaking the defense.

It is important to note that the shadow register switching does not depend on these signals, so register values from the enclave cannot leak to the ISR, even during a WDT interrupt.

c) Mitigation: This issue highlights the attention to be given to interface signals. The padding implementation is activated based on a signal that is too specific. If the detection happened based on a signal that is raised for any type of incoming interrupt, this attack would not be possible.

VI. Security analysis of VRASED

A. Identifying falsifiable and hidden assumptions

As explained earlier, the VRASED [15] security argument relies on only 7 explicit assumptions (cf. Appendix B) that reportedly encapsulate all assumptions placed on the functionality of the core and the compiler. Favorably, these assumptions are explicitly listed and not scattered throughout the paper, which makes them easier to validate. However, our analysis revealed several imprecisely formulated or missing assumptions, as well as important unmodeled attacker interactions. Table II lists falsified and exploitable assumptions and unmodeled features, as well as a deductive error. We refer to Appendix C for remaining assumptions that were not found to be directly exploitable.

B. Validating the implementation

As a first important observation, it is interesting to note that, while our analysis of Sancus$_V$ revealed seven implementation-model mismatches, we found fewer such exploitable errors in VRASED. This shows the power of extracting the model directly from HW-Mod's Verilog implementation.

1) Incorrect DMA address translation: One of the signals HW-Mod monitors from the core is the DMA address bus (`dma_addr`). If the address on this bus falls within the key region, a reset is triggered before the value can be read out.

a) Broken assumption: The internal `dma_addr` bus of the 16-bit openMSP430 core measures only 15 bits. This is because DMA accesses are always word-aligned, making the last (zero) bit of the address redundant. However, our audit revealed that, in the crucial connection of the verified HW-Mod to the unverified openMSP430 core, the 15-bit `dma_addr` signal was incorrectly zero-extended (instead of left-shifted) into a 16-bit signal as follows: `{1'b0, dma_addr[15:1]}`.

b) Attack: We experimentally validated that the incorrect, zero-extended address comparison in HW-Mod allows untrusted DMA peripherals to trivially read out the entire value of the secret key without triggering a reset. This means a complete bypass of the main VRASED security goal (P1).

c) Mitigation: This issue reinforces the importance of interface signals between verified and unverified components and more generally highlights the limitations of automated model generation of only a subset of the core. The implementation should adhere to the model by adding the extension bit in the correct place, i.e., `{dma_addr[15:1], 1'b0}`.

2) Inconsistent key sizes: VRASED specifies [15, Definition 2] a security parameter l, which equals the key size, as well as the challenge and HMAC digest sizes. The VRASED implementation, furthermore, uses HMAC-SHA256 with explicit challenge and digest sizes of 256 bits ($l = 32$ bytes).

a) Broken assumption: In contrast to the formal security parameter definition above, the unverified secure key ROM module of the modified core defines a 64-byte master key. This entire 64-byte master key is used by SW-Att to derive a 32-byte challenge-dependent key, which is securely stored on the exclusive stack and is subsequently used to calculate the attestation HMAC.

Crucially, however, in the verified HW-Mod component, a master key size of only `KMEM_SIZE = 31` bytes is used for access control to the secure key ROM, leaving the second half of the master key completely unprotected. To make matters worse, even the first half of the master key is not completely protected, as HW-Mod's bounds checking excludes the byte at `key[KMEM_SIZE]`, i.e., the 32^{nd} byte.

b) Attack: We experimentally validated that the incorrect access control in HW-Mod allows untrusted code outside SW-Att to directly read out all 33 affected bytes at `key[31:63]`.

c) Mitigation: A consistent master key size should be used throughout the implementation and the verification code.

These findings further highlight the limitations of interactions between verified and unverified components, and maintaining consistent parameters between them. Importantly, this crucial oversight was not detected because VRASED's verification is parameterized with the same erroneous 31-byte key size as HW-Mod.

C. Exploiting missing attacker capabilities

1) Secure metadata corruption with a peripheral: Both the APEX [20] and RATA [21] VRASED extensions[3] store important metadata in secure MMIO peripheral registers.

a) Unmodeled capability: The secure proof-of-execution register in APEX is implemented as a read-only peripheral device. On openMSP430, all peripherals are connected to the core via shared buses, both for addressing and for transferring output values. If multiple peripherals output data in the same cycle, the values are combined with a bitwise `or` operation.

We experimentally validated that a compromised DMA device connected via the shared peripheral buses can interfere with the values read from other peripherals. This includes APEX's secure proof-of-execution flag, which resides at a fixed address in the MMIO space. APEX returns a secure flag value of '1' when the attested program has successfully completed, while '0' is used to signal an error or tampering.

b) Attack: Whenever the secure APEX peripheral MMIO register address is seen on the shared address bus, our proof-of-concept compromised peripheral puts a '1' on the shared output bus. A '1', when `or`-ed with the actual output value of the APEX peripheral will always result in a value of '1', signaling successful execution to the core. This is the case even if the attacker has previously tampered with the execution and the value of the flag in the secure register is '0'.

c) Mitigation: This issue highlights the risks of storing security-sensitive data outside the security perimeter, among untrusted and unverified components. Since this issue is a result of the original openMSP430 design, it is not straightforward to fix. A possible workaround is to avoid the untrusted peripheral bus altogether and store crucial attestation metadata on-chip, within the trust boundaries of the core itself.

2) Key leakage through stack pointer poisoning: The HACL* [37] cryptographic library requires a stack to save temporary state, including secrets. VRASED's HW-Mod, therefore, enforces an exclusive stack XS for SW-Att.

a) Unmodeled capability: At the hardware level, only HW-Mod is verified, VRASED reuses the existing HACL* proofs [37] to claim full security for its trusted SW-Att software component. However, SW-Att is also responsible for setting up the execution environment expected by HACL* before invoking the actual cryptographic primitives. Unfortunately, this crucial trusted wrapper code remains entirely unverified.

Even worse, our security audit of the implementation revealed that the secure stack pointer is set up by *untrusted* code before invoking SW-Att. The trusted SW-Att entry code does not validate that the stack points to XS as expected.

b) Attack: This vulnerability allows an attacker to freely change the value of the stack pointer before entering SW-Att. Since SW-Att can only write to XS and a shared memory region MR to store the HMAC result, a logical choice is to point the stack to MR, as it is also accessible by untrusted software. SW-Att will now fill MR with sensitive stack frames, potentially including the secret key. Once the stack overflows MR, an illegal write will happen and the CPU will reset. However, the secure reset does not clear MR, which allows the attacker to retrieve the leaked values after the reset.

We experimentally found that, with the predefined size of MR and without changing any configuration parameters, the CPU resets during zero-initialization of the local variable holding the key, i.e., before sensitive values are leaked. However, by changing the optimization level of the compiler (e.g., as confirmed with `msp430-clang` v4.0.1 at `-O1`), this redundant zero-initialization may be skipped. Note that this specific zero-initialization is entirely redundant, and, hence, can be safely removed without affecting functional correctness (A7). We experimentally validated that, when the zero initialization is skipped, the first 22 key bytes are copied into MR before reset, thereby breaking VRASED's no-leakage property (P2).

c) Mitigation: This issue highlights the risks of combining proofs (i.e., HW-Mod and HACL*) without a rigorous holistic security argument. In this specific case, a more explicit argument should be made about how VRASED intends to fulfill the assumptions of the HACL* proof. The assumption for stack-pointer initialization should be fulfilled when entering SW-Att. This can be done either transparently at the hardware level, within the verified HW-Mod logic; or inside the SW-Att software component, using a trusted (and preferably also verified) assembly entry stub that sanitizes the ABI expected by the C compiler, similar to Intel SGX shielding runtimes [52]. This stub should also properly cleanse caller-save registers when exiting SW-Att (cf. Appendix C1).

3) Timing side channel in authentication: To protect against denial-of-service attacks, VRASED$_{\text{A}}$ [15] extends SW-Att with verifier authentication. Specifically, VRASED$_{\text{A}}$ only executes the expensive attestation if a correct authentication token, calculated from the challenge and the secret key, is supplied with the request (cf. Appendix A1). Importantly, the verifier authentication primitive provided by VRASED$_{\text{A}}$ is also tightly coupled with the security of the more recent RATA [21] VRASED extension.

a) Unmodeled capability: VRASED$_{\text{A}}$ is entirely implemented in trusted C code that is included at the entry point of SW-Att and invokes the required HACL* cryptographic primitives as shown in Listing 4. However, similar to the previous issue, the C code itself remains entirely unmodeled and unverified. Any security argument or assumptions about this wrapper code are missing. Hence, a single vulnerability in the trusted wrapper C code may invalidate SW-Att's claimed guarantees built on HACL*'s functional correctness, memory safety, and secret-independent timing behavior.

```
1 if (memcmp(CHALL_ADDR, CTR_ADDR, 32) > 0) {
2   hacl_hmac(mac, key, CHALL_ADDR));
3   if (!memcmp(VRF_AUTH, mac, 32)) {
4     attest();
5     memcpy(CTR_ADDR, CHALL_ADDR, 32);
6   }
7 }
```

Listing 4. Authentication code in VRASED$_{\text{A}}$ (simplified).

[3]The source code of RATA was only released after we conducted our research, so we did not analyze it beyond confirming that it uses the same shared peripheral bus and is thus potentially vulnerable to a similar attack, which is left as future work.

b) Attack: Observe that line 3 is vulnerable to timing attacks as it uses the standard `memcmp` function from `libc` to determine whether the attacker-provided value `VRF_AUTH` matches the expected `mac` authentication tag computed using the secret key. This function terminates at the first mismatching byte pair, thus allowing an attacker to guess the secret authentication `mac` value byte-by-byte, reducing the effort from an exponential problem (256^{32}) to a linear one ($256 \cdot 32$).

We experimentally validated (cf. Table III in Appendix D). that an attacker measuring SW-Att's start-to-end execution time can deterministically extract the expected `mac` value with little effort and without key knowledge, thereby entirely bypassing the main VRASED_A security goal.

c) Mitigation: This specific timing vulnerability can be patched by using a constant-time `memcmp` implementation. In general, however, this shows that any code included in SW-Att needs to be thoroughly checked to not leak the key or any information about it, which is difficult to generalize.

More broadly, this issue once again highlights the importance of rigorous argumentation when reusing existing security proofs (e.g., that of HACL*) combined with seemingly simple wrapper code. This timing issue could have been prevented if the wrapper were not developed in C, but instead written in F* and properly integrated into the HACL* verification.

4) Nemesis side-channel leakage: Given that VRASED_A is vulnerable to timing side channels, and given that the Nemesis [32] interrupt latency attack on Sancus is the result of the underlying openMSP430 architecture, a logical question to ask is whether this side channel affects VRASED as well. In VRASED, interrupts are not allowed during the execution of SW-Att and should result in an immediate, secure CPU reset.

a) Unmodeled capability: HW-Mod contains the formally verified Verilog logic that will reset the CPU in case of an interrupt request during SW-Att execution. However, the way HW-Mod is wired to the core allows a Nemesis-type side-channel leakage. The signal monitored by HW-Mod is `irq_detect`, which is only raised upon instruction retirement. In other words, the reset will indeed always correctly happen, but it will be delayed until the cycle when the interrupt handling would normally start. This can also be seen in Figure 9, where the interrupt arrives and the `irq[8]` signal changes in the 4th cycle, but the `irq_detect` signal is only raised at the end of the executing instruction, followed by the CPU reset.

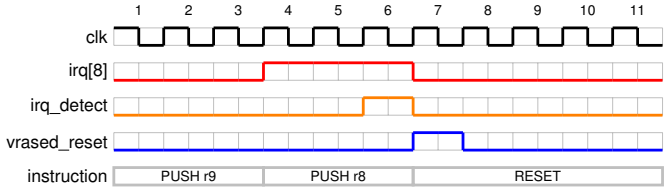

Fig. 9. Interrupt is only detected at the last cycle of the `push` instruction.

b) Attack: Since a CPU reset also zeroes benign timer peripherals, the reset delay cannot be directly measured from software. However, we experimentally developed a proof-of-concept compromised DMA peripheral that is capable of detecting CPU resets and saving information across them. This allows a Nemesis-style "reset-latency" attacker to reconstruct execution lengths for every individual instruction in SW-Att.

The effects on VRASED are arguably less severe than on Sancus_V, since the HACL* code does not contain any secret-dependent branches. As such, the only currently practical use of this instruction-granular leakage would be to break a naively patched `memcmp` solution that may be proposed against the start-to-end timing attack presented in Section VI-C3.

c) Mitigation: This issue again highlights the attention to be given to interface signals. Similar to Section V-C2, the attack is enabled by wiring the wrong interrupt signal: connecting HW-Mod with a signal that is immediately raised for all interrupt sources would have ensured that the reset is not delayed until instruction retirement.

More generally, this attack, along with the previous and next ones, illustrates the ramifications and pitfalls of not modeling time measurement as an important attacker capability.

5) DMA side-channel leakage: Just as with Nemesis, the DMA side channel from Section IV is also a result of the underlying openMSP430 architecture. Similar to interrupts, DMA requests during the execution of SW-Att are disallowed.

a) Unmodeled capability: HW-Mod contains the formally verified Verilog logic that will trigger a secure CPU reset whenever detecting any DMA activity during SW-Att execution. While this reset indeed prohibits direct data leakage, we found that it, as with interrupts, does not prevent timing leakage.

There are two signals connected to DMA peripherals that are driven by the core. First, a bus contains the data one cycle after read requests. This is the same cycle as when the reset is triggered, which also clears the data bus; this behavior ensures that no sensitive data can be read out during SW-Att execution.

The other signal to the peripheral driven by the core is the `dma_ready` signal, which shows whether the request was successfully completed. This is the signal the DMA attacker will monitor instead of the data bus. If this signal is high, there was no contention on the memory bus. This signal is raised in the same cycle when the DMA request is issued, so it is *not* masked by the reset, which only happens in the next cycle.

b) Attack: We experimentally validated that an attacker with untrusted peripheral access can fully exploit the DMA side channel on VRASED to learn cycle-accurate memory bus utilization for every instruction in SW-Att. This may have severe effects if the code of SW-Att changes, while remaining undetected by the HW-Mod security proof.

c) Mitigation: Since VRASED already disallows DMA during SW-Att execution (in contrast to upstream Sancus [31] which allows untrusted DMA accesses as a performance optimization during enclave execution), a relatively easy fix would be to properly mask the `dma_ready` signal as well, so no side-channel information leaks to the rogue peripheral.

D. Deductive errors

VRASED does not formalize the operational semantics of the original openMSP430 core. Any core implementation that satisfies five short assumptions A1-A5 (cf. Appendix B)

should, hence, be covered by the proof. The end-to-end RA security argument depends on two premises: the soundness of the remote attestation scheme and the claim that the attacker cannot learn the key. In the following, we argue that these premises do not follow from the stated assumptions (for a dissection of the full argument, we refer to Appendix E).

The claim that the attacker cannot learn the key is centered around a lemma stating that a reset is caused if the key is read directly, or if SW-Att writes outside the HMAC result region MR. This lemma is rigorously formulated in LTL, and a machine-checked proof is provided that shows that this lemma follows from the LTL security invariants that were previously shown to be enforced by HW-Mod. Specifically, the LTL rules used in this part of the proof were shown to be enforced by the state machine that was directly generated from HW-Mod's Verilog implementation, and we did not find any issues with this mechanized part of the proof. However, the premises leading to this lemma and the further conclusions drawn from it are formulated more informally in writing, and they can be bypassed by modifying unverified parts of the core.

First, leading up to the lemma, it is informally argued that the key may only leak through *(i)* registers, supposedly covered by A6; *(ii)* timing, supposedly covered by HACL*'s secret-independent timing; and *(iii)* memory, covered by the lemma. However, consider an alternative core implementation that, when reading the key during the HMAC calculation, delays this load operation by exactly as many clock cycles as is the value of the key being read. Such a core satisfies A1-A5, which do not specify constraints about timing, but does not conform to HACL*'s machine model that should support constant-time code and, hence, trivially leaks the key.

Second, following the lemma, it is assumed that raising the reset signal always prevents key leakage. However, consider an alternative core that, upon reset caused by a DMA read, still outputs the read value on the DMA data bus without masking; or, alternatively, properly clears registers during a reset (A4), but simply restores them, for instance, 100 cycles after the reset, thus leaking the key without violating A1-A5.

We want to stress that the point is not whether such alternative cores are realistic or not (indeed, the actual openMSP430 implementation may very well be secure on these points). What is important, however, is that none of these misbehaving cores would cause the proof of VRASED to fail and the end-to-end security claim does, hence, *not* follow from the assumptions A1-A5, as claimed.

VII. Discussion

Our case study suggests some concrete, actionable guidelines that can help in avoiding vulnerabilities and providing better security assurance for systems. We structure them by vulnerability category.

A. Implementation/model mismatches

A successful attack on the real system that can be represented in the formal model, but fails there, implies a certain disconnect between the real system and the model. Since the real system is not a mathematical object, it is fundamentally *not* possible to find all these issues through more rigorous use of formal deductive methods alone. However, for software-based systems, many implementation artifacts of the real system can be given a mathematical interpretation. Code in a programming or a hardware description language *becomes* a mathematical object by defining a semantics for these languages. This, of course, introduces an assumption that the real system will execute the code according to the defined semantics. But in turn, verifying security properties of the source code as used in the implementation of the real system becomes a deductive problem that can – at least in principle – be handled rigorously using mathematical methods.

Our case study provides substantial evidence for the fact that implementation/model mismatches can be avoided by maintaining a strong connection between the model and the implementation code. For Sancus_V, where this connection is weak, we found a considerable number of mismatches (V-B1 to V-B7). In contrast, the model for HW-Mod in VRASED was automatically derived from the Verilog code, and indeed no model/implementation mismatches were found *within* the HW-Mod component. However, other parts of VRASED where the connection was again weak, including the interface between HW-Mod and the untrusted core, suffer from several issues. Many of the implementation/model mismatches from our case study are relatively simple errors that could have been avoided or discovered with a more rigorous connection.

Guideline: Avoid implementation/model mismatches by maintaining strong connections between code used in the implementation, and the model used for verification.

While the systems in our case study only used automatic derivation of a model to maintain this connection, we believe other techniques can be very useful in reducing implementation/model mismatches. Some important ones include: deriving the implementation code from the model [53], systematic testing of executable models against the real system [54], or directly verifying the code itself [55].

Provable security results that do not provide evidence for the connection between model and implementation, like Sancus_V, can of course still be very useful. They show the absence of mistakes or oversights in one specific aspect of the system (like the design of the interrupt padding mechanism in Sancus_V). But they only provide weak assurance about the security of the real implementation.

B. Missing attacker capabilities

The existence of attacks on the real system that *cannot* be represented in the formal model shows that the formal model is incomplete in some sense. This is the most challenging category of errors, and deductive methods can fundamentally not find all these issues. Domain and attack expertise are essential to assess whether a model captures all relevant attacks, and it can never be ruled out that new kinds of realistic, creative attack techniques are invented.

Our case study provides evidence for this by introducing a fundamentally new attack technique that exploits the con-

tention between the CPU and a DMA device to break some security properties of both systems considered in our study. The literature provides other examples, such as the discovery of Spectre attacks [56], which invalidated proofs of many confidentiality properties based on execution models that did not consider speculative execution.

In the absence of a systematic way of avoiding model incompleteness, we have to rely on heuristic guidelines and rules of thumb to assess the completeness of models.

Guideline: Useful sanity checks to avoid model incompleteness include:

- the model should either represent attacks from the literature or explicitly argue why they are not represented;
- directly modeling specific attack scenarios should be avoided, focus on modeling attacker capabilities and ways of composing these capabilities into attack scenarios. This will typically allow the model to represent a wider range of attack scenarios where the attacker can also compose capabilities in unexpected ways;
- interfaces between verified and unverified components should be audited for attacker-controlled inputs.

Our case study provides evidence for these rules of thumb: three out of five model incompleteness issues that we found in VRASED could have been avoided if the model had included the realistic capability that the attacker can measure cycle-accurate time. Additionally, at least three of the issues we found in VRASED are directly related to untrusted interfaces.

It is, however, important to emphasize that modeling more attacker capabilities can also impact the feasibility of the verification: verification techniques do not necessarily scale well as more attacker capabilities are modeled.

C. Deductive errors

If a successful attack on a real system can also be performed on the model of the system, this implies that there is an error in the formalization itself: the proof that no attacks exist in the model must be flawed. One of the strengths of formal methods is that they allow for very systematic checking for these kinds of errors, and by moving to more rigorous proof methods, ideally machine-checked proofs, these errors can be avoided. In the case of VRASED (cf. Section VI-D), moving to a stricter formal argument would have uncovered that the end-to-end RA security property does not follow from the assumptions. A possible solution may be to rigorously prove that the core adheres to an operational semantics.

Other research also supports the claim that rigorous use of formal methods is very useful in avoiding this class of errors. For instance, an extensive effort to uncover compilation bugs [57] in 11 compilers found no bugs in the machine-checked formally verified CompCert [53] middle-end, but found bugs in all other compilers *and* in the unverified parts of CompCert.

Guideline: Avoid deductive errors with rigorous reasoning, ideally using end-to-end machine-checked proofs.

VIII. Related work

The use of formal methods to increase assurance in the security of computer systems goes back to the early days of computer security with the US Military's desire to build multi-level secure operating systems [58]. The Multics operating system is perhaps the most influential example of a system whose security assurance combined formal methods [59] with serious attack research [60]. Experience with the evaluation of Multics and related systems ultimately led to the development of the Common Criteria [61] as a standard for the evaluation of the security of systems. The highest assurance level of the Common Criteria requires a combination of deductive evidence (a formally verified design) and inductive evidence (thorough testing) – but very few products aim for that highest assurance level. While the Common Criteria has had some success in the evaluation and certification of very security-sensitive commercial products, like smart cards or hardware security modules, it also has widely recognized limitations in terms of cost, bureaucracy, and lack of agility in the certification process [62].

From the eighties onward, formal methods have been successfully used in the fields of protocol design and cryptography. Security properties of protocols were formalized as early as the eighties [63], [64]. Currently, protocols are often formalized during the design phase to detect mistakes before deployment, for example in the NIST Post-Quantum Cryptography Standardization Process [10] or for TLS 1.3 [9].

In parallel, researchers have also discovered that formal approaches can still lead to mistakes. The Needham-Schroeder protocol, formalized in 1989 [64], was shown to be broken seven years later using an automated tool [65]. In the years since, many attacks have been discovered against formally proven protocol schemes [66], [67]. Koblitz and Menezes [67] list multiple reasons why proofs can go wrong, some of which, such as implicit and incorrect assumptions, also appeared in our case study. Other papers, like the KRACK attacks [68] show how protocol implementations deviating from the formal descriptions can enable attacks, while pen-and-paper security proofs have been described as "alarmingly fragile" [69].

In contemporary systems security research, formalization of security properties is not yet as common, even if calls have often been made to use more scientific reasoning and formal methods for security guarantees [1]–[3], [70]. Notable recent systems with public security proofs include Komodo [12] and the seL4 microkernel [71]. Intel SGX also has (non-public) proofs about the linearizability of its instructions [13] and about some security properties [72].

IX. Conclusion

For systems of the size considered in this paper, we believe the time is ripe to work towards complete open-source hardware-software implementations whose security is supported by a combination of *(i)* deductive evidence based on models derived from, or strongly connected to the source code, and *(ii)* inductive evidence based on attack research, supporting the assumptions for the deductive reasoning.

Acknowledgment

We would like to thank the designers of the Sancus and VRASED architectures for making their systems open-source. We are grateful to Job Noorman, Thomas Van Strydonck, and the anonymous reviewers for their insightful comments on different versions of this paper. This research is partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by a gift from Intel Corporation. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

References

[1] D. I. Good, "The foundations of computer security: We need some," in *Medieval Propaganda Pamphlet*. University of Texas, 1986.

[2] B. D. Snow, "We need assurance!" in *21st Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2005, pp. 3–10.

[3] C. Herley and P. C. van Oorschot, "Science of security: Combining theory and measurement to reflect the observable," *IEEE Security & Privacy*, vol. 16, no. 1, pp. 12–22, 2018.

[4] ——, "Sok: Science, security and the elusive goal of security as a scientific pursuit," in *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2017, pp. 99–120.

[5] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium*, Aug. 2019, pp. 249–266.

[6] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010, pp. 270–283.

[7] D. A. Basin and S. Capkun, "The research value of publishing attacks," *Communications of the ACM*, vol. 55, no. 11, pp. 22–24, 2012.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 48–62.

[9] K. G. Paterson and T. van der Merwe, "Reactive and proactive standardisation of TLS," in *Security Standardisation Research - Third International Conference (SSR)*, ser. Lecture Notes in Computer Science, vol. 10074. Springer, 2016, pp. 160–186.

[10] G. Alagic, G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta *et al.*, "Status report on the first round of the nist post-quantum cryptography standardization process," 2019.

[11] P. Subramanyan, R. Sinha, I. A. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 2435–2450.

[12] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 287–305.

[13] R. Leslie-Hurd, D. Caspi, and M. Fernandez, "Verifying linearizability of Intel® software guard extensions," in *Computer Aided Verification - 27th International Conference (CAV)*, ser. Lecture Notes in Computer Science, vol. 9207. Springer, 2015, pp. 144–160.

[14] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Provably secure isolation for interruptible enclaved execution on small microprocessors," in *33rd IEEE Computer Security Foundations Symposium (CSF)*, Jun. 2020, pp. 262–276.

[15] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *28th USENIX Security Symposium*, 2019, pp. 1429–1446.

[16] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–33, 2017.

[17] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient component authentication and software isolation for automotive control networks," in *33rd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2017, pp. 225–237.

[18] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Aion: Enabling open systems through strong availability guarantees for enclaves," in *28th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2021, pp. 1357–1372.

[19] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "PURE: using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," in *International Conference on Computer-Aided Design (ICCAD)*. ACM, 2019, pp. 1–8.

[20] ——, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium*. USENIX Association, 2020, pp. 771–788.

[21] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," in *28th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2021, pp. 2921–2936.

[22] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 641–646.

[23] J. Certes and B. Morgan, "Remote attestation of bare-metal microprocessor software: A formally verified security monitor," in *International Conference on Database and Expert Systems Applications*. Springer, 2021, pp. 42–51.

[24] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede, "Soteria: Offline software protection within low-cost embedded devices," in *31st Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 241–250.

[25] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, 2018.

[26] O. Girard, *openMSP430*, 1.17 ed., https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf, Nov. 2017.

[27] 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 Transactions on Programming Languages and Systems (TOPLAS)*, vol. 43, no. 3, pp. 12:1–12:77, 2021.

[28] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: a security architecture for tiny embedded devices," in *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014, pp. 10:1–10:14.

[29] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *52nd ACM/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[30] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, "Secure interrupts on low-end microcontrollers," in *25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 147–152.

[31] "Sancus-core: Minimal openmsp430 hardware extensions for isolation and attestation," https://github.com/sancus-tee/sancus-core, accessed 2021-08-06.

[32] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *25th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018, pp. 178–195.

[33] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*, 1996, pp. 104–113.

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

[35] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing a dynamic) root of trust." in *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012, pp. 1–15.

[36] X. Carpent, G. Tsudik, and N. Rattanavipanon, "ERASMUS: efficient remote attestation via self-measurement for unattended settings," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1191–1194.

[37] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *24th ACM Confer-*

ence on Computer and Communications Security (CCS). ACM, 2017, pp. 1789–1806.
[38] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet *et al.*, "Verified low-level programming embedded in F," *ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–29, 2017.
[39] S. Underwood, *mspgcc: A port of the GNU tools to the Texas Instruments MSP430 microcontrollers*, http://mspgcc.sourceforge.net/manual/, 2003.
[40] M. Howard and S. Lipner, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8.
[41] A. Boileau, "Hit by a bus: Physical access attacks with firewire," *Presentation, Ruxcon*, vol. 3, 2006.
[42] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," in *26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.
[43] G. Kupfer, "Iommu-resistant DMA attacks," Master's thesis, Computer Science Department, Technion, 2018.
[44] B. Ruytenberg, "Breaking Thunderbolt Protocol Security: Vulnerability Report," Apr. 2020. [Online]. Available: https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf
[45] M. van Dijk, S. K. Haider, C. Jin, and P. H. Nguyen, "Advanced power side channel, cache side channel attacks, DMA attacks," *Presentation, Department of Electrical & Computer Engineering, University of Connecticut*, 2017.
[46] D. R. E. Gnad, J. Krautter, and M. B. Tahoori, "Leaky noise: New side-channel attack vectors in mixed-signal IoT devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, pp. 305–339, 2019.
[47] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.
[48] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 603–615, 2015.
[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *25th USENIX Security Symposium*, 2016, pp. 565–581.
[50] D. Ustiugov, P. Petrov, M. R. S. Katebzadeh, and B. Grot, "Bankrupt covert channel: Turning network predictability into vulnerability," in *14th USENIX Workshop on Offensive Technologies, WOOT*, Aug. 2020.
[51] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 225–236.
[52] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *26th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2019, pp. 1741–1758.
[53] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.
[54] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *ECOOP*, 2010.
[55] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens, "Software verification with verifast: Industrial case studies," *Sci. Comput. Program.*, vol. 82, pp. 77–97, 2014.
[56] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.
[57] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI*, 2011.
[58] D. MacKenzie and G. Pottinger, "Mathematics, technology, and trust: Formal verification, computer security, and the us military," *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 41–59, 1997.
[59] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and multics interpretation," Mitre Corporation, Tech. Rep., 1976.
[60] P. A. Karger and R. R. Schell, "Thirty years later: Lessons from the multics security evaluation," in *18th Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2002, pp. 119–126.
[61] "Common criteria for information technology security evaluation," https://www.commoncriteriaportal.org/, accessed 2021-08-18.
[62] R. J. Anderson, *Security engineering - a guide to building dependable distributed systems (3. ed.)*. Wiley, 2020.
[63] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.
[64] M. Burrows, M. Abadi, and R. M. Needham, "A logic of authentication," *Royal Society of London: Mathematical and Physical Sciences*, vol. 426, no. 1871, pp. 233–271, 1989.
[65] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1996, pp. 147–166.
[66] J. P. Degabriele, K. Paterson, and G. Watson, "Provable security in the real world," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 33–41, 2010.
[67] N. Koblitz and A. Menezes, "Critical perspectives on provable security: Fifteen years of "another look" papers," *Advances in Mathematics of Communications*, vol. 13, no. 4, p. 517, 2019.
[68] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in wpa2," in *24th ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1313–1328.
[69] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *42nd IEEE Symposium on Security and Privacy*, 2020.
[70] D. McMorrow, "Science of cyber-security," MITRE Corporation JASON Program Office, Tech. Rep. JSR-10-102, Nov. 2010.
[71] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 207–220.
[72] A. Goel, S. Krstic, R. Leslie, and M. R. Tuttle, "Smt-based system verification with dvf." in *SMT@ IJCAR*, 2012, pp. 32–43.

APPENDIX

A. Extensions to VRASED

Multiple derived architectures have been published that are built on the open-source VRASED research prototype and use its security arguments as the basis of their own.

1) VRASED$_A$ [15]: Verifier authentication: To prevent an attacker from generating many attestation requests to overwhelm the prover's computational resources, a modification to SW-Att is proposed in the original VRASED paper [15]. This variant, referred to as VRASED$_A$ in our paper, authenticates attestation requests before starting the expensive attestation. Attestation requests need to contain an authentication token, which is calculated by calling HMAC on the challenge with the shared master key. Hence, in VRASED$_A$, guessing the correct authentication value for a given request should be computationally infeasible without key knowledge.

2) PURE [19]: Proofs of update, reset and erasure: In addition to the malware detection provided by VRASED, PURE also offers remote capabilities to erase the data section of the device, update the program code, and reset the device – steps that need to be taken if malware is detected, or simply in case of a software update.

3) APEX [20]: Proofs of execution: APEX provides authenticated sensor readings and actuation: proof that the device executed the desired program (with a freshness guarantee) and that the results of the execution have not been tampered with. This is meant to solve the problem of malware infections between the time of attestation and execution.

4) RATA [21]: TOCTOU avoidance: Built on VRASED$_A$, this extension addresses the above time-of-check-time-of-use (TOCTOU) problem more broadly. It not only allows executing the software combined with an attestation, but it keeps track of whether the software has been tampered with between two attestations. It also allows for performance improvements, since subsequent attestations do not require running HMAC again as long as the software has not been altered.

5) Tiny-CFA [22]: Control-flow attestation: Tiny-CFA enables the verifier to conduct control flow attestation on the prover device. This is the only VRASED extension whose additional security properties were not verified.

B. VRASED assumptions

VRASED [15] explicitly assumes the following to hold for the implementation of the core:

A1 Program counter: The PC register contains the address of the executing instruction.

A2 Memory address: Whenever memory is accessed, the address bus contains its address, and the W_{en}/R_{en} signals are active.

A3 DMA: Whenever DMA accesses the memory, the DMA address bus contains the accessed memory address and the DMA_{en} signal is high.

A4 MCU reset: The reset handling cannot be modified, and registers are zeroed during reset.

A5 Interrupts: Triggering an interrupt sets the corresponding `irq` signal.

Furthermore, VRASED formulates two additional assumptions for the trusted compiler:

A6 Callee-Saves-Register: All registers used in a function are cleared before exiting.

A7 Semantic preservation: Functional correctness is preserved during compilation from C to MSP430 assembly.

C. Additional (not directly exploitable) VRASED flaws

This appendix lists an additional falsified assumption and an unmodeled feature that we did not find to be directly exploitable within the attacker model.

1) Compiler not clearing dirty register values: Assumption A6 unequivocally states that the compiler should clear all registers that are used in a function. The paper also claims that the `msp430-gcc` compiler used in the implementation satisfies this assumption.

a) Broken assumption: We found, however, that registers `r12-r15` are explicitly designated as "caller-save" in the `msp430-gcc` application binary interface (ABI) [39, §Register usage]. This means that their value may be clobbered, and the compiler is *not* required to restore or clear them at the end of the function.

If the HMAC function uses any of these registers to temporarily save key-dependent values, those may leak out and be visible to untrusted code, since the register values are not cleaned up manually by SW-Att either.

b) Attack: We experimentally confirmed that caller-save registers are indeed clobbered after execution of SW-Att. However, in our experiments, no sensitive data was leaked with the current implementation of the HMAC function and compiler settings, but this is not guaranteed to always be true if the compiler or the SW-Att implementation changes.

c) Mitigation: The most straightforward solution is to insert a custom assembly stub at the trusted exit point of SW-Att to clear all registers that can contain key-dependent data. Such an ABI sanitization stub resembles existing security solutions and best practices to prevent leakage through CPU registers in Sancus [16] and, more generally, in Intel SGX enclave shielding runtimes [52].

2) Reading the key with the debug unit: The openMSP430 architecture comes equipped with a debug unit connected to the core through UART or I^{2}C. This unit enables its user to read or write data in memory, pause the execution of the CPU, and read register contents.

a) Unmodeled capability: While the debug unit is not mentioned at all in the paper, it is included in the open-source implementation of VRASED.

b) Attack: Operating the debug unit requires physical access, so strictly speaking, it lies outside the attacker model of VRASED. However, since the debug unit lies outside VRASED's verification perimeter, it could also be extended to be controlled from software without violating any of the core assumptions in Appendix B (cf. the discussion in Section VI-D). As a more concrete example, even when the debug unit would adhere to the memory interface (A2) and interrupt (A5) assumptions, it could still be configured to schedule breakpoints or trivially read out CPU registers to leak the key.

c) Mitigation: This issue highlights the security risks of development interfaces that fall outside the verification perimeter. The easiest fix is to remove the debug unit altogether from the implementation, as it only causes possible sources of information leakage.

D. Timing attack on VRASED$_A$

Conforming to the attack described in Section VI-C3, Table III shows the number of cycles the entire execution of SW-Att takes with different authentication tokens given for the same key-challenge pair. For the given pair, the correct token starts with the bytes {`0x59, 0x76`}, as can be seen from the increasing execution times. Following this guessing for one byte at a time, the entire VRASED$_A$ authentication token can be extracted in linear effort.

TABLE III. Execution time of VRASED$_A$ for authentication guesses.

VRF_AUTH[32]	Execution time (cycles)
{0x1}	210,641
{0x0}	210,641
{0x59}	210,654
{0x59, 0x75}	210,654
{0x59, 0x76}	210,667

E. Analysis of end-to-end RA security argument

Soundness of remote attestation: VRASED's end-to-end remote attestation security argument [15] first relies on a separate argument that proves the soundness of the RA scheme. The soundness argument crucially relies on the proven functional correctness of the HACL* cryptographic library [37]. However, no argumentation is given how the unverified openMSP430 core satisfies the machine model assumed by HACL*. Particularly, even the formally verified HACL* library will clearly break when executed on a core which, for instance, performs subtractions for `add` or reverses the direction of `jmp` instructions. Observe that there is no assumption whatsoever that forbids this by saying that the core should be bug-free, free of hardware Trojans, or even adhere to some (formalized) MSP430 ISA specification. Hence, the soundness of the attestation and the proven guarantees of HACL* are trivially broken by a malicious core that e.g., performs subtractions for addition instructions and vice versa.

Security of remote attestation: We first provide an overview of how the end-to-end RA security property relies on two assumptions, which are further decomposed into sub-arguments. Enumeration items on the same level represent preconditions (which all need to be satisfied) for their parent item. '✗' indicates that the given step could be bypassed by a misbehaving core that still satisfies A1-A5. Steps for which the proof uses a model derived from Verilog are typeset in *italics*.

1) RA soundness ✗
2) The attacker does not learn the key ✗
   a) The key can only be learned through the memory ✗
      i) The key can only be learned through registers, memory, or SW-Att timing ✗
      ii) A6 forbids leakage through registers after exiting SW-Att ✗
      iii) HACL* prevents all possible SW-Att timing leakages ✗
   b) *Lemma 2: reading the key directly, or data that SW-Att wrote to unprotected regions will cause a reset* ✓
   c) Resets do not leak the key ✗
   d) The shared MR region cannot contain the key ✗

We already argued above how soundness (step 1) may be broken. Section VI-D, furthermore, contains a high-level description of how a misbehaving core could be constructed that still satisfies A1-A5, but breaks the assumption that HACL* prevents the key from leaking through SW-Att timing (step 2(a)iii). That section similarly describes how such a core could break the assumption that a reset never leaks the key (step 2c).

Step 2(a)i can be falsified similarly to the previous timing example, but in this case the timing of an unprotected instruction can change depending on the previously saved value of the key to falsify the claim in a misbehaving core.

For step 2(a)ii, leakage through register values is claimed to be eliminated by the assumption A6 and the secure reset property (P3). These prevent register values to be leaked after both successful and unsuccessful executions of SW-Att. Assumption A6 is directly falsified in Appendix C1. Furthermore, even when the A6 and P3 conditions are met, we can once again construct a malicious core to leak the key by not following the implicitly expected read and write semantics of registers. This core saves the value of the key in a shadow register during the HMAC calculation, and writes it to one of the real registers after the cleanup at the end of SW-Att execution.

For step 2d, it is stated that writes by SW-Att to the HMAC region do not have to be covered by the lemma, as this region cannot contain the key. This is directly falsified by our stack pointer overwrite attack (Section VI-C2). Moreover, a core could conceivably be constructed that simply dumps the key in the MR memory region after SW-Att execution.