Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors

Matteo Busi*, Job Noorman†, Jo Van Bulck†,
Letterio Galletta‡, Pierpaolo Degano*, Jan Tobias Mühlberg† and Frank Piessens†
* Dept. of Computer Science, Università di Pisa, Italy
† imec-DistriNet, Dept. of Computer Science, KU Leuven, Belgium
‡ IMT School for Advanced Studies Lucca, Italy

***Abstract*—Computer systems often provide hardware support for isolation mechanisms like privilege levels, virtual memory, or enclaved execution. Over the past years, several successful software-based side-channel attacks have been developed that break, or at least significantly weaken the isolation that these mechanisms offer. Extending a processor with new architectural or micro-architectural features, brings a risk of introducing new such side-channel attacks.**

This paper studies the problem of extending a processor with new features *without* weakening the security of the isolation mechanisms that the processor offers. We propose to use full abstraction as a formal criterion for the security of a processor extension, and we instantiate that criterion to the concrete case of extending a microprocessor that supports enclaved execution with secure interruptibility of these enclaves. This is a very relevant instantiation as several recent papers have shown that interruptibility of enclaves leads to a variety of software-based side-channel attacks. We propose a design for interruptible enclaves, and prove that it satisfies our security criterion. We also implement the design on an open-source enclave-enabled microprocessor, and evaluate the cost of our design in terms of performance and hardware size.

I. Introduction

Many computing platforms run programs coming from a number of different stakeholders that do not necessarily trust each other. Hence, these platforms provide mechanisms to prevent code from one stakeholder to interfere with code from other stakeholders in undesirable ways. These *isolation mechanisms* are intended to confine the interactions between two isolated programs to a well-defined communication interface. Examples of such isolation mechanisms include process isolation, virtual machine monitors, or enclaved execution [1].

However, security researchers have shown that many of these isolation mechanisms can be attacked by means of *software-exploitable side-channels*. Such side-channels have been shown to violate integrity of victim programs [2], [3], [4], as well as their confidentiality on both high-end processors [5], [6], [7], [8] and on small microprocessors [9]. In fact, over the past two years, many major isolation mechanisms have been successfully attacked: Meltdown [6] has broken user/kernel isolation, Spectre [7] has broken process isolation and software defined isolation, and Foreshadow [8] has broken enclaved execution on Intel processors.

The class of software-exploitable side-channel attacks is complex and varied. These attacks often exploit, or at least rely on, specific hardware features or hardware implementation details. Hence, for complex state-of-the-art processors there is a wide potential attack surface that should be explored (see for instance [10] for an overview of just the attacks that rely on transient execution). Moreover, the potential attack vectors vary with the attacker model that a specific isolation mechanism considers. For instance, enclaved execution is designed to protect enclaved code from malicious operating system software whereas process isolation assumes that the operating system is trusted and not under control of the attacker. As a consequence, protection against software-exploitable side-channel attacks is much harder for enclaved execution [11].

Hence, no silver-bullet solutions against this class of attacks should be expected, and countermeasures will likely be as varied as the attacks. They will depend on attacker model, performance versus security trade offs, and on the specific processor feature that is being exploited.

The objective of this paper is to study how to design and prove secure such countermeasures. In particular, we rigorously study the resistance of enclaved execution on small microprocessors [12], [13] against interrupt-based attacks [9], [14], [15]. This specific instantiation is important and challenging. First, interrupt-based attacks are very powerful against enclaved execution: fine-grained interrupts have been a key ingredient in many attacks against enclaved execution [16], [8], [17], [9]. Second, to the best of our knowledge, all existing implementations of interruptible enclaved execution are vulnerable to software-exploitable side-channels, including implementations that specifically aim for secure interruptibility [18], [13].

We base our study on the existing open-source Sancus platform [19], [12] that supports *non-interruptible* enclaved execution. We illustrate that achieving security is non-trivial through a variety of attacks enabled by supporting interruptibility of enclaves. Next, we provide a formal model of the existing Sancus and we then extend it with interrupts. We prove that this extension does not break isolation properties by instantiating full abstraction [20].

Roughly, we show that what the attacker can learn from (or do to) an enclave is exactly the same *before* and *after* adding the support for interrupts. In other words, adding interruptibility does not open new avenues of attack. Finally, we implement the secure interrupt handling mechanism as

DOI 10.1109/CSF49147.2020.00026

an extension to Sancus, and we show that the cost of the mechanism is low, in terms of both hardware complexity and performance.

In summary, the novel contributions of this paper are:

- We propose a specific design for extending Sancus, an existing enclaved execution system, with interrupts.
- We propose to use full abstraction [20] as a formal criterion of what it means to maintain the security of isolation mechanisms under processor extensions. Also, we instantiate it for proving that the mechanism of enclaved execution, extended to support interrupts, complies with our security definition.
- We implement the design on the open source Sancus processor, and evaluate cost in terms of hardware size and performance impact.[1]

The paper is structured as follows: in Section II we provide background information on enclaved execution and interrupt-based attacks. Section III provides an informal overview of our approach. Section IV discusses our formalization and sketches the proof, pointing to the online extended version [21] for full details. Then, in Section V we describe and evaluate our implementation. Section VI and VII discuss limitations, and the connection to related work. Finally, Section VIII offers our conclusions and plans for future work.

II. Background

a) Enclaved execution: Enclaved execution is a security mechanism that enables *secure remote computation* [22]. It supports the creation of *enclaves* that are initialized with a software module, and that have the following security properties. First, the software module in the enclave is isolated from all other software on the same platform, including system software such as the operating system. Second, the correct initialization of an enclave can be *remotely attested*: a remote party can get cryptographic assurance that an enclave was properly initialized with a specific software module (characterized by a cryptographic hash of the binary module). These security properties are guaranteed while relying on a small trusted computing base, for instance trusting only the hardware [12], [1], or possibly also a small hypervisor [23], [24].

The remote attestation aspect of enclaved execution is important for the secure initialization of enclaves, and for setting up secure communication channels to the enclave. However, it does not play an important role for the interrupt-driven attacks that we study in this paper, and hence we will focus here on the isolation aspect of enclaves only. Other papers describe in detail how remote attestation and secure communication work on large [22] or small systems [12], [13].

The isolation guarantees offered to an enclaved software module are the following. The module consists of two contiguous memory sections, a *code section*, initialized with the machine code of the module, and a *data section*. The data section is initialized to zero, and loading of confidential data happens through a secure channel to the enclave, after attesting the correct initialization of the module. For instance, confidential data can be restored from cryptographically sealed storage, or can be obtained from a remote trusted party.

The enclaved execution platform guarantees that: (1) the data section of an enclave is *only* accessible while executing code from the code section, and (2) the code section can only be entered through one or more designated *entry points*.

These isolation guarantees are simple, but they offer the useful property that *data of a module can only be manipulated by code of the same module*, i.e., an encapsulation property similar to what programming languages offer through classes and objects. Untrusted code residing in the same address space as the enclave but outside the enclave code and data sections can interact with the enclave by jumping to an entry point. The enclave can return control (and computation results) to the untrusted code by jumping back out.

b) Interrupt-based attacks: Enclaved execution is designed to be resistant against a very strong attacker that controls all other software on the platform, including privileged system software. While isolating enclaves is well-understood at the architectural level, including even successful formal verification efforts [24], [25], researchers have shown that it is challenging to protect enclaves against side-channels. Particularly, a recent line of work on *controlled channel* attacks [11], [15], [9], [26], [16] has demonstrated a new class of powerful, low-noise side-channels that leverage the adversary's increased control over the untrusted operating system.

A specific consequence of this strong model is that the attacker also controls the scheduling and handling of interrupts: the attacker can precisely schedule interrupts to arrive during enclaved execution, and can choose the code to handle these interrupts. This power has been put to use for instance to single-step through an enclave [15], or to mount a new class of ingenious *interrupt latency* attacks [9], [14] that derive individual enclaved instruction timings from the time it takes to dispatch to the untrusted operating system's interrupt handler. We provide concrete examples of interrupt-based attacks in the next section, after detailing our model of enclaved execution.

While advanced CPU features such as virtual memory [11], [26], [8], branch prediction [16], [17] or caching [27] are known to leak information on high-end processors, pure interrupt-based attacks such as interrupt latency measurements are the *only* known controlled-channel attack against low-end enclaved execution platforms lacking these advanced features. Moreover, they have been shown to be very powerful: e.g., Van Bulck et al. [9] have shown how to efficiently extract enclave secrets like passwords or PINs from embedded enclaves.

Some enclaved execution designs avoid the problem of interrupt-based attacks by completely disabling interrupts during enclave execution [12], [25]. This has the important downside that system software can no longer guarantee availability: if an enclaved module goes into an infinite loop, the system cannot progress. All designs that do support interruptibility of enclaves [18], [13] are vulnerable to these attacks.

[1]Our implementation is available online at https://github.com/sancus-pma/sancus-core/tree/nemesis.

Instr. i	Meaning	Cycles	Size
RETI	Returns from interrupt.	5	1
NOP	No-operation.	1	1
HLT	Halt.	1	1
NOT r	$r \leftarrow \neg r$. (Emulated in MSP430)	2	2
IN r	Reads word from the device and puts it in r.	2	1
OUT r	Writes word in register r to the device.	2	1
AND r_1 r_2	$r_2 \leftarrow r_1 \,\&\, r_2$.	1	1
JMP &r	Sets pc to the value in r.	2	1
JZ &r	Sets pc to the value in r if bit 0 in sr is set.	2	1
MOV r_1 r_2	$r_2 \leftarrow r_1$.	1	1
MOV @r_1 r_2	Loads in r_2 the word in starting in location pointed by r_1.	2	1
MOV r_1 0(r_2)	Stores the value of r_1 starting at location pointed by r_2.	4	2
MOV #w r_2	$r_2 \leftarrow w$.	2	2
ADD r_1 r_2	$r_2 \leftarrow r_1 + r_2$.	1	1
SUB r_1 r_2	$r_2 \leftarrow r_1 - r_2$.	1	1
CMP r_1 r_2	Zero bit in sr set if $r_2 - r_1$ is zero.	1	1

Table I
SUMMARY OF THE ASSEMBLY LANGUAGE CONSIDERED.

III. OVERVIEW OF OUR APPROACH

We set out to design an interruptible enclaved execution system that is provably resistant against interrupt-based attacks. This section discusses our approach informally, later sections discuss a formalization with security proofs, and report on implementation and experimental evaluation.

We base our design on Sancus [12], an existing open-source enclaved execution system. We first describe our Sancus model, and discuss how extending Sancus with interrupts leads to the attacks mentioned in Section II-b. In other words, we show how extending Sancus with interrupts breaks some of the isolation guarantees provided by Sancus.

Then, we propose a formal security criterion that defines what it means for interruptibility to *preserve the isolation properties*, and we illustrate that definition with examples.

Finally, we propose a design for an interrupt handling mechanism that is resistant against the considered attacks and that satisfies our security definition. Crucial to our design is the assumption that the timing of individual instructions is predictable, which is typical of "small" microprocessors, like Sancus. Although tailored here on a specific architecture and a specific class of attacks, we expect our approach of ensuring that the same attacks are possible before and after an architecture extension to be applicable in other settings too.

A. Sancus model

a) Processor: Sancus is based on the TI MSP430 16-bit microprocessor [28], with a classic von Neumann architecture where code and data share the same address space. We formalize the subset of instructions summarized in Table I that is rich enough to model all the attacks we care about. We have a subset of memory-to-register and register-to-memory transfer instructions; a comparison instruction; an unconditional and a conditional jump; and basic arithmetic instructions.

b) Memory: Sancus has a byte addressable memory of at most 64KB, where a finite number of enclaves can be defined. The bound on the number of enclaves is a parameter set at processor synthesis time. In our model, we assume that there is only a single enclave, made of a *code section*, initialized with the machine code of the module, and a *data section*. A data section is securely provisioned with data by relying on remote attestation and secure communication, not modeled here as they play no role in the interrupt-based attacks we care about in this paper. Instead, our model allows direct initialization of the data section with confidential enclave data. All the other memory is *unprotected memory*, and will be considered to be under control of the attacker.

Enclaves have a single entry point; the enclave can only be entered by jumping to the first address of the code section. Multiple *logical entry points* can easily be implemented on top of this single physical entry point. Control flow can leave the enclave by jumping to any address in unprotected memory. Obviously, a compiler can implement higher-level abstractions such as enclave function calls and returns, or out-calls from the enclave to functions in the untrusted code [12].

Sancus enforces program counter (pc) based memory access control. If the pc is in unprotected memory, the processor can not access any memory location within the enclave – the only way to interact with the enclave is to jump to the entry point. If the pc is within the code section of the enclave, the processor can only access the enclave data section for reading/writing and the enclave code section for execution. This access control is faithfully rendered in our model, via the predicate MAC in Table II.

c) I/O devices: Sancus uses memory-mapped I/O to interact with peripherals. One important example of a peripheral for the attacks we study is a cycle accurate timer, which allows software to measure time in terms of the number of CPU cycles. In our model, we include a single very general I/O device that behaves as a state machine running synchronously to CPU execution. In particular, it is trivial to instantiate this general I/O device to a cycle-accurate timer.

Instead of modeling memory-mapped I/O, we introduce two special instructions that allow writing/reading a word to/from the device (see Table I). Actually these instructions are shorthands, which are easy to macro-expand, at the price of dealing with special cases in the execution semantics for any memory operation. For instance, software could read the current cycle timer value from a timer peripheral by using the IN instruction.

The I/O devices can request to interrupt the processor with single-cycle accuracy. The original Sancus disables interrupts during enclaved execution. One of the key objectives of this paper is to propose a Sancus extension that does handle such interrupts without weakening security. Hence, we will define two models of Sancus, one that ignores interrupts, and one that handles them even during enclaved execution.

B. Security definitions

a) Attacker model: An attacker controls the entire *context* of an enclave, that is: he controls (1) all of unprotected memory (including code interacting with the enclave, as well as data in unprotected memory), and (2) the connected device. This is the standard attacker model for enclaved execution. In particular, it implies that the attacker has complete control over the Interrupt Service Routines.

b) Contextual equivalence formalizes isolation: Informally, our security objective is extending the Sancus processor without weakening the isolation it provides to enclaves. What isolation achieves is that attackers can not see "inside" an enclave, so making it possible to "hide" enclave data or implementation details from the attacker. We formalize this concept of isolation precisely by using the notion of *contextual equivalence* or *contextual indistinguishability* (as first proposed by Abadi [20]). Two enclaved modules M_1 and M_2 are contextually equivalent, if the attacker can not distinguish them, i.e., if there exists no context that tells them apart. We discuss this on the following example.

Example 1 (Start-to-end timing)**.** *The following enclave compares a user-provided password in* $\mathtt{R}_{15}$ *with a secret in-enclave password at address* pwd_adrs*, and stores the user-provided value in* $\mathtt{R}_{14}$ *into the enclave location at* $store\_adrs$ *if the user password was correct.*

```
1  enclave_entry:
2      /* Load addresses for comparison */
3      MOV #store_adrs, r10     ; 2 cycles
4      MOV #access_ok, r11      ; 2 cycles
5      MOV #endif, r12          ; 2 cycles
6      MOV #pwd_adrs, r13       ; 2 cycles
7      /* Compare user vs. enclave password */
8      MOV @r13, r13            ; 2 cycles
9      CMP r13, r15             ; 1 cycle
10     JZ  &r11                 ; 2 cycles
11 access_fail:   /* Password fail: return */
12     JMP &r12                 ; 2 cycles
13 access_ok:  /* Password ok: store user val */
14     MOV r14, 0(r10)          ; 4 cycles
15 endif:  /* Clear secret enclave password */
16     SUB r13, r13             ; 1 cycle
17 enclave_exit:
```

In the absence of a timer device, this enclave successfully hides the in-enclave password. If we take enclaves M_1 and M_2 to be two instances of Example 1, differing only in the value for the secret password, then M_1 and M_2 are indistinguishable for any context that does not have access to a cycle accurate timer: all such a context can do is call the entry point, but the context does not get any indication whether the user-provided password was correct. This formalizes that enclave isolation successfully "hides" the password.

However, with the help of a cycle accurate timer, the attacker can distinguish M_1 and M_2 as follows. The attacker can create a context that measures the start-to-end execution time of an enclave call: the context reads the timer right before jumping to the enclave. On enclave exit, the context reads the timer again to compute the total time spent in the enclave.

In order to reason about execution timing, we represent enclaved executions as an ordered array of individual instruction timings. (Table I conveniently specifies how many cycles it takes to execute each instruction.) Hence the two possible control flow paths of the above program are: `ok=[2,2,2,2,2,1,2,4,1]` for the "access_ok" branch, or `fail=[2,2,2,2,2,1,2,2,1]` for the "access_fail" branch. Since `sum(ok) = 18` and `sum(fail) = 16`, the context can distinguish the two control flow paths, and hence can distinguish M_1 and M_2 (and by launching a brute-force attack [29], can also extract the secret password).

This example illustrates how contextual equivalence formalizes isolation. It also shows that the original Sancus already has some side-channel vulnerabilities under our attacker model. Since we assume the attacker can use any I/O device, he can choose to use a timer device and mount the start-to-end timing attack we discussed.

It is important to note that it is *not* our objective in this paper to close these existing side-channel vulnerabilities in Sancus. Our objective is to make sure that extending Sancus with interrupts does not introduce *additional* side-channels, i.e., that this does not *weaken* the isolation properties of Sancus.

For existing side-channels, like the start-to-end timing side-channel, countermeasures can be applied by the enclave programmer. For instance, the programmer can balance out the various secret-dependent control-flow paths as in Example 2.

Example 2 (Interrupt latency)**.** *Consider the program of Example 1, balanced in terms of overall execution time by adding two* `NOP` *instructions at lines 13-14. The two possible control flow paths are:* `ok=[2,2,2,2,2,1,2,4,1]` *vs.* `fail=[2,2,2,2,2,1,2,1,1,2,1]`*. Since* `sum(ok)` *is equal to* `sum(fail)`*, the start-to-end timing attack is mitigated.*

```
1  enclave_entry:
2      /* Load addresses for comparison */
3      MOV #store_adrs, r10     ; 2 cycles
4      MOV #access_ok, r11      ; 2 cycles
5      MOV #endif, r12          ; 2 cycles
6      MOV #pwd_adrs, r13       ; 2 cycles
7      /* Compare user vs. enclave password */
8      MOV @r13, r13            ; 2 cycles
9      CMP r13, r15             ; 1 cycle
10     JZ  &r11                 ; 2 cycles
11 access_fail:
12     /* Password fail: constant time return */
13     NOP                      ; 1 cycle
14     NOP                      ; 1 cycle
15     JMP &r12                 ; 2 cycles
16 access_ok: /* Password ok: store user val */
17     MOV r14, 0(r10)          ; 4 cycles
18 endif:   /* Clear secret enclave password */
19     SUB r13, r13             ; 1 cycle
20 enclave_exit:
```

c) Interrupts can weaken isolation: We now show that a straightforward implementation of interrupts in the Sancus processor would significantly weaken isolation. Consider an implementation of interrupts similar to the TI MSP430: on arrival of an interrupt, the processor first completes the ongoing instruction, and then jumps to an interrupt service routine.

The program in Example 2 is secure on Sancus without interrupts. However, it is not secure against a malicious context that can schedule interrupts to be handled while the enclave executes. To see why, assume that an interrupt is scheduled by the malicious context to arrive within the first cycle after the conditional jump at line 10. If the jump was taken then the instruction being executed is the 4-cycle `MOV` at line 18, otherwise the current instruction is the 1-cycle `NOP` at line 13. Now, since the attacker's interrupt handler will only be called *after* completion of the current instruction, the adversary observes an interrupt latency difference of 3 cycles, depending on the secret branch condition inside the enclave. Researchers [9]

have shown how interrupt latency can be practically measured to precisely reconstruct individual enclave instruction timings on both high-end and low-end enclave processors.

Using this attack technique, a context can again distinguish two instances of the module with a different password, and hence the addition of interrupts has *weakened* isolation.

A strawman solution to fix the above timing leakage is to modify the implementation of interrupt handling in the processor to always dispatch interrupt service routines in constant time T, i.e., regardless of the execution time of the interrupted instruction. We show in the two examples below, however, that this is a necessary but not sufficient condition.

Example 3 (Resume-to-end timing)**.** *Consider the program from Example 2 executed on a processor which always dispatches interrupts in constant time* T*. The attacker schedules an interrupt to arrive in the first cycle after the* JZ *instruction, yielding constant interrupt latency* T*. Next, the context resumes the enclave and measures the time it takes to let the enclave run to completion* without *further interrupts. While interrupt latency timing differences are properly masked, the time to complete enclave execution after resume from the interrupt is 1 cycle for the* `ok` *path and 4 cycles for the* `fail` *path.*

Example 4 (Interrupt-counting attack)**.** *An alternative way to attack the program from Example 2 even when interrupt latency is constant, is to* count *how often the enclave execution can be interrupted, e.g., by scheduling a new interrupt 1 cycle after resuming from the previous one. Since interrupts are handled on instruction boundaries, this allows the attacker to count the number of instructions executed in the enclave, and hence to distinguish the two possible control flow paths.*

d) Defining the security of an extension: The examples above show how a new processor feature (like interrupts) can weaken isolation of an existing isolation mechanism (like enclaved execution), and this is exactly what we want to avoid. Here we propose and implement a provably secure defense against these attacks. With this background, our security definition is now obvious. Given an original system (like Sancus), and an extension of that system (like interruptible Sancus), that extension is secure if and only if it does not change the contextual equivalence of enclaves. Enclaves that are contextually equivalent in the original system must be contextually equivalent in the extended system and vice versa (we shall formalize this as a *full abstraction* property later on).

C. Secure interruptible Sancus

Designing an interrupt handling mechanism that is secure according to our definition above is quite subtle. We illustrate some of the subtleties. In particular, we provide an intuition on how an appropriate use of padding can handle the various attacks discussed above. We also discuss how other design aspects are crucial for achieving security. In this section, we just provide intuition and examples. The ultimate argument that our design is secure is our proof, discussed later.

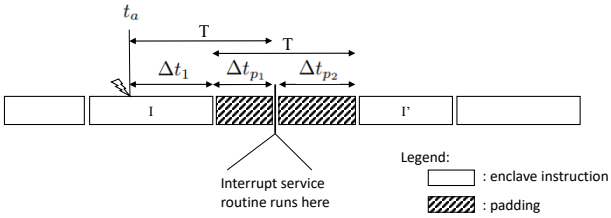

Figure 1. The secure padding scheme.

a) Padding: We already discussed that it is insufficient for security to naively pad interrupt latency to make it constant. We need a padding approach that handles all kinds of attacks, including the example attacks discussed above.

The following padding scheme works (see Figure 1). Suppose the attacker schedules the interrupt to arrive at t_a, during the execution of instruction I in the enclave. Let Δt_1 be the time needed to complete execution of I. To make sure the attacker can not learn anything from the interrupt latency, we introduce padding for Δt_{p_1} cycles where Δt_{p_1} is computed by the interrupt handling logic such that $\Delta t_1 + \Delta t_{p_1}$ is a constant value T. This value T should be chosen as small as possible to avoid wasting unnecessary time, but must be larger than or equal to the maximal instruction cycle time `MAX_TIME` (to make sure that no negative padding is required, even when an interrupt arrives right at the start of an instruction with the maximal cycle time). This first padding ensures that an attacker always measures a constant interrupt latency.

But this alone is not enough, as an attacker can now measure resume-to-end time as in Example 3. Thus, we provide a second kind of padding. On return from an interrupt, the interrupt handling logic will pad again for Δt_{p_2} cycles, ensuring that $\Delta t_{p_1} + \Delta t_{p_2}$ is again the constant value T (i.e., $\Delta t_{p_2} = \Delta t_1$). This makes sure that the resume-to-end time measured by the attacker does not depend on the instruction being interrupted.

This description of our padding scheme is still incomplete: it is also important to specify what happens if a new interrupt arrives while the interrupt handling logic is still performing padding because of a previous interrupt. This is important to counter attacks like that of Example 4. We refer to the formal description for the complete definition.

Intuitively, the property we get is that (1) an attacker can schedule an interrupt at any time t_a during enclave execution, (2) that interrupt will always be handled with a constant latency T, (3) the resume-to-end time is always exactly the time the enclave still would have needed to complete execution from point t_a if it had not been interrupted.

This double padding scheme is a main ingredient of our secure interrupt handling mechanism, but many other aspects of the design are important for security. We briefly discuss a number of other issues that came up during the security proof.

b) Saving execution state on interrupt: When an enclaved execution is interrupted, the processor state (contents of the registers) is saved (to allow resuming the execution

once the interrupt is handled) and is cleared (to avoid leaking confidential register contents to the context). A straightforward implementation would be to store the processor state on the enclave stack. However, the proof of our security theorem showed that storing the processor state in enclave accessible memory is not secure: consider two enclaved modules that monitor the content of the memory area where processor state is saved, and behave differently on observing a change in the content of this memory area. These modules are contextually equivalent in the absence of interrupts (as the contents of this memory area will never change), but become distinguishable in the presence of interrupts. Hence, our design saves processor state in a storage area *inaccessible* to software.

c) No access to unprotected memory from within an enclave: Most designs of enclaved execution allow an enclave to access unprotected memory (even if this has already been criticized for security reasons [30]). However, for a single core processor, interruptibility significantly weakens contextual equivalence for enclaves that can access unprotected memory. Consider an enclave M_1 that always returns a constant 0, and an enclave M_2 that reads twice from the same unprotected address and returns the difference of the values read. On a single-core processor without interrupts, M_2 will also always return 0, and hence is indistinguishable from M_1. But an interrupt scheduled to occur between the two reads from M_2 can change the value returned by the second read, and hence M_1 and M_2 become distinguishable. Hence, our design forbids enclaves to access unprotected memory.

For similar reasons, our design forbids an interrupt handler to reenter the enclave while it has been interrupted, and forbids the enclave to directly interact with I/O devices.

Finally, we prevent the interrupt enable bit (`GIE`) in the status register from being changed by software in the enclave, as such changes are unobservable in the original Sancus and they would be observable once interruptibility is added.

While the security proof is a significant amount of effort, an important benefit of this formalization is that it forced us to consider all these cases and to think about secure ways of handling them. We made our design choices to keep model and proof simple, and these choices may seem restrictive. Section VI discusses the practical impact of these choices.

IV. Formalization and Security Proofs

We proceed to formally define two Sancus models, one describing the original, uninterruptible Sancus (Sancus$^{\mathsf{H}}$, Sancus-High) and one describing the secure interruptible Sancus (**Sancus$^{\mathsf{L}}$**, Sancus-Low).[2] The two share most of their structure and just differ in the way they deal with interrupts.

Given the semantics of Sancus$^{\mathsf{H}}$ and **Sancus$^{\mathsf{L}}$**, we formally show that the two versions of Sancus actually provide the same security guarantees, i.e., the isolation mechanism is not broken by adding a carefully designed interruptible enclaved execution. Technically, this is done through the *full abstraction* theorem between Sancus$^{\mathsf{H}}$ and **Sancus$^{\mathsf{L}}$** (Theorem IV.1). Note that, our theorem guarantees that the *same* program has the *same* security guarantees both in Sancus$^{\mathsf{H}}$ and **Sancus$^{\mathsf{L}}$**.

Space limitations prevent us from discussing all the details of our formalization and we refer the reader to the extended version [21] for all the missing details.

A. Setting up our formal framework

a) Memory and memory layout: The memory is modeled as a (finite) function mapping 2^{16} locations to bytes b. Given a memory $\mathcal{M}$, we denote the operation of retrieving the byte associated to the location l as $\mathcal{M}(l)$. On top of that, we define read and write operations on words (i.e., pairs of bytes) and we write $w = b_1 b_0$ to denote that the most significant byte of a word w is b_1 and its least significant byte is b_0.

The read operation is standard: it retrieves two consecutive bytes from a given memory location l (in a little-endian fashion, as in the MSP430):

$$\mathcal{M}[l] \triangleq b_1 b_0 \quad \text{if } \mathcal{M}(l) = b_0 \wedge \mathcal{M}(l+1) = b_1$$

We define the write operation as follows

$$(\mathcal{M}[l \mapsto b_1 b_0])(l') \triangleq \begin{cases} b_0 & \text{if } l' = l \\ b_1 & \text{if } l' = l+1 \\ \mathcal{M}(l') & \text{o.w.} \end{cases}$$

Writing $b_0 b_1$ in location l in $\mathcal{M}$ means to build an updated memory mapping l to b_0, $l+1$ to b_1 and unchanged otherwise.

Note that reads and writes to $l =$ `0xFFFF` are undefined ($l+1$ would overflow hence it is undefined). The memory access control explicitly forbids these accesses (see below). Also, the write operation deals with unaligned memory accesses (cfr. case $l' = l + 1$). We faithfully model these aspects to prove that they do not lead to potential attacks.

A memory layout $\mathcal{L} \triangleq \langle ts, te, ds, de, isr \rangle$ describes how the enclave and the *interrupt service routine* (ISR) are placed in memory and is used to check memory accesses during the execution of each instruction (see below). The protected code section is denoted by $[ts, te)$, $[ds, de)$ is the protected data section, and isr is the address of the ISR. The protected code and data sections do not overlap and the first address of the protected code section is the single entry point of the enclave. Finally, we reserve the location `0xFFFE` to store *the address of* the first instruction to be executed when the CPU starts or when an exception happens, reflecting the behavior of MSP430. Thus, `0xFFFE` must be outside the enclave sections and different from isr.

b) Registers: There are sixteen 16-bit registers, three of which $\mathtt{R}_0$, $\mathtt{R}_1$, $\mathtt{R}_2$ have dedicated functions, whereas the others are for general use. ($\mathtt{R}_3$ is a constant generator in the MSP430, but we ignore that use in our formalization.) More precisely, $\mathtt{R}_0$ (hereafter denoted as `pc`) is the program counter and points to the next instruction to be executed. Instruction accesses are performed by word and the `pc` is aligned to even addresses. The register $\mathtt{R}_1$ (`sp` hereafter) is the stack pointer and is aligned

[2] The *high* and *low* terminology is inherited from the field of *secure compilation* of *high* source languages to *low* target ones. Also, for readability we hereafter highlight in blue, sans-serif font elements of Sancus$^{\mathsf{H}}$, in **red, bold** font elements of **Sancus$^{\mathsf{L}}$** and in black those that are in common.

to even addresses. Since for the time being we do not model instructions for procedure calls, the only special use of the stack pointer in our model is to store the state while handling an interrupt (see below). The register $\mathtt{R}_2$ (`sr` hereafter) is the status register and contains different pieces of information encoded as flags. The most important for us is the fourth bit, called `GIE`, set to `1` when interrupts are enabled. Other bits signal, e.g., when an operation produces a carry or when an operation returns zero.

Formally, our *register file* $\mathcal{R}$ is a function that maps each register `r` to a word. While read operation is standard, the write operation models some invariants enforced by the hardware:

$$\mathcal{R}[\mathtt{r}] \triangleq w \text{ if } \mathcal{R}(\mathtt{r}) = w$$

$$\mathcal{R}[\mathtt{r} \mapsto w] \triangleq \lambda[\mathtt{r}'].\begin{cases} w\&\mathtt{0xFFFE} \\ \qquad \text{if } \mathtt{r}' = \mathtt{r} \wedge (\mathtt{r} = \mathtt{pc} \vee \mathtt{r} = \mathtt{sp}) \\ (w\&\mathtt{0xFFF7}) \mid (\mathcal{R}[\mathtt{sr}]\&\mathtt{0x8}) \\ \qquad \text{if } \mathtt{r}' = \mathtt{r} = \mathtt{sr} \wedge \mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{PM} \\ w \qquad \text{if } \mathtt{r}' = \mathtt{r} \wedge (\mathtt{r} \neq \mathtt{pc} \wedge \mathtt{r} \neq \mathtt{sp}) \\ \mathcal{R}[\mathtt{r}'] \quad \text{o.w.} \end{cases}$$

More specifically, the least-significant bit of the program counter and of the stack pointer are *always* masked to 0 (as is also the case in the MSP430), and the `GIE` bit of the status register is always masked to its previous value when in protected mode (i.e., it cannot be changed when the CPU is running protected code, cf. the discussion in Section III). Note that in the definition above we use the relation $\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{m}$, for $\mathtt{m} \in \{\mathtt{PM}, \mathtt{UM}\}$ made precise below: roughly it denotes that the execution is in *protected* or in *unprotected* mode (i.e, execution is within, respectively outside the enclave).

c) I/O Devices: I/O devices are (simplified) *deterministic I/O automata* $\mathcal{D} \triangleq \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle$ over a common signature A containing the following actions a (below, w is a word): (i) ϵ, a silent, internal action; (ii) $rd(w)$, an output action (i.e., read request from the CPU); (iii) $wr(w)$, an input action (i.e., write request from the CPU); (iv) $int?$, an output action indicating an interrupt is raised. The transition function $\delta \overset{a}{\rightsquigarrow}_D \delta'$ models the device in state δ performing action $a \in A$ and moving to state δ', and δ_{init} is the initial state.

d) Contexts, software modules and whole programs: We call *software module* a memory $\mathcal{M}_M$ containing both protected data and code sections. A *context* C is a pair $\langle \mathcal{M}_C, \mathcal{D} \rangle$, where $\mathcal{D}$ is a device and $\mathcal{M}_C$ defines the contents of all memory locations *outside* the protected sections of the layout, thus disjoint from $\mathcal{M}_M$. Intuitively, the context is the part of the whole program that can be manipulated by an attacker. Given a context C and a software module $\mathcal{M}_M$, we define a *whole program* as $C[\mathcal{M}_M] = \langle \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{D} \rangle$.

e) Instruction set: We consider a subset of the MSP430 instructions plus our I/O instructions; they are in Table I. For each instruction the table includes its operands, an informal description of its semantics, its duration and the number of words it occupies in memory. The durations are used to define the function $cycles(i)$. In our model, we let $\mathtt{MAX\_TIME} = 6$, because the longest MSP430 instructions take 6 cycles (typically those for moving words within memory [28], none of which are displayed in Table I). Instructions are stored in the memory $\mathcal{M}$. We use the meta-function $decode(\mathcal{M}, l)$ that decodes the contents of the cell(s) starting at location l, returning an instruction in the table if any and $\perp$ otherwise.

f) Configurations: Given an I/O device $\mathcal{D}$, the state of the Sancus system is described by configurations of the form:

$$c \triangleq \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \in \mathbb{C}, \quad \text{where}$$

(i) δ is the current state of the I/O device; (ii) t is the current time of the CPU; (iii) t_a is either the arrival time of the last pending interrupt, or $\perp$ if there are none (this value may persist across multiple instructions); (iv) $\mathcal{M}$ is the current memory; (v) $\mathcal{R}$ is the current content of the registers; (vi) pc_{old} is the value of the program counter before executing the current instruction; (vii) $\mathcal{B}$ is called the *backup*, is software inaccessible storage space to save enclave state (registers, the old program counter and the remaining padding time) while handling an interrupt raised in protected mode.

The initial configuration for a whole program $C[\mathcal{M}_M] = \langle \mathcal{M}, \mathcal{D} \rangle$ is:

$$\text{INIT}_{C[\mathcal{M}_M]} \triangleq \langle \delta_{\text{init}}, 0, \perp, \mathcal{M}, \mathcal{R}^{init}_{\mathcal{M}_C}, \mathtt{0xFFFE}, \perp \rangle \text{ where}$$

(i) the state of the I/O device $\mathcal{D}$ is δ_{init}; (ii) the initial value of the clock is 0 and no interrupt has arrived yet; (iii) the memory is initialized to the whole program memory $\mathcal{M}_C \uplus \mathcal{M}_M$; (iv) all the registers are set to 0 except that `pc` is set to `0xFFFE` (the address from which the CPU gets the initial program counter), and that `sr` is set to `0x8` (the register is clear except for the `GIE` flag); (v) the previous program counter is also initialized to `0xFFFE`; (vi) the backup is set to $\perp$ to indicate absence of any backup.

Dually, HALT is the only configuration denoting termination, more specifically it is an opaque and distinguished configuration that indicates graceful termination.

Also, we define *exception handling* configurations, that model what happens on soft reset of the machine (e.g. on a memory access violation, or a halt in protected mode). On such a soft reset, control returns to the attacker by jumping to the address stored in location `0xFFFE`:

$$\text{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle} \triangleq \\ \langle \delta, t, \perp, \mathcal{M}, \mathcal{R}_0[\mathtt{pc} \mapsto \mathcal{M}[\mathtt{0xFFFE}]], \mathtt{0xFFFE}, \perp \rangle.$$

g) I/O device wrapper: Since the class of interrupt-based attacks requires a cycle-accurate timer, it is convenient to synchronize the CPU and the device time by forcing the device to take as many steps as the number of cycles consumed for each instruction by the CPU. The following "wrapper" around the device $\mathcal{D}$ models this synchronization:

$$\mathcal{D} \vdash \delta, t, t_a \curvearrowright^k_D \delta', t', t'_a$$

Assuming that the device was in state δ, at time t, and the last pending interrupt was raised at time t_a, then this wrapper defines for k cycles later: the new time $t' = t + k$, the new last

		t			
		Entry Point	Prot. code	Prot. Data	Other
f	Entry Point/Prot. code	r-x	r-x	rw-	–x
	Other	–x	—	—	rwx

Table II
DEFINITION OF $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$, WHERE f AND t ARE LOCATIONS.

pending interrupt time t'_a, and the new device state δ'. When no interrupt has to be handled, t_a and t'_a are $\perp$.

h) CPU mode and memory access control: The last two relations used by the main transition systems are the *CPU mode* and the *memory access control*, MAC. The first tells when a given program counter value, pc, is an address in the protected code memory (PM) or in the unprotected one (UM):

$$pc \vdash_{mode} \texttt{m}, \text{ with } \texttt{m} \in \{\texttt{PM}, \texttt{UM}\}$$

(Also, for simplicity, the relation is lifted to configurations.) The second one

$$i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK}$$

holds whenever the instruction i can be executed in a CPU configuration in which the previous program counter is pc_{old}, the registers are $\mathcal{R}$ and the backup is $\mathcal{B}$. More precisely, it uses the predicate $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$ (see Table II) that holds whenever from the location f we have the rights rght on location t. The predicate checks that (1) the code we came from (i.e., that in location pc_{old}) can actually execute the instruction i located at $\mathcal{R}[\texttt{pc}]$; (2) i can be executed in current CPU mode; and (3) we have the rights to perform i from $\mathcal{R}[\texttt{pc}]$, when i is a memory operation.

B. Sancus$^{\text{H}}$: a model of the original Sancus

Our models of Sancus are defined by means of two transition systems: a main one and an auxiliary one. The first system defines the operational semantics of instructions, and relies on the auxiliary system to specify the behavior upon interrupts.

a) Main transition system: The main transition system describes how the Sancus$^{\text{H}}$ configurations evolve during the execution, whose steps are represented by transitions of the following form, where $\mathcal{D}$ is an I/O device and $c, c' \in \mathbb{C}$:

$$\mathcal{D} \vdash c \to c'$$

Figure 2 reports some selected rules among those defining the main transition system. The first shows how the model deals with violations in protected mode: if an instruction can not be executed according to the memory-access control relation then a transition to the *exception handling* configuration happens. Rule (CPU-MovL) is for when the current instruction i loads in $\texttt{r}_2$ the word in memory at the position pointed by $\texttt{r}_1$. Its first premise checks if the instruction can be executed; the second one increments the program counter by 2 and loads in $\texttt{r}_2$ the value $\mathcal{M}[\texttt{r}_1]$; the third premise registers in the device that i requires $cycles(i)$ cycles to complete; and the last one executes the interrupt logic to check whether an interrupt needs to be handled or not (see comment below). Another interesting rule is (CPU-IN) that deals with the case in which the instruction reads a word from the device and puts the result in r. Its second premise holds when the device sends the word w to the CPU; the others are similar to those of (CPU-MovL).

b) Interrupt logic: The auxiliary transition system for Sancus$^{\text{H}}$ specifies the interrupt logic, and has the form:

$$\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc_{old}, \mathcal{B}' \rangle.$$

Since Sancus$^{\text{H}}$ ignores all interrupts, even in unprotected mode, the transition system always leaves the configuration unchanged.

Actually, one could remove the premise with the auxiliary transition system from all the rules defining the semantics of Sancus$^{\text{H}}$, as it always holds. However, it is convenient keeping them both to ease the presentation of the transition system of Sancus$^{\text{L}}$, and for technical reasons, as well.

C. Sancus$^{\text{L}}$: secure interruptible Sancus

We now define the semantics of Sancus$^{\text{L}}$, the *secure interruptible Sancus*, formalizing the mitigation outlined in Section III. We start by describing the main difference with that of Sancus$^{\text{H}}$, i.e., the way interrupts are handled.

a) Interrupt logic: Figure 3 shows the relevant rules of the auxiliary transition system describing the interrupt logic of Sancus$^{\text{L}}$. Now interrupts are handled both in unprotected and protected mode, modeled by the rules (INT-UM-P) and (INT-PM-P), resp. For the first case there is the premise $pc_{old} \vdash_{mode} \texttt{UM}$, for the second $pc_{old} \vdash_{mode} \texttt{PM}$ (i.e., the mode in which the last instruction was executed). Both rules have a premise requiring that the GIE bit of the status register is set to 1 and that an interrupt is on ($t_a \neq \perp$). (If this is not the case, two further rules, not displayed, just leave the configuration untouched, and keep the value of t_a unchanged.) A premise of (INT-UM-P) concerns registers: the program counter gets the entry point of the handler; the status register gets 0; and the top of the stack is moved 4 positions ahead. Accordingly, the new memory $\mathcal{M}'$ updates the locations pointed by the relevant elements of the stack with the current program counter and the contents of the status register. The last premise specifies that this interrupt handling takes 6 cycles.

The rule (INT-PM-P) is more interesting. Besides assigning the entry point of the handler to the program counter, it computes the padding time for mitigation of interrupt-based timing attacks and saves the backup in $\mathcal{B}'$. The padding k is then used, causing interrupt handling to take $6 + k$ steps. Such a padding is needed to implement the first part of the mitigation (see Section III-C) and is computed so as to make the dispatching time of interrupts constant. Note that the padding never gets negative. When an interrupt arrives in protected mode two cases may arise. Either $\texttt{GIE} = 1$, and the padding is non-negative because the interrupt is handled at the end of the current instruction; or $\texttt{GIE} = 0$, and no padding is needed because the interrupt is handled as soon as GIE becomes 1, which is only possible in unprotected mode. The

(CPU-VIOLATION-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \not\vdash_{mac} \texttt{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \to \text{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) \neq \bot$$

(CPU-MOVL)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{M}[\mathcal{R}[\texttt{r}_1]]] \\ \mathcal{D} \vdash \delta, t, t_a \frown_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \to \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}])) = \texttt{MOV @r}_1\ \texttt{r}_2$$

(CPU-IN)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r} \mapsto w] \\ \mathcal{D} \vdash \delta', t, t_a \frown_D^{cycles(i)} \delta'', t', t'_a \qquad \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \to \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

Figure 2. Selected rules from the main transition system.

(INT-UM-P)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \qquad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \qquad t_a \neq \bot \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto isr, \texttt{sr} \mapsto 0, \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}] - 4] \qquad \mathcal{M}' = \mathcal{M}[\mathcal{R}[\texttt{sp}] - 2 \mapsto \mathcal{R}[\texttt{pc}], \mathcal{R}[\texttt{sp}] - 4 \mapsto \mathcal{R}[\texttt{sr}]] \qquad \mathcal{D} \vdash \delta, t, \bot \frown_D^6 \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc_{old}, \mathcal{B} \rangle}$$

(INT-PM-P)

$$\frac{\begin{array}{c}k = \texttt{MAX\_TIME} - (t - t_a) \\ pc_{old} \vdash_{mode} \texttt{PM} \qquad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \qquad t_a \neq \bot \qquad \mathcal{R}' = \mathcal{R}_0[\texttt{pc} \mapsto isr] \qquad \mathcal{D} \vdash \delta, t, \bot \frown_D^{6+k} \delta', t', t'_a \qquad \mathcal{B}' = \langle \mathcal{R}, pc_{old}, t - t_a \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta', t', \bot, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

Figure 3. Selected rules for the interrupt logic in Sancus$^{\text{L}}$.

backup stores part of the CPU configuration ($\mathcal{R}$ and pc_{old}) and $t_{pad} = t - t_a$. The value of t_{pad} will then be used as further padding before returning, so fully implementing the mitigation (cf. Section III-C). The register file $\mathcal{R}_0$ is $\{\texttt{pc} \mapsto 0, \texttt{sp} \mapsto 0, \texttt{sr} \mapsto 0, \texttt{R}_3 \mapsto 0, \ldots, \texttt{R}_{15} \mapsto 0\}$.

b) The main transition system: The rules defining the main transition system of Sancus$^{\text{L}}$ are those of Sancus$^{\text{H}}$, with a non-trivial transition system for interrupt logic and mitigation — this explains why also Sancus$^{\text{H}}$ rules have the premise $\mathcal{D} \vdash \cdot \hookrightarrow_{\text{I}} \cdot$ for interrupts.

There are new rules for the new RETI instruction, shown in Figure 4. Rule (CPU-RETI) deals with a return from an interrupt that was handled in unprotected mode, i.e., when $i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$ and there is no backup. Its first premise checks that the RETI instruction is indeed permitted. The second one requires that the program counter is set to the contents of the memory location pointed by the second element from the top of the stack (that grows downwards); that the status register is set to the contents of the memory location pointed by the top of the stack; and that two words are popped from the stack. Finally, the third one registers that $cycles(i)$ steps are needed to complete this task. Rule (CPU-RETI-CHAIN) executes the interrupt handler in unprotected mode when the CPU discovers that another interrupt arrived, while returning from a handler whose interrupt was raised in protected mode (via the interrupt logic). The most interesting rules are the last two. They deal with the case in which the CPU is returning from the handling of an interrupt raised in protected mode, but no new interrupt arrived afterwards (or the GIE bit is off, cf. fourth premise of rule (CPU-RETI-PREPAD)). First, rule (CPU-RETI-PREPAD) restores registers and pc_{old} from the backup $\mathcal{B}$, then rule (CPU-RETI-PAD) (which is the only one applicable after (CPU-RETI-PREPAD)) applies the remaining padding (recorded in the backup) to rule out resume-to-end timing attacks (note that this last padding is interruptible, as witnessed by the last premise). We model the mechanism of restoring registers, pc_{old} and of applying the remaining padding with two rules instead of just one for technical reasons (see the extended version [21] for details). Note that this last padding is applied even if the configuration reached through rule (CPU-RETI-PREPAD) is in unprotected mode (i.e., the interrupted instruction was a jump out of protected mode). Indeed, if it was not the case, the attacker would be able to discover the value of the padding applied *before* the interrupt service routine.

D. Security theorem

Our security theorem states that what an attacker can learn from an enclave is exactly the same before and after adding the support for interrupts. Technically, we show that the semantics of Sancus$^{\text{L}}$ is *fully abstract* w.r.t. the semantics of Sancus$^{\text{H}}$, i.e., all the attacks that can be carried out in Sancus$^{\text{L}}$ can also be carried out in Sancus$^{\text{H}}$, and viceversa. Even though the technical details are specific to our case study, the security definition applies also to other architectures. Before stating the full abstraction theorem and giving the sketch of its proof, we introduce some further notations.

(CPU-RETI)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]+2], \texttt{sr} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]], \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}]+4] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \bot \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-RETI-CHAIN)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \mathcal{B} \neq \bot \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{R}[\texttt{sr.GIE}] = 1 \qquad t'_a \neq \bot \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-RETI-PREPAD)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{B} \neq \bot \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad (\mathcal{R}[\texttt{sr.GIE}] = 0 \;\vee\; t'_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{B}.\mathcal{R}, \mathcal{B}.pc_{old}, \langle \bot, \bot, \mathcal{B}.t_{pad} \rangle \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-RETI-PAD)

$$\frac{\mathcal{B} = \langle \bot, \bot, t_{pad} \rangle \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{t_{pad}} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

Figure 4. Some rules from the operational semantics of Sancus$^{\mathrm{L}}$.

Recall that a whole program $C[\mathcal{M}_M]$ consists of a module $\mathcal{M}_M$ and a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, where $\mathcal{M}_C$ contains the unprotected program and data and $\mathcal{D}$ is the I/O device.

Let $C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$ denote a *converging computation in* Sancus$^{\mathrm{H}}$, i.e., a sequence of transitions of the whole program that reaches the halting configuration from the initial one. Also, let two software modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ be *contextually equivalent in* Sancus$^{\mathrm{H}}$, written $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$, if and only if for all contexts C, $C[\mathcal{M}_M]\Downarrow^{\mathrm{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{H}}$. Similarly, we define $C[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ and $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$ for Sancus$^{\mathrm{L}}$. Roughly, the notion of contextual equivalence formalizes the intuitive notion of *indistinguishability*: two modules are contextually equivalent if they behave in the same way under any attacker (i.e., context). Due to the quantification over *all* contexts, it suffices to consider just terminating and non-terminating executions as distinguishable, since any other distinction can be reduced to it. We can state the theorem that guarantees the absence of interrupt-based attacks:

Theorem IV.1 (Full abstraction)**.**
$\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ (\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'})$.

First we prove $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$ and then $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$. Below we only intuitively describe the proof steps (all the details are in the extended version [21]).

a) Proof sketch for $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$: Since programs in Sancus$^{\mathrm{H}}$ behave like those in Sancus$^{\mathrm{L}}$ with no interrupts, proving this implication is not too hard. It suffices to introduce the notion of *interrupt-less* context C_I for Sancus$^{\mathrm{L}}$ that behaves as C, but never raises interrupts. The thesis follows because an enclave hosted in a interrupt-less context terminates in Sancus$^{\mathrm{L}}$ whenever it does in Sancus$^{\mathrm{H}}$, as interrupt-less contexts are a strict subset of all the contexts.

b) Proof sketch for $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$: We first introduce the notion of observable behavior, in terms of the traces that $C[\mathcal{M}_M]$ can perform according to the

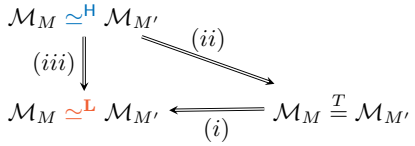

Figure 5. The steps for proving preservation of behavior.

Sancus$^{\mathrm{L}}$ semantics. Traces are built using three observables: (i) $\bullet$ denotes that the computation halts; (ii) $\texttt{jmpIn?}(\mathcal{R})$ denotes that the CPU enters the protected mode, where $\mathcal{R}$ are the observed registers and (iii) $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ denotes the exit from protected mode with observed registers $\mathcal{R}$ and with Δt representing the end-to-end time measured by an attacker for code running in protected mode.

The proof then follows the steps in Figure 5, where $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ means that $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ have the same traces. Implication (i) shows that the attacker in Sancus$^{\mathrm{L}}$ at most observes *as much as* traces say; implication (ii) shows that the attacker in Sancus$^{\mathrm{H}}$ is *at least as powerful as* described by traces; finally implication (iii) is our thesis that follows by transitivity. The proof of (i) $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$ roughly goes as follows. First the mitigation is shown to guarantee that the behavior of the context (in unprotected mode) does not depend on the behavior of the enclave (in protected mode) and vice versa (Lemmata III.4 and III.5 of [21]). The thesis follows because if $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ and $C[\mathcal{M}_M]$ has a trace $\overline{\beta}$, then also $C[\mathcal{M}'_M]$ has the same trace $\overline{\beta}$.

The proof of (ii) $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ is by contraposition: if two modules have different traces, there exists a context that distinguishes them, and we build such a context through a *backtranslation*. Because of the strong limitations – for instance because only 64KB of memory is

available – building such a context in unprotected memory only is infeasible and the strong attacker model that enclaved execution is built for is actually helpful here. The backtranslation defines and uses both the unprotected memory (Algorithm 1 of [21]), and the I/O device, which has unrestricted memory (Algorithm 2 of [21]). Very roughly, the idea is to take a trace of $\mathcal{M}_M$ and one of $\mathcal{M}_{M'}$ that differ for one observable, and build a context C such that $\mathcal{M}_M$ converges and $\mathcal{M}_{M'}$ does not, so contradicting the hypothesis $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$.

V. Implementation and Evaluation

We provide a full implementation of our approach based on the Sancus [12] architecture which, in turn, is based on the openMSP430, an open source implementation of the TI MSP430 ISA. Our implementation can be divided in two parts. First, we adapted the execution unit's state machine to add padding cycles whenever an interrupt happens in protected mode and when we return from such interrupts. Second, we added a protected storage area corresponding to $\mathcal{B}$.

a) Cycle padding: To implement cycle padding, we added three counters to the processor's frontend. The first, $\mathtt{C}_{\mathtt{reti\_nxt}}$, tracks the number of cycles to be padded on the next `RETI`. Whenever an *interrupt request* (IRQ) occurs, this counter is initialized to zero and is subsequently incremented every cycle until the current instruction completes. Thus, at the end of an instruction, this counter holds $t - t_a$, which corresponds to t_{pad} in $\mathcal{B}$ (cf. the **(INT-PM-P)** rule in Figure 3).

The second counter, $\mathtt{C}_{\mathtt{irq}}$, holds the number of cycles that needs to be padded when an IRQ occurs. It is initialized to $\mathtt{MAX\_TIME} - \mathtt{C}_{\mathtt{reti\_next}}$ (`MAX_TIME` is 6 in our case) when the instruction during which an IRQ occurred finishes execution. That is, it holds the value k from rule **(INT-PM-P)** in Figure 3 after the instruction finishes. From this point on, the counter is decremented every cycle and the execution unit's state machine is kept in a wait state until the counter reaches zero. Only then is it allowed to progress and start handling the IRQ.

Lastly, a third counter, $\mathtt{C}_{\mathtt{reti}}$, is added that holds the number of cycles that needs to be padded for the current `RETI` instruction. Whenever a `RETI` is executed while handling an IRQ from protected mode, this counter is initialized with the value of $\mathtt{C}_{\mathtt{reti\_nxt}}$. Then, after restoring the processor state from $\mathcal{B}$ (see Section V-b), this counter is decremented every cycle until it reaches zero. After these padding cycles, the next instruction is fetched, from $\mathcal{R}[\mathtt{pc}]$ restored from $\mathcal{B}$, and executed. Note that these padding cycles behave as any t_{pad}-cycle instruction from the perspective of the padding logic. That is, they can be interrupted and, hence, padded as well. This is the reason why we need two counters to hold padding information for `RETI`: $\mathtt{C}_{\mathtt{reti}}$ is used to pad the current `RETI` instruction and $\mathtt{C}_{\mathtt{reti\_nxt}}$ is used – concurrently, if an IRQ occurs – to count t_{pad} for the *next* `RETI`.

b) Saving and restoring processor state: Whenever an IRQ in protected mode occurs, the processor's register state needs to be saved in a location inaccessible from software. Our current implementation uses a shadow register file to this end. We duplicate all registers $\mathtt{R}_0, \ldots, \mathtt{R}_{15}$ (except $\mathtt{R}_3$, the constant generator, which does not store state). On an IRQ, all registers are first copied to the shadow register file and then cleared. When a subsequent `RETI` is executed, registers are restored from their copies. For the other values in $\mathcal{B}$, pc_{old} is handled the same as registers, and t_{pad} is saved from $\mathtt{C}_{\mathtt{reti\_nxt}}$ and restored to $\mathtt{C}_{\mathtt{reti}}$, as explained in Section V-a. Besides the values in $\mathcal{B}$, we add a single bit to indicate if we are currently handling an IRQ from protected mode, allowing us to test if $\mathcal{B} \neq \bot$.

The current implementation allows to save or restore the processor state in a single cycle at the cost of approximately doubling the size of the register file. If this increase in area is unacceptable, the state could be stored in a protected memory area. Implementing this directly in hardware would increase the number of cycles needed to save and restore a state to one cycle per register. Of course, one should make sure that this memory area is inaccessible from software by adapting the memory access control logic of the processor accordingly.

c) Evaluation: To evaluate the performance impact of our implementation, we only need to quantify the overhead on handling interrupts and returning from them, as an uninterrupted flow of instructions is not impacted by our design.

When an IRQ occurs, as well as when the subsequent `RETI` is executed, there is a maximum of `MAX_TIME` padding cycles executed. This variable part of the overhead is thus bounded by `MAX_TIME` cycles for both cases. The fixed part – saving and restoring the processor's state – turns out to be 0 in our current implementation: since the fetch unit's state machine needs at least one extra cycle to do a jump in both cases, copying the state is done during this cycle and causes no extra overhead. Of course, if the register state is stored in memory, as described in Section V-b, the fixed overhead grows accordingly.

To evaluate the impact on area, we synthesized our implementation on a Xilinx XC6SLX25 Spartan-6 FPGA with a speed grade of -2 using Xilinx ISE Design Suite optimizing for area. The baseline is an unaltered Sancus 2.0 core configured with support for a single protected module and 64-bit keys for remote attestation. The unaltered core could be synthesized using 1239 slice registers and 2712 slice LUTs. Adding support for saving and restoring the processor state increases the area to 1488 slice registers and 2849 slice LUTs and the implementation of cycle padding further increases it to 1499 slice registers and 2854 slice LUTs. It is clear that the largest part of the overhead comes from saving the processor state which is necessary for any implementation of secure interrupts and can be optimized as discussed in Section V-b. The implementation of cycle padding, on the other hand, does not have a significant impact on the processor's area.

VI. Discussion

A. On the use of full abstraction a security objective

The security guarantee that our approach offers is quite strong: an attack is possible in $\mathsf{Sancus}^{\mathsf{H}}$ if and only if it is possible at $\mathbf{Sancus}^{\mathsf{L}}$. Full abstraction fits naturally with our goal, because isolation is defined in terms of contextual equiva-

lence, and full abstraction specifies that contextual equivalence is preserved and reflected.

The *if*-part, namely preservation, guarantees that extending Sancus^{H} with interrupts opens no new vulnerabilities. Reflection, i.e., the *only if*-part is needed because otherwise two enclaves that are distinguishable in Sancus^{H} *become* indistinguishable in **Sancus^{L}**. Although this mainly concerns functionality and not security, a problem emerges: adding interrupts is not fully "backwards compatible." Indeed, reflection rules out mechanisms that while closing the interrupt side-channels also close other channels. We believe the situation is very similar for other extensions: adding caches, pipelining, etc. should not strengthen existing isolation mechanisms either.

Actually, full abstraction enables us to take the security guarantees of Sancus^{H} as the specification of the isolation required after an extension is added.

An alternative approach to full abstraction would be to require (a non interactive version of) robust preservation of timing-sensitive non-interference [31]. This can also guarantee resistance against the example attacks in Section III. However, this approach offers a strictly weaker guarantee: our full abstraction result implies that timing-sensitive non-interference properties of Sancus^{H} programs are preserved in **Sancus^{L}**, as far as non-interference takes as secret the whole enclave, i.e., its memory and code, and the initial state, as well. In addition, full abstraction implies that isolation properties that rely on code confidentiality are preserved, and this matters for enclave systems that guarantee code confidentiality, like the Soteria system [32]. An advantage however might be that robust preservation of timing-sensitive non-interference might be easier to prove.

In case full abstraction is considered too strong as a security criterion, it is possible to selectively weaken it by modifying Sancus^{H}. For instance, to specify that code confidentiality is not important, one can modify Sancus^{H} to allow contexts to read the code of an enclave.

B. The impact of our simplifcations

The model and implementation we discussed in this paper make several simplifying assumptions. A first important observation that we want to make is that some simplifications of our model with respect to our implementation are straightforward to remove. For instance, supporting more MSP430 instructions in our model would not affect the strong security guarantees offered by our approach, and only requires straightforward, yet tedious technical work.

However, there are also other assumptions that are more essential, and removing these would require additional research. Here, we discuss the impact of these assumptions on the applicability of our results to real systems.

First, we scoped our work to only consider "small" microprocessors. The essential assumption our work relies on is that the timing of individual instructions is predictable (as shown, e.g., in Table I for the MSP430). This is typically only true of small microprocessors. As soon as a processor implements performance enhancing features like caching or speculation, the timing of an individual instruction will be variable, e.g., a load will be faster if can be served from the cache. Our model and proof do not apply to such more advanced processors. However, we do believe that the padding countermeasure that we proved to be secure on simple processors is a very good *heuristic* countermeasure, also for more advanced processors. It has been shown that for instance interrupt-latency attacks are relevant for modern Intel Core processors supporting SGX enclaves [9]. Interrupt latency is not deterministic on these processors, but is instead a complex function of the micro-architectural state at the point of interruption, and it is hard to determine an upper bound on the maximal latency that could be observed. Still, padding to a fixed length on interrupt and complementary padding on resume will significantly raise the bar for interrupt latency attacks. We are aware that it would be very hard, if not impossible at all, to carry over to these settings the strong security guarantees offered by full abstraction for "small" microprocessors. Consider for instance the leaks made possible by the persistent micro-architectural state that we do not model in this paper. However, implementing our countermeasure will likely make attacks harder also in high-end microprocessors.

Second, our model made some simplifying assumptions about the enclave-based isolation mechanism. We did not model support for cryptographic operations and for attestation. This means that we assume that the loading and initialization of an enclave can be done as securely in **Sancus^{L}** as it can be done in Sancus^{H}. Our choice separates concerns, and it is independent of the security criterion adopted. Modelling both memory access control and cryptography would only increase the complexity of the model, as two security mechanisms rather than one would be in order. Also their interactions should be considered to prevent, e.g., leaks of cryptographic keys unveiling secrets protected by memory access control, and viceversa. Also, we assumed the simple setting where only a single enclave is supported. We believe these simplifications are acceptable, as they reduce the complexity of the model significantly, and as none of the known interrupt-driven attacks relies on these features. It is also important to emphasize that these are model-limitations, and that an implementation can easily support attestation and multiple enclaves. However, for implementations that do this, our current proof does not rule out the presence of attacks that rely on these features.

A more fundamental limitation of the model is that it forbids reentering an enclave that has been interrupted, via $\vdash_{mac}$. Allowing reentrancy essentially causes the same complications as allowing multi-threaded enclaves, and these are substantial complications that also lead to new kinds of attacks [33]. We leave investigation of these issues to future work.

Third, our model and implementation make other simplifications that we believe to be non-essential and that could be removed with additional work but without providing important new insights. For instance, we assumed that enclaves have no read/write access to untrusted memory. A straightforward alternative is to allow these accesses, but to also make them observable to the untrusted context in Sancus^{H}. Essentially,

this alternative forces the enclave developer to be aware of the fact that accessing untrusted memory is an interaction with the attacker. A better alternative (putting less security responsibility with the enclave developer) is to rely on a trusted run-time that can access unprotected memory to copy in/out parameters and results, and then turn off access to unprotected memory before calling enclaved code. This is very similar to how Supervisor Mode Access Prevention prevents the kernel from the security risks of accessing user memory. Our model could easily be extended to model such a trusted run-time by considering memory copied in/out as a large CPU register. It is important to emphasize however that the implementation of such trusted enclave runtime environments has been shown to be error-prone [34].

Another such non-essential limitation is the fact that we do not support nested interrupts, or interrupt priority. It is straightforward to extend our model with the possibility of multiple pending interrupts and a policy to select which of these pending interrupts to handle. One only has to take care that the interrupt arrival time used to compute padding is the arrival time of the interrupt that will be handled first.

In summary, to provide hard mathematical security guarantees, one often abstracts from some details and provable security only provides assurance to the extent that the assumptions made are valid and the simplifications non-essential. The discussion above shows that this is the case for a relevant class of attacks and systems, and hence that our countermeasure for these attacks is well-designed. Since there is no 100% security, attacks remain possible for more complex systems (e.g. including caches and speculation), or for more powerful attackers (e.g. with physical access to the system).

VII. Related Work

Our work is motivated by the recent wave of software-based side-channel attacks and controlled-channel attacks that rely on architectural or micro-architectural processor features. The area is too large to survey here, but good recent surveys include Ge et al. [5] for timing attacks, Gruss' PhD thesis [35] for software-based microarchitectural attacks before Spectre/Meltdown, and [10] for transient execution based attacks. The attacks most relevant to this paper are the pure interrupt-based attacks. Van Bulck et al. [9] were the first to show how just measuring interrupt latency can be a powerful attack vector against both high-end enclaved execution systems like Intel SGX, and against low-end systems like the Sancus system that we based our work on. Independently, He et al. [14] developed a similar attack for Intel SGX.

There is an extensive body of work on defenses against software-based side-channel attacks. The three surveys mentioned above ([5], [35], [10]) also survey defenses, including both software-based defenses like the constant-time programming model and hardware-based defenses such as cache-partitioning. To the best of our knowledge, our work proposes the first defense specifically designed and proved to protect against pure interrupt-based side-channel attacks. De Clerck et al. [18] have proposed a design for secure interruptibility of enclaved execution, but they have not considered side-channels – their main concern is to make sure that there are no direct leaks of, e.g., register contents on interrupts. Most closely related to ours is the work on SecVerilog [36] that also aims for formal assurances. To guarantee timing-sensitive non-interference properties, SecVerilog uses a security-typed hardware description language. However, this approach has not yet been applied to the issue of interrupt-based attacks. Similarly, Zagieboylo et al. [37] describe an ISA with information-flow labels and use it to guarantee timing-insensitive information flow at the architectural level.

An alternative approach to interruptible secure remote computation is pursued by VRASED [25]. In contrast to enclaved execution, their design only relies on memory access control for the attestation key, not for the software modules being attested. They prove that a carefully designed hardware/software co-design can securely do remote attestation.

Our security criterion is directly influenced by a long line of work that considers *full abstraction* as a criterion for secure compilation. The idea was first coined by Abadi [20], and has been applied in many settings, including compilation to JavaScript [38], various intermediate compiler passes [39], [40], and compilation to platforms that support enclaved execution [41], [42], [43]. But none of these works consider timing-sensitivity or interrupts: they study compilations higher up the software stack than what we consider in this paper. Patrignani et al. [44] have provided a good survey of this entire line of work on secure compilation.

VIII. Conclusions and future work

We have proposed an approach to formally assure that extending a microprocessor with a new feature does not weaken the isolation mechanisms that the processor offers. We have shown that the approach is applicable to an IoT-scale microprocessor, by showing how to design interruptible enclaved execution that is as secure as uninterruptible enclaved execution. Despite this successful case study, some limitations of the approach remain, and we plan to address them in future.

First, as discussed in Section VI, our approach currently applies only to "small" micro-processors for which we can define a cycle-accurate operational semantics. While this obviously makes it possible to rigorously reason about timing-based side-channels, it is also difficult to scale to larger processors. To handle larger processors, we need models that can abstract away many details of the processor implementation, yet keeping enough detail to model relevant micro-architectural attacks. A very recent and promising example of such a model was proposed by Disselkoen et al. [45]. An interesting avenue for future work is to consider such models for our approach instead of the cycle-accurate models.

Second, the security criterion we proposed is binary: an extension is either secure, or it is not. The criterion does not distinguish *low bandwidth* side-channels from *high-bandwidth* side-channels. An important challenge for future work is to introduce some kind of *quantification* of the weakening of

security, so that it becomes feasible to allow the introduction of some bounded amount of leakage.

ACKNOWLEDGEMENTS

We would like to thank the anonymous referees and the paper shepherd for their insightful comments and detailed suggestions that helped to greatly improve our presentation. Matteo Busi and Pierpaolo Degano have been partially supported by the University of Pisa project PRA_2018_66 *DECLware: Declarative methodologies for designing and deploying applications*. This research is partially funded by the Research Fund KU Leuven, by the Agency for Innovation and Entrepreneurship (Flanders), and by a gift from Intel Corporation. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). Letterio Galletta has been partially supported by EU Horizon 2020 project No 830892 *SPARTA* and by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

REFERENCES

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, R. B. Lee and W. Shi, Eds. ACM, 2013, p. 10.

[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 361–372.

[3] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1057–1074. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[4] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[5] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[7] 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'19)*, 2019.

[8] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[9] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 178–195. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243822

[10] C. Canella, J. V. 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, USENIX Security 2019*, 2019.

[11] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656.

[12] J. Noorman, J. V. 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 Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, Jul. 2017. [Online]. Available: http://doi.acm.org/10.1145/3079763

[13] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan, "Trustlite: a security architecture for tiny embedded devices," in *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, D. C. A. Bulterman, H. Bos, A. I. T. Rowstron, and P. Druschel, Eds. ACM, 2014, pp. 10:1–10:14.

[14] W. He, W. Zhang, S. Das, and Y. Liu, "SGXlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 2018, pp. 108–114.

[15] J. V. Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017, pp. 4:1–4:6.

[16] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 557–574. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution."

[18] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, "Secure interrupts on low-end microcontrollers," in *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014*. IEEE Computer Society, 2014, pp. 147–152.

[19] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 479–494. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman

[20] M. Abadi, "Protection in programming-language translations," in *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, ser. Lecture Notes in Computer Science, J. Vitek and C. D. Jensen, Eds., vol. 1603. Springer, 1999, pp. 19–34.

[21] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Provably secure isolation for interruptible enclaved execution on small microprocessors: Extended version," *CoRR*, vol. abs/2001.10881, 2020. [Online]. Available: http://arxiv.org/abs/2001.10881

[22] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[23] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 143–158.

[24] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 287–305.

[25] I. 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, USENIX Security 2019*, 2019.

[26] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1041–1056. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[27] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[28] T. Instruments, "MSP430x1xx Family: User Guide," http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[29] T. Goodspeed, "Practical attacks against the MSP430 BSL," in *Twenty-Fifth Chaos Communications Congress.*, 2008.

[30] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel SGX," *CoRR*, vol. abs/1902.03256, 2019. [Online]. Available: http://arxiv.org/abs/1902.03256

[31] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, 2019, pp. 256–271.

[32] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede, "Soteria: Offline software protection within low-cost embedded devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 241–250. [Online]. Available: http://doi.acm.org/10.1145/2818000.2856129

[33] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, 2016, pp. 440–457.

[34] J. V. 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 2019, pp. 1741–1758.

[35] D. Gruss, "Software-based microarchitectural attacks," Ph.D. dissertation, Graz University of Technology.

[36] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 503–516.

[37] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an ISA that controls timing channels," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, 2019, pp. 272–287. [Online]. Available: https://doi.org/10.1109/CSF.2019.00026

[38] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits, "Fully abstract compilation to javascript," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds. ACM, 2013, pp. 371–384.

[39] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, 2008, pp. 157–168.

[40] ——, "An equivalence-preserving CPS translation via multi-language semantics," in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, 2011, pp. 431–444.

[41] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, S. Chong, Ed. IEEE Computer Society, 2012, pp. 171–185.

[42] M. Patrignani and D. Clarke, "Fully abstract trace semantics for protected module architectures," *Computer Languages, Systems & Structures*, vol. 42, pp. 22–45, 2015.

[43] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 6:1–6:50, 2015.

[44] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," *ACM Comput. Surv.*, vol. 51, no. 6, 2019. [Online]. Available: https://doi.org/10.1145/3280984

[45] C. Disselkoen, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely, "The code that never ran: Modeling attacks on speculative evaluation," in *Proc. IEEE Symp. Security and Privacy*, 2019.