Securing Interruptible Enclaved Execution on Small Microprocessors

MATTEO BUSI, Dept. of Computer Science, Università di Pisa
JOB NOORMAN, imec-DistriNet, Dept. of Computer Science, KU Leuven
JO VAN BULCK, imec-DistriNet, Dept. of Computer Science, KU Leuven
LETTERIO GALLETTA, IMT School for Advanced Studies Lucca
PIERPAOLO DEGANO, Dept. of Computer Science, Università di Pisa and IMT School for Advanced Studies Lucca
JAN TOBIAS MÜHLBERG, imec-DistriNet, Dept. of Computer Science, KU Leuven
FRANK PIESSENS, imec-DistriNet, Dept. of Computer Science, KU Leuven

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 software-based 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. Our solution is heavily based on techniques from research on programming languages. More specifically, we propose to use the programming language concept of full abstraction as a general formal criterion for the security of a processor extension. We instantiate the proposed criterion to the concrete case of extending a microprocessor that supports enclaved execution with secure interruptibility. 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.

CCS Concepts: • **Security and privacy** → **Formal methods and theory of security**; **Embedded systems security**.

Additional Key Words and Phrases: language-based security, enclaves, full abstraction, secure compilation

ACM Reference Format:
Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2022. Securing Interruptible Enclaved Execution on Small Microprocessors. 1, 1 (March 2022), 76 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Matteo Busi, matteo.busi@di.unipi.it, Dept. of Computer Science, Università di Pisa, Pisa, Italy; Job Noorman, job.noorman@kuleuven.be, imec-DistriNet, Dept. of Computer Science, KU Leuven, Leuven, Belgium; Jo Van Bulck, jo.vanbulck@kuleuven.be, imec-DistriNet, Dept. of Computer Science, KU Leuven, Leuven, Belgium; Letterio Galletta, letterio.galletta@imtlucca.it, IMT School for Advanced Studies Lucca, Lucca, Italy; Pierpaolo Degano, pierpaolo.degano@unipi.it, Dept. of Computer Science, Università di Pisa and IMT School for Advanced Studies Lucca, Lucca, Italy; Jan Tobias Mühlberg, jantobias.muehlberg@kuleuven.be, imec-DistriNet, Dept. of Computer Science, KU Leuven, Leuven, Belgium; Frank Piessens, frank.piessens@kuleuven.be, imec-DistriNet, Dept. of Computer Science, KU Leuven, Leuven, Belgium.

1 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 interfering 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 [41].

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 [33, 43, 54], as well as their confidentiality on both high-end processors [9, 23, 34, 38] and on small microprocessors [56]. In fact, over the past two years, many major isolation mechanisms have been successfully attacked: Meltdown [38] has broken user/kernel isolation, Spectre [34] has broken process isolation and software defined isolation, and Foreshadow [9] 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 [14] 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 [60].

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 [35, 45] against interrupt-based attacks [11, 29, 56]. 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 [9, 15, 36, 56]. 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, 35]. For our study, we rely on programming language techniques developed in the field of secure compilation [48].

We base our study on the existing open-source Sancus platform [44, 45], a small microprocessor with predictable timing of individual instructions, 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, called hereafter Sancus^{H}, and we then extend it with interrupts, dubbed Sancus^{L}. We prove that this extension does not break isolation properties by instantiating full abstraction [1]. Full abstraction is a good fit for this study, as Sancus is fully deterministic, including deterministic timing. Moreover, the attacks we consider rely on distinguishing code paths of programs when they have different execution time, which is closely related to distinguishing different programs.

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 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 [1] 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 specialize the proof technique called *backtranslation* [47] to encode the attack logic within the I/O device, so as to construct an attacker at Sancus^{H} given one at Sancus^{L}. The novelty of our backtranslation consists in using the unlimited state space of the (attacker-controlled) I/O device to work around the 64KB memory limit of the processor.
- We implement our countermeasures 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 2 we provide background information on enclaved execution and interrupt-based attacks. Section 3 provides an informal overview of our approach. Section 4 introduces our formalization, and Section 5 presents the semantics of Sancus without and with interrupts. The proof that enclaved executions are resistant to interrupt-based attacks is in Section 6; some auxiliary definitions and proofs are presented in full detail in the Appendix. Section 7 shows how our full abstraction result implies some other security notions when tailored to our setting. In Section 8 we describe and evaluate our implementation. Sections 9 and 10 discuss limitations, and the connection to related work. Finally, Section 11 offers our conclusions and plans for future work.

This is an extended version of the paper [13]. Here we include all the results of the conference paper, and additionally include (1) a detailed outline of our formal model and full abstraction proof, (2) additional results that make explicit how our full abstraction result relates to the preservation of (variants of) non-interference and other security properties, and (3) a more detailed discussion of the lessons that can be learned for other, more complex, enclaved execution systems, and the challenges that remain there.

2 BACKGROUND

2.1 Enclaved execution

Enclaved execution is a security mechanism that enables *secure remote computation* [17]. 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 [41, 45], or possibly also a small hypervisor [21, 40].

The remote attestation aspect is important for the secure initialization of enclaves, and for setting up secure communication channels with them. 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

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

isolation aspect of enclaves only. Other papers describe in detail how remote attestation and secure communication work on large [17] or small systems [35, 45].

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 the loading of confidential data happens through a secure channel, 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 code and data sections of an enclave are *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. Actually, untrusted code may reside in the same address space of the enclave, but outside its code and data sections. Untrusted code can only 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.

2.2 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. Isolating enclaves is well-understood at the architectural level, including even successful formal verification efforts [21, 46]. As a matter of fact, researchers have shown that it is challenging to protect enclaves against side channels. Particularly, a recent line of work on *controlled-channel* attacks [11, 12, 36, 42, 56, 60] 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 them. This power has been exploited for instance to single-step through an enclave [11], or to mount a new class of ingenious *interrupt latency* attacks [29, 56] 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 [9, 12, 42, 60], branch prediction [15, 36] or caching [53] 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. [56] 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 [45, 46]. 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, 35] are vulnerable to these attacks.

Instr. *i*	**Meaning**	**Cycles**	**Size in words**
`RETI`	Returns from interrupt.	5	1
`NOP`	No-operation.	1	1
`HLT`	Halt.	1	1
`NOT` $\mathtt{r}$	$\mathtt{r} \leftarrow \neg\mathtt{r}$. (Emulated in MSP430)	2	2
`IN` $\mathtt{r}$	Reads word from the device and puts it in $\mathtt{r}$.	2	1
`OUT` $\mathtt{r}$	Writes word in register $\mathtt{r}$ to the device.	2	1
`AND` $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1$ & $\mathtt{r}_2$.	1	1
`JMP &r`	Sets $\mathtt{pc}$ to the value in $\mathtt{r}$.	2	1
`JZ &r`	Sets $\mathtt{pc}$ to the value in $\mathtt{r}$ if bit 0 in $\mathtt{sr}$ is set.	2	1
`MOV` $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1$.	1	1
`MOV` @$\mathtt{r}_1$ $\mathtt{r}_2$	Loads in $\mathtt{r}_2$ the word starting in location pointed to by $\mathtt{r}_1$.	2	1
`MOV` $\mathtt{r}_1$ `0`($\mathtt{r}_2$)	Stores the value of $\mathtt{r}_1$ starting at location pointed to by $\mathtt{r}_2$.	4	2
`MOV` #w $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow w$.	2	2
`ADD` $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1 + \mathtt{r}_2$.	1	1
`SUB` $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1 - \mathtt{r}_2$.	1	1
`CMP` $\mathtt{r}_1$ $\mathtt{r}_2$	Zero bit in $\mathtt{sr}$ set if $\mathtt{r}_2 - \mathtt{r}_1$ is zero.	1	1

Table 1. Summary of the assembly language considered.

3 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 [45], an existing open-source enclaved execution system. We first describe our Sancus model, and discuss how naively extending Sancus with interrupts leads to the attacks mentioned in Section 2.2. In other words, we show how extending Sancus with interrupts breaks some of the isolation guarantees provided by the original architecture.

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 (whose memory has only 64KB). Our approach of ensuring that the same attacks are possible before and after an architecture extension is tailored here on a specific architecture and on a specific class of attacks, however we expect it to be applicable in other settings too, as briefly discussed in Section 9.3.

3.1 Sancus model

Processor. Sancus is based on the TI MSP430 16-bit microprocessor [30], with a classic von Neumann architecture where code and data share the same address space. We formalize the subset of instructions summarized in Table 1 that is rich enough to model all the attacks on Sancus we care about (see also Section 9). 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.

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 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 of interest in this paper. Instead, our model allows direct initialization of the data section with confidential enclave data. All the other memory is *unprotected memory* that is under full 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 [45].

Sancus enforces memory access control based on program counter (pc). If the pc points to unprotected memory, the processor cannot 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, see Section 4.8 for the full definition of the relevant mechanism.

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 the two special instructions `IN` and `OUT` that allow writing/reading a word to/from the device (see Table 1). Actually these instructions are short-hands, 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.

3.2 Security definitions

Attacker model. An attacker controls the entire execution environment, aka the *context* of an enclave: he controls (1) the whole 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, i.e., pieces of code that the CPU invokes when an interrupt is raised.

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 cannot see "inside" an enclave, so making it possible to "hide" enclave data or implementation details from the attacker. We precisely formalize this concept of isolation by using the notion of *contextual equivalence* or *contextual indistinguishability*, as done by Abadi [1]. Contextual equivalence (as opposed to alternatives based on for instance non-interference) also covers confidentiality of the code in the enclave, which some enclaved execution systems guarantee [25]. Two enclaved modules M_1 and M_2 are contextually equivalent if there exists no context that tells them apart. We discuss this on the following example.

Example 3.1 (Start-to-end timing). The following enclave compares a user-provided password in the register R_{15} with a secret in-enclave password at address *pwd_adrs*, and stores the user-provided value in the register R_{14} into the enclave location at *store_adrs* if the user password was correct.

```
enclave_entry:
    /* Load addresses for comparison */
    MOV #store_adrs, r10    ; 2 cycles
    MOV #access_ok, r11     ; 2 cycles
    MOV #endif, r12         ; 2 cycles
    MOV #pwd_adrs, r13      ; 2 cycles
    /* Compare user vs. enclave password */
    MOV @r13, r13           ; 2 cycles
    CMP r13, r15            ; 1 cycle
    JZ  &r11                ; 2 cycles
access_fail:   /* Password fail: return */
    JMP &r12                ; 2 cycles
access_ok:  /* Password ok: store user val */
    MOV r14, 0(r10)         ; 4 cycles
endif:  /* Clear secret enclave password */
    SUB r13, r13            ; 1 cycle
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 the above only differing in the value of 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 calling the entry point, but the context gets no 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 time, we represent enclaved executions as an ordered array of individual instruction timings. (Table 1 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 [24], 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 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 adding 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 or by a security-aware compiler [7]. For instance, the programmer can balance out the various secret-dependent control-flow paths as in Example 3.2.

Example 3.2 (Interrupt latency). Consider the program of Example 3.1, balanced in terms of overall execution time by adding two `NOP` instructions at lines 13-14 below. 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:
```

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 TI MSP430. The processor checks for the presence of pending interrupts after the completion of each instruction. Hence, if an interrupt arrives while the processor is executing a multi-cycle instruction, it will only be handled once that instruction is completed. If there is an interrupt, the processor saves some essential state (like where to resume after the interrupt is handled) and then sets the program counter to the interrupt service routine. The interrupt service routine performs any actions required to handle the interrupt and then uses the RETI instruction to resume execution at the instruction following the interrupted instruction.

The program in Example 3.2 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, consider the following attack. The attacker schedules an interrupt to arrive within the first cycle after the conditional jump at line 10 (call this clock cycle t_0), and the attacker measures when control flow arrives in the interrupt service routine (clock cycle t_1). The attacker can then compute the interrupt latency $t_1 - t_0$. If the jump was taken then the instruction being interrupted is the 4-cycle `MOV` at line 18, otherwise it is the 1-cycle `NOP` at line 13. Now, since the attacker's interrupt service routine 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 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 [56].

Using this attack technique, illustrated in Figure 1, an attacker 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 that this is however a necessary but not sufficient condition.

Example 3.3 (Resume-to-end timing). Consider the program from Example 3.2 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

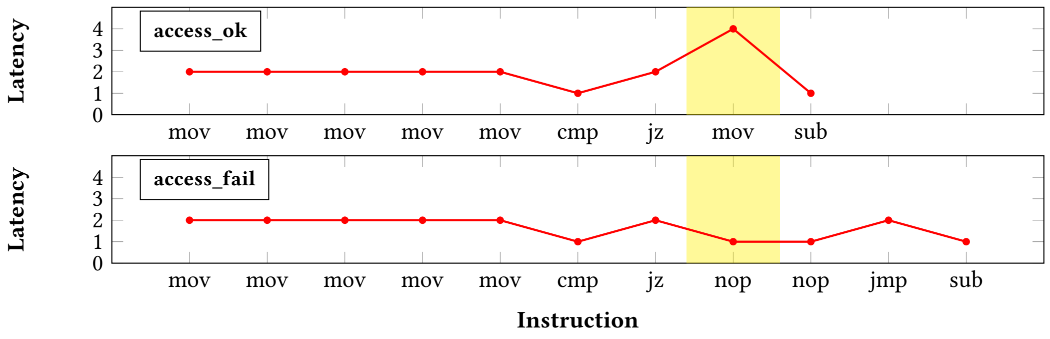


Fig. 1. Interrupt latency traces corresponding to the conditional control-flow paths in Example 3.2. When interrupting after the 7th instruction, the adversary observes a distinct latency difference for the 4-cycle MOV instruction vs. the 1-cycle NOP instruction.

time to complete enclave execution after resume from the interrupt is 1 cycle for the ok path and 4 cycles for the `fail` path (cf. Figure 1). Hence, like in Example 3.2, the compiler's or developer's effort to equalize both branches is undermined.

Example 3.4 (Interrupt-counting attack). An alternative way to attack the program from Example 3.2 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 (cf. Figure 1). Such interrupt counting attacks [42] have been shown to be dangerous even on enclaved execution systems like Intel SGX, where timing measurements are noisy.

Defining the security of an extension. The examples above show how a new processor feature (like interrupts) can weaken an existing isolation mechanism (like enclaved execution), and this is exactly what we want to avoid. Here we propose and implement a defense against these attacks and formally prove that it is indeed secure. Our security definition should now be clear: 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).

3.3 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 time 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.

Padding. We already discussed that it is insufficient for security to naively pad interrupt latency to make it constant, while we need a padding approach that handles all kinds of attacks.

Our padding scheme (see Figure 2) is as follows. 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 the execution of I. To make sure the attacker cannot learn anything from the interrupt latency, we introduce padding for Δt_{p_1} cycles where Δt_{p_1} is computed by the interrupt handling

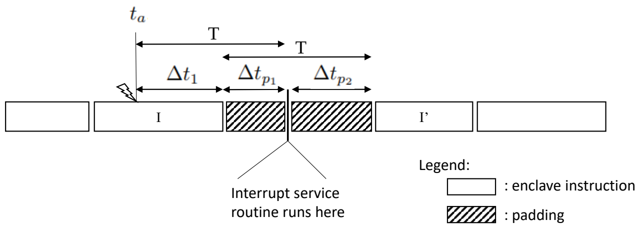


Fig. 2. The secure padding scheme.

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 3.3. 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 3.4.

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. Interrupt counting attacks become useless, as the number of times an execution path can be interrupted does no longer depend on the number of instructions in that path.

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, leading to the refinement of the implementation of Sancus.

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 a stack in the enclave accessible memory. However, the proof of our security theorem showed that this solution 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.

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 [52]). 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 in the status register from being changed by the 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 the model simple and the proof manageable, although some of them may seem restrictive. Section 9 discusses the practical impact of these choices and possible ways of relaxing some limitations.

4 THE FORMAL MODEL OF THE ARCHITECTURE

Here we set up the formal model of the architecture that runs both the original, uninterruptible Sancus ($\mathsf{Sancus^H}$, Sancus-High) and the secure interruptible Sancus ($\mathbf{Sancus^L}$, Sancus-Low).[2] The next section will define the semantics of of $\mathsf{Sancus^H}$ and $\mathbf{Sancus^L}$, and then we will formally show that the two versions of Sancus actually provide the same security guarantees, i.e., the isolation mechanism is not broken by adding our carefully designed interruptible enclaved execution.

4.1 Memory and memory layout

Recall from Section 3.1 that MSP430 has a 16-bit architecture, thus we model its memory 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 with 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 $w = b_1b_0$ 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 MSP430):

$$\mathcal{M}[l] \triangleq b_1b_0 \quad \text{if } \mathcal{M}(l) = b_0 \land \mathcal{M}(l+1) = b_1$$

We define the write operation as follows

$$(\mathcal{M}[l \mapsto b_1b_0])(l') \triangleq \begin{cases} b_0 & \text{if } l' = l \\ b_1 & \text{if } l' = l + 1 \\ \mathcal{M}(l') & \text{o.w.} \end{cases}$$

Writing b_0b_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 in $l = \texttt{0xFFFF}$ 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.

[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 $\mathsf{Sancus^H}$, in **red, bold** font elements of $\mathbf{Sancus^L}$ and in black those that are in common.

Since modeling the memory as a function gives no clues on how the enclave is organized, we assume a fixed *memory layout* $\mathcal{L} \triangleq \langle ts, te, ds, de, isr \rangle$. It describes how the enclave and the *interrupt service routine* (ISR) are placed in non-fragmented portions of memory and is used to check memory accesses during the execution of each instruction (see below). To reflect the memory segmentation of the real Sancus, we have two protected memory sections, containing the code and the data of the enclave. The protected code section is denoted by $[ts, te)$, while $[ds, de)$ is the protected data section, and they are placed in non-overlapping memory sections. The first address of the protected code section is the single entry point of the enclave. The last component of the tuple $\mathcal{L}$, isr, is the address of the ISR. 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. Note that memory operations enforce no memory access control w.r.t. $\mathcal{L}$, since these checks are performed during the execution of each instruction (see below).

Summing up, a memory layout is defined as

$$\mathcal{L} \triangleq \langle ts, te, ds, de, isr \rangle, \quad \text{where}$$

- $[ts, te)$ and $[ds, de)$ are the protected code and data sections, resp., with $[ts, te) \cap [ds, de) = \emptyset$;
- $isr \notin [ts, te) \cup [ds, de)$ is the entry point for the ISR;
- $isr \neq \texttt{0xFFFE}$, and $\texttt{0xFFFE} \notin [ts, te) \cup [ds, de)$. The address `0xFFFE` is the one from which the CPU starts executing on boot, or on an exception.

4.2 Register files

Sancus$^{\mathsf{H}}$, just like the original Sancus, has sixteen 16-bit registers three of which R_0, R_1, R_2 are used for dedicated functions, whereas the others are for general use. (R_3 is a constant generator in the real MSP430 machine, but we ignore that use in our formalization.) More precisely, 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 R_1 (sp hereafter) is the stack pointer and it is used, as usual, by the CPU to store the pointer to the activation record of the current procedure. Also the stack pointer is aligned to even addresses. The register R_2 (sr hereafter) is the status register and contains different pieces of information encoded as flags. The most important here is the fourth bit, called `GIE`, set to 1 when interrupts are enabled. Other bits are set, e.g., when an operation produces a carry or when the result of an operation is zero.

Formally, our *register file* $\mathcal{R}$ is a function that maps each register r to a word. The read operation is standard:

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

The write operation requires instead accommodating the hardware itself and our security requirements:

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

In the definition above & and | denote the standard *and* and *or* bitwise operators, and we use the relation $\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{m}$, for $m \in \{\mathsf{PM}, \mathsf{UM}\}$ that is defined in Section 4.7. It indicates that the execution is carried on in protected or in unprotected mode. Note that word alignment is enforced because the least-significant bit of the program counter and of the stack pointer are *always* masked to 0 (as it happens in MSP430). Also, 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 in protected code (resulting from the bitwise *or* between $w\&\texttt{0xFFF7}$ - masking the `GIE` bit of w - and $\mathcal{R}[\texttt{sr}]\&\texttt{0x8}$ - masking everything except the value of the `GIE` bit of the status register).

Finally, it is convenient defining the following special register files:

$$\mathcal{R}_0 \triangleq \{\texttt{pc} \mapsto 0, \texttt{sp} \mapsto 0, \texttt{sr} \mapsto 0, \texttt{R}_3 \mapsto 0, \ldots, \texttt{R}_{15} \mapsto 0\}$$

$$\mathcal{R}_{\mathcal{M}}^{init} \triangleq \{\texttt{pc} \mapsto \mathcal{M}[\texttt{0xFFFE}], \texttt{sp} \mapsto 0, \texttt{sr} \mapsto \texttt{0x8}, \texttt{R}_3 \mapsto 0, \ldots, \texttt{R}_{15} \mapsto 0\}$$

where

- `pc` is set to $\mathcal{M}[\texttt{0xFFFE}]$ as it does in the MSP430;
- `sp` is set to 0 and we expect untrusted code to set it up in a setup phase, if any;
- `sr` is set to `0x8`, i.e., register is clear except for the `GIE` flag.

Register file $\mathcal{R}_0$ is used when we jump out from the enclave to zero the processor state; $\mathcal{R}_M$ denotes the initial file register of the CPU, when it starts executing.

4.3 I/O Devices

Recall from the previous section that the attacker can raise an interrupt and observe the effects it has on the execution of the enclave. This kind of attack usually requires a software component and a hardware one. The software component is settled in the unprotected memory and is detailed below. The hardware component is a physical device that interacts with the processor through synchronous I/O operations. Additionally, the progress of I/O devices is tied to that of the CPU, making them cycle-accurate and allowing to model the full power of the attacker considered in the real Sancus (e.g., to use a cycle-accurate timer). In our case it is a Sancus I/O device, and we model it as a (simplified) *deterministic I/O automaton* [39], as follows:

$$\mathcal{D} \triangleq \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle, \quad \text{where}$$

- $a \in A$, with A a signature that includes the following actions (below w is a word):
  - ϵ, a silent, internal action;
  - $rd(w)$, an output action (i.e., read request from the CPU);
  - $wr(w)$, an input action (i.e., write request from the CPU);
  - $int?$ an output action telling that an interrupt was raised in the last state;
- $\Delta \neq \emptyset$ is the *finite* set of internal states of the device;
- $\delta_{\text{init}} \in \Delta$ is the *single* initial state;
- $\delta \overset{a}{\rightsquigarrow}_D \delta' \subseteq \Delta \times A \times \Delta$ is the transition function that takes one step in the device while doing action $a \in A$, starting in state δ and ending in state δ'. (We write $\overline{a}$ for a string of actions and we omit ϵ when unnecessary.) The transition function is such that $\forall \delta$ either $\delta \overset{\epsilon}{\rightsquigarrow}_D \delta'$ or $\delta \overset{int?}{\rightsquigarrow}_D \delta''$ (i.e., one and only one of the two transitions must be possible), also at most one $rd(w)$ action must be possible starting from a given state.

4.4 Software modules, contexts and whole programs

A module contains both protected code and protected data.

Definition 4.1. A *software module* is a memory $\mathcal{M}_M$ containing both protected code and protected data sections.

Intuitively, the context is the part of the whole program that can be manipulated by an attacker, i.e., the software component and the physical device:

Definition 4.2. 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.

Filling in a context hole with a software module yields a whole program.

Definition 4.3. Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ and a software module $\mathcal{M}_M$ such that $\mathtt{dom}(\mathcal{M}_C) \cap \mathtt{dom}(\mathcal{M}_M) = \emptyset$, a *whole program* is

$$C[\mathcal{M}_M] \triangleq \langle \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{D} \rangle.$$

4.5 Instruction set

The instruction set *Inst* is the same for both Sancus^{L} and Sancus^{H} and is (almost) that of the MSP430. An overview of the instruction set is in Table 1. For each instruction i the table includes its operands, an intuitive meaning of its semantics, its duration and the number of words it occupies in memory. The durations are used to define the function $cycles(i)$ and implicitly determine a value `MAX_TIME`, greater than or equal to the duration of longest instruction. Here we choose `MAX_TIME` $= 6$, in order to maintain the compatibility with MSP430 (whose longest instruction takes 6 cycles). Since instructions are stored in either the unprotected or in the protected code section of the memory $\mathcal{M}$, for getting them 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 $\bot$ otherwise.

4.6 Configurations

Given an I/O device $\mathcal{D}$, the internal state of the entire system is described by configurations of the form:

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

- δ is the current state of the I/O device;
- t is the current clock cycle, i.e., a natural number denoting the time elapsed since the CPU started its execution;
- t_a is the arrival time (clock cycle) of the last pending interrupt, set to $\bot$ if there are none;
- $\mathcal{M}$ is the current memory;
- $\mathcal{R}$ is the current content of the registers;
- pc_{old} is the value of the program counter *before* executing the current instruction
- $\mathcal{B}$ is the *backup* that can assume the following values:
  - $\bot$, indicating that the CPU is either handling no interrupt or it is handling one originated in unprotected mode;
  - $\langle \mathcal{R}, pc_{old}, t_{pad} \rangle$, indicating that the interrupt handler is managing an interrupt raised in protected mode. The triple includes the register file $\mathcal{R}$, the program counter pc_{old} at the time the interrupt was originated, and the value t_{pad}, which indicates the remaining padding time that must be applied before returning into protected mode;
  - $\langle \bot, \bot, t_{pad} \rangle$, indicating that the CPU is currently padding the resumption from an interrupt.

The initial states of the CPU are represented by the initial configurations from which the computation starts. The initial configuration for a whole program $C[\mathcal{M}_M] = \langle \mathcal{M}, \mathcal{D} \rangle$ is:

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

- the state of the I/O device $\mathcal{D}$ is δ_{init};
- the initial value of the clock is 0 and no interrupt has arrived yet;
- the memory is initialized to the whole program memory $\mathcal{M}_C \uplus \mathcal{M}_M$;

- all the registers are initialized to `0`, their initial value, 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);
- the "old" program counter is also initialized to `0xFFFE`;
- the backup is set to $\bot$, as no interrupt has been raised yet.

Dually, HALT is the only configuration denoting termination. More precisely, we feel free to use this distinguished and opaque configuration for representing 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`:

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

4.6.1 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:

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

Intuitively, assume the device be in state δ, the clock time be t and the last interrupt be raised at time t_a. Then, after k cycles the new clock time will be $t' = t + k$, the last interrupt was raised at time t'_a and the new state will be δ'; when no interrupt has to be handled, $t_a = t'_a = \bot$. Formally:

$$\frac{a \in \{\epsilon, int?\} \qquad \bigwedge_{i=0}^{k-1} \delta_i \overset{a}{\rightsquigarrow}_D \delta_{i+1} \qquad t'_a = \begin{cases} t + j & \text{if } \exists 0 \le j < k.\, \delta_j \overset{int?}{\rightsquigarrow}_D \delta_{j+1} \land \\ & \forall j' < j.\, \delta_{j'} \overset{\epsilon}{\rightsquigarrow}_D \delta_{j'+1} \\ t_a & \text{o.w.} \end{cases}}{\mathcal{D} \vdash \delta_0, t, t_a \curvearrowright^k_D \delta_k, (t + k), t'_a}$$

4.7 CPU mode

We now specify when the CPU is running in protected or in unprotected mode. Actually, the mode $\mathtt{m} \in \{\mathtt{PM}, \mathtt{UM}\}$ is determined by the value of the program counter, which can be in either code section:

$$\frac{pc \in [\mathcal{L}.ts, \mathcal{L}.te)}{pc \vdash_{mode} \mathtt{PM}} \qquad \frac{pc \notin [\mathcal{L}.ts, \mathcal{L}.te) \cup [\mathcal{L}.ds, \mathcal{L}.de)}{pc \vdash_{mode} \mathtt{UM}}$$

Also, we lift the definition to configurations as follows:

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{m}}{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \vdash_{mode} \mathtt{m}} \qquad \frac{}{\mathrm{HALT} \vdash_{mode} \mathtt{UM}}$$

Note in passing that no mode is defined when the program counter points within the data section, because the memory access control introduced below prevents the program counter to assume values therein.

4.8 Memory access control

We formalize the *memory access control* (MAC) mechanism of Sancus using the predicate $MAC_{\mathcal{L}}(f, \mathtt{rght}, t)$ in Table 2. Roughly, this predicate holds whenever the address that the CPU is trying to read is within the same memory partition as the program counter of the last completed instruction (pc_{old}); in other words, whenever from the location f (usually pc_{old}) we have the rights `rght` on location

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

Table 2. Definition of $MAC_{\mathcal{L}}(f, \mathtt{rght}, t)$ function, where f and t are locations.

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{sp}] \neq 2^{16}-1 \quad \mathcal{R}[\mathtt{sp}]+2 \neq 2^{16}-1 \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \\ MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+1) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+2) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+3)\end{array}}{\mathtt{RETI}, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}i \in \{\mathtt{NOP}, \mathtt{AND}\ \mathtt{r_1}\ \mathtt{r_2}, \mathtt{ADD}\ \mathtt{r_1}\ \mathtt{r_2}, \mathtt{SUB}\ \mathtt{r_1}\ \mathtt{r_2}, \mathtt{CMP}\ \mathtt{r_1}\ \mathtt{r_2}, \mathtt{MOV}\ \mathtt{r_1}\ \mathtt{r_2}, \mathtt{JMP}\ \&\mathtt{r}, \mathtt{JZ}\ \&\mathtt{r}\} \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)\end{array}}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}i \in \{\mathtt{NOT}\ \mathtt{r}, \mathtt{MOV}\ \#w\ \mathtt{r}\} \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+2) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+3)\end{array}}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{i \in \{\mathtt{IN}\ \mathtt{r}, \mathtt{OUT}\ \mathtt{r}\} \quad \mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{r_1}] \neq 2^{16}-1 \quad \mathcal{R}[\mathtt{r_1}]+1 \neq 2^{16}-1 \\ MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{r_1}]) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{r_1}]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)\end{array}}{\mathtt{MOV}\ @\mathtt{r_1}\ \mathtt{r_2}, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{r_2}] \neq 2^{16}-1 \quad \mathcal{R}[\mathtt{r_2}]+1 \neq 2^{16}-1 \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{w}, \mathcal{R}[\mathtt{r_2}]) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{w}, \mathcal{R}[\mathtt{r_2}]+1) \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+2) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+3)\end{array}}{\mathtt{MOV}\ \mathtt{r_1}\ \mathtt{0(r_2)}, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{i \neq \mathtt{RETI} \quad \mathcal{B} \neq \bot \quad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK} \quad \mathcal{R}[\mathtt{sr}].\mathtt{GIE} = \mathtt{0} \quad \mathcal{R}[\mathtt{pc}] \neq ts}{i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK}} \qquad \frac{\mathcal{B} \neq \bot}{\mathtt{RETI}, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK}}$$

Fig. 3. The rules defining the memory access control.

t, reflecting the mechanism provided by Sancus. Note that when f is within unprotected code, $MAC_{\mathcal{L}}(f, \mathtt{rght}, t)$ grants it no rights on a location t in the protected memory.

Building on the above, we define the following relation

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

that 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 $\mathcal{B}$. We check that (1) when transitioning from pc_{old} to $\mathcal{R}[\mathtt{pc}]$, the CPU has execution rights to execute instruction i, i.e., $MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+j)$ for $j \in \{0, ..., size(i)-1\}$; (2) if i is an I/O instruction, it can be executed in current CPU mode; and (3) if i is a memory operation (i.e., either $\mathtt{MOV}\ \mathtt{r_1}\ \mathtt{0(r_2)}$ or $\mathtt{MOV}\ @\mathtt{r_1}\ \mathtt{r_2}$) from $\mathcal{R}[\mathtt{pc}]$ we have the appropriate rights to perform it. The predicate MAC is the minimal relation satisfying the inference rules in Figure 3. Note that (i) for each word that is accessed in memory we also check that the first location is not the last byte of the memory (except for the program counter, for which the decode function would fail since it would try to access undefined memory); (ii) word accesses must be checked once for each byte of the word; and (iii) checks on $\mathtt{pc}$ guarantee that a memory violation does not happen while decoding. We briefly comment on the rule for $i \in \{\mathtt{IN}\ \mathtt{r}, \mathtt{OUT}\ \mathtt{r}\}$, the others being self-explanatory. The pre-conditions say that (i) the current value of the program counter is in unprotected mode; (*ii*) that the instructions pointed to by pc_{old} and $\mathcal{R}[\mathsf{pc}]$ are executable, according to $MAC_{\mathcal{L}}$; and (*iii*) that the same holds for pc_{old} and $\mathcal{R}[\mathsf{pc}] + 1$, i.e., the next instruction.

5 THE SEMANTICS OF Sancus$^{\mathsf{H}}$ AND **Sancus$^{\mathbf{L}}$** AND THEIR INTERRUPT LOGIC

As anticipated, we proceed to formally define the semantics of Sancus$^{\mathsf{H}}$ and **Sancus$^{\mathbf{L}}$**. The two share most of their structure and just differ in the way they deal with interrupts, because Sancus$^{\mathsf{H}}$ has none of them and so the handler is trivial, while **Sancus$^{\mathbf{L}}$** has an appropriate interrupt logic, based on the mitigation intuitively introduced in Section 3. Each version of Sancus is endowed with two transitions systems: the main one specifies the operational semantics of instructions, while the other is auxiliary and describes the relevant interrupt logic. Therefore, we will factorize as much as possible the inference rules shared by the main transition systems, and only indicate the differences using the mentioned code: blue, sans-serif font for Sancus$^{\mathsf{H}}$ and in **red, bold** font elements for **Sancus$^{\mathbf{L}}$**.

More precisely, assume hereafter as given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, where M_C defines the contents of the memory locations of the unprotected section and $\mathcal{D}$ is an I/O device, and let $c, c' \in \mathbb{C}$ be two configurations. Then, the main transition system of Sancus$^{\mathsf{H}}$ has the transitions on the left and its auxiliary one the transitions on the right:

$$\mathcal{D} \vdash c \rightarrow c' \qquad\qquad\qquad \mathcal{D} \vdash c \hookrightarrow_{\mathsf{I}} c'$$

while the main and the auxiliary transition systems of **Sancus$^{\mathbf{L}}$** have the transitions on the left and on the right, respectively:

$$\mathcal{D} \vdash c \rightarrow c' \qquad\qquad\qquad \mathcal{D} \vdash c \hookrightarrow_{\mathbf{I}} c'$$

5.1 The Operational Semantics of Sancus$^{\mathsf{H}}$

We first present the auxiliary transition system implementing the logic that decides what happens when an interrupt arrives, and then we formalize how the instructions are executed in Sancus$^{\mathsf{H}}$.

5.1.1 Interrupts in Sancus$^{\mathsf{H}}$*.* Interrupts in Sancus$^{\mathsf{H}}$ are *always* ignored, thus the configuration is left unchanged, and we have the following trivial rule:

$$\textsf{INT}\quad \frac{}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

5.1.2 Main transition system. The transitions of the main transition system describe how the Sancus$^{\mathsf{H}}$ configurations evolve during the execution. Figure 4 shows selected inference rules of the transition system, on which we briefly comment below; the other rules can be found in the Appendix.

The rule (CPU-HLT-UM) is the only one that halts the CPU and only applies when an HLT instruction is executed in unprotected mode. Dually the rule (CPU-HLT-PM) deals with the case in which an HLT instruction is to be executed in protected mode. In such a case, the *exception handling* configuration is reached, allowing for a cleanup and a graceful termination. The rule (CPU-Violation-PM) takes care of the violations in protected mode: the transition in the conclusion of the rule leads to the *exception handling* configuration if there is a non-empty backup (first premise) and if the instruction i does not pass the memory-access control relation (second premise). The rule (CPU-MovL) is for when the current instruction i loads in r_2 the word in memory at the position pointed to by r_1. Its first premise checks that the CPU is not currently padding interrupt resumption time (more details on that later on, it can be safely ignored for now); the second one if the instruction can be executed; the third one increments the program counter by 2 and loads in r_2 the value $\mathcal{M}[\mathsf{r}_1]$; the fourth 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 below). Rules dealing with jumps are quite standard. Upon a `JZ &r` instruction (*jump if zero*), the CPU checks the content of the Z (*zero*) bit of the status register. If $\mathcal{R}[\mathtt{sr}].Z$ is 0, then the rule (CPU-Jz0) is triggered and $\mathcal{R}[\mathtt{pc}]$ is left unchanged, otherwise the rule (CPU-Jz1) applies and the content of the register `r` is copied into `pc`, so performing the jump. 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 third premise holds when the device sends the word w to the CPU; the others are similar to those of (CPU-MovL). Dually, the rule (CPU-Out) deals with outputs to the device. Note that, the CPU is forced to halt when the I/O device is not ready for a read or a write (rules (CPU-NoIn) and (CPU-NoOut)). As a matter of fact, this can only happen in unprotected mode, since the MAC relation forbids I/O operations inside enclaves. Note also that the current time of the CPU is *always* incremented by the time needed to complete the current instruction.

5.2 The Operational Semantics of Sancus$^{\mathrm{L}}$

In Sancus$^{\mathrm{L}}$ interrupts can be raised and must be properly handled securely both in protected and unprotected mode, and for that we define a non-trivial auxiliary transition system. Although the rules of the main transition system are largely the same of Sancus$^{\mathrm{H}}$, the new auxiliary transitions affect the behaviour of the instruction for returning from interrupts.

5.2.1 Interrupts in Sancus$^{\mathrm{L}}$. The inference rules in Figure 5 formalize the mitigation outlined in Section 3 as a defense against interrupt-based attacks, regardless of the CPU being in unprotected or protected mode. To intuitively clarify how our rules realize the secure padding schema, we refer again to Figure 2. We still denote the time when the interrupt is raised with t_a. Instead, the intervals Δt_1 (in the rules $t - t_a$) and Δt_{p_1} (in the rules $k = \mathtt{MAX\_TIME} - (t - t_a)$) represent the time to complete the current instruction and the padding before the ISR starts, respectively. The interval Δt_{p_2} (in the rules t_{pad}) completes the padding making sure that mitigation always amounts to `MAX_TIME` (recall from Section 4.5 that the longest instruction takes 6 cycles). Note also that MSP430 takes 6 cycles to set up the call to the interrupt handler (which is not displayed in Figure 2).

All the semantic rules have a premise checking the mode in which the last instruction was executed ($pc_{old} \vdash_{mode} \mathtt{UM}$ or $pc_{old} \vdash_{mode} \mathtt{PM}$).

The rules **(INT-UM-NP)** and **(INT-PM-NP)** take care of when the `GIE` bit of the status register is set to 0, i.e., interrupts are disabled, or there is none ($t_a = \bot$). In this case the configurations are simply left untouched.

When instead $\mathtt{GIE} = 1$ and an interrupt is on ($t_a \neq \bot$), either rule **(INT-UM-P)** or **(INT-PM-P)** handles it. When in unprotected mode, 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 to allocate the activation record of the interrupt handler.

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 reflects that setting up this interrupt handling takes 6 cycles.

The rule **(INT-PM-P)** is for protected mode and it 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 implements the first part of the mitigation (see Section 3.3) 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 $\mathtt{GIE} = 1$, and the padding is non-negative because the interrupt is handled at the end of the current instruction; or $\mathtt{GIE} = 0$, and no padding is needed because the interrupt is handled as soon as GIE becomes 1, which is only possible in unprotected mode. The 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 3.3). Recall that the register file $\mathcal{R}_0$ is $\{\mathtt{pc} \mapsto 0, \mathtt{sp} \mapsto 0, \mathtt{sr} \mapsto 0, \mathtt{R}_3 \mapsto 0, \dots, \mathtt{R}_{15} \mapsto 0\}$.

It might be worthy to briefly describe what happens upon "corner cases:"

- Whenever an interrupt has to be handled in protected mode, but the current instruction drives the CPU in unprotected mode, the padding mechanism is applied as in the rule (CPU-Reti) *including* the padding after the RETI. Indeed, if partial padding (resp. no padding at all) was applied then the duration of the padding (resp. of the last instruction) would be leaked to the attacker (cf. Figure 5).
- Interrupts are ignored when arising during the time spent in padding and *before* invoking the interrupt service routine. This is because the padding duration and the instruction duration would be leaked otherwise. To avoid that, the rule (INT-PM-P) ignores any interrupts raised during the cycles needed for the interrupt logic and for the padding. A viable alternative would require to buffer interrupts and handle them later on.
- Interrupts happening *during* the execution of the interrupt service routine are simply "chained" and handled as soon as the current routine completes (see rule (CPU-Reti-Chain)).
- Finally, interrupts raised during the padding time and *after* the interrupt service routine are handled as any other interrupt happening in protected mode (see rule (CPU-Reti-Pad)).

5.2.2 *Main transition system.*

The rules of the main transition system of Sancus^{L} are exactly the same used for the semantics of Sancus^{H}, except for the blue arrows turned into red, notably those for the interrupt logic: the red arrow $\hookrightarrow_{\text{I}}$ replaces the blue arrow $\hookrightarrow_{\text{I}}$ in the premises.

Figure 6 shows the rules dealing with the cases that may happen when the interrupt handler returns and the processor gives the control back to the code that was executing before the interrupt was raised. The first rule (CPU-Reti), deals with the actual return from an interrupt. In this case the processor restores the status register and sets the program counter to the instruction following the interrupted one. The previous values of these registers are stored in the current activation record on the stack (i.e., $\mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{M}[\mathcal{R}[\mathtt{sp}] + 2], \mathtt{sr} \mapsto \mathcal{M}[\mathcal{R}[\mathtt{sp}]]]$). Instead, rule (CPU-Reti-Chain) applies if an interrupt arrived while returning from handling an interrupt raised in protected mode (third and fifth premises). In this case the CPU directly jumps to the handler of the new interrupt with no further padding. Finally, we discuss the rules (CPU-Reti-PrePad) and (CPU-Reti-Pad). Their combination deals with the case in which the CPU is returning from handling an interrupt raised in protected mode, and no new interrupt arrived afterwards (or the GIE bit is off, cf. the fourth premise of rule (CPU-Reti-PrePad)). First, the rule (CPU-Reti-PrePad) restores registers and pc_{old} from the backup $\mathcal{B}$, so enabling the application of the rule (CPU-Reti-Pad) (note that no other rule is applicable because of the contents of $\mathcal{B}$). Then, through the rule (CPU-Reti-Pad) the remaining padding (recorded in the backup) is applied so to prevent resume-to-end timing attacks (note that this last padding is interruptible, as witnessed by the last premise). This last padding is applied even though the configuration reached through rule (CPU-Reti-PrePad) is in unprotected mode (i.e., when the interrupted instruction was a jump out of protected mode). Otherwise, the attacker may discover the value of the padding applied *before* the interrupt service routine. Actually, 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.

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathsf{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{HLT}$$

(CPU-NoIN)

$$\frac{\delta \overset{rd(w)}{\not\rightsquigarrow}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{IN}\ \mathsf{r}$$

(CPU-NoOUT)

$$\frac{\delta \overset{wr(\mathcal{R}[\mathsf{r}])}{\not\rightsquigarrow}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{OUT}\ \mathsf{r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathsf{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{HLT}$$

(CPU-Decode-Fail)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}}$$

(CPU-Violation-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \nvdash_{mac} \mathsf{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) \neq \bot$$

(CPU-MovL)

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

(CPU-Jz0)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_{D}^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \land \mathcal{R}[\mathsf{sr}].Z = 0$$

(CPU-Jz1)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_{D}^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \land \mathcal{R}[\mathsf{sr}].Z = 1$$

(CPU-In)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \delta \overset{rd(w)}{\rightsquigarrow}_{D} \delta' \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_{D}^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{IN}\ \mathsf{r}$$

(CPU-Out)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \delta \overset{wr(\mathcal{R}[\mathsf{r}])}{\rightsquigarrow}_{D} \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_{D}^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{OUT}\ \mathsf{r}$$

Fig. 4. Some rules of the main transition system for Sancus$^{\mathsf{H}}$.

(INT-UM-P)

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

(INT-UM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \qquad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = \texttt{0} \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(INT-PM-P)

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

(INT-PM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{PM} \qquad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = \texttt{0} \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

Fig. 5. The transition system for handling interrupts in Sancus$^{\mathrm{L}}$.

(CPU-Reti)

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

(CPU-Reti-Chain)

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

(CPU-Reti-PrePad)

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

(CPU-Reti-Pad)

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

Fig. 6. Some rules from the operational semantics of Sancus$^{\mathrm{L}}$.

5.3 A progress property

As a sanity check we prove the following progress theorem showing that both Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$ get stuck only if the CPU reaches the distinguished configuration HALT. Its proof is in the Appendix:

Theorem 5.1 (Progress). *For all* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, $\mathcal{M}_M$ *and configuration* c

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \nrightarrow \implies c = \mathrm{HALT} \qquad \textit{and} \qquad \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \nrightarrow \implies c = \mathrm{HALT}.$$

6 THE SECURITY THEOREM

In this section we establish that Sancus$^{\mathrm{L}}$ enjoys the following security property: 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$^{\mathrm{L}}$ is *fully abstract* w.r.t. the semantics of Sancus$^{\mathrm{H}}$; in other words all the attacks that can be carried out in Sancus$^{\mathrm{L}}$ can also be carried out in Sancus$^{\mathrm{H}}$, and viceversa.

Before stating the full abstraction theorem, we introduce some further notations, which also help in the main steps of its proof; additional, minor lemmata and definitions for completing the proofs are in the Appendix. Recall from Section 4.4 that $C[\mathcal{M}_M]$ is a whole program, where $\mathcal{M}_M$ is the software module and $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ represents the context ($\mathcal{M}_C$ contains the unprotected program and data and $\mathcal{D}$ is the I/O device).

We first define the notion of convergence of whole programs.

Definition 6.1. Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ be a context, and $\mathcal{M}_M$ be a software module. A whole program $C[\mathcal{M}_M]$ converges in Sancus$^{\mathrm{H}}$ (written $C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$) *iff*

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* \mathrm{HALT}.$$

Similarly, the same whole program converges in Sancus$^{\mathrm{L}}$ (written $C[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$) *iff*

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* \mathrm{HALT}.$$

The following definition introduces the notion of contextual equivalence of two software modules. Roughly, the notion of contextual equivalence formalizes the intuitive notion of *indistinguishability*: two modules are contextually equivalent if they behave in the same way when they interact with an arbitrary, attacker-controlled 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.

Definition 6.2. Two software modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are contextually equivalent in Sancus$^{\mathrm{H}}$, written $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$, *iff*

$$\forall C. \left(C[\mathcal{M}_M]\Downarrow^{\mathrm{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{H}} \right).$$

Similarly, $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are contextually equivalent in Sancus$^{\mathrm{L}}$, written $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$, *iff*

$$\forall C. \left(C[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{L}} \right).$$

Finally, we state and prove the main theorem establishing the security of our mitigation:

Theorem 6.3 (Full abstraction). $\forall \mathcal{M}_M, \mathcal{M}_{M'}. \ (\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'})$.

Proof. Here we only present the "surface" of the proof by stating the main properties, whose proofs often require many other auxiliary definitions and properties that are detailed in the Appendix. Actually, the proof that our mitigation guarantees absence of interrupt-based attacks is rather long, and has the following steps. We first establish reflection of behaviors: $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \Leftarrow$

$$\begin{tikzcd}
\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \arrow[d, Rightarrow, "(iii)"'] \arrow[dr, Rightarrow, "(ii)"] & \\
\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'} & \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \arrow[l, Rightarrow, "(i)"]
\end{tikzcd}$$

Fig. 7. An illustration of the proof strategy of preservation of behaviors.

$\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ (Lemma 6.6 in Section 6.1). Then, the other implication, i.e., preservation of behaviors is proved by Lemma 6.15 in Section 6.2 following the strategy summarized in Figure 7. We rely on the well-known notion of traces, i.e., the sequences of actions performed by a module $\mathcal{M}_M$ plugged in a context that can be observed by an attacker. In particular we focus on the invocations of $\mathcal{M}_M$ and on the returns from it. In both cases our traces also carry information about the contents of the registers and for returns also the flow of time. We then say that two modules $\mathcal{M}_M$ and $\mathcal{M}'_M$ are trace equivalent, in symbols $\mathcal{M}_M \overset{T}{=} \mathcal{M}'_M$, if they exhibit the same traces (see Definition 6.7). Proving preservation is then done in two steps, the composition of which gives (iii) in Figure 7. First Lemma 6.14 establishes (ii) in Figure 7: two modules equivalent in Sancus$^{\mathsf{H}}$ are trace equivalent. The proof technique that we adopt specializes backtranslation of [47], applied to the contrapositive of (ii). Roughly, we construct a context in Sancus$^{\mathsf{H}}$ distinguishing two modules when they are not trace equivalent. Then Lemma 6.12 establishes (i) in Figure 7: two modules that are trace equivalent are also equivalent in Sancus$^{\mathsf{L}}$. The proof of this lemma is rather technical: essentially, it consists in showing that neither the context affects the behavior of the module, nor the module affects that of the context.

Summing up:

- *Case* $\Leftarrow$. Reflection of behaviors follows from Lemma 6.6 in Section 6.1.
- *Case* $\Rightarrow$. Preservation of behaviors follows from Lemma 6.15 in Section 6.2. □

6.1 Reflection of behaviors

Recall that Sancus$^{\mathsf{L}}$ differs from Sancus$^{\mathsf{H}}$ because of its interrupt handling mechanism, only. Consequently, to prove the reflection of behaviors, i.e., that for all $\mathcal{M}_M, \mathcal{M}_{M'}$ we have that $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ implies $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ it suffices to inhibit interrupts in Sancus$^{\mathsf{L}}$. For establishing that, we introduce the notion of *interrupt-less context* $C_{\not I}$ for a context C. Intuitively, $C_{\not I}$ behaves as C but never raises any interrupt. When a module is plugged in an interrupt-less context, it terminates according to the low level semantics if and only if it does in the high level semantics. Technically, to obtain the interrupt-less version of a context C it suffices to remove from the device the transitions that may raise an interrupt.

Definition 6.4. Let $\mathcal{D} = \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle$ be an I/O device. Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, we define its corresponding *interrupt-less context* as $C_{\not I} = \langle \mathcal{M}_C, \overset{a}{\rightsquigarrow}_{D\not I} \rangle$ where:

- $\mathcal{D}_{\not I} = \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_{D\not I} \rangle$, and
- $\overset{a}{\rightsquigarrow}_{D\not I} \triangleq \overset{a}{\rightsquigarrow}_D \cup \{(\delta, \epsilon, \delta') \mid (\delta, int?, \delta') \in \overset{a}{\rightsquigarrow}_D\} \setminus \{(\delta, int?, \delta') \mid (\delta, int?, \delta') \in \overset{a}{\rightsquigarrow}_D\}$.

Note that $\mathcal{D}_{\not I}$ is actually a device, due to the constraints on its transition function.

The behavior of interrupt-less contexts in Sancus$^{\mathsf{L}}$ directly correspond to the behavior of their standard counterparts in Sancus$^{\mathsf{H}}$ as stated below.

Lemma 6.5. *For any module $\mathcal{M}_M$, context C, and corresponding interrupt-less context $C_{\not I}$:*

$$C_{\not I}[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$$

Reflection now follows, because whole programs in Sancus$^{\mathrm{H}}$ behave just like a subset of whole programs in Sancus$^{\mathrm{L}}$.

Lemma 6.6 (Reflection). $\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ (\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'})$.

6.2 Preservation of behaviors

Here, we prove the preservation of behaviors, i.e., the chain of implications (ii) and then (i), resulting in (iii) in Figure 7. More precisely we perform the following steps.

In Section 6.2.1 we first define two notions of traces: the *fine-grained* and *coarse-grained* traces. The first is an auxiliary notion that directly derives from the semantics of Sancus$^{\mathrm{L}}$ and facilitates the proofs. Intuitively, it takes into account all the actions performed by the system. The second kind of traces only records the actions that attackers can observe, and are easily derived from the fine-grained ones. Also, we call trace equivalent two modules with the same set of coarse-grained traces. Using the fine-grained traces, we state and prove the key yet rather technical Property 6.1 ensuring that our mitigation reflects the intuition described in Figure 2. This property also helps in showing that, roughly speaking, the observed actions of the enclave are not influenced by those of the context (Lemma 6.10), as well as in proving the correctness of the backtranslation algorithm (Property A.22). For proving both facts, we use the timing information recorded in the coarse-grained traces that result from assembling those in the fine-grained traces.

Then we prove in Section 6.2.2 that trace equivalence implies contextual equivalence at Sancus$^{\mathrm{L}}$ (the implication (i) of Figure 7). For that Lemma 6.11 is crucial, since it ensures that two trace equivalent modules still produce the same traces when plugged in a given context.

Next, in Section 6.2.3 we prove that contextual equivalence implies trace equivalence at Sancus$^{\mathrm{H}}$ (the implication (ii) of Figure 7). This is achieved by defining a *backtranslation* [47] that given an attacker (a context that differentiate two modules) at Sancus$^{\mathrm{L}}$ returns an attacker at Sancus$^{\mathrm{H}}$.

Finally, Section 6.2.4 immediately concludes our proof of item (iii) in Figure 7.

6.2.1 *Fine-grained and coarse-grained traces.*

We consider the fine-grained and the coarse-grained traces. The first traces record the relevant actions performed by the processors including those concerned with interrupt handling. The coarsed-grained, instead, record what the attacker is able to observe, i.e., jumping in and out an enclave.

The fine-grained observables are defined as follows:

$$\alpha ::= \xi \mid \tau(k) \mid \texttt{reti?}(k) \mid \texttt{handle!}(k) \mid \bullet \mid \texttt{jmpIn?}(\mathcal{R}) \mid \texttt{jmpOut!}(k;\mathcal{R}).$$

Above, $k \in \mathbb{N}$ indicates that the observed action takes k cycles. Intuitively, ξ denotes unobservable actions performed by the context; $\tau(k)$ indicates an internal action; $\texttt{handle!}(k)$ and $\texttt{reti?}(k)$ denote when the processor starts executing the interrupt service routine from protected mode and when it returns from it, respectively. Then, the observable $\bullet$ indicates that termination occurred; $\texttt{jmpIn?}(\mathcal{R})$ and $\texttt{jmpOut!}(k;\mathcal{R})$ record when the CPU enters and exits from protected mode, respectively, where $\mathcal{R}$ is the contents of the register file when the action ends.

The relation $\xRightarrow{\alpha}$ in Figure 8 extracts observables from the execution of a whole program. Note that each transition $\mathcal{D} \vdash c \rightarrow c'$ has a corresponding transition $\mathcal{D} \vdash c \xRightarrow{\alpha} c'$ for some α, possibly the silent ξ. The transitive and reflexive closure of $\xRightarrow{\alpha}$ is $\xRightarrow{\overline{\alpha}}{}^{*}$, where $\overline{\alpha}$ is a trace, i.e., a sequence of actions (ϵ is the empty trace).

Note that in any trace $\overline{\alpha}$, *only* the observables $\tau(k)$, $\texttt{reti?}(k)$ or $\texttt{handle!}(k)$ may occur between a $\texttt{jmpIn?}(\mathcal{R})$ and a $\texttt{jmpOut!}(k;\mathcal{R})$. When an interrupt has to be handled, the observed trace starts

(Obs-Internal-PM)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{PM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\tau(k)} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle}$$

(Obs-JmpIn)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}')} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle}$$

(Obs-Reti)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{B} \neq \bot \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\mathtt{reti?}(k)} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle}$$

(Obs-JmpOut)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{PM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{jmpOut!}(k;\mathcal{R}')} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-JmpOut-PostPoned)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle \xRightarrow{\mathtt{jmpOut!}(k;\mathcal{R}')} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-Handle)

$$\frac{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{B}' \neq \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{handle!}(k)} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-Internal-UM)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B} \rangle \qquad \mathcal{R}'[\mathsf{pc}] \vdash_{mode} \mathsf{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\xi} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B} \rangle}$$

(Obs-Final)

$$\frac{\mathcal{R}[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\bullet} \mathrm{HALT}}$$

Fig. 8. The relation $\xRightarrow{\alpha}$ for fine-grained observables.

with handle!(·), followed by a sequence of ξ and then a reti?(k), provided that a RETI is executed (k always has value *cycles*(RETI)). If the interrupted instruction was a jump from protected to unprotected mode, the reti?(·) is followed by a jmpOut!(·; ·) (cf. rules (Obs-Handle), (Obs-Internal-UM), (Obs-Reti) and (Obs-JmpOut-PostPoned)); otherwise a $\tau(\cdot)$ – or a handle!(·) if an interrupt has to be handled.

Actually, an attacker (i.e., the context) cannot observe all α's, but only the following coarse-grained observables, where jmpIn?($\mathcal{R}$) and jmpOut!(Δt;$\mathcal{R}$) represent invoking a module and returning from it.

$$\beta ::= \bullet \mid \mathtt{jmpIn?}(\mathcal{R}) \mid \mathtt{jmpOut!}(\Delta t; \mathcal{R}).$$

In Figure 9 we define the relation $\overset{\beta}{\Longrightarrow\!\!\!\!\!\!\Longrightarrow}$, under the assumption that both the existentially quantified configuration c and the configuration c' are reachable from the same initial configuration. Essentially, we remove the observables for interrupts and silent actions from the fine-grain traces, making them not visible any longer. More in detail, all the actions in between a jmpIn?(·) and the immediately following jmpOut!(k; $\mathcal{R}$) (or a •) are dropped; similarly for the fine-grained observables in between a jmpOut!(k;$\mathcal{R}$) (or the very first observable from the initial configuration) and the next jmpIn?(·). In addition, the parameter k is replaced by Δt in the observable jmpOut!(Δt;$\mathcal{R}$) to model that

$$\frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\xi\cdots\xi\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c}{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c} \qquad \frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\xi\cdots\xi\cdot\bullet}{}^{*} \text{HALT}}{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\bullet} \text{HALT}}$$

$$\frac{\exists c.\, \mathcal{D} \vdash c \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\xi\cdots\xi\cdot\texttt{jmpIn?}(\mathcal{R}'')}{}^{*} c''}{\mathcal{D} \vdash c' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c''} \qquad \frac{\exists c.\, \mathcal{D} \vdash c \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\xi\cdots\xi\cdot\bullet}{}^{*} \text{HALT}}{\mathcal{D} \vdash c' \xRightarrow{\bullet} \text{HALT}}$$

$$\frac{\begin{array}{c}\exists c.\, \mathcal{D} \vdash c \xRightarrow{\texttt{jmpIn?}(\mathcal{R}')} c' \\ \mathcal{D} \vdash c' \xRightarrow{\alpha^{(0)}\cdots\alpha^{(n-1)}\cdot\texttt{jmpOut!}(k'';\mathcal{R}'')}{}^{*} c'' \qquad \forall 0 \le i < n.\, \alpha_i \notin \{\texttt{jmpOut!}(\_;\_), \bullet\} \qquad \Delta t = k'' + \sum_{i=0}^{n-1} time(\alpha^{(i)})\end{array}}{\mathcal{D} \vdash c' \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}'')} c''}$$

$$\frac{\exists c.\, \mathcal{D} \vdash c \xRightarrow{\texttt{jmpIn?}(\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\alpha_0\cdots\alpha_{n-1}\cdot\bullet}{}^{*} \text{HALT} \qquad \forall 0 \le i < n.\, \alpha_i \notin \{\texttt{jmpOut!}(\_;\_), \bullet\}}{\mathcal{D} \vdash c' \xRightarrow{\bullet} \text{HALT}}$$

$$\text{where} \quad time(\alpha) = \begin{cases} k & \text{if } \alpha \in \{\texttt{reti?}(k), \texttt{handle!}(k), \tau(k), \texttt{jmpOut!}(k;\mathcal{R})\} \\ 0 & \text{o.w.} \end{cases}$$

Fig. 9. The relation $\xRightarrow{\beta}$ for coarse-grained observables.

an attacker can only measure the end-to-end time of a piece of code running in protected mode. The value Δt is computed by accumulating the values $time(\alpha^{(i)})$ that are the number of cycles associated with the observable $\alpha^{(i)}$.

Then, we take its reflexive and transitive closure $\xRightarrow{\overline{\beta}}{}^{*}$ (where traces $\overline{\beta}$ are strings of β's), and we use it to eventually define when two modules are trace equivalent:

Definition 6.7. Two modules are *(coarse-grained) trace equivalent*, written $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, *iff*

$$Tr(\mathcal{M}_M) = Tr(\mathcal{M}_{M'}).$$

where $Tr(\mathcal{M}_M) \triangleq \{\overline{\beta} \mid \exists C = \langle \mathcal{M}_C, \mathcal{D}\rangle.\, \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c'\}$.

Notation. Hereafter let $x \in \{1, 2\}$; let $c, c_1, c_2, \ldots$, possibly dashed, be configurations; and let $c_x^{(n)} = \langle \delta_x^{(n)}, t_x^{(n)}, t_{a_x}^{(n)}, \mathcal{M}_x^{(n)}, \mathcal{R}_x^{(n)}, pc_{old_x}^{(n)}, \mathcal{B}_x^{(n)}\rangle$ be the configuration reached after n execution steps from the initial configuration $c_x^{(0)}$. We will index the elements of a trace and the components of a context C_x in a similar way. Finally, let $c_x^{(i)}$ be the configuration *right before* the action of index i in a given (fine- or coarse-grained) trace.

To prove a crucial property of our mitigation, it is convenient to introduce the notion of *complete interrupt segments* of a fine-grained trace, which are those starting with an `handle!`$(\cdot)$ action and ending with a `reti?`$(\cdot)$ action (see Definition A.1 in the Appendix). Also, let $|\mathbb{I}_{\overline{\alpha}}|$ be the number of the complete interrupt segments in a given trace $\overline{\alpha}$.

The property below characterizes how our mitigation affects the execution time of a module. Intuitively, it ensures that handling each interrupt contributes to the time spent in protected mode with a constant number of cycles equal to 11 + MAX_TIME. This is crucial to guarantee a constant delay *before and after* interrupt handling, otherwise an attacker would be able to observe different timings as it happens in Examples 3.1 and 3.3. In its statement $\gamma(c)$ is the time taken by the current protected-mode instruction in the given configuration to be executed (cf. Figure 2). Also, recall from Figure 9 that $time(\alpha^{(i)})$ indicates the number of cycles associated with the observable $\alpha^{(i)}$.

Note that when the observables are `reti?`(k) and `handle!`(k), the value k takes care of `MAX_TIME`, as dictated by the interrupt logic of Sancus$^{\mathrm{L}}$ in Figure 5.

Property 6.1. *If* $c^{(0)} \vdash_{mode} \mathsf{PM}$ *and* $\mathcal{D} \vdash c^{(0)} \xRightarrow{\overline{\alpha}}{}^{*} c^{(n+1)}$*, with* $\overline{\alpha} = \alpha^{(0)} \cdots \alpha^{(n-1)} \cdot \mathtt{jmpOut!}(k^{(n)}; \mathcal{R}')$*, then* $k^{(n)} + \sum_{i=0}^{n-1} time(\alpha^{(i)}) = \sum_{i=0}^{n} \gamma(c^{(i)}) + (11 + \mathtt{MAX\_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}}|$*, where*

$$\gamma(c) \triangleq \begin{cases} cycles(decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}])) & if\ c \vdash_{mode} \mathsf{PM} \wedge \mathcal{B} = \bot \\ 0 & o.w. \end{cases}$$

Proof. By definition of the interrupt logic and the operational semantics of Sancus$^{\mathrm{L}}$, for each interrupt handled in protected mode we perform a $0 \leq k \leq \mathtt{MAX\_TIME}$ padding *before* invoking the interrupt service routine and an additional padding of $(\mathtt{MAX\_TIME} - k)$ cycles *after* its execution, i.e., the padding time introduced for each complete interrupt segment amounts to `MAX_TIME`. Also, since the interrupt logic always requires 6 cycles to jump to the interrupt service routine and 5 cycles are required upon `RETI` it easily follows that:

$$k^{(n)} + \sum_{i=0}^{n-1} time(\alpha^{(i)}) = \sum_{i=0}^{n} \gamma(c^{(i)}) + (11 + \mathtt{MAX\_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}}|. \qquad \square$$

6.2.2 *Trace equivalence implies contextual equivalence at* Sancus$^{\mathrm{L}}$.

Here we prove the implication (i) of Figure 7, i.e., that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$. We rely on the following proposition to ensure that a terminating program generates a coarse-grained trace ending with $\bullet$, and vice versa.

Proposition 6.8. $C[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ iff $\exists \overline{\beta}.\ \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}\cdot\bullet}{}^{*} \mathrm{HALT}$.

Proof. The *only-if* part holds trivially. For the other direction, the definition of $C[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ implies that $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} \mathrm{HALT}$ and the definitions of fine- and coarse-grained traces (Figures 8 and 9) guarantee that the last observed action is $\bullet$ as requested. $\square$

Consider two whole programs that share the same context. The lemma below states that if they perform the same sequence of actions reaching a unprotected configuration, then their next action, if any, will be the same (its proof relies on Property A.19). Intuitively, this is because the context is deterministic and because our mitigation makes the context behavior independent of the module. Recall that coarse-grained traces record timing information, and therefore this lemma and the next one also express timing independence between contexts and modules.

Lemma 6.9. *Let* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*. If* $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1 \xRightarrow{\beta} c_1'$, $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ *and* $c_2 \vdash_{mode} \mathsf{UM}$*, then there exists* c_2' *such that* $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$.

The following lemma shows the viceversa: the isolation mechanism offered by the enclave guarantees that the behavior of the module is not influenced by the context:

Lemma 6.10. *Let* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*. If* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1'' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}_1)} c_1 \xRightarrow{\beta} c_1'$ *and* $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2'' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}_2)} c_2$*, then there exists* c_2' *such that* $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$.

The two lemmata above imply that two whole programs obtained by plugging two trace equivalent modules in the same context C produce the same traces:

Lemma 6.11. *Given a context* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ *and two modules* $\mathcal{M}_M$ *and* $\mathcal{M}_{M'}$*. If* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ *and* $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1$*, then there exists a* c_2 *such that* $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2$.

Proof. We show this by induction on the length n of $\overline{\beta}$.

- *Case* $n = 0$. Since $\overline{\beta} = \varepsilon$, by definition of $\dot{\Longrightarrow}^*$, we have $c_1 = \mathrm{INIT}_{C[\mathcal{M}_M]} = c_1$. Again, by definition of $\dot{\Longrightarrow}^*$, we choose $c_2 = \mathrm{INIT}_{C[\mathcal{M}_{M'}]}$ and get the thesis.
- *Case* $n = n' + 1$. The induction hypothesis (IHP) is then:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xrightarrow{\overline{\beta}'}{}^{*} c_1' \Rightarrow \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xrightarrow{\overline{\beta}'}{}^{*} c_2'$$

and we must show that

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xrightarrow{\overline{\beta}'}{}^{*} c_1' \xrightarrow{\beta} c_1 \Rightarrow \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xrightarrow{\overline{\beta}'}{}^{*} c_2' \xrightarrow{\beta} c_2$$

By cases on the CPU mode in c_1' and c_2':
– $\mathcal{R}_1'[\mathtt{pc}] \vdash_{mode} \mathtt{UM}$ *and* $\mathcal{R}_2'[\mathtt{pc}] \vdash_{mode} \mathtt{UM}$*:* Follows by (IHP) and Lemma 6.9;
– $\mathcal{R}_1'[\mathtt{pc}] \vdash_{mode} \mathtt{PM}$ *and* $\mathcal{R}_2'[\mathtt{pc}] \vdash_{mode} \mathtt{PM}$*:* Follows by (IHP) and Lemma 6.10;
– $\mathcal{R}_1'[\mathtt{pc}] \vdash_{mode} \mathtt{m}$ *and* $\mathcal{R}_2'[\mathtt{pc}] \vdash_{mode} \mathtt{m}'$ *and* $\mathtt{m} \neq \mathtt{m}'$*:* It never happens, as observed in Proposition A.6. □

Finally, we conclude with the proof that if two modules are trace equivalent then they are contextually equivalent in Sancus$^{\mathrm{L}}$ (arrow (i) in Figure 7):

Lemma 6.12. *If* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ *then* $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$*.*

Proof. Expanding the definition of $\simeq^{\mathrm{L}}$, the statement becomes:

$$\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \Rightarrow (\forall C = \langle \mathcal{M}_C, \mathcal{D} \rangle . C[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{L}})$$

We split the double implication and we show the two cases independently.

- *Case* $\Rightarrow$. By Proposition 6.8 there exists $\overline{\beta}$ such that $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xrightarrow{\overline{\beta}\cdot\bullet}{}^{*} \mathrm{HALT}$. Since $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, we know by Lemma 6.11 that $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xrightarrow{\overline{\beta}\cdot\bullet}{}^{*} \mathrm{HALT}$. Thus, again by Proposition 6.8, we have $C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{L}}$;
- *Case* $\Leftarrow$. Symmetric to the previous one. □

6.2.3 Contextual equivalence at Sancus$^{\mathrm{H}}$ *implies trace equivalence.* Here we prove by contraposition that $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \implies \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, i.e., implication (ii) of Figure 7.

We first define those traces, if any, that *distinguish* a given a pair of modules, i.e., one converges while the other does not. Given a context in Sancus$^{\mathrm{L}}$ that keeps two modules apart through two such traces, we define two algorithms: the first builds a memory and the other a device. Once put together, they implement a *backtranslation* [47] and return a context differentiating the two modules in Sancus$^{\mathrm{H}}$. Because of the strong limitations of MSP430 (e.g., it only has 64KB of memory) building such a context in unprotected memory only is infeasible. Since the attacker model we assumed has the strong power of controlling everything except the enclave, it is also assumed to control the I/O device that has unlimited memory. Therefore, the backtranslation takes full advantage of such a strength to build a distinguishing context. Of course, this also implies that only isolation properties proved on Sancus$^{\mathrm{H}}$ under this attacker model with unbounded device memory are guaranteed to be preserved in Sancus$^{\mathrm{L}}$.

We start from two distinguishing traces, that consist of a common prefix followed by two further traces starting with two different observables. We then make sure that there always exist such traces for two modules that are kept apart in Sancus$^{\mathrm{L}}$.

Definition 6.13 (Distinguishing traces). Let $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ be two modules, and let $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e \in Tr(\mathcal{M}_M)$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e \in Tr(\mathcal{M}_{M'})$. We say that $\overline{\beta}$ and $\overline{\beta}'$ are *distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$* iff there exist a context $C^L = \langle \mathcal{M}_{C^L}, \mathcal{D}^L \rangle$ such that

- $\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c$ and $\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}'}{}^{*} c'$, for some c, c';
- $\overline{\beta} \notin Tr(\mathcal{M}_{M'})$, $\overline{\beta}' \notin Tr(\mathcal{M}_M)$ and $\beta \neq \beta'$.

Property 6.2. *If $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are two modules such that $\mathcal{M}_M \not\eqsim^{\mathrm{L}} \mathcal{M}_{M'}$, then there always exist $\overline{\beta}$ and $\overline{\beta}'$ that are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$.*

First algorithm: memory initialization. Intuitively, given two modules and two distinguishing traces $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$ for them, the Algorithm 1 builds the memory of the wanted distinguishing context. Actually, this memory only contains the code that cooperates with the I/O device built in Algorithm 2 to mimic the target context and to differentiate the two modules at hand. Intuitively, the generated code communicates the state of the CPU to the I/O device enabling it to drive the context execution and thus the behavior of the processor.

Assume as given an *assembler* function *encode* that returns the encoding of any assembly instruction as one or two words – according to the size specified in Table 1. Also, assume that the unprotected memory is large enough to contain the code of the context we are building (there is no lack of generality since the space required for this code is bounded by a constant ($\leq$ 25 words) plus the number of different addresses to which the protected code jumps – kept anyway in the unprotected memory). Suppose also to have the five constants `A_HALT`, `A_LOOP`, `A_JIN`, `A_EP` and `A_RDIFF` representing addresses in the unprotected memory: they are assumed different from (i) each other, (ii) `0xFFFE` and (iii) any address $\mathcal{R}[\texttt{pc}]$ such that $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ occurs in either input distinguishing traces. Finally, assume for simplicity that the modules never jump to `0xFFFE`.[3]

First, the algorithm initializes the memory $\mathcal{M}_C$ by filling it with the code in Figure 10. It consists of five parts. The first two are for convergence (line 1) and divergence (line 3). The next part (lines 5 to 20) inputs the registers values from the device and then jumps into the enclave. Line 25 specifies that the first instruction to be executed is at the address specified by `A_EP`. Finally, the code in lines 22 and 23 interacts with the device to get the next instruction to execute.

Then, the algorithm inspects β and β', and generates a piece of code that orchestrates the interactions between the distinguishing context and the I/O device. Roughly, the generated code operates as follows: it (i) writes to the I/O device the different content of the register that distinguishes the traces; and (ii) reads from the I/O device a new value for the program counter (either `A_HALT` or `A_LOOP`), which causes the context to either diverge or terminate.

If they are both $\texttt{jmpOut!}(\cdot;\cdot)$ and at least one register has different values in the observables, two cases arise:

- If one of the registers differentiating β and β' is $\texttt{r} \neq \texttt{pc}$, then we store in $\mathcal{M}_C$ the instructions to ask the device a new program counter (that will depend on the value of r), starting at the address `A_RDIFF` (line 7). Note that in this case *joutd* and *joutd'* are left undefined;
- Otherwise, the register differentiating $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ and $\beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R}')$ is pc. In this case, we store in $\mathcal{M}_C$ the instructions to ask the device a new program counter at the address $\mathcal{R}[\texttt{pc}]$ and $\mathcal{R}'[\texttt{pc}]$ for the first and second module, respectively (lines 15 – 19). Also, we record the differentiating values of the program counter in *joutd* and *joutd'*, to be used by Algorithm 2.

[3] Slightly changing Algorithm 1 suffices to remove this limitation: upon the jump into protected mode right before jumping to `0xFFFE`, the context writes the right code to deal with it in `0xFFFE` and, afterwards, restores the old content of that address.

Finally, the algorithm adds the code to deal with jumps out from the protected module to unprotected code for any $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ in $\overline{\beta_s} \cdot \beta$ or $\overline{\beta_s} \cdot \beta'$ such that $\mathcal{R}[\texttt{pc}] \neq \mathit{joutd}$ and $\mathcal{R}[\texttt{pc}] \neq \mathit{joutd}'$. Since the code cannot track timing directly, we delegate the device to deal with the case when the observables differ on timings, i.e., when $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ and $\beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R})$ with $\Delta t \neq \Delta t'$ (see Algorithm 2). Eventually, the algorithm returns the memory built and the values of *joutd* and *joutd'* (if any), used by Algorithm 2 to build the distinguishing device.

Algorithm 1 Builds the memory of the distinguishing context.

```
procedure BuildMem($\overline{\beta} = \overline{\beta_s} \cdot \beta \cdot \overline{\beta_e}, \overline{\beta}' = \overline{\beta_s} \cdot \beta' \cdot \overline{\beta_e}'$)
        ▷ $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces w. common prefix $\overline{\beta_s}$
    $joutd = joutd' = \bot$
    $\mathcal{M}_C$ = filled as described in Figure 10
    if $\beta = \texttt{jmpOut!}(\Delta t;\mathcal{R}) \land \beta' = \texttt{jmpOut!}(\Delta t;\mathcal{R}') \land (\exists \texttt{r}. \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$ then
        if $\texttt{r} \neq \texttt{pc}$ then
            $\mathcal{M}_C = \mathcal{M}_C[\texttt{A\_RDIFF} \mapsto encode(\texttt{OUT r}), \texttt{A\_RDIFF}+1 \mapsto encode(\texttt{IN pc})]$
        else
            $joutd = \mathcal{R}[\texttt{pc}]$
            $joutd' = \mathcal{R}'[\texttt{pc}]$
            $\mathcal{M}_C = \mathcal{M}_C[joutd \mapsto encode(\texttt{OUT pc}), joutd+1 \mapsto encode(\texttt{IN pc})]$
            $\mathcal{M}_C = \mathcal{M}_C[joutd' \mapsto encode(\texttt{OUT pc}), joutd'+1 \mapsto encode(\texttt{IN pc})]$
        end if
    end if
    for $\texttt{jmpOut!}(\Delta t;\mathcal{R}) \in \overline{\beta_s} \cdot \beta, \overline{\beta_s} \cdot \beta'$ do
        if $\mathcal{R}[\texttt{pc}] \neq joutd \land \mathcal{R}[\texttt{pc}] \neq joutd'$ then
            $\mathcal{M}_C = \mathcal{M}_C[\mathcal{R}[\texttt{pc}] \mapsto encode(\texttt{IN pc})]$
        end if
    end for
    return $(\mathcal{M}_C, joutd, joutd')$
end procedure
```

Second algorithm: device construction. This second algorithm iteratively builds a device that cooperates with the memory of the context given by Algorithm 1 to distinguish $\mathcal{M}_M$ from $\mathcal{M}_{M'}$.[4] The algorithm is in the Appendix, and here we only briefly comment on it, for space reasons.

Let *joutd* and *joutd'* be the addresses returned by Algorithm 1 (if any) and that represent the differentiating values of the program counter; let $\overline{\beta} = \overline{\beta_s} \cdot \beta \cdot \overline{\beta_e}$ and $\overline{\beta}' = \overline{\beta_s} \cdot \beta' \cdot \overline{\beta_e}'$ $(\beta \neq \beta')$ be two distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ under C^L; finally, let *term* (resp. *term'*) denote whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode. The algorithm starts with an empty device and iterates over the observables β_i in $\overline{\beta_s}$:

- *Case* $\beta_i = \texttt{jmpIn?}(\mathcal{R})$.
  In this case either this is the first observable or $\beta_{i-1} = \texttt{jmpOut!}(\cdot;\cdot)$. According to Algorithm 1, in both cases we reach the instruction `IN pc` (either at address `A_EP` or those of jumps out of protected mode), waiting for the next program counter. The algorithm appends the behavior described in Figure 11a to the device built so far. Intuitively, the device ignores possible write

[4]The interactions between the module and the I/O device need not to be preserved because the module runs in protected mode and therefore it cannot access the I/O device in the first place.

```
A_HALT. HLT

A_LOOP. JMP pc

A_JIN . IN sp
      . IN sr
      . IN R3
      . IN R4
      . IN R5
      . IN R6
      . IN R7
      . IN R8
      . IN R9
      . IN R10
      . IN R11
      . IN R12
      . IN R13
      . IN R14
      . IN R15
      . IN pc

A_EP  . OUT pc
      . IN pc

0xFFFE. A_EP  ; the content of 0xFFFE is A_EP

```

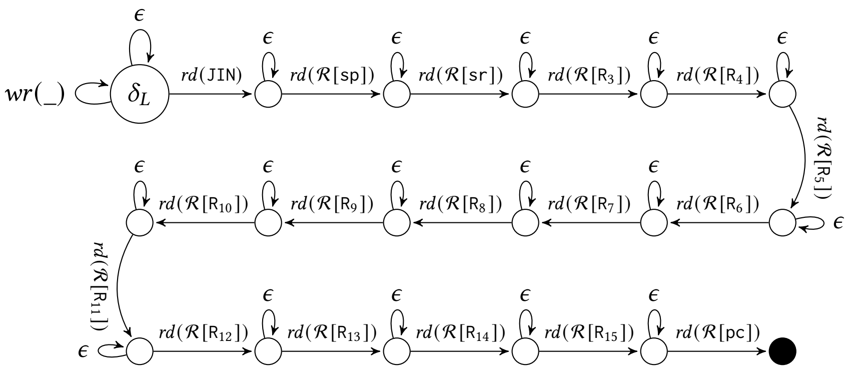
Fig. 10. Initial content of unprotected memory as used by Algorithm 1.

operations and outputs the special address A_JIN. Then, it starts sending the values of the registers in $\mathcal{R}$, so to simulate in Sancus^{H} what happens in Sancus^{L} and to match the code requests.

- *Case* $\beta_i = \texttt{jmpOut!}(\Delta t; \mathcal{R})$.
  The device is simply updated with an ϵ-loop on the last added state δ_L and ignores write operations (so as to deal with $\mathcal{R}[\texttt{pc}] = \mathit{joutd}$ or $\mathcal{R}[\texttt{pc}] = \mathit{joutd}'$). Figure 11b pictorially represents this case.

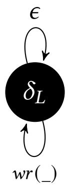
Then, as soon as β and β' show up, the algorithm sets up the device to differentiate the modules:

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \land \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}') \land (\exists \texttt{r}. \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$.
  In this case the differentiation is due to a register, and two further sub-cases may arise. If the register is pc then the device gets the differentiating value from the context (executing code at *joutd* and *joutd'* by construction); based on that value, it outputs either A_HALT or A_LOOP (see Figure 12a). For any other register than pc, the context waits for the next program counter and replies with the address A_RDIFF. This address points to the code that sends the differentiating register value and, based on that, the device replies with either A_HALT or A_LOOP (see Figure 12b).
- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \land \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}) \land \Delta t \neq \Delta t'$.
  Since different timings in Sancus^{L} correspond to different timings in Sancus^{H} (see Property A.22), we program the device so as to either reply with A_HALT or with A_LOOP depending on the time value (Figure 12c).
- *Case* $\beta = \bullet \land \beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$.
  In this case $\bullet$ may occur during an interrupt service routine. Two sub-cases may arise, depending on whether the first module $\mathcal{M}_M$ terminates or not when executed in a context with no interrupts after the last jump into protected mode. Note that the value of *term* differentiates the two sub-cases. When *term* holds, $\mathcal{M}_M$ causes a transition to an exception handling configuration from which there is a a jump to A_EP, and the device instructs the



ϵ
$wr(\_)$
δ_L
$rd(\mathtt{JIN})$
$rd(\mathcal{R}[\mathtt{sp}])$
$rd(\mathcal{R}[\mathtt{sr}])$
$rd(\mathcal{R}[\mathtt{R}_3])$
$rd(\mathcal{R}[\mathtt{R}_4])$
$rd(\mathcal{R}[\mathtt{R}_5])$
$rd(\mathcal{R}[\mathtt{R}_6])$
$rd(\mathcal{R}[\mathtt{R}_7])$
$rd(\mathcal{R}[\mathtt{R}_8])$
$rd(\mathcal{R}[\mathtt{R}_9])$
$rd(\mathcal{R}[\mathtt{R}_{10}])$
$rd(\mathcal{R}[\mathtt{R}_{11}])$
$rd(\mathcal{R}[\mathtt{R}_{12}])$
$rd(\mathcal{R}[\mathtt{R}_{13}])$
$rd(\mathcal{R}[\mathtt{R}_{14}])$
$rd(\mathcal{R}[\mathtt{R}_{15}])$
$rd(\mathcal{R}[\mathtt{pc}])$

(a) The case of $\beta_i = \beta'_i = \mathtt{jmpIn?}(\mathcal{R})$.



(b) The case of $\beta_i = \beta'_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R})$.

Fig. 11. A graphical representation of the algorithm building the I/O device for β_i and β'_i being in the longest common prefix. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle.

code to jump to `A_HALT`. Instead, the second module jumps to another location different from the distinguished address `A_EP`, thus a jump to `A_LOOP` occurs (Figure 12d). When *term* does not hold, $\mathcal{M}_M$ diverges and the second module makes the CPU jump to a location in unprotected code and the CPU is instructed to jump to `A_HALT` (Figure 12e).

- *Case* $\beta = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \varepsilon$.
  Analogous to the above, with *term'* replacing *term*.
- *Otherwise.* No other cases may arise (see Property A.21).

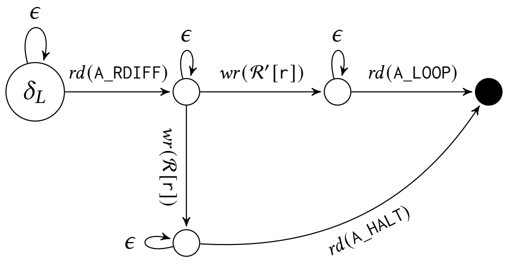
At the end, the algorithm returns a device built as just summarized.

The correctness of the two algorithms is established by the Properties A.21 and A.22 in the Appendix. The first states that, under the stated conditions, BuildDevice always produces an actual I/O device. The second property guarantees that the context built by joining together the results of the two algorithms is indeed a distinguishing one.

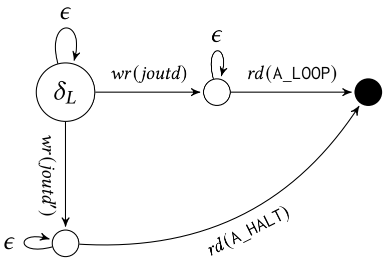
We finally prove that if two modules are contextually equivalent in Sancus$^{\mathrm{H}}$, then they are trace equivalent (implication (ii) in Figure 7).

Lemma 6.14. *If* $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$ *then* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$.

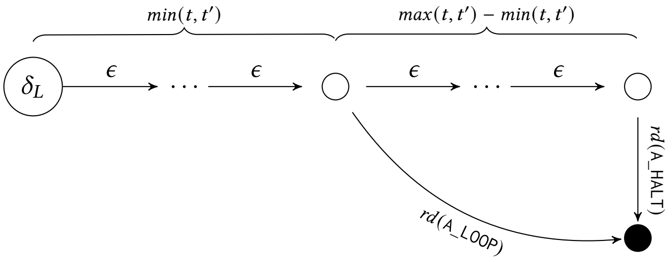
Proof. We prove the contrapositive: $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$ then $\mathcal{M}_M \not\simeq^{\mathrm{H}} \mathcal{M}_{M'}$. By Property 6.2, since $\overset{T}{\neq}$ there exists a pair of distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$. Algorithm 1 and 2 witness the existence of a context C^H that is an actual context and is guaranteed to differentiate $\mathcal{M}_M$ from $\mathcal{M}_{M'}$, i.e., $C^H[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$ and $C^H[\mathcal{M}_{M'}]\not\Downarrow^{\mathrm{H}}$ (or vice versa). Thus, by definition of contextually equivalent modules in Sancus$^{\mathrm{H}}$, we get $\mathcal{M}_M \not\simeq^{\mathrm{H}} \mathcal{M}_{M'}$ as requested. □



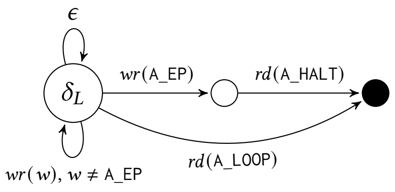
(a) The case of β_i = jmpOut!($\Delta t;\mathcal{R}$) $\wedge$ β'_i = jmpOut!($\Delta t';\mathcal{R}'$) $\wedge$ ($\exists$r $\neq$ pc. $\mathcal{R}$[r] $\neq$ $\mathcal{R}'$[r]).



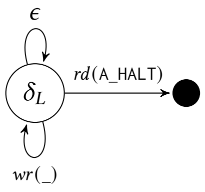
(b) The case of β_i = jmpOut!($\Delta t;\mathcal{R}$) $\wedge$ β'_i = jmpOut!($\Delta t';\mathcal{R}'$) $\wedge$ $\mathcal{R}$[pc] $\neq$ $\mathcal{R}'$[pc].



(c) The case of β_i = jmpOut!($\Delta t;\mathcal{R}$) $\wedge$ β'_i = jmpOut!($\Delta t';\mathcal{R}$) $\wedge$ $\Delta t \neq \Delta t'$. Let t and t' be the timing differences observed by the attacker at Sancus^L, see Algorithm 2 in the Appendix for details.



(d) The case of
$\beta_i = \bullet \wedge \beta'_i$ = jmpOut!($\Delta t;\mathcal{R}$)$\wedge$ $term$.



(e) The case of
$\beta_i = \bullet \wedge \beta'_i$ = jmpOut!($\Delta t;\mathcal{R}$) $\wedge \neg term$.

Fig. 12. A graphical representation of the algorithm building the I/O device for β_i and β'_i being the distinguishing observables. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle.

6.2.4 Preservation of behaviours. The last step of this long proof consists in stating and proving the lemma that guarantees preservation of behaviours, i.e., the implication (iii) in Figure 7:

Lemma 6.15 (Preservation).

$$\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ (\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}).$$

Proof. Just compose the implications (i) and (ii) of Figure 7 (i.e., Lemmata 6.14 and 6.12, resp.). □

7 PRESERVATION OF HYPERPROPERTIES

This section shows that our full abstraction result allows us to easily derive the preservation of some notions of *non-interference* and *hypersafety* when passing from Sancus$^{\mathrm{H}}$ to Sancus$^{\mathrm{L}}$. Since we are dealing with enclaves, the standard notions will be adapted to our framework.

From now onwards, we will use the following equivalence relation to express configurations that are equivalent from an attacker's point of view. According to this relation two configurations are equivalent if they cannot be distinguished by code running in unprotected mode (e.g., the contents of unprotected memory). In its definition we use the auxiliary equivalence of memories that holds when their public portions coincide.

Definition 7.1. Let c and c' be two configurations, and let $\mathcal{M} \overset{U}{=} \mathcal{M}'$ *iff* $\forall l \notin [ts, te) \cup [ds, de).\ \mathcal{M}[l] = \mathcal{M}'[l]$.

Then we define $c \overset{L}{=} c'$ *iff* $(c = c' = \mathrm{HALT}) \vee (c.\delta = c'.\delta\ \wedge\ c.t = c'.t\ \wedge\ c.t_a = c'.t_a\ \wedge\ c.\mathcal{M} \overset{U}{=} c'.\mathcal{M}\ \wedge\ c.\mathcal{R} = c'.\mathcal{R})$.

7.1 Take one: termination-insensitive, time-sensitive non-interference

We now tailor the notion of termination-insensitive, time-sensitive non-interference (inspired by [32]) to fit our framework. Roughly, two modules are non-interferent if and only if no context can distinguish them by examining the content of their public memories right before they terminate. Formally:

Definition 7.2. Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are termination-insensitive, time-sensitive non-interferent (*ISNI*) in Sancus$^{\mathrm{H}}$ (written $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$) *iff* for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \rightarrow \mathrm{HALT}\ \wedge\ \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \rightarrow^* c' \rightarrow \mathrm{HALT} \implies c \overset{L}{=} c'.$$

Similarly, we define *ISNI* in Sancus$^{\mathrm{L}}$, $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.

Commonly termination-insensitive non-interference is a property of a single program, to which our definition actually reduces when considering initial configurations as programs whose public input is a context and secret input is a module. Indeed this is a good model of what happens in reality: contexts are controlled by the attackers, whereas modules are securely deployed (i.e., we model the situation where both code and data are confidentially deployed, as can be done in Sancus 2.0 [45] and in Soteria [25]). Note in passing that our definition and results still hold if the code is made public before being loaded in the enclave (see also the discussion at the end of Section 7.2).

In the following, we clarify the relation between non-interference as defined in Definition 7.2 and our instance of full abstraction established in Theorem 6.3. Note that contextual equivalence requires *equi*convergence, and thus it is termination *sensitive*. On the contrary, *ISNI* is termination *insensitive* by definition and can relate modules that are not contextually equivalent. Therefore, preservation of termination-insensitive non-interference cannot be directly derived from full abstraction. As a matter of fact, requiring equiconvergence transforms Definition 7.2 in Definition 7.5 that introduces the more demanding notion of termination-sensitive non-interference, which *is* preserved (see below). Nevertheless, we can establish a couple of interesting properties. First, we relate contextual equivalence with non-interference in Sancus$^{\mathrm{L}}$:

LEMMA 7.3. *If* $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$ *then* $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.

From that it easily follows that non-interference in Sancus$^{\mathrm{L}}$ is guaranteed when two modules are contextual equivalent in Sancus$^{\mathrm{H}}$:

THEOREM 7.4. *If* $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$ *then* $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.

7.2 Take two: termination- and time-sensitive non-interference

In this section we consider a notion of non-interference inspired from [19] and distinguishes terminating modules from non-terminating ones. In the standard notion the program is public and the memory is split in a public and a secret segment: an attacker cannot discover any secret data by running the code with different public data. In our framework however also the code is protected, being hosted in the enclave. We first adapt the standard definition to our case, where the entire module is protected, and at the end of this section, we discuss how to recover the classic notion, where the code is public and some data secret.

Definition 7.5 (SSNI). Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are termination- and time-sensitive non-interferent (*SSNI*) in Sancus$^{\mathrm{H}}$ (written $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$) *iff* for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, and configurations c both implications hold:

- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \rightarrow \mathrm{HALT} \implies \exists c'. (\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \rightarrow^* c' \rightarrow \mathrm{HALT} \wedge c \overset{L}{=} c')$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \rightarrow^* c \rightarrow \mathrm{HALT} \implies \exists c'. (\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c' \rightarrow \mathrm{HALT} \wedge c \overset{L}{=} c')$

Similarly, we define *SSNI* in Sancus$^{\mathrm{L}}$, $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

The following theorem is easily established:

THEOREM 7.6.

(1) If $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$ *and* *(2) if* $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$

Thus, due to Theorem 6.3, the preservation of *SSNI* holds:

COROLLARY 7.7. $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'} \implies \mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

To recover the standard notion of termination- and time-sensitive non-interference, we only consider the code part of a module. In this way, we model the fact that the code needs not to be kept confidential, even though it is part of the enclave.

Recall the notion of layout $\mathcal{L} = \langle ts, te, ds, de, isr \rangle$, which is fixed in our model, and let $\mathcal{M}_P$ and $\mathcal{M}_D$ be two modules. By $\mathcal{M}_P \blacktriangleleft \mathcal{M}_D$ denote the module resulting from the composition of the code of $\mathcal{M}_P$ (called *module program*) and the data of $\mathcal{M}_D$ (called *module data*). Formally:

Definition 7.8. Given two modules $\mathcal{M}_P$ and $\mathcal{M}_D$, let

$$\mathcal{M}_P \blacktriangleleft \mathcal{M}_D \triangleq \lambda l. \begin{cases} \mathcal{M}_P(l) & \text{if } l \in [ts, te) \\ \mathcal{M}_D(l) & \text{if } l \in [ds, de) \end{cases}$$

Since the standard notion of termination- and time-sensitive non-interference only predicates on single programs. To recover it, we first tailor our *SSNI* to consider a single module program:

Definition 7.9 (Unary SSNI). A module program $\mathcal{M}_P$ is termination- and time-sensitive non-interferent (*USSNI*) in Sancus$^{\mathrm{H}}$ (written $\vdash^{\mathrm{H}}_{\mathrm{USSNI}} \mathcal{M}_P$) *iff* for all module data $\mathcal{M}_D, \mathcal{M}_{D'}$, for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, and for all configurations c:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]} \rightarrow^* c \rightarrow \mathrm{HALT} \implies \exists c'. (\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'}]} \rightarrow^* c' \rightarrow \mathrm{HALT} \ \wedge \ c \overset{L}{=} c').$$

Similarly, we define *USSNI* in Sancus$^{\mathrm{L}}$, $\vdash^{\mathrm{L}}_{\mathrm{USSNI}} \mathcal{M}_P$.

The following theorem then relates *USSNI* with our contextual equivalence, both in Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$:

Theorem 7.10. *Let $\mathcal{M}_P$ be a module program, then*
(1) if $\forall \mathcal{M}_D, \mathcal{M}_{D'}. (\mathcal{M}_P \blacktriangleleft \mathcal{M}_D) \simeq^{\mathrm{L}} (\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'})$, then $\vdash^{\mathrm{L}}_{\mathrm{USSNI}} \mathcal{M}_P$; and
(2) if $\vdash^{\mathrm{H}}_{\mathrm{USSNI}} \mathcal{M}_P$, then $\forall \mathcal{M}_D, \mathcal{M}_{D'}. (\mathcal{M}_P \blacktriangleleft \mathcal{M}_D) \simeq^{\mathrm{H}} (\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'})$.

Finally, the preservation of *USSNI* easily follows by Theorem 6.3:

Corollary 7.11. *If $\vdash^{\mathrm{H}}_{\mathrm{USSNI}} \mathcal{M}_P$, then $\vdash^{\mathrm{L}}_{\mathrm{USSNI}} \mathcal{M}_P$.*

As a final remark, note that both Corollary 7.7 and 7.11 hold under the hypothesis that Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$ are fully abstract.

7.3 Take three: stepwise termination- and time-sensitive non-interference

Since the attacker in our model is able to interrupt execution at *every* CPU cycle, one might wonder about the preservation of a stronger, *stepwise* notion of non-interference.

For that, we start from *SSNI* and introduce *stepwise termination- and time-sensitive non-interference.* It stipulates that two modules are non-interferent whenever their public memories are kept equivalent while stepping between successive unprotected configurations. For that, we first need the following definition:

Definition 7.12. $\mathcal{D} \vdash c \twoheadrightarrow_k c'$ *iff*

$$\mathcal{D} \vdash c \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \rightarrow c' \ \wedge \ c, c' \vdash_{mode} \mathsf{UM} \ \wedge \ \forall 1 \leq i \leq n.\ c_i \vdash_{mode} \mathsf{PM} \ \wedge \ k = \begin{cases} 0 & n = 0 \\ 2 & \text{o.w.} \end{cases}$$

Also, let $\mathcal{D} \vdash c_1 \twoheadrightarrow^t_K c_t$, where $K = \sum_{i=1}^{t} k_i$, be the shorthand for $\mathcal{D} \vdash c_1 \twoheadrightarrow_{k_1} \ldots \twoheadrightarrow_{k_t} c_t$. Similarly, we define $\mathcal{D} \vdash c \twoheadrightarrow_k c'$ and $\mathcal{D} \vdash c \twoheadrightarrow^t_K c'$.

Intuitively k counts the interactions between the context and the module ($k = 0$ if there are none and $k = 2$ if there is one entry and one exit) whereas the arrows $\twoheadrightarrow_k$ and $\twoheadrightarrow_k$ ignore all the steps taken in protected mode and just take into account the actions of the context.

We can now define the new notion of stepwise termination- and time-sensitive non-interference:

Definition 7.13. Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are stepwise termination- and time-sensitive non-interferent (*SSSNI*) in Sancus$^{\mathrm{H}}$ (written $\mathcal{M}_M \approx_{\mathrm{SSS}} \mathcal{M}_{M'}$) *iff* for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ both implications hold

- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c \implies \exists c'. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^t_K c' \ \wedge \ c \overset{L}{=} c'$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^t_K c' \implies \exists c. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c \ \wedge \ c \overset{L}{=} c'$

Similarly, we define *SSSNI* in Sancus$^{\mathrm{L}}$, $\mathcal{M}_M \approx_{\mathrm{SSS}} \mathcal{M}_{M'}$.

Since the arrows $\twoheadrightarrow^t_K$ and $\twoheadrightarrow^t_K$ are in a clear relation with $\overset{\overline{\beta}}{\Longrightarrow}$ (see Property A.24), we can prove the following results, leading to the preservation of *SSSNI*:

Lemma 7.14.

(1) if $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ and (2) if $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$

Thus, due to Theorem 6.3, the preservation of *SSSNI* holds:

Corollary 7.15. *If $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$.*

We note in passing that the same considerations made at the end of Section 7.2 suffice to show that our contextual-equivalence coincides with this notion of non-interference when the code and some data are deemed public.

7.4 Take five: hypersafety [5]

In this section we briefly sketch how to reduce our notion of full abstraction to the preservation of a much wider family of security properties than just non-interference, building on the work of Patrignani and Garg [50].

We first recall some notation. A compiler is seen in [50] as a mapping $[\![\cdot]\!]$ from source to target programs. Our compiler is actually the identity function, since any module $\mathcal{M}_M$ in Sancus$^{\mathsf{H}}$ is mapped into the *same* module $\mathcal{M}_M$ in Sancus$^{\mathsf{L}}$. Also, Patrignani and Garg [50], give the following notion of trace equivalence, which we call *whole program trace equivalence*:

Definition 7.16 (Definition 19 [50]). We say that

$$\mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'} \iff \forall C. WTr(C[\mathcal{M}_M]) = WTr(C[\mathcal{M}_{M'}]),$$

where $WTr(C[\mathcal{M}_M]) \triangleq \{\overline{\beta} \mid \exists c. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c\}$.

Contrary to our notion of Definition 6.7, their definition of trace equivalence requires the programs to produce the same set of traces under a *fixed context* (i.e., it is defined on whole programs).

Crucially, the following theorem links the notion of whole program trace equivalence with ours:

Theorem 7.17. *The following relations are equivalent:*

(1) $\mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$ (2) $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ (3) $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ (4) $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$

As a consequence of Theorem 7.17, our coarse-grained traces (Figure 9) are thus a fully abstract trace semantics for both Sancus$^{\mathsf{H}}$ and Sancus$^{\mathsf{L}}$, according Definition 20 of [50]:

Corollary 7.18.

(1) $\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$;

(2) $\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ \mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'} \iff \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$.

Due to this corollary, both **Assumption 1** (i.e., trace semantics of Sancus$^{\mathsf{H}}$ is fully abstract) and **Assumption 2** (i.e., trace semantics of Sancus$^{\mathsf{L}}$ is fully abstract) from [50] hold. Recall that the compiler from Sancus$^{\mathsf{H}}$ to Sancus$^{\mathsf{L}}$ implicitly used throughout the paper *is* an identity compiler (mapping a module in itself). By the assumption above, it also follows that Sancus$^{\mathsf{H}}$ and Sancus$^{\mathsf{L}}$ share the same set of fully abstract traces. Our identity compiler is therefore trivially *correct*, and it also is a *fail-safe-behavior compiler* – roughly, a compiler producing target programs that always halt after an invalid input (Definition 16 of [50]). We finally conclude that all the safety hyperproperties that hold for whole programs in Sancus$^{\mathsf{H}}$ also hold in Sancus$^{\mathsf{L}}$ (Theorems 10 and 6 of [50]).

[5]Not *four* as a homage to Dave Brubeck and his Quartet.

Note that the above is just a sketch of how one could prove the preservation of hypersafety following the approach of [50]. Indeed, a more formal and complete treatment of hyperproperty preservation would require traces to be the *ground truth* concerning what an attacker might observe. This might call for a complete reworking of the notion of traces in our setting, that are now a mere tool, beneficial to the proof. The same considerations hold for other principles of secure compilation based on (robust) hyperproperty preservation, such as those in [2, 3]; see also Section 9.1.

8 IMPLEMENTATION AND EVALUATION

We provide a full implementation[6] of our approach based on the Sancus [45] architecture which, in turn, is based on the openMSP430, an open source implementation of the TI MSP430 ISA. Our implementation has 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 $\mathcal{B}$.

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 of Sancus^{L}).

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 the rule (INT-PM-P) of Sancus^{L} 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 it is allowed to progress and start handling the IRQ.

Lastly, the third counter, $\mathtt{C}_{\mathtt{reti}}$, 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 $\mathcal{B}$ 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`.

Saving and restoring the 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 $\mathsf{R}_0, \ldots, \mathsf{R}_{15}$ (except 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 $\mathtt{C}_{\mathtt{reti}}$. 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 saves and restores 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 the protected memory area. Directly implementing this in hardware would increase the number of cycles needed to save and restore a state to one cycle per register. Of course,

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

one should make sure that this memory area is inaccessible from software by adapting the memory access control logic of the processor accordingly.

Evaluation. To evaluate the impact on the performance 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 above, the fixed overhead grows accordingly.

To evaluate the impact on hardware cost, we consider the scenario where the Sancus processor is synthesized on an FPGA, and we count the number of FPGA resources required for this synthesis. More specifically, we count the number of registers required (a measure for the amount of hardware state, i.e., flip-flops) and the number of lookup-tables (LUTs) required (a measure for the amount of hardware combinational logic).

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 registers and 2712 LUTs. Adding support for saving and restoring the processor state increases the area to 1488 registers and 2849 LUTs and the implementation of cycle padding further increases it to 1499 registers and 2854 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 8. The implementation of cycle padding, on the other hand, does not have a significant impact on the processor's area.

9 DISCUSSION

9.1 On the use of full abstraction as a security objective

The security guarantee that our approach offers is quite strong: an attack is possible in Sancus^{H} if and only if it is possible at Sancus^{L}. Since isolation is defined in term of contextual equivalence, full abstraction fits nicely in our setting, in that it ensures preservation and reflection of contextual equivalence.

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.

A property alternative to full abstraction would be to require (a non interactive version of) robust preservation of timing-sensitive non-interference [3]. This can also guarantee resistance against the example attacks in Section 3. However, this property 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}, provided that non-interference takes as secret the whole enclave, i.e., its memory, code, and initial state (see also the discussion in Section 7 about the role of full abstraction in preservation of non-interference).

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 [25]. 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 (see also the discussion at the end of Section 7.2).

9.2 The impact of our simplifications

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 of them are straightforward to remove. For instance, supporting more MSP430 instructions 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. We discuss the impact of this simplification in Section 9.3.

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 loading and initializing 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 rely 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 [59]. 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 runtime 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 deal with such a trusted runtime 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 [10]. A further alternative is considering the secure compartmentalizing compilation proposed by Juglaret et al. [31], who also use full abstraction to prove security.

Another 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).

9.3 Extending to more complex processors

Both our formal models, as well as the design and security proof of our interrupt padding countermeasure focus very much on enclaved execution on small microprocessors, like the Sancus system. An interesting question is to what extent the insights of our countermeasure design can be applied to more complex enclaved execution platforms like Intel SGX. While the designs of Intel SGX and Sancus are similar at a very high level, there are also major differences. The two most important differences that are not captured by our model are:

(1) The execution time of instructions on a high-end processor is not deterministic. The use of caches, and the use of processor optimization techniques like pipelining, speculative execution, micro-coding and so forth implies that the execution time of instructions can vary widely, both in terms of wall-clock time and in the number of processor cycles (or at least, modelling the processor with sufficient detail to make execution time deterministic would make the model *very* complex, as it would need to model state of the cache, the pipeline, the branch predictor, and so forth).
(2) These high-end processor optimizations also typically imply that attackers have many more ways to observe side effects of enclaved execution. In our model, the only thing a context (attacker) learns about enclaved execution is timing, either end-to-end timing of an enclave call or resume, or interrupt latency time. On higher-end enclaved execution systems, like Intel SGX, enclaved execution has other side effects visible to attackers, like the occurrence of page faults, or contention for other shared micro-architectural resources [60]. Such side effects could even be caused by *transient* enclaved execution, i.e., by instructions that are executed speculatively but never committed [14].

The first aspect, the fact that execution times are non-deterministic is to some extent a disadvantage for the attacker. Since interrupt latency attacks rely on measuring execution times, the fact that these measurements become non-deterministic will make it harder for the attacker to draw conclusions. However, even on high-end Intel x86 processors, it has been shown [56] that averaging interrupt latency measurements over multiple runs still leak significant information about the instruction being executed and about the micro-architectural state of the processor at the point of the interrupt. So it is just a matter for the attacker to improve his measurement techniques, and some form of padding on handling of (and resuming from) interrupts would still be useful. On the one hand, the non-deterministic nature of instruction execution time makes it hard to choose a good value for `MAX_TIME`. Especially since the *worst-case* execution time on a complex processor can be quite high, choosing `MAX_TIME` to be higher than any possible instruction may be prohibitively expensive.[7] On the other hand, it might be fine to choose `MAX_TIME` to be smaller than the actual worst-case longest instruction execution time. In this case, one can think of the choice of `MAX_TIME` as a trade-off between performance and security against the leakage through interrupt latency. The higher `MAX_TIME`, the less an attacker can learn from a specific interrupt latency measurement: only in the (presumably very few) cases where interrupt latency exceeds `MAX_TIME` the attacker does learn something. However, in cases where the attacker can influence the execution time of instructions (for instance, by flushing caches), the precise security gains are hard to estimate.

Alternatively, one could consider adding *random* padding to interrupt handling and resume, together with measures to make it impossible for the attacker to execute the same measurement many times (thus making it impossible to do statistical analysis).

Our current formal model and proof obviously do not apply to these more complex settings, and it is unclear whether the strong guarantees that full abstraction provides are compatible with the pragmatic or heuristic solutions suggested above.

The second aspect, the existence of other side effects visible to attackers of the enclaved execution, is even trickier. For instance, by spying on page table accesses [12, 42, 60] or via cache-based side channels [23], an attacker can reliably observe enclave memory accesses at some granularity. This is an important disadvantage for the defender, as it is less obvious that a padding countermeasure would provide a substantial benefit in the presence of such observable side effects. For instance, if the attacker can distinguish padding from regular instructions by observing side effects, the countermeasure becomes useless. Making such a distinction could for instance be done by monitoring accesses to code memory: if the processor is just padding, no instruction load needs to happen. So an implementation of our countermeasure on a complex processor would have to make sure that padding is not distinguishable from instruction execution through any kind of side effect that instructions might have, which we consider to be a significant challenge. From a security point of view, the ideal scenario would be to remove all the possible side effects through which enclaved executions leak information. However, just like the end-to-end timing side channel, closing other side channels completely will likely be too expensive, if not entirely impossible. So instead, the question is whether we can *accept* some bounded side-channel leakage, i.e., leave it up to the software developer to deal with the remaining channels (for instance by means of orthogonal defenses or by using some form of constant-time programming, like we did for the end-to-end timing channel), but at the same time guarantee that the power to precisely schedule and handle interrupts does not *increase* the power of these attacks. For instance, we might accept the fact that memory accesses leak at the granularity of pages, while at the same time making sure that the precision or bandwidth of that channel does not get amplified for interrupt adversaries, as has been shown repeatedly in the case of Intel SGX [11, 42]. It is again an open question for future work whether this could be formulated usefully as a full abstraction theorem, where we model the side channels that we accept in the high model, similarly to how we modeled end-to-end timing attacks in the high model in this paper.

[7]To give an idea of the complexity involved, consider that on modern Intel x86 processors with hardware virtualization extensions, a single address translation can in itself already require up to 20 memory accesses, which in the worst-case would all miss the cache hierarchy and have to be served from slow DRAM memory [17].

10 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. [23] for timing attacks, Gruss' PhD thesis [26] for software-based micro-architectural attacks before Spectre/Meltdown, Canella et al. [14] for transient execution based attacks, and Van Bulck's PhD thesis [55] for Intel SGX attacks. The attacks most relevant to this paper are the pure interrupt-based attacks. Van Bulck et al. [56] 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. [29] developed a similar attack for Intel SGX.

There is an extensive body of work on defenses against software-based side-channel attacks. The four surveys mentioned above ([14, 23, 26, 55]) 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 Clercq 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 [62] 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. [61] 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 [46]. 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 [1], and has been applied in many settings, including compilation to JavaScript [22], various intermediate compiler passes [5, 6], and compilation to platforms that support enclaved execution [4, 47, 49]. 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. [48] have provided a good survey of this entire line of work on secure compilation.

Still higher up in the computational stack, Tomé Cortiñas et al. [16] extended the MAC library [58] – a Haskell *information-flow control* library – with asynchronous exceptions. Akin to interrupts in our setting, asynchronous exceptions can be raised at any time and may break security properties of the running code. To ensure that this never happens, they introduced a variant of non-interference and proved that it is satisfied by their extension of the MAC library.

Other authors applied secure compilation techniques to prove security against side-channel attacks. For instance, Barthe et al. [7] proved that a suitably modified version of the CompCert compiler [37] preserves the constant-time policy. For that, they identified the passes of CompCert that did not preserve constant-time and modified them accordingly; afterwards, they proved them to be constant-time preserving using variants of the proof techniques proposed in [8]. Very recently, Patrignani and Guarnieri [51] proved secure a couple of mitigations against Spectre v1 [34] by specializing hyperproperty preservation principles of [3] to preserve *speculative non-interference* [28].

One could consider the addition of speculation and out-of-order execution as a new processor feature, similar to how our work considers extending a processor with the feature of interrupts. It would be reasonable to investigate under what conditions this new feature does not introduce new information leaks. To apply our approach to this problem seems to require a relatively precise model of how these features work. Existing work on dealing with speculative leaks using programming language techniques instead works with more abstract models of speculation. For example, Spectector [28] is a symbolic execution tool that analyses `x86_64` assembly programs and detects the presence of possible speculative leaks or proves their absence. Guanciale et al. [27] present a formal model capturing out-of-order execution and speculation in single core processors. Using this model they discover three new (possible) vulnerabilities and assess the security of existing countermeasures. Vassena et al. [57] define a static type system that labels each expression of their language as either *transient* or *stable* (i.e., that may include transient values or not, respectively). Crucially, their type system rejects programs that possibly contain speculative leaks. Also, they introduce the **protect** construct that ensures that assignments containing it are performed only once their right-hand side is stable. Furthermore, the same paper proposes an algorithm that automatically synthesizes the minimal number of **protect**s to be inserted in given program to fix all the potential speculative leaks.

11 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. More precisely, we advocate full abstraction as a formal criterion of what it means to maintain the security of isolation mechanisms under processor extensions. We have applied our approach to an IoT-scale microprocessor: first we have designed an extension of Sancus with interruptible enclaves (Sancus^{L}) and then we have proved it fully abstract with respect to the original Sancus without them (Sancus^{H}). Remarkably, the full abstraction proof relies on the strong power of our attacker that controls the unprotected memory, which is limited to 64 KB, and the I/O device which instead has unlimited memory. Indeed, the backtranslation encodes the attack logic within the I/O device that then drives a fixed piece of code in unprotected memory, namely the software component of the attacker.

To further assess our full abstraction-based security criterion we have compared its guarantees with those of some notions of non-interference preservation presented in the literature: we have proved that they are implied by our full abstraction theorem. We have also outlined how to prove that our results preserve hyperproperties, thus ensuring that modules executed in interruptible Sancus enjoy the same hyperproperties as they would when executed by the uninterruptible one.

Despite this successful case study, some limitations of the approach remain. A first challenging issue to be addressed in the future concerns the formal treatment of the extensions discussed Section 9.3. As a matter of fact, our model and full abstraction result seem to be a good starting point, although they currently apply only to "small" microprocessors for which we can define a cycle-accurate operational semantics. While this obviously makes it possible to rigorously reason about timing-based side channels, scaling our approach to larger processors is however not trivial. Indeed, to handle larger processors, we need models that can abstract away many details of the processor implementation, yet keeping enough details to model micro-architectural attacks of interest. A promising example of model with such features was proposed by Disselkoen et al. [20] that could replace our cycle-accurate model.

In our proposal, the security criterion is binary: an extension is either secure, or it is not. Therefore *low bandwidth* side channels are not kept apart from *high-bandwidth* side channels. An important challenge for future work is to introduce some kind of *measure* on the weakening of security, so as to allow security policies that consider some bounded amount of leakage acceptable.

ACKNOWLEDGMENTS

The authors warmly thank the anonymous referees for their extremely careful comments and helpful suggestions.

Matteo Busi has been partially supported by the research grant on *Incremental type systems for secure compilation* from the Department of Computer Science of the University of Pisa. 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). Pierpaolo Degano has been partially supported by the MIUR project PRIN 2017FTXR7S *IT MATTERS* (*Methods and Tools for Trustworthy Smart Systems*). 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.

REFERENCES

[1] Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects (Lecture Notes in Computer Science)*, Jan Vitek and Christian Damsgaard Jensen (Eds.), Vol. 1603. Springer, 19–34.

[2] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Springer, Dublin, Ireland, 1–28. https://doi.org/10.1007/978-3-030-44914-8_1

[3] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. 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*. IEEE, Hoboken, NJ, USA, 256–271.

[4] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, Stephen Chong (Ed.). IEEE Computer Society, Cambridge, MA, USA, 171–185.

[5] Amal Ahmed and Matthias Blume. 2008. 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*. Association for Computing Machinery, Victoria, BC, Canada, 157–168.

[6] Amal Ahmed and Matthias Blume. 2011. 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*. Association for Computing Machinery, Tokyo, Japan, 431–444.

[7] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[8] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE, Oxford, United Kingdom, 328–343.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. 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.*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, Baltimore, MD, USA, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[10] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. 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*. Association for Computing Machinery, London, UK, 1741–1758. https://doi.org/10.1145/3319535.3363206

[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. 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, Shanghai, China, 4:1–4:6. https://doi.org/10.1145/3152701.3152706

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. 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.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, Vancouver, BC, Canada, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[13] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2020. Provably Secure Isolation for interruptible Enclaved Execution on Small Microprocessors. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, Boston, MA, USA, 262–276.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, Santa Clara, CA, USA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Stockholm, Sweden, 142–157.

[16] Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo. 2020. Securing Asynchronous Exceptions. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, Boston, MA, USA, 214–229. https://doi.org/10.1109/CSF49147.2020.00023

[17] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86. http://eprint.iacr.org/2016/086

[18] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. 2014. 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, Zurich, Switzerland, 147–152. https://doi.org/10.1109/ASAP.2014.6868649

[19] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, Berleley/Oakland, California, USA, 109–124. https://doi.org/10.1109/SP.2010.15

[20] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, San Francisco, CA, USA, 1238–1255. https://doi.org/10.1109/SP.2019.00047

[21] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. 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, Shanghai, China, 287–305. https://doi.org/10.1145/3132747.3132782

[22] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. 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*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, Rome, Italy, 371–384.

[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering* 8, 1 (2018), 1–27. https://doi.org/10.1007/s13389-016-0141-6

[24] Travis Goodspeed. 2008. Practical attacks against the MSP430 BSL. In *Twenty-Fifth Chaos Communications Congress*. Verlag Art d'Ameublement, Bielefeld, Berlin, Germany, 6.

[25] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. 2015. Soteria: Offline Software Protection Within Low-cost Embedded Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference* (Los Angeles, CA, USA) *(ACSAC 2015)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2818000.2856129

[26] Daniel Gruss. 2017. *Software-based Microarchitectural Attacks*. Ph.D. Dissertation. Graz University of Technology.

[27] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. IEEE, Virtual Event, USA, 1853–1869. https://doi.org/10.1145/3372297.3417246

[28] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, San Francisco, CA, USA, 1–19. https://doi.org/10.1109/SP40000.2020.00011

[29] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. 2018. 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, Orlando, FL, USA, 108–114. https://doi.org/10.1109/ICCD.2018.00025

[30] Texas Instruments. 2016. MSP430x1xx Family: User Guide. http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[31] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE, Lisbon, Portugal, 45–60. https://doi.org/10.1109/CSF.2016.11

[32] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, Berkeley, California, USA, 413–428. https://doi.org/10.1109/SP.2011.19

[33] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. 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, Minneapolis, MN, USA, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, San Francisco, CA, USA, 1–19. https://doi.org/10.1109/SP.2019.00002

[35] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: a security architecture for tiny embedded devices. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel (Eds.). ACM, Amsterdam, The Netherlands, 10:1–10:14. https://doi.org/10.1145/2592798.2592824

[36] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. 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.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, Vancouver, BC, Canada, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[37] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, Baltimore, MD, USA, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[39] Nancy A. Lynch and Mark R. Tuttle. 1989. An introduction to input/output automata. *CWI Quarterly* 2 (1989), 219–246.

[40] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. 2010. 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, Berleley/Oakland, California, USA, 143–158. https://doi.org/10.1109/SP.2010.17

[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. 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*, Ruby B. Lee and Weidong Shi (Eds.). ACM, Tel-Aviv, Israel, 10. https://doi.org/10.1145/2487726.2488368

[42] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA, 469 – 486.

[43] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, Virtual event, USA, 1466–1482.

[44] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. 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*, Samuel T. King (Ed.). USENIX Association, Washington, DC, USA, 479–494. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman

[45] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* 20, 3, Article 7 (July 2017), 33 pages. https://doi.org/10.1145/3079763

[46] Ivan Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium, USENIX Security 2019*. USENIX Association, Santa Clara, CA, USA, 1429–1446.

[47] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 6:1–6:50.

[48] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (2019), 36 pages. https://doi.org/10.1145/3280984

[49] Marco Patrignani and Dave Clarke. 2015. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures* 42 (2015), 22–45.

[50] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, Santa Barbara, CA, USA, 392–404.

[51] Marco Patrignani and Marco Guarnieri. 2019. Exorcising Spectres with Secure Compilers. *CoRR* abs/1910.08607 (2019), 82. arXiv:1910.08607 http://arxiv.org/abs/1910.08607

[52] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. *CoRR* abs/1902.03256 (2019), 20. arXiv:1902.03256 http://arxiv.org/abs/1902.03256

[53] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Bonn, Germany, 3–24.

[54] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, Vancouver, BC, Canada, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[55] Jo Van Bulck. 2020. *Microarchitectural Side-Channel Attacks for Privileged Software Adversaries*. Ph.D. Dissertation. KU Leuven, Leuven, Belgium. https://lirias.kuleuven.be/3047121

[56] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 178–195. https://doi.org/10.1145/3243734.3243822

[57] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proc. ACM Program. Lang.* 5, POPL, Article 49 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434330

[58] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. Mac a verified static information-flow control library. *Journal of logical and algebraic methods in programming* 95 (2018), 148–180.

[59] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. 2016. 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*. Springer, Heraklion, Greece, 440–457. https://doi.org/10.1007/978-3-319-45744-4_22

[60] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. 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, San Jose, CA, USA, 640–656. https://doi.org/10.1109/SP.2015.45

[61] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. 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*. IEEE, Hoboken, NJ, USA, 272–287. https://doi.org/10.1109/CSF.2019.00026

[62] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. 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*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, Istanbul, Turkey, 503–516. https://doi.org/10.1145/2694344.2694372

A ADDITIONAL DEFINITIONS AND RESULTS

A.1 The device of Section 4.6.1 is deterministic

Property A.1. *If* $\mathcal{D} \vdash \delta, t, t_a \curvearrowright^{k}_{D} \delta', t', t'_a$ *and* $\mathcal{D} \vdash \delta, t, t_a \curvearrowright^{k}_{D} \delta'', t'', t''_a$*, then* $\delta' = \delta''$, $t' = t''$ *and* $t'_a = t''_a$.

Proof. Trivial. □

A.2 Complete operational semantics rules of Sancus^H (Section 5.1)

INT

$$\frac{}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \hookrightarrow_{I} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle}$$

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \vdash_{mode} \mathtt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{HLT}$$

(CPU-NoIN)

$$\frac{\delta \not\curvearrowright^{rd(w)}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{IN\ r}$$

(CPU-NoOUT)

$$\frac{\delta \not\curvearrowright^{wr(\mathcal{R}[\mathtt{r}])}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{OUT\ r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \vdash_{mode} \mathtt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle}} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{HLT}$$

(CPU-Decode-Fail)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle}}$$

(CPU-Violation-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \nvdash_{mac} \mathtt{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \text{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle}} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) \neq \bot$$

(CPU-MovL)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{M}[\mathcal{R}[\mathtt{r}_1]]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright^{cycles(i)}_{D} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B}\rangle \hookrightarrow_{I} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV\ @r_1\ r_2}$$

(CPU-MovS)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 4] \\ \mathcal{M}' = \mathcal{M}[\mathcal{R}[\mathtt{r}_2] \mapsto \mathcal{R}[\mathtt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright^{cycles(i)}_{D} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B}\rangle \hookrightarrow_{I} \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV\ r_1\ 0(r_2)}$$

(CPU-Mov)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright^{cycles(i)}_{D} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B}\rangle \hookrightarrow_{I} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV\ r_1\ r_2}$$

Fig. 13. Rules of the main transition system for Sancus^H incl. interrupt logic. (part I)

(CPU-MovI)

$$\dfrac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 4][\mathtt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV}\ \#w\ \mathtt{r}$$

(CPU-Cmp) $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{CMP}\ \mathtt{r}_1\ \mathtt{r}_2$

$$\dfrac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r}_2] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r}_2] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r}_2] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}$$

(CPU-Nop)

$$\dfrac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{NOP}$$

(CPU-Reti-Chain)

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

(CPU-Reti-PrePad)

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

(CPU-Reti-Pad)

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

(CPU-Reti)

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

Fig. 14. Rules of the main transition system for Sancus$^{\mathrm{H}}$ incl. interrupt logic. (part II)

(CPU-Jz0)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \wedge \mathcal{R}[\mathsf{sr}].Z = 0$$

(CPU-Jz1)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \wedge \mathcal{R}[\mathsf{sr}].Z = 1$$

(CPU-Jmp)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JMP}\ \&\mathsf{r}$$

(CPU-In)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{IN}\ \mathsf{r}$$

(CPU-Out)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \delta \overset{wr(\mathcal{R}[\mathsf{r}])}{\rightsquigarrow}_D \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{OUT}\ \mathsf{r}$$

(CPU-Not)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r} \mapsto \neg\mathcal{R}[\mathsf{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{NOT}\ \mathsf{r}$$

(CPU-And)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r}_2 \mapsto \mathcal{R}[\mathsf{r}_1] \& \mathcal{R}[\mathsf{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathsf{sr.N} \mapsto \mathcal{R}'[\mathsf{r}_2]\&\mathtt{0x8000}, \mathsf{sr.Z} \mapsto (\mathcal{R}'[\mathsf{r}_2] == 0), \mathsf{sr.C} \mapsto (\mathcal{R}'[\mathsf{r}_2] \neq 0), \mathsf{sr.V} \mapsto 0] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{AND}\ \mathsf{r}_1\ \mathsf{r}_2$$

(CPU-Add) $i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{ADD}\ \mathsf{r}_1\ \mathsf{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r}_2 \mapsto \mathcal{R}[\mathsf{r}_1] + \mathcal{R}[\mathsf{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathsf{sr.N} \mapsto (\mathcal{R}'[\mathsf{r}_2] < 0), \mathsf{sr.Z} \mapsto (\mathcal{R}'[\mathsf{r}_2] == 0), \mathsf{sr.C} \mapsto (\mathcal{R}'[\mathsf{r}_2] \neq 0), \mathsf{sr.V} \mapsto overflow(\mathcal{R}[\mathsf{r}_1] + \mathcal{R}[\mathsf{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle}$$

(CPU-Sub) $i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{SUB}\ \mathsf{r}_1\ \mathsf{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r}_2 \mapsto \mathcal{R}[\mathsf{r}_1] - \mathcal{R}[\mathsf{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathsf{sr.N} \mapsto (\mathcal{R}'[\mathsf{r}_2] < 0), \mathsf{sr.Z} \mapsto (\mathcal{R}'[\mathsf{r}_2] == 0), \mathsf{sr.C} \mapsto (\mathcal{R}'[\mathsf{r}_2] \neq 0), \mathsf{sr.V} \mapsto overflow(\mathcal{R}[\mathsf{r}_1] - \mathcal{R}[\mathsf{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle}$$

Fig. 15. Rules of the main transition system for Sancus$^{\mathrm{H}}$ incl. interrupt logic. (part III)

A.3 Complete operational semantics rules of Sancus$^{\mathrm{L}}$ (Section 5.2)

(INT-UM-P)

$$\frac{\begin{array}{c} pc_{old} \vdash_{mode} \mathsf{UM} \quad \mathcal{R}[\mathsf{sr}].\mathsf{GIE} = 1 \quad t_a \neq \bot \quad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto isr, \mathsf{sr} \mapsto 0, \mathsf{sp} \mapsto \mathcal{R}[\mathsf{sp}] - 4] \\ \mathcal{M}' = \mathcal{M}[\mathcal{R}[\mathsf{sp}] - 2 \mapsto \mathcal{R}[\mathsf{pc}], \mathcal{R}[\mathsf{sp}] - 4 \mapsto \mathcal{R}[\mathsf{sr}]] \quad \mathcal{D} \vdash \delta, t, \bot \curvearrowright^{6}_{D} \delta', t', t'_a \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc_{old}, \mathcal{B} \rangle}$$

(INT-UM-NP)

$$\frac{pc_{old} \vdash_{mode} \mathsf{UM} \quad (\mathcal{R}[\mathsf{sr}].\mathsf{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(INT-PM-P)

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

(INT-PM-NP)

$$\frac{pc_{old} \vdash_{mode} \mathsf{PM} \quad (\mathcal{R}[\mathsf{sr}].\mathsf{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathsf{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{HLT}$$

(CPU-NoIN)

$$\frac{\delta \not\xrightarrow{rd(w)}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{IN}\ \mathsf{r}$$

(CPU-NoOUT)

$$\frac{\delta \not\xrightarrow{wr(\mathcal{R}[\mathsf{r}])}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \quad decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{OUT}\ \mathsf{r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathsf{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t + cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{HLT}$$

(CPU-Decode-Fail)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}}$$

(CPU-Violation-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad i, \mathcal{R}, pc_{old}, \mathcal{B} \nvdash_{mac} \mathsf{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t + cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) \neq \bot$$

(CPU-MovL)

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

(CPU-MovS)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \quad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 4] \\ \mathcal{M}' = \mathcal{M}[\mathcal{R}[\mathsf{r}_2] \mapsto \mathcal{R}[\mathsf{r}_1]] \quad \mathcal{D} \vdash \delta, t, t_a \curvearrowright^{cycles(i)}_{D} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \quad i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{MOV}\ \mathsf{r}_1\ \mathsf{0}(\mathsf{r}_2)$$

Fig. 16. Rules of the main transition system for Sancus$^{\mathrm{L}}$ incl. interrupt logic. (part I)

(CPU-Mov)

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

(CPU-MovI)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 4][\mathsf{r} \mapsto w] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{MOV}\ \#w\ \mathsf{r}$$

(CPU-Cmp) $i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{CMP}\ \mathsf{r}_1\ \mathsf{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2][\mathsf{r}_2 \mapsto \mathcal{R}[\mathsf{r}_1] - \mathcal{R}[\mathsf{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathsf{sr.N} \mapsto (\mathcal{R}'[\mathsf{r}_2] < 0), \mathsf{sr.Z} \mapsto (\mathcal{R}'[\mathsf{r}_2] == 0), \mathsf{sr.C} \mapsto (\mathcal{R}'[\mathsf{r}_2] \neq 0), \mathsf{sr.V} \mapsto overflow(\mathcal{R}[\mathsf{r}_1] - \mathcal{R}[\mathsf{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle}$$

(CPU-Nop)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{NOP}$$

(CPU-Reti-Chain)

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

(CPU-Reti-PrePad)

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

(CPU-Reti-Pad)

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

(CPU-Reti)

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

(CPU-Jz0)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \wedge \mathcal{R}[\mathsf{sr}].Z = 0$$

(CPU-Jz1)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathsf{OK} \\ \mathcal{R}' = \mathcal{R}[\mathsf{pc} \mapsto \mathcal{R}[\mathsf{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathsf{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathsf{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathsf{pc}]) = \mathsf{JZ}\ \&\mathsf{r} \wedge \mathcal{R}[\mathsf{sr}].Z = 1$$

Fig. 17. Rules of the main transition system for Sancus$^{\mathrm{L}}$ incl. interrupt logic. (part II)

(CPU-Jmp)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{JMP\ \&r}$$

(CPU-In)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{IN\ r}$$

(CPU-Out)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2] \qquad \delta \overset{wr(\mathcal{R}[\mathtt{r}])}{\rightsquigarrow}_D \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)-1} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{OUT\ r}$$

(CPU-Not)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r} \mapsto \neg\mathcal{R}[\mathtt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{NOT\ r}$$

(CPU-And)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r_2} \mapsto \mathcal{R}[\mathtt{r_1}]\&\mathcal{R}[\mathtt{r_2}]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto \mathcal{R}'[\mathtt{r_2}]\&\mathtt{0x8000}, \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r_2}] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r_2}] \neq 0), \mathtt{sr.V} \mapsto 0] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{AND\ r_1\ r_2}$$

(CPU-Add) $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{ADD\ r_1\ r_2}$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r_2} \mapsto \mathcal{R}[\mathtt{r_1}]+\mathcal{R}[\mathtt{r_2}]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r_2}] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r_2}] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r_2}] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r_1}]+\mathcal{R}[\mathtt{r_2}])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}$$

(CPU-Sub) $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{SUB\ r_1\ r_2}$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r_2} \mapsto \mathcal{R}[\mathtt{r_1}]-\mathcal{R}[\mathtt{r_2}]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r_2}] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r_2}] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r_2}] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r_1}]-\mathcal{R}[\mathtt{r_2}])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}$$

Fig. 18. Rules of the main transition system for Sancus$^{\mathrm{L}}$ incl. interrupt logic. (part III)

A.4 Proof of progress of Section 5.3

Theorem 5.1 (Progress). *For all $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, $\mathcal{M}_M$ and configuration c*

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \mathrm{HALT} \qquad and \qquad \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \mathrm{HALT}.$$

Proof. Since no conclusion of the Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$ semantic rules has HALT as starting configuration, this distinguished configuration is trivially stuck.

Also, HALT is the only stuck configuration because any configuration $c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \neq$ HALT can progress. We show this for Sancus$^{\mathrm{H}}$; for Sancus$^{\mathrm{L}}$ just substitute $\rightarrow$ for $\rightarrow$.

If $\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle$, the following three cases arise:

(1) If $decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \bot$, then (CPU-Decode-Fail) applies.

(2) If $decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) \neq \bot \wedge i, \mathcal{R}, pc_{old}, \mathcal{B} \nvdash_{mac} \mathtt{OK}$, then (CPU-Violation-PM) applies.
(3) If the device is not willing to synchronize with the CPU, either rule (CPU-NoIN) or rule (CPU-NoOUT) applies.
(4) Otherwise, there is a rule for each $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}])$ leading to a target configuration. Indeed, all the cases that may arise are covered by the premises that:
   - check well-formedness of i and non-violation of MAC; and
   - are all mutually exclusive (e.g., $\mathcal{B} \neq \bot$ in (CPU-Reti-Chain) and (CPU-Reti-PrePad) is dealt with in rule (CPU-Reti) or the requirements of the values of $\mathcal{R}[\mathtt{sr.GIE}]$ and t'_a in (CPU-Reti-Chain) appear negated in (CPU-Reti-PrePad)); and
   - require the existence of values either built explicitly (e.g., the value of $\mathtt{sr.N}$ in (CPU-And)) or through relations that are always defined (e.g., through the transition system for interrupts).

Otherwise, $\mathcal{B} = \langle \bot, \bot, t_{pad} \rangle$ and the rule (CPU-Reti-Pad) applies. □

A.5 Proofs and additional definition for Section 6.1

Lemma 6.5. *For any module $\mathcal{M}_M$, context C, and corresponding interrupt-less context C_I:*

$$C_I[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$$

Proof. By definition of $\mathcal{D} \vdash \cdot \curvearrowright_D^k \cdot$, the value t_a in the CPU configuration (that signals the presence of an unhandled interrupt) is changed only when an interrupt has been raised since the last time it was checked.

Since any $int?$ action has been substituted with an ϵ, t_a is never changed from its initial $\bot$ value.

Since the only difference in behavior between the two levels is in the interrupt logic, and since the ISR in C_I is never invoked (thus, it does not affect the program behavior), $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathrm{I}} \cdot$ behaves exactly as $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathrm{I}} \cdot$. So, $C_I[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ implies $C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$ and vice versa. □

Lemma 6.6 (Reflection). $\forall \mathcal{M}_M, \mathcal{M}_{M'}. (\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'})$.

Proof. We can expand the hypothesis using the definition of $\simeq^{\mathrm{L}}$ and $\simeq^{\mathrm{H}}$ as follows:

$$(\forall C.\, C[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathrm{L}}) \implies (\forall C'.\, C'[\mathcal{M}_M]\Downarrow^{\mathrm{H}} \iff C'[\mathcal{M}_{M'}]\Downarrow^{\mathrm{H}}).$$

For any C' we can build the corresponding interrupt-less context C'_I.
Since interrupt-less contexts are a (strict) subset of all the contexts, by hypothesis:

$$C'_I[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C'_I[\mathcal{M}_{M'}]\Downarrow^{\mathrm{L}}.$$

But from Lemma 6.5 it follows that

$$C'[\mathcal{M}_M]\Downarrow^{\mathrm{H}} \iff C'[\mathcal{M}_{M'}]\Downarrow^{\mathrm{H}}.$$

□

Definition A.1 (Complete interrupt segments). Let $\overline{\alpha} = \alpha_0 \cdots \alpha_n$ be a fine-grained trace. The set $\mathbb{I}_{\overline{\alpha}}$ of *complete interrupt segments* of $\overline{\alpha}$ is defined as follows:

$$\mathbb{I}_{\overline{\alpha}} \triangleq \{(i,j) \mid \alpha_i = \mathtt{handle!}(k) \;\wedge\; \alpha_j = \mathtt{reti?}(k') \;\wedge\; i < j \;\wedge\; \forall i < l < j.\, \alpha_l = \xi\}.$$

A.6 Preliminary definitions and proofs for Lemmata 6.9 and 6.10

Roughly, we define two configurations be P-equivalent (U-equivalent, resp.) if they cannot be kept apart by looking at those parts that can be inspected when the CPU is operating in protected (unprotected, resp.) mode.

Definition A.2. We say that two configurations are *P-equivalent* (written $c \overset{P}{\approx} c'$) *iff*

$$(c = c' = \text{HALT}) \vee$$

$$(c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \ \wedge \ c' = \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}'\rangle \ \wedge \ \mathcal{M} \overset{P}{=} \mathcal{M}' \ \wedge$$

$$pc_{old} \vdash_{mode} \mathtt{m} \ \wedge \ pc'_{old} \vdash_{mode} \mathtt{m} \ \wedge \ \mathcal{R} \overset{\mathtt{PM}}{\asymp}_{\mathtt{m}} \mathcal{R}' \ \wedge \ \mathcal{B} \bowtie \mathcal{B}')$$

where

- $\mathcal{M} \overset{P}{=} \mathcal{M}'$ iff $\forall l \in [ts, te) \cup [ds, de).\ \mathcal{M}[l] = \mathcal{M}'[l]$.
- $\mathcal{R} \overset{\mathtt{PM}}{\asymp}_{\mathtt{m}} \mathcal{R}'$ iff $(\mathtt{m} = \mathtt{PM} \implies \mathcal{R} = \mathcal{R}')$
- $\mathcal{B} \bowtie \mathcal{B}'$ iff $(\mathcal{B} = \bot \iff \mathcal{B}' = \bot)$.

Definition A.3. We say that two configurations are *U-equivalent* (written $c \overset{U}{\approx} c'$) *iff*

$$(c = c' = \text{HALT}) \vee$$

$$(c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \ \wedge \ c' = \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}'\rangle \ \wedge \ \mathcal{M} \overset{U}{=} \mathcal{M}' \ \wedge$$

$$c \vdash_{mode} \mathtt{m} \ \wedge \ c' \vdash_{mode} \mathtt{m} \ \wedge \ \delta = \delta' \ \wedge \ t = t' \ \wedge \ t_a = t'_a \ \wedge \ \mathcal{R} \overset{\mathtt{UM}}{\asymp}_{\mathtt{m}} \mathcal{R}' \ \wedge \ \mathcal{B} \bowtie \mathcal{B}')$$

where

- $\mathcal{M} \overset{U}{=} \mathcal{M}'$ iff $\forall l \notin [ts, te) \cup [ds, de).\ \mathcal{M}[l] = \mathcal{M}'[l]$
- $\mathcal{R} \overset{\mathtt{UM}}{\asymp}_{\mathtt{m}} \mathcal{R}'$ iff $(\mathtt{m} = \mathtt{UM} \implies \mathcal{R} = \mathcal{R}') \ \wedge \ \mathcal{R}[\mathtt{sr.GIE}] = \mathcal{R}'[\mathtt{sr.GIE}]$
- $\mathcal{B} \bowtie \mathcal{B}'$ iff $(\mathcal{B} = \bot \iff \mathcal{B}' = \bot)$

The following property easily follows from the above definitions:

Property A.2. *Both $\overset{P}{\approx}$ and $\overset{U}{\approx}$ are equivalence relations.*

Proof. Trivial. □

A.6.1 Properties of Definition A.2. The first property says that if a configuration can take a step, also another P-equivalent configuration can.

Property A.3. *If $c_1 \overset{P}{\approx} c_2, c_1 \vdash_{mode} \mathtt{PM}, \mathcal{D}' \vdash c_1 \rightarrow c'_1$ then $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$ and $\mathcal{D}' \vdash c_2 \rightarrow c'_2$.*

Proof. Since $c_1 \overset{P}{\approx} c_2$ and $c_1 \vdash_{mode} \mathtt{PM}$, it also holds that $c_2 \vdash_{mode} \mathtt{PM}$. Also, the instruction $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}])$ is decoded in both $\mathcal{M}_1$ and $\mathcal{M}_2$ at the same protected address, hence $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$, and $\mathcal{D}' \vdash c_2 \rightarrow c'_2$. □

Property A.4. *If $c_1 \overset{P}{\approx} c_2$, $c_1 \vdash_{mode} \mathtt{PM}$, $\mathcal{D} \vdash c_1 \rightarrow c'_1$, $\mathcal{D}' \vdash c_2 \rightarrow c'_2$ and $\mathcal{B}'_1 \bowtie \mathcal{B}'_2$ then $c'_1 \overset{P}{\approx} c'_2$.*

Proof. Since $c_1 \overset{P}{\approx} c_2$, $c_1 \vdash_{mode} \mathtt{PM}$ and $\mathcal{D} \vdash c_1 \rightarrow c'_1$, by Property A.3, $i = decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$ and $\mathcal{D}' \vdash c_2 \rightarrow c'_2$.

Since $\mathcal{B}'_1 \bowtie \mathcal{B}'_2$, we have two cases:

(1) *Case $\mathcal{B}'_1 = \mathcal{B}'_2 = \bot$.* In this case we know that no interrupt handling started during the step, and by exhaustive cases on i we can show $c'_1 \overset{P}{\approx} c'_2$:
   - *Case $i \in \{\mathtt{HLT}, \mathtt{IN}\ \mathtt{r}, \mathtt{OUT}\ \mathtt{r}\}$.* In both cases we have $c'_1 = \text{EXC}_{c_1} \overset{P}{\approx} \text{EXC}_{c_2} = c'_2$.
   - *Otherwise.* The relevant values in c'_1 and c'_2 just depend on values that coincide also in c_1 and c_2. Hence, by determinism of the rules, we get $c'_1 \overset{P}{\approx} c'_2$.

(2) *Case* $\mathcal{B}_1' \neq \bot$ *and* $\mathcal{B}_2' \neq \bot$. In this case an interrupt was handled, but the same instruction was indeed executed in protected mode, hence $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$. Also, $\mathcal{R}_1' \overset{\mathsf{PM}}{\asymp}_{\mathsf{UM}} \mathcal{R}_2'$ holds trivially, $\mathcal{B}_1' \bowtie \mathcal{B}_2'$ by hypothesis and $pc_{old_1}' \vdash_{mode} \mathsf{UM}$ and $pc_{old_2}' \vdash_{mode} \mathsf{UM}$. Thus, $c_1' \overset{P}{\approx} c_2'$.

□

Some sequences of fine-grained traces preserve P-equivalence.

Property A.5. *If* $c_1 \overset{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1}}{}^* c_1' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R})} c_1''$, $\mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R})} c_2''$, *then* $c_1'' \overset{P}{\approx} c_2''$.

Proof. We show by Noetherian induction over (ℓ_1, ℓ_2) that $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$. For that, we use well-founded relation $(\ell_1, \ell_2) \prec (\ell_1', \ell_2')$ *iff* $\ell_1 < \ell_1' \wedge \ell_2 < \ell_2'$.

- *Case* $(0,0)$. Trivial.
- *Case* $(0, \ell_2)$, *with* $\ell_2 > 0$. *(and symmetrically* $(\ell_1, 0)$, *with* $\ell_1 > 0$*)* We have to show that

$$\mathcal{D} \vdash c_1 \xRightarrow{\varepsilon}{}^* c_1' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2' \Rightarrow \mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$$

  Since from c_1 there is no step, $c_1 = c_1'$. Moreover a sequence of ξ was observed starting from c_2, and since both configurations are in unprotected mode and no violation occurred (see Table 2) the protected memory is unchanged. Thus, by transitivity of $\overset{P}{=}$, we have $\mathcal{M}_1' = \mathcal{M}_1 \overset{P}{=} \mathcal{M}_2 \overset{P}{=} \mathcal{M}_2'$.
- *Case* $(\ell_1, \ell_2) = (\ell_1' + 1, \ell_2' + 1)$. If

$$\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1''' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2''' \Rightarrow \mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2''' \text{ (IHP)}$$

  then

$$\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1''' \xRightarrow{\xi} c_1' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2''' \xRightarrow{\xi} c_2' \Rightarrow \mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'.$$

  By (IHP) we know that $\mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2'''$. Indeed, since we observed ξ it means that $pc_{old_1}' \vdash_{mode} \mathsf{m} \wedge pc_{old_2}' \vdash_{mode} \mathsf{m}$. Moreover (see Figure 8) since ξ was observed starting from c_1''' and from c_2''' and since both configurations are in unprotected mode, protected memory is unchanged. Thus, $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2''' \overset{P}{=} \mathcal{M}_2'$.

Since the instruction generating $\alpha = \mathtt{jmpIn?}(\mathcal{R})$ was executed in unprotected mode, we have that $\mathcal{M}_1'' \overset{P}{=} \mathcal{M}_2''$. Also $\mathcal{R}_1'' = \mathcal{R} \overset{\mathsf{PM}}{\asymp}_{\mathsf{PM}} \mathcal{R} = \mathcal{R}_2''$, $pc_{old_1}'' \vdash_{mode} \mathsf{UM}$, $pc_{old_2}'' \vdash_{mode} \mathsf{UM}$ and $\mathcal{B}_1'' \bowtie \mathcal{B}_2''$. □

Property A.6. *If* $c_1 \overset{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \xRightarrow{\mathtt{handle!}(k_1)}{}^* c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1}}{}^* c_1'' \xRightarrow{\mathtt{reti?}(k_1')} c_1'''$,

$\mathcal{D}' \vdash c_2 \xRightarrow{\mathtt{handle!}(k_2)}{}^* c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2'' \xRightarrow{\mathtt{reti?}(k_2')} c_2'''$, *then* $c_1''' \overset{P}{\approx} c_2'''$.

Proof. Since upon observation of handle!(k_x) the protected memory cannot be modified, we know that $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$.

We show by Noetherian induction over (ℓ_1, ℓ_2) that $\mathcal{M}_1'' \overset{P}{=} \mathcal{M}_2''$. For that, we use well-founded relation $(\ell_1, \ell_2) \prec (\ell_1', \ell_2')$ *iff* $\ell_1 < \ell_1' \wedge \ell_2 < \ell_2'$.

- *Case* $(0, 0)$. Trivial.
- *Case* $(0, \ell_2)$*, with* $\ell_2 > 0$ *(and symmetrically* $(\ell_1, 0)$*, with* $\ell_1 > 0$*).* We have to show that

$$\mathcal{D} \vdash c_1' \xRightarrow{\varepsilon}{}^* c_1'' \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2'' \Rightarrow \mathcal{M}_1'' \overset{P}{=} \mathcal{M}_2''$$

Since from c_1' there is no step, $c_1'' = c_1'$. Moreover a sequence of ξ was observed starting from c_2', and since both configurations are in unprotected mode and no violation occurred (see Table 2) the protected memory is unchanged. Thus, by transitivity of $\overset{P}{=}$, we have $\mathcal{M}_1'' = \mathcal{M}_1' \overset{P}{=} \mathcal{M}_2' \overset{P}{=} \mathcal{M}_2''$.
- *Case* $(\ell_1, \ell_2) = (\ell_1' + 1, \ell_2' + 1)$. If

$$\mathcal{D} \vdash c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1^{iv} \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2^{iv} \Rightarrow \mathcal{M}_1^{iv} \overset{P}{=} \mathcal{M}_2^{iv} \text{ (IHP)}$$

then

$$\mathcal{D} \vdash c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1^{iv} \xRightarrow{\xi} c_1'' \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2^{iv} \xRightarrow{\xi} c_2'' \Rightarrow \mathcal{M}_1'' \overset{P}{=} \mathcal{M}_2''.$$

By (IHP) we know that $\mathcal{M}_1^{iv} \overset{P}{=} \mathcal{M}_2^{iv}$. Indeed, since we observed ξ it means that ${pc_{old}}_1'' \vdash_{mode}$ UM $\wedge$ $\vdash_{mode}$ UM${pc_{old}}_2''$. Moreover (see Figure 8) since ξ was observed starting from c_1^{iv} and from c_2^{iv} and since both configurations are in unprotected mode, no violation occurred and by Table 2 protected memory is unchanged. Thus, by transitivity of $\overset{P}{=}$, we have $\mathcal{M}_1'' \overset{P}{=} \mathcal{M}_1^{iv} \overset{P}{=} \mathcal{M}_2^{iv} \overset{P}{=} \mathcal{M}_2''$.

Thus, we have that $\mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2'''$, since $\alpha =$ reti?$(\cdot)$ does not modify protected memory. Also $\mathcal{R}_1''' \overset{\mathsf{PM}}{\asymp}_{\mathsf{UM}} \mathcal{R}_2'''$, $\mathcal{B}_1''' \bowtie \mathcal{B}_2'''$, $pc_{old1}' \vdash_{mode}$ UM and $pc_{old2}' \vdash_{mode}$ UM, by definition of $\alpha =$ reti?$(\cdot)$. □

Property A.7. *If* $c_1 \overset{P}{\approx} c_2$*,* $c_1 \vdash_{mode}$ PM*,* $\mathcal{D} \vdash c_1 \xRightarrow{\alpha_1} c_1'$*,* $\mathcal{D}' \vdash c_2 \xRightarrow{\alpha_2} c_2'$*,* $\alpha_1, \alpha_2 \neq$ handle!$(\cdot)$ *then* $\alpha_1 = \alpha_2$ *and* $c_1' \overset{P}{\approx} c_2'$.

Proof. By definition of fine-grained traces we know that the transition leading to the observation of α_1 happens upon the execution of an instruction that must also be executed starting from c_2 (by Property A.3) and that $c_1' \overset{P}{\approx} c_2'$ (by Property A.4). Also, since $c_1 \vdash_{mode}$ PM, we know that $\alpha_1 \in \{\tau(k_1),$ jmpOut!$(k_1; \mathcal{R}_1)\}$. Thus, in both cases and since by hypothesis $\alpha_2 \neq$ handle!$(\cdot)$, it must be that $\alpha_2 = \alpha_1$. □

Property A.8. *If* $c_1 \overset{P}{\approx} c_2$*,* $\mathcal{D} \vdash c_1 \xRightarrow{\tau(k_1^{(0)}) \cdots \tau(k_1^{(n_1-1)}) \cdot \alpha_1}{}^* c_1'$*,* $\mathcal{D}' \vdash c_2 \xRightarrow{\tau(k_2^{(0)}) \cdots \tau(k_2^{(n_2-1)}) \cdot \alpha_2}{}^* c_2'$*, and* $\alpha_1, \alpha_2 \neq$ handle!$(\cdot)$ *then* $\tau(k_1^{(0)}) \cdots \tau(k_1^{(n_1-1)}) \cdot \alpha_1 = \tau(k_2^{(0)}) \cdots \tau(k_2^{(n_2-1)}) \cdot \alpha_2$ *and* $c_1' \overset{P}{\approx} c_2'$.

Proof. Corollary of Property A.7. □

P-equivalence is preserved by complete interrupt segments (recall Definition A.1). Indeed, from now onwards denote

$$\begin{aligned}\overline{\alpha}_x \in & \{\varepsilon\} \cup \\ & \{\alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \mid n_x \geq 1 \wedge \alpha_x^{(n_x-1)} = \mathtt{reti?}(k_x^{(n_x-1)}) \wedge \\ & \qquad \forall i.\, 0 \leq i \leq n_x - 1.\, \alpha_x^{(i)} \notin \{\bullet, \mathtt{jmpIn?}(\mathcal{R}_x^{(i)}), \mathtt{jmpOut!}(k_x^{(i)}; \mathcal{R}_x^{(i)})\}\}.\end{aligned}$$

Property A.9. *Let $\mathcal{D}$ and $\mathcal{D}'$ be two devices.*
If $c_1^{(0)} \overset{P}{\approx} c_2^{(0)}$, $\mathcal{D} \vdash c_1 \xRightarrow{\mathtt{jmpIn?}(\mathcal{R})} c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)}$ and $\mathcal{D}' \vdash c_2 \xRightarrow{\mathtt{jmpIn?}(\mathcal{R})} c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_2^{(n_2)}$ then $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$.

Proof. We first show by induction on $|\mathbb{I}_{\overline{\alpha}_1}|$ (see Definition A.1) that

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_1^{(n_2)} \Rightarrow c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$$

assuming wlog that $|\mathbb{I}_{\overline{\alpha}_2}| \leq |\mathbb{I}_{\overline{\alpha}_1}|$.

- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = 0$. Trivial.
- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}'_1}| + 1$. If

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}'_1}{}^{*} c_1^{(n'_1)} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}'_2}{}^{*} c_2^{(n'_2)} \Rightarrow c_1^{(n'_1)} \overset{P}{\approx} c_2^{(n'_2)} \text{ (IHP)}$$

then

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_2^{(n_2)} \Rightarrow c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$$

Now let (i_1, j_1) be the new interrupt segment of $\overline{\alpha}_1$ that we split it as follows:

$$\overline{\alpha}_1 = \overline{\alpha}'_1 \cdot \tau(k_1^{(n'_1)}) \cdots \tau(k_1^{(i_1-1)}) \cdot \mathtt{handle!}(k_1^{(i_1)}) \cdots \mathtt{reti?}(k_1^{(j_1)})$$

The following two exhaustive cases may arise.

(1) *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$. For some (i_2, j_2) we then have:

$$\overline{\alpha}_2 = \overline{\alpha}'_2 \cdot \tau(k_2^{(n'_2)}) \cdots \tau(k_2^{(i_2-1)}) \cdot \mathtt{handle!}(k_2^{(i_2)}) \cdots \mathtt{reti?}(k_2^{(j_2)})$$

By Properties A.8 and A.6 we know that $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$, being reached through $\alpha_1^{(j_1)}$ and $\alpha_2^{(j_2)}$.

(2) *Case* $|\mathbb{I}_{\overline{\alpha}_2}| < |\mathbb{I}_{\overline{\alpha}_1}|$. In this case we have

$$\overline{\alpha}_2 = \overline{\alpha}'_2 \cdot \tau(k_2^{(n'_2)}) \cdots \tau(k_2^{(n_2-2)}) \cdot \tau(k_2^{(n_2-1)})$$

with $c_1^{\ell} \overset{P}{\approx} c_2^{\ell}$ for $n'_2 \leq \ell \leq n_2 - 2 = i_1 - 1$, where the last equality holds because the module is executing from configurations that are P-equivalent. As soon as the interrupt arrives, the same instruction is executed (Property A.3) that causes the same changes in the registers, the old program counter and the protected memory. In turn the first two are stored in the backup before handling the interrupt. They are then restored by the RETI, observed as $\alpha_1^{(j_1)}$, while the protected memory is left untouched. Consequently, we have that $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$, that are the configurations reached through $\alpha_1^{(j_1)}$ and $\tau(k_2^{(n_2)-1})$.

□

Finally, we can show that P-equivalence is preserved by coarse-grained traces:

Property A.10. *If* $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_1$ *and* $\mathcal{D}' \vdash \text{INIT}_{C'[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_2$ *then* $c_1 \overset{P}{\approx} c_2$.

Proof. By definition of coarse-grained traces, we have that in both premises $\texttt{jmpIn?}(\mathcal{R})$ is preceded by a sequence of ξ actions (possibly in different numbers). Since neither ξ actions nor $\texttt{jmpIn?}(\mathcal{R})$ ever change the protected memory (by definition of memory access control) and since the $\texttt{jmpIn?}(\mathcal{R})$ sets the registers to the values in $\mathcal{R}$, it follows that $c_1 \overset{P}{\approx} c_2$. □

The following definition gives an equality up to timings among coarse-grained traces:

Definition A.4. Let $\overline{\beta} = \beta_0 \dots \beta_n$ and $\overline{\beta}' = \beta'_0 \dots \beta'_{n'}$ be two coarse-grained traces. We say that $\overline{\beta}$ is *equal up to timings* to $\overline{\beta}'$ (written $\overline{\beta} \approx \overline{\beta}'$) *iff*

$$n = n' \land (\forall i \in \{0, \dots, n\}. \beta_i = \beta'_i \lor (\beta_i = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \land \beta'_i = \texttt{jmpOut!}(\Delta t'; \mathcal{R}))).$$

Finally, the property below shows that the traces that are equal up to timings preserve the P-equivalence:

Property A.11. *If* $c_1 \overset{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow}{}^* c'_1$, $\mathcal{D}' \vdash c_2 \overset{\overline{\beta}'}{\Longrightarrow}{}^* c'_2$ *and* $\overline{\beta} \approx \overline{\beta}'$ *then* $c'_1 \overset{P}{\approx} c'_2$.

Proof. The thesis easily follows from Property A.5 and Property A.9. □

A.6.2 Properties of Definition A.3. Also for U-equivalent configurations it holds that when one takes a step, also the other does.

Property A.12. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ *then* $decode(\mathcal{M}_1, \mathcal{R}_1[\mathsf{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathsf{pc}])$.

Proof. Since $c_1 \overset{U}{\approx} c_2$ and $c_1 \vdash_{mode} \mathsf{UM}$, it also holds that $c_2 \vdash_{mode} \mathsf{UM}$. Also, the instruction $decode(\mathcal{M}_1, \mathcal{R}_1[\mathsf{pc}])$ is decoded in both $\mathcal{M}_1$ and $\mathcal{M}_2$ at the same unprotected address, hence $decode(\mathcal{M}_1, \mathcal{R}_1[\mathsf{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathsf{pc}])$. □

Next we prove that $\overset{U}{\approx}$ is preserved by unprotected-mode steps of the Sancus$^{\text{L}}$ operational semantics:

Property A.13. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ *and* $\mathcal{D} \vdash c_1 \rightarrow c'_1$, *then* $\mathcal{D} \vdash c_2 \rightarrow c'_2 \ \land \ c'_1 \overset{U}{\approx} c'_2$.

Proof. Since $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ and $\mathcal{D} \vdash c_1 \rightarrow c'_1$, by Property A.12, $i = decode(\mathcal{M}_1, \mathcal{R}_1[\mathsf{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathsf{pc}])$.

To show that $c'_1 \overset{U}{\approx} c'_2$, we consider the following exhaustive cases:

- *Case* $i = \bot$. Since $c_1 \overset{U}{\approx} c_2$ we get $c_2 \vdash_{mode} \mathsf{UM}$ and by definition of $\cdot \vdash \cdot \rightarrow \cdot$ we get $c'_1 = \text{EXC}_{c_1}$ and $c'_2 = \text{EXC}_{c_2}$. However, by definition of EXC., we have that $\mathcal{M}'_1 \overset{U}{=} \mathcal{M}'_2$, $c'_1 \vdash_{mode} \mathsf{UM}$, $c'_2 \vdash_{mode} \mathsf{UM}$, $\delta'_1 = \delta_1 = \delta_2 = \delta'_2$, $t'_1 = t_1 = t_2 = t'_2$, $t'_{a_1} = t_{a_1} = t_{a_2} = t'_{a_2}$, $\mathcal{R}'_1 \overset{\mathsf{UM}}{\asymp}_{\mathsf{m}} \mathcal{R}'_2$, and $\bot = \mathcal{B}'_1 \bowtie \mathcal{B}'_2 = \bot$, i.e., $c'_1 \overset{U}{\approx} c'_2$.
- *Case* $i = \mathsf{HLT}$. Trivial, since $c'_1 = \text{HALT} = c'_2$.
- *Case* $i \neq \bot$. We have the following exhaustive sub-cases, depending on c'_1:
  - *Case* $c'_1 = \text{EXC}_{c_1}$. In this case a violation occurred, i.e., $i, \mathcal{R}_1, pc_{old1}, \mathcal{B}_1 \nvdash_{mac} \mathsf{OK}$. However, the same violation also occurs for c_2, since the only parts that may keep c_1 apart from c_2 are pc_{old} and $\mathcal{B}$, and thus $c'_1 \overset{U}{\approx} c'_2$ because:
    * $pc_{old2} \neq pc_{old1}$, cannot cause a failure since unprotected code is executable from anywhere,

  - $\mathcal{B}_1 = \langle \mathcal{R}_1, pc_{old1}, t_{pad_1} \rangle \neq \langle \mathcal{R}_2, pc_{old2}, t_{pad_2} \rangle = \mathcal{B}_2$, cannot cause a failure since the additional conditions on the configuration imposed by the memory access control only concern values that are the same in both configurations.
- *Case* $c_1' \neq \mathrm{EXC}_{c_1}$ *and* $i = \texttt{RETI}$. If $\mathcal{B}_1 = \bot$, then $\mathcal{B}_1 = \mathcal{B}_2 = \mathcal{B}_1' = \mathcal{B}_2' = \bot$, hence rule (CPU-Reti) applies and we get $c_1' \overset{U}{\approx} c_2'$ since $\mathcal{R}_1' = \mathcal{R}_2'$ and $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$ is a deterministic relation (Property A.1). If $\mathcal{B}_1 \neq \bot$ it must also be that $\mathcal{B}_2 \neq \bot$ by U-equivalence, so either rule (CPU-Reti-Chain) or rule (CPU-Reti-PrePad) applies. In the first case we get $c_1' \overset{U}{\approx} c_2'$ because $c_1 \overset{U}{\approx} c_2$ and by determinism of $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$ and $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathrm{I}} \cdot$. In the second case we get $c_1' \overset{U}{\approx} c_2'$ since $\langle \bot, \bot, t'_{pad_1} \rangle = \mathcal{B}_1' \bowtie \mathcal{B}_2' = \langle \bot, \bot, t'_{pad_2} \rangle$ and $\mathcal{R}_1' \overset{\mathsf{UM}}{\asymp}_{\mathsf{PM}} \mathcal{R}_2'$ holds since we restored the register files from backups in which the interrupts were enabled (otherwise the CPU would not have handled the interrupt it is returning from).
- *Case* $c_1' \neq \mathrm{EXC}_{c_1}$ *and* $i \notin \{\bot, \texttt{HLT}, \texttt{RETI}\}$. All the other rules depend on both (i) parts of the configurations that are equal due to $c_1 \overset{U}{\approx} c_2$, and on (ii) $\mathcal{D} \vdash \cdot \curvearrowright_D^5 \cdot$ and $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathrm{I}} \cdot$ which are deterministic and have the same inputs (since $c_1 \overset{U}{\approx} c_2$). Hence, $c_1' \overset{U}{\approx} c_2'$ as requested.

□

The above property carries on fine-grained traces, provided that the computation is carried on in unprotected mode:

Property A.14. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$, $\mathcal{D} \vdash c_1 \overset{\alpha}{\Longrightarrow} c_1'$ *then* $\mathcal{D} \vdash c_2 \overset{\alpha}{\Longrightarrow} c_2'$ *and* $c_1' \overset{U}{\approx} c_2'$.

Proof. By Properties A.12 and A.13, $c_1' \overset{U}{\approx} c_2'$ and $i = decode(\mathcal{M}_1, \mathcal{R}_1[\mathsf{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathsf{pc}])$.

Thus, since the same i is executed under U-equivalent configurations and since $c_1' \overset{U}{\approx} c_2'$, we have that $\mathcal{D} \vdash c_2 \overset{\alpha}{\Longrightarrow} c_2'$. □

Property A.15. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$, $\mathcal{D} \vdash c_1 \xRightarrow{\xi\cdots\xi\cdot\alpha}{}^* c_1'$ *and* $\alpha \in \{\xi, \bullet, \texttt{jmpIn?}(\mathcal{R}), \texttt{reti?}(k)\}$ *then* $\mathcal{D} \vdash c_2 \xRightarrow{\xi\cdots\xi\cdot\alpha}{}^* c_2'$ *and* $c_1' \overset{U}{\approx} c_2'$.

Proof. The proof goes by induction on the length n of $\xi \cdots \xi$.

- *Case* $n = 0$. Property A.14 applies.
- *Case* $n' = n+1$. By induction hypothesis for some c_1''', c_2''', c_1'' and c_2'' we have $\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi\cdots\xi}^{n'}} c_1''' \overset{\alpha}{\Longrightarrow} c_1''$, $\mathcal{D} \vdash c_2 \xRightarrow{\overbrace{\xi\cdots\xi}^{n'}} c_2''' \overset{\alpha}{\Longrightarrow} c_2''$ and $c_1'' \overset{U}{\approx} c_2''$. Thus, if $\mathcal{D} \vdash c_1''' \overset{\xi}{\Longrightarrow} c_1^{iv}$ (i.e., we observe a further ξ starting from c_1), by Property A.14 we get $\mathcal{D} \vdash c_2''' \overset{\xi}{\Longrightarrow} c_2^{iv}$ and $c_1^{iv} \overset{U}{\approx} c_2^{iv}$. Finally, by Property A.14 applies on c_1^{iv} and c_2^{iv} we get the thesis.

□

Now we move our attention to $\texttt{handle!}(\cdot)$.

Property A.16. *If* $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$, $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\tau(k_1^{(0)})\ \cdots\ \tau(k_1^{(n_1-1)})\cdot\texttt{handle!}(k_1^{(n_1)})}{}^* c_1^{(n_1+1)}$ *and* $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\tau(k_2^{(0)})\ \cdots\ \tau(k_2^{(n_2-1)})\cdot\texttt{handle!}(k_2^{(n_2)})}{}^* c_2^{(n_2+1)}$ *then* $c_1^{(n_1+1)} \overset{U}{\approx} c_2^{(n_2+1)}$.

Proof.

- By definition of fine-grained semantics, $\texttt{handle!}(k_x^{(n_x)})$ only happens when an interrupt is handled with $c_x^{(n_x)}$ in protected mode.
- By definition of $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathrm{I}} \cdot$, $\mathcal{R}_1^{(n_1+1)} = \mathcal{R}_2^{(n_2+1)} = \mathcal{R}_0[\texttt{pc} \mapsto isr]$.
- Since unprotected memory cannot be changed by protected mode actions without causing a violation (that would cause the observation of a $\texttt{jmpOut!}(\cdot;\cdot)$) and is not changed upon RETI when it happens in a configuration with backup different from $\bot$ (cf. rules **(CPU-Reti-*)**), $\mathcal{M}_1^{(n_1+1)} \overset{U}{=} \mathcal{M}_2^{(n_2+1)}$.
- Since we observe $\texttt{handle!}(k_x^{(n_x)})$ it must be that $\texttt{GIE} = 1$ and it had to be such also in $c_x^{(0)}$ (because by definition the operations on registers cannot modified this flag in protected mode). Hence, $t_{a_x}^i = \bot$ for $0 \leq i \leq n_x$. Let $t_{a_1}^{int}$ and $t_{a_2}^{int}$ be the arrival times of the interrupt that originated the observations $\texttt{handle!}(k_1^{(n_1)})$ and $\texttt{handle!}(k_2^{(n_2)})$, resp. By definition of $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$, $t_{a_1}^{int}$ and $t_{a_2}^{int}$ are the first absolute times after $t_1^{(n_1)}$ and $t_2^{(n_2)}$ in which an interrupt was raised and, since $\mathcal{D}$ is deterministic and $t_{a_x}^{(i)} = \bot$ for $0 \leq i \leq n_x$, it must be that $t_{a_1}^{int} = t_{a_2}^{int} = t^{int}$ (recall that $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and that IN or OUT instructions are forbidden in protected mode).

  Assume now that the instruction during which the interrupt occurred ended at time t_x^f. Then we can write $t^{(n_x+1)}$ as:

$$t^{(n_x+1)} = t^{(n_x)} + k_x^{(n_x)} = t^{(n_x)} + \underbrace{t^{int} - t^{(n_x)} + t_x^f - t^{int}}_{\text{Duration of the instruction}} + \underbrace{\texttt{MAX\_TIME} - t_x^f + t^{int}}_{\text{Mitigation from } \textbf{(INT-PM-P)}} + 6$$
$$= \cancel{t^{(n_x)}} + t^{int} - \cancel{t^{(n_x)}} + \cancel{t_x^f} - \cancel{t^{int}} + \texttt{MAX\_TIME} - \cancel{t_x^f} + \cancel{t^{int}} + 6$$
$$= t^{int} + \texttt{MAX\_TIME} + 6$$

  and therefore $t^{(n_1+1)} = t^{(n_2+1)}$.
- Since $t^{(n_1+1)} = t^{(n_2+1)}$, $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and no interaction with $\mathcal{D}$ via IN or OUT can occur in protected mode, the deterministic device $\mathcal{D}$ performed the same number of steps in both computations, and then $t_{a_1}^{(n_1+1)} = t_{a_2}^{(n_2+1)}$ and $\delta_1^{(n_1+1)} = \delta_2^{(n_2+1)}$.

Hence, $c_1^{(n_1+1)} \overset{U}{\approx} c_2^{(n_2+1)}$ as requested. □

The following properties show that the combination of U-equivalence and trace equivalence induces some useful properties of modules and sequences of complete interrupt segments. Before doing that we define the $(\overline{a}, n)$-interrupt-limited version of a context C as the context that behaves as C but such that (i) the transition relation of its device results from unrolling at most n steps of its transition relation and (ii) its device never raises interrupts *after* observing the sequence of actions $\overline{a}$:

Definition A.5. Let $\mathcal{D} = \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle$ be an I/O device. Let $\overline{a}$ be a string over the signature A of I/O devices and denote ℓ as the function that associates to each string over A a unique natural number (e.g., its position in a suitable lexicographic order). Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, we define its corresponding $(\overline{a}, n)$*-interrupt-limited context* as $C_{\leq \overline{a},n} = \langle \mathcal{M}_C, \mathcal{D}_{\leq \overline{a},n} \rangle$ where $\mathcal{D}_{\leq \overline{a},n} = \langle img(\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n}) \cup dom(\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n}), 0, \overset{a}{\rightsquigarrow}_{D \leq \overline{a},n} \rangle$ and

$$\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n} \triangleq \Big(\{(p, a, p') \mid \forall \overline{a}'. p = \ell(\overline{a}') \land p' = \ell(\overline{a}' \cdot a) \land \delta_{init} \overset{\overline{a}'}{\rightsquigarrow}{}^*_D \delta \overset{a}{\rightsquigarrow}_D \delta' \land |\overline{a}' \cdot a| \leq n\} \setminus$$
$$\{(p, int?, p') \mid \forall \overline{a}'. p = \ell(\overline{a} \cdot \overline{a}') \land p' = \ell(\overline{a} \cdot \overline{a}' \cdot int?)\} \Big) \cup$$
$$\{(p, \epsilon, p') \mid \forall \overline{a}'. p = \ell(\overline{a} \cdot \overline{a}') \land p' = \ell(\overline{a} \cdot \overline{a}' \cdot int?) \land \delta_{init} \overset{\overline{a} \cdot \overline{a}'}{\rightsquigarrow}{}^*_D \delta \overset{int?}{\rightsquigarrow}_D \delta' \land |\overline{a} \cdot \overline{a}' \cdot int?| \leq n\}.$$

(Note that any $(\overline{a}, n)$-interrupt-limited context is actually a device, due to the constraint on its transition function).

Now, let

$$\overline{\alpha}_x \in \{\varepsilon\} \cup \{\alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \mid n_x \geq 1 \wedge \alpha_x^{(n_x-1)} = \texttt{reti?}(k_x^{(n_x-1)}) \wedge \\ \forall i.\, 0 \leq i \leq n_x - 1.\, \alpha_x^{(i)} \notin \{\bullet, \texttt{jmpIn?}(\mathcal{R}_x^{(i)}), \texttt{jmpOut!}(k_x^{(i)}; \mathcal{R}_x^{(i)})\}\}.$$

Property A.17. *If*

- $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[M_M]} \xRightarrow{\overline{\beta}\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_1^{(0)}$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[M_{M'}]} \xRightarrow{\overline{\beta}\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_2^{(0)}$
- $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$
- *for some* $m_1 \geq 0$, $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1\cdot\tau(k_1^{(n_1)})\cdots\tau(k_1^{(n_1+m_1-1)})\cdot\texttt{jmpOut!}(k_1^{(n_1+m_1)};\mathcal{R}')}{}^{*} c_1^{(n_1+m_1+1)}$
- *for some* $m_2 \geq 0$, $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2\cdot\tau(k_2^{(n_2)})\cdots\tau(k_2^{(n_2+m_2-1)})\cdot\texttt{jmpOut!}(k_2^{(n_2+m_2)};\mathcal{R}')}{}^{*} c_2^{(n_2+m_2+1)}$

then $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$.

Proof. We show this property by contraposition, by showing that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \neq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ then $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$. For that it suffices to show that

$$\exists C'.\mathcal{D}' \vdash \mathrm{INIT}_{C'[\mathcal{M}_M]} \xRightarrow{\overline{\beta}\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_3^{(0)} \xRightarrow{\texttt{jmpOut!}(\Delta t_3;\mathcal{R}_3^{(n_3+m_3)})} c_3^{(n_3+m_3+1)}$$

(i.e., $\mathcal{D} \vdash c_3^{(0)} \xRightarrow{\overline{\alpha}_3\cdot\tau(k_3^{(n_3)})\cdots\tau(k_3^{(n_3+m_3-1)})\cdot\texttt{jmpOut!}(k_3^{(n_3+m_3)};\mathcal{R}_3^{(n_3+m_3)})}{}^{*} c_3^{(n_3+m_3+1)}$)
such that

$$\forall C''.\, \mathcal{D}'' \vdash \mathrm{INIT}_{C''[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_4^{(0)} \xRightarrow{\texttt{jmpOut!}(\Delta t_4;\mathcal{R}_4^{(n_4+m_4+1)})} c_4^{(n_4+m_4+1)} \quad \text{with } \Delta t_3 \neq \Delta t_4$$

(i.e., $\mathcal{D} \vdash c_4^{(0)} \xRightarrow{\overline{\alpha}_4\cdot\tau(k_4^{(n_4)})\cdots\tau(k_4^{(n_4+m_4-1)})\cdot\texttt{jmpOut!}(k_4^{(n_4+m_4)};\mathcal{R}_4^{(n_4+m_4)})}{}^{*} c_4^{(n_4+m_4+1)}$).

Assume wlog that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) < \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$. Noting that the first observable of $\overline{\beta} \cdot \texttt{jmpIn?}(\mathcal{R})$ must be a $\texttt{jmpIn?}(\cdot)$, by Properties A.10 and A.11, we have that $c_1^{(0)} \overset{P}{\approx} c_3^{(0)}$ and, similarly, $c_2^{(0)} \overset{P}{\approx} c_4^{(0)}$. Thus, as a consequence of Properties A.3, A.9 and A.8, $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_3+m_3} \gamma(c_3^{(i)})$ and $\sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) = \sum_{i=0}^{n_4+m_4} \gamma(c_4^{(i)})$.

Let $n \in \mathbb{N}$ be greater than the number of steps over the relation $\dot{\rightsquigarrow}_D$ in the computation $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c_1^{(n_1+m_1+1)}$ and let $\overline{a}$ be the sequence of actions over $\dot{\rightsquigarrow}_D$ in the computation $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c_1^{(0)}$. Choosing $C' = C_{\leq \overline{a},n}$ we get $\Delta t_3 = \sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_3+m_3} \gamma(c_3^{(i)})$. Any other context C'' that allows to observe the same $\overline{\beta} \cdot \texttt{jmpIn?}(\mathcal{R})$ from $\mathrm{INIT}_{C''[\mathcal{M}_{M'}]}$ raises 0 or more interrupts "after" c_4^0, hence taking additional $S \geq 0$ cycles on top of those required for the instructions to be executed. Thus $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, since $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) < \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ and $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \Delta t_3 < \Delta t_4 = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) + S$. □

Property A.18. *If*

- $\mathcal{D} \vdash \text{INIT}_{C[M_M]} \xRightarrow{\overline{\beta}\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_1^{(0)}$
- $\mathcal{D} \vdash \text{INIT}_{C[M_{M'}]} \xRightarrow{\overline{\beta}'\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c_2^{(0)}$
- $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$
- $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1\cdot\tau(k_1^{(n_1)})\cdots\tau(k_1^{(n_1+m_1-1)})\cdot\alpha_1}{}^{*} c_1^{(n_1+m_1+1)}$ *for some* $m_1 \geq 0$ *and* $\alpha_1 \in \{\texttt{jmpOut!}(k_1^{(n_1+m_1)};\mathcal{R}'), \texttt{handle!}(k_1^{(n_1+m_1)})\}$
- $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2\cdot\tau(k_2^{(n_2)})\cdots\tau(k_2^{(n_2+m_2-1)})\cdot\alpha_2}{}^{*} c_2^{(n_2+m_2+1)}$ *for some* $m_2 \geq 0$ *and* $\alpha_2 \in \{\texttt{jmpOut!}(k_2^{(n_2+m_2)};\mathcal{R}'), \texttt{handle!}(k_2^{(n_2+m_2)})\}$

then

(1) $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$
(2) $c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)}$.

Proof. Assume wlog that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \leq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$, and we prove by induction on $|\mathbb{I}_{\overline{\alpha}_1}|$ that

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \;\wedge\; \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_1^{(n_2)} \quad \text{imply} \quad c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)} \;\wedge\; |\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$$

- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = 0$. Since no complete interrupt segment was observed it means that $\overline{\alpha}_1$ cannot end with a $\texttt{reti?}(\cdot)$, so it must be $\overline{\alpha}_1 = \varepsilon$. Moreover, since $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and the value of the `GIE` bit cannot be changed in protected mode, we know that:
  - *Case* $\mathcal{R}_1^{(0)}[\texttt{sr.GIE}] = \mathcal{R}_2^{(0)}[\texttt{sr.GIE}] = \texttt{0}$. Then no $\texttt{handle!}(\cdot)$ can be observed in $\overline{\alpha}_2$, hence it must be that $\overline{\alpha}_2 = \varepsilon$ and the two thesis easily follow.
  - *Case* $\mathcal{R}_1^{(0)}[\texttt{sr.GIE}] = \mathcal{R}_2^{(0)}[\texttt{sr.GIE}] = 1$. Then it means that no interrupt was raised by the device in the computation starting with $c_1^{(0)}$ and the same must happen in $c_2^{(0)}$ because of U-equivalence and $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \leq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$. Hence it must be that $\overline{\alpha}_2 = \varepsilon$ and the two thesis easily follow.
- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}'_1}| + 1$. If

  $$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}'_1}{}^{*} c_1^{(n'_1)} \;\wedge\; \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}'_2}{}^{*} c_2^{(n'_2)} \quad \text{imply} \quad c_1^{(n'_1)} \overset{U}{\approx} c_2^{(n'_2)} \;\wedge\; |\mathbb{I}_{\overline{\alpha}'_1}| = |\mathbb{I}_{\overline{\alpha}'_2}| \text{ (IHP)}$$

  then

  $$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \;\wedge\; \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_2^{(n_2)} \quad \text{imply} \quad c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)} \;\wedge\; |\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$$

  Now let (i_1, j_1) be the new interrupt segment of $\overline{\alpha}_1$, that we split as follows:

  $$\overline{\alpha}_1 = \overline{\alpha}'_1 \cdot \tau(k_1^{(n'_1)}) \cdots \tau(k_1^{(i_1-1)}) \cdot \texttt{handle!}(k_1^{(i_1)}) \cdots \texttt{reti?}(k_1^{(j_1)}).$$

  Since by (IHP) $c_1^{(n'_1)} \overset{U}{\approx} c_2^{(n'_2)}$ and $\mathcal{D}$ is deterministic and no successfully I/O ever happens in protected mode, the first new interrupt (i.e., the one leading to the observation of $\texttt{handle!}(k_1^{(i_1)})$) is raised at the same cycle in both computations. Call $c_2^{(i_2)}$ the configuration at the beginning of the step of computation in which such interrupt was raised (the choice of indexes will be clear below). From this configuration only three cases for the fine-grained action might be observed:
  - *Case* $\tau(\cdot)$ *and* $\texttt{jmpOut!}(\cdot;\cdot)$. Never happens, since $\mathcal{B}_2^{(i_2+1)} \neq \bot$.
  - *Case* $\texttt{handle!}(k_2^{(i_2)})$. Property A.16 ensures that $c_2^{(i_2+1)} \overset{U}{\approx} c_1^{(i_1+1)}$, and Property A.15 that at some index j_2 a $\texttt{reti?}(k_2^{(j_2)})$ is observed in $\overline{\alpha}_2$, i.e., a new interrupt segment (i_2, j_2) is

observed. Thus, $|\mathbb{I}_{\overline{\alpha}_2}| = |\mathbb{I}_{\overline{\alpha}'_2}|+1 = |\mathbb{I}_{\overline{\alpha}'_1}|+1 = |\mathbb{I}_{\overline{\alpha}_1}|$ (where the second equality holds by (IHP)).
Finally, by definition of $\overline{\alpha}_2$, we have that $n_1 = j_1 + 1$ and $n_2 = j_2 + 2$, hence $c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)}$.

□

The following property states that U-equivalent unprotected-mode configurations perform the same single coarse-grained action:

Property A.19. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ *and* $\mathcal{D} \vdash c_1 \overset{\beta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} c'_1$, *then* $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} c'_2$ *and* $c'_1 \overset{U}{\approx} c'_2$.

Proof. Since $c_1 \vdash_{mode} \mathsf{UM}$, the segment of fine-grained trace that originated β (see Figure 9) is in the form:

$$\mathcal{D} \vdash c_1 \xRightarrow{\xi\cdots\xi\cdot\alpha}{}^{*} c'_1$$

with either $\alpha = \bullet$ or $\alpha = \mathtt{jmpIn?}(\mathcal{R})$.
Property A.15 guarantees that:

$$\mathcal{D} \vdash c_2 \xRightarrow{\xi\cdots\xi\cdot\alpha}{}^{*} c'_2 \wedge c'_1 \overset{U}{\approx} c'_2.$$

Thus, $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} c'_2$ and $c'_1 \overset{U}{\approx} c'_2$. □

Finally, we can show that U-equivalence is preserved by coarse-grained traces:

Property A.20. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$, $\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c'_1$, $\mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c'_2$, $c'_1 \vdash_{mode} \mathsf{UM}$ *and* $c'_2 \vdash_{mode} \mathsf{UM}$ *then* $c'_1 \overset{U}{\approx} c'_2$.

Proof. We show the property by induction on n, the length of $\overline{\beta}$:

- *Case* $n = 0$. By definition of $\overset{\varepsilon}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*}$ we know that it must be $c'_1 = c_1$ and $c'_2 = c'_2$ and the thesis easily follows.
- *Case* $n = n' + 1$. The only case in which a coarse-grained trace can be extended by just one action, while remaining in unprotected mode, is when the action is $\bullet$. In this case the hypothesis easily follows from the definition of $\bullet$ and U-equivalence.
- *Case* $n = n' + 2$. If

$$\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c''_1 \;\wedge\; \mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c''_2 \;\wedge\; \mathcal{R}''_1[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \;\wedge\; \mathcal{R}''_2[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \text{ imply } c''_1 \overset{U}{\approx} c''_2$$

then

$$\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c''_1 \overset{\beta\beta'}{\Longrightarrow\!\!\!\!\!\!\rightarrow} c'_1 \wedge \mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\rightarrow}{}^{*} c''_2 \overset{\beta\beta'}{\Longrightarrow\!\!\!\!\!\!\rightarrow} c'_2 \wedge \mathcal{R}'_1[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \wedge \mathcal{R}'_2[\mathsf{pc}] \vdash_{mode} \mathsf{UM} \text{ imply } c'_1 \overset{U}{\approx} c'_2.$$

By cases on $\beta\beta'$:

- *Case* $\beta\beta' = \mathtt{jmpIn?}(\mathcal{R})\ \bullet$. Directly follows from definition of $\bullet$ and $\overset{U}{\approx}$.
- *Case* $\beta\beta' = \mathtt{jmpIn?}(\mathcal{R})\ \mathtt{jmpOut!}(\Delta t; \mathcal{R}')$. By definition they are originated by

$$\mathcal{D} \vdash c''_1 \xRightarrow{\xi\cdots\xi\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_1^{(0)} \xRightarrow{\alpha_1^{(0)}\ \cdots\ \alpha_1^{(n_1-1)}}{}^{*} c_1^{(n_1)} \xRightarrow{\mathtt{jmpOut!}(k_1^{(n_1)}; \mathcal{R}')} c'_1$$

$$\mathcal{D} \vdash c''_2 \xRightarrow{\xi\cdots\xi\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_2^{(0)} \xRightarrow{\alpha_2^{(0)}\ \cdots\ \alpha_2^{(n_2-1)}}{}^{*} c_2^{(n_2)} \xRightarrow{\mathtt{jmpOut!}(k_2^{(n_2)}; \mathcal{R}')} c'_2.$$

By *(IHP)* and by Property A.15 we can conclude that $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$.
Let $c_x^{(M_x)}$ be the configuration generated by the last $\mathtt{reti?}(\cdot)$ in $\alpha_x^{(0)}\ \cdots\ \alpha_x^{(n_x-1)}$. By Property A.18 the number of completely handled interrupts is the same in the two traces and $c_1^{(M_1)} \overset{U}{\approx} c_2^{(M_2)}$. Also:

- $*$ By definition of $\texttt{jmpOut!}(k_1^{(n_1)};\mathcal{R}')$ and $\texttt{jmpOut!}(k_2^{(n_2)};\mathcal{R}')$ we trivially get $\mathcal{R}_1' = \mathcal{R}_2' = \mathcal{R}'$.
- $*$ Since unprotected memory cannot be changed in protected mode (see Table 2) and $c_1^{(M_1)} \overset{U}{\approx} c_2^{(M_2)}$, $\mathcal{M}_1' \overset{U}{=} \mathcal{M}_2'$.
- $*$ Let $\overline{\alpha}_x = \alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \cdot \texttt{jmpOut!}(k_x^{(n_x)};\mathcal{R}')$. By definition of $\beta = \texttt{jmpOut!}(\Delta t;\mathcal{R}')$:

$$t_1' = t_1^{(0)} + \Delta t + \sum_{(i_1,j_1)\in|\mathbb{I}_{\overline{\alpha}_1}|} (t_1^{(j_1)} - t_1^{(i_1+1)})$$

$$t_2' = t_2^{(0)} + \Delta t + \sum_{(i_2,j_2)\in|\mathbb{I}_{\overline{\alpha}_2}|} (t_2^{(j_2)} - t_2^{(i_2+1)})$$

But $t_1^{(0)} = t_2^{(0)}$ since $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$. Also, each operand in $(t_1^{(j_1)} - t_1^{(i_1+1)})$ equals the corresponding $(t_2^{(j_2)} - t_2^{(i_2+1)})$ because for each (p^{th} element) $(i_1, j_1) \in \mathbb{I}_{\overline{\alpha}_1}$ and corresponding $(i_2, j_2) \in \mathbb{I}_{\overline{\alpha}_2}$, Property A.16 guarantees that $t_1^{(i_1+1)} = t_2^{(i_2+1)}$ and Property A.15 guarantees that $t_1^{(j_1)} = t_2^{(j_2)}$.
- $*$ Finally, since no interaction with $\mathcal{D}$ via INor OUToccurs in protected mode and since the same deterministic device performed the same number of steps (starting from $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$), it follows that $t_{a_1}' = t_{a_2}'$ and $\delta_1' = \delta_2'$.

□

A.7 Proofs of Lemmata 6.9 and 6.10 of Section 6.2.2

Proposition A.6. *Let $C = \langle \mathcal{M}_C, \mathcal{D}\rangle$. If $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}^* c_1$ and $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}^* c_2$, then $c_1 \vdash_{mode}$ m and $c_2 \vdash_{mode}$ m.*

Proof. Let β the last observable of $\overline{\beta}$. By definition c_1 and c_2 are such that, for some c_1' and c_2':

$$\mathcal{D} \vdash c_1' \overset{\alpha}{\Longrightarrow} c_1 \qquad \mathcal{D} \vdash c_2' \overset{\alpha}{\Longrightarrow} c_2$$

with α equal to $\bullet$, $\texttt{jmpIn?}(\cdot)$ or $\texttt{jmpOut!}(\cdot;\cdot)$ (depending on the value of β). In either case, since c_1' and c_1' are the configuration *right after* α and by definition of fine-grained traces, we have $c_1 \vdash_{mode}$ m and $c_2 \vdash_{mode}$ m. □

Proposition A.7. *For any context $C = \langle \mathcal{M}_C, \mathcal{D}\rangle$ and module $\mathcal{M}_M$, if $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xrightarrow{\beta_0 \cdots \beta_n}^* c$ with $n \geq 0$, then the observables occurring*

(i) in even positions ($\beta_0, \beta_2, \dots$) are either $\bullet$ or $\texttt{jmpIn?}(\mathcal{R})$ (for some $\mathcal{R}$)
(ii) in odd positions ($\beta_1, \beta_3, \dots$) are either $\bullet$ or $\texttt{jmpOut!}(\Delta t;\mathcal{R})$ (for some Δt and $\mathcal{R}$)

Proof. Both easily follow from Figures 8 and 9. □

First, we show that, due to the mitigation, the behavior of the context does not depend on the behavior of the module:

Lemma 6.9. *Let $C = \langle \mathcal{M}_C, \mathcal{D}\rangle$. If $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}^* c_1 \overset{\beta}{\Longrightarrow} c_1'$, $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}^* c_2$, $c_1 \vdash_{mode}$ UM and $c_2 \vdash_{mode}$ UM, then there exists c_2' such that $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow} c_2'$.*

Proof. First, observe that $\text{INIT}_{C[\mathcal{M}_M]} \overset{U}{\approx} \text{INIT}_{C[\mathcal{M}_{M'}]}$, because

$$\text{INIT}_{C[\mathcal{M}_M]} = \langle \delta_{\text{init}}, 0, \bot, \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{R}^{init}_{\mathcal{M}_C}, \texttt{0xFFFE}, \bot\rangle$$

$$\text{INIT}_{C[\mathcal{M}_{M'}]} = \langle \delta_{\text{init}}, 0, \bot, \mathcal{M}_C \uplus \mathcal{M}_{M'}, \mathcal{R}^{init}_{\mathcal{M}_C}, \texttt{0xFFFE}, \bot\rangle.$$

Since $\mathrm{INIT}_{C[\mathcal{M}_M]} \vdash_{mode} \mathsf{UM}$, $\mathrm{INIT}_{C[\mathcal{M}_M]} \overset{U}{\approx} \mathrm{INIT}_{C[\mathcal{M}_{M'}]}$, $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1$, $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2$, $c_1 \vdash_{mode} \mathsf{UM}$ and $c_2 \vdash_{mode} \mathsf{UM}$, by Property A.20 we have $c_1 \overset{U}{\approx} c_2$. Finally, since $\mathcal{D} \vdash c_1 \xRightarrow{\beta} c_1'$ and by Property A.19 we get $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$. □

Then the following lemma shows that the isolation mechanism offered by the enclave guarantees that the behavior of the module is not influenced by the one of the context:

Lemma 6.10. *Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$. If $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1'' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}_1)} c_1 \xRightarrow{\beta} c_1'$ and $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2'' \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}_2)} c_2$, then there exists c_2' such that $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$.*

Proof. Noting that $c_1 \vdash_{mode} \mathsf{PM}$ and that the last observable of $\overline{\beta}$ is a $\mathtt{jmpIn?}(\cdot)$, by definition of coarse-grained traces (see Figure 9) we have the following fine-grained traces starting from c_1'':

$$\mathcal{D} \vdash c_1'' \xRightarrow{\xi \cdots \xi \cdot \mathtt{jmpIn?}(\mathcal{R}_1)}{}^{*} c_1 \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \xRightarrow{\tau(k_1^{(n_1)}) \ \cdots \ \tau(k_1^{(n_1+m_1-1)}) \cdot \overline{\alpha}_1'}{}^{*} c_1'$$

with $\overline{\alpha}_1' \in \{\mathtt{jmpOut!}(k_1; \mathcal{R}_1'), \mathtt{handle!}(k_1) \cdot \xi \cdots \xi \cdot \bullet\}$.

Similarly for c_2 it must be:

$$\mathcal{D} \vdash c_2'' \xRightarrow{\xi \cdots \xi \cdot \mathtt{jmpIn?}(\mathcal{R}_2)}{}^{*} c_2 \xRightarrow{\overline{\alpha}_2}{}^{*} c_2^{(n_2)} \xRightarrow{\tau(k_2^{(n_2)}) \ \cdots \ \tau(k_1^{(n_2+m_2-1)}) \cdot \overline{\alpha}_2'}{}^{*} c_2'.$$

with $\overline{\alpha}_2' \in \{\mathtt{jmpOut!}(k_2; \mathcal{R}_2'), \mathtt{handle!}(k_2) \cdot \xi \cdots \xi \cdot \bullet\}$.

We have now two cases:

- *Case $\beta = \mathtt{jmpOut!}(\Delta t; \mathcal{R})$.* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ implies the existence of a context $C' = \langle \mathcal{M}_{C'}, \mathcal{D}' \rangle$ that allow us to observe $\mathcal{D}' \vdash \mathrm{INIT}_{C'[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}} c_3 \xRightarrow{\beta} c_3'$, i.e.,

  $$\mathcal{D}' \vdash c_3 \xRightarrow{\overline{\alpha}_3}{}^{*} c_3^{(n_3)} \xRightarrow{\tau(k_3^{(n_3)}) \ \cdots \ \tau(k_3^{(n_3+m_3-1)}) \cdot \overline{\alpha}_3'} c_3'$$

  with $\overline{\alpha}_3' \in \{\mathtt{jmpOut!}(k_3; \mathcal{R}_3'), \mathtt{handle!}(k_3) \cdot \xi \cdots \xi \cdot \bullet\}$.

  By Properties A.10 and A.11 we have that $c_2 \overset{P}{\approx} c_3$, and by Property A.9 we conclude that $c_3^{(n_3)} \overset{P}{\approx} c_2^{(n_2)}$.

  Property A.8 guarantees that

  $$\tau(k_2^{(n_2)}) \ \cdots \ \tau(k_2^{(n_2+m_2-1)}) \cdot \overline{\alpha}_2' = \tau(k_3^{(n_3)}) \ \cdots \ \tau(k_3^{(n_3+m_3-1)}) \cdot \overline{\alpha}_3'.$$

  Since $\overline{\alpha}_2' = \overline{\alpha}_3' = \mathtt{jmpOut!}(k_3; \mathcal{R}_1)$, we know that $\mathcal{D} \vdash c_2^{(n_2)} \xRightarrow{\mathtt{jmpOut!}(\Delta t'; \mathcal{R}_1)} c_2'$.

  By Property 6.1, we have

  $$\Delta t = \sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) + (11 + \mathsf{MAX\_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}_1}|$$

  $$\Delta t' = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) + (11 + \mathsf{MAX\_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}_2}|.$$

  Since by Properties A.17 and A.18 we have $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ and $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$, we get $\Delta t = \Delta t'$ as requested.
- *Case $\beta = \bullet$.* It must be that $\overline{\alpha}_1' = \mathtt{handle!}(k_1) \cdot \xi \cdots \xi \cdot \bullet$ and $\overline{\alpha}_2' = \mathtt{handle!}(k_2) \cdot \xi \cdots \xi \cdot \bullet$. If this was not the case (i.e., if $\overline{\alpha}_2' = \mathtt{jmpOut!}(k_2; \mathcal{R}_2')$), then c_2 could be swapped with c_1 (and c_1 with c_2) in the the statement of this Lemma and the previous case would apply. Thus, the thesis follows.

□

A.8 Proof of Property 6.2 and Algorithm 2 of Section 6.2.3

From now onwards, we simply write $\beta = \varepsilon$ (resp. $\beta' = \varepsilon$) if $\overline{\beta}$ (resp. $\overline{\beta}'$) is shorter than $\overline{\beta}'$ (resp. $\overline{\beta}$).

Property 6.2. *If $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are two modules such that $\mathcal{M}_M \not\eqsim^{\mathrm{L}} \mathcal{M}_{M'}$, then there always exist $\overline{\beta}$ and $\overline{\beta}'$ that are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$.*

Proof. From the contrapositive of Lemma 6.12 we know that $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, i.e., there exist $\overline{\beta} \in Tr(\mathcal{M}_M)$ and $\overline{\beta}' \in Tr(\mathcal{M}_{M'})$ such that $\overline{\beta} \notin Tr(\mathcal{M}_{M'})$ and $\overline{\beta} \in Tr(\mathcal{M}_M)$. Also, since $\mathcal{M}_M \not\eqsim^{\mathrm{L}} \mathcal{M}_{M'}$, we have that there exists a context C^L such that $C^L[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathrm{L}}$ (or vice versa) — assume wlog $C^L[\mathcal{M}_M]\Downarrow^{\mathrm{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathrm{L}}$.

Thus, by Proposition 6.8:

$$\mathcal{D}^L \vdash \mathrm{INIT}_{C^L[\mathcal{M}_M]} \overset{\overline{\beta}''}{\Longrightarrow}{}^{*} \mathrm{HALT}$$

$$\mathcal{D}^L \vdash \mathrm{INIT}_{C^L[\mathcal{M}_{M'}]} \overset{\overline{\beta}'''}{\Longrightarrow}{}^{*} c \neq \mathrm{HALT}$$

for some $\overline{\beta}''$ (ending in $\bullet$), c and for all $\overline{\beta}'''$ that can be observed.
Indeed, we can always write that $\overline{\beta}'' = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}''' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$ where:

- $\overline{\beta}_s$ is the longest (possibly empty) common prefix of the two traces
- β and $\beta' \neq \bullet$ are the first different observables – one of the two may be ε or, by Proposition 6.8, it may be $\beta = \bullet$
- $\overline{\beta}_e$ and $\overline{\beta}'_e$ are the (possibly empty) remainders of the two traces

Thus, since $\overline{\beta}''$ and $\overline{\beta}'''$ are also observed under the same context C^L, they are distinguishing traces. □

The first two parameters of BuildDevice – *joutd* and *joutd′* – are differentiating `jmpOut!`$(\cdot;\cdot)$ addresses (if any), as returned by the BuildMem (Algorithm 1). Parameters $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ generated under the context C^L (cf. Definition 6.13). Finally, *term* (resp. *term′*) denotes whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode.

The first two lines define the initial set of states, which will be a finite subset of $\mathbb{N}$ in the end, and the initial *empty* transition function.

Line 7 defines δ_L that records the last state that was added to the I/O device. At the beginning it is initialized to 0.

The algorithm then proceeds by iterating over all the observables in $\overline{\beta}_s$ (all the steps below also update Δ and δ_L, but we omit to state it explicitly):

- *Case* $\beta_i = \beta'_i =$ `jmpIn?`$(\mathcal{R})$. In this case we know that either this is the first observable or previous one was a `jmpOut!`$(\cdot;\cdot)$. Since the memory is obtained following Algorithm 1, we know that in both cases we reach the instruction `IN pc` (either at address `A_EP` or those of jumps out of protected mode), waiting for the next program counter (sometimes before that we perform a write, which shall be ignored). Thus, the device ignores any write operation and replies with `A_JIN` (line 12). Then it starts to send the values of the registers in $\mathcal{R}$, so to simulate in Sancus$^{\mathrm{H}}$ what happens in Sancus$^{\mathrm{L}}$ and to match the requests from the code. To help the intuition Figure 11a depicts how the transition function looks after the update (the solid black state denotes the new value of δ_L).

Algorithm 2 Builds the device of the distinguishing context.

```
procedure BuildDevice($joutd, joutd', \overline{\beta} = \beta_0 \cdots \beta_{n-1} \cdot \beta \cdot \overline{\beta}_e, \overline{\beta}' = \beta_0 \cdots \beta_{n-1} \cdot \beta' \cdot \overline{\beta}'_e, term, term', C^L$)
        ▷ $joutd, joutd'$ are differentiating jmpOut!(·; ·) addresses, if any
        ▷ $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces generated by the context $C^L$
        ▷ $term$ (resp. $term'$) denotes whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode
    $\Delta = \{0\}$
    $\overset{\cdot}{\rightsquigarrow}_D = \emptyset$
    $\delta_L = 0$                    ▷ This variable keeps track of the last added device state.
    for $i \in 0..n-1$ do
        if $\beta_i =$ jmpIn?($\mathcal{R}$) then
            $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 17\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, wr(w), \delta_L) \mid w \in Word\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A\_JIN}), \delta_L + 1)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 1, rd(\mathcal{R}[\texttt{sp}]), \delta_L + 2)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\mathcal{R}[\texttt{sr}]), \delta_L + 3)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + i, rd(\mathcal{R}[\texttt{i}]), \delta_L + i + 1) \mid 3 \leq i \leq 15\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 16, rd(\mathcal{R}[\texttt{pc}]), \delta_L + 17)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 16\}$
            $\delta_L = \delta_L + 17$
        else if $\beta_i =$ jmpOut!($\Delta t;\mathcal{R}$) then
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, \epsilon, \delta_L)\} \cup \{(\delta_L, wr(w), \delta_L) \mid w \in Word\}$
        end if
    end for
    if $\beta =$ jmpOut!($\Delta t;\mathcal{R}$) $\wedge \beta' =$ jmpOut!($\Delta t';\mathcal{R}'$) $\wedge (\exists \texttt{r}. \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$ then
        if r ≠ pc then
            $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 4\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A\_RDIFF}), \delta_L + 1)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 1, wr(\mathcal{R}[\texttt{pc}]), \delta_L + 2)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 1, wr(\mathcal{R}'[\texttt{pc}]), \delta_L + 3)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\texttt{A\_HALT}), \delta_L + 4)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 3, rd(\texttt{A\_LOOP}), \delta_L + 4)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 3\}$
            $\delta_L = \delta_L + 4$
        else
            $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 3\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, wr(joutd), \delta_L + 1)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L, wr(joutd'), \delta_L + 2)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 1, rd(\texttt{A\_HALT}), \delta_L + 3)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\texttt{A\_LOOP}), \delta_L + 3)\}$
            $\overset{\cdot}{\rightsquigarrow}_D = \overset{\cdot}{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 2\}$
            $\delta_L = \delta_L + 3$
        end if
        continues ...
```

```
43:             ... continued
44:         else if $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}) \wedge \Delta t \neq \Delta t'$ then
45:                 ▷ Let $\mathcal{D}^L \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s}{}^* c_1$ and $\mathcal{D}^L{}_I \vdash c_1 \xRightarrow{\texttt{jmpOut!}(\Delta t_I; \mathcal{R})} c_1'$.
46:                 ▷ Let $\mathcal{D}^L \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s}{}^* c_2$ and $\mathcal{D}^L{}_I \vdash c_2 \xRightarrow{\texttt{jmpOut!}(\Delta t'_I; \mathcal{R})} c_2'$.
47:             $t = t_1' - t_1$
48:             $t' = t_2' - t_2$
49:             $\Delta = \Delta \cup \{\delta_L + 1, \dots, \delta_L + max(t, t') + 1\}$
50:             $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + min(t, t'), rd(\texttt{A\_HALT}), \delta_L + max(t, t') + 1)\}$
51:             $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + max(t, t'), rd(\texttt{A\_LOOP}), \delta_L + max(t, t') + 1))\}$
52:             $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + k, \epsilon, \delta_L + k + 1) \mid 0 \leq k \leq max(i, i')\}$
53:             $\delta_L = \delta_L + max(t, t') + 1$
54:         else if $\beta = \bullet \wedge \beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ then
55:             if $term$ then
56:                 $\Delta = \Delta \cup \{\delta_L + 1, \dots, \delta_L + 2\}$
57:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(\texttt{A\_EP}), \delta_L + 1)\}$
58:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 1, rd(\texttt{A\_HALT}), \delta_L + 2)\}$
59:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A\_LOOP}), \delta_L + 2)\}$
60:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(w), \delta_L) \mid w \in Word \setminus \{\texttt{A\_EP}\}\}$
61:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 1\}$
62:                 $\delta_L = \delta_L + 2$
63:             else
64:                 $\Delta = \Delta \cup \{\delta_L + 1\}$
65:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A\_HALT}), \delta_L + 1)\}$
66:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(w), \delta_L) \mid w \in Word\}$
67:                 $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, \epsilon, \delta_L)\}$
68:                 $\delta_L = \delta_L + 2$
69:             end if
70:         else if $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \varepsilon$ then
71:                 ▷ As the previous case, with $term'$ in place of $term$.
72:         else
73:             return $\perp$
74:         end if
75:         $\mathcal{D} = \langle \Delta, 0, \dot{\rightsquigarrow}_D \rangle$
76:         return $\mathcal{D}$
77: end procedure
```

- *Case* $\beta_i = \beta_i' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$. The device is simply updated with a loop on δ_L with action ϵ and ignores any write operation (so as to deal with $\mathcal{R}[\texttt{pc}] = joutd$ or $\mathcal{R}[\texttt{pc}] = joutd'$). Figure 11b pictorially represents this case.

Then, when $\overline{\beta}_s$ ends, the algorithm analyses β and β' and sets up the device to differentiate the two modules:

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge (\exists \texttt{r}. \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$. In this case the differentiation is due to a register, and two further sub-cases may arise, depending on whether it is pc. If the register is pc then the device waits for the differentiating value for the context (that is executing code at *joutd* and *joutd'* by construction) and based on that value, it replies with either `A_HALT` (line 37) or `A_LOOP` (line 38). Instead, if the differentiation register is not pc then the code of the context is waiting for the next program counter and the context replies with `A_RDIFF`. From this address we find the code that sends the differentiating register and, based on that value, the device replies with either `A_HALT` (line 29) or `A_LOOP` (line 30). Figures 12a and 12b may help the intuition.

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \land \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}) \land \Delta t \neq \Delta t'$. This case is probably the most interesting since differentiation happens in Sancus^{L} due to timings. However, different timings in Sancus^{L} correspond to different timings in Sancus^{H} (as observed in proof of Property A.22), and the device is programmed to reply with either `A_HALT` (line 50) or `A_LOOP` (line 51) depending on the time value. Figure 12c intuitively depicts this situation.
- *Case* $\beta = \bullet \land \beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$. In this case $\bullet$ may occur during an interrupt service routine. We then have two sub-cases, depending on whether the first module terminates when executed in a context with no interrupts after the last jump into protected mode or not (i.e., encoded by the value of *term*). When *term* holds, the first module makes the CPU go through an exception handling configuration that jumps to `A_EP` and the device instructs the code to jump to `A_HALT` (line 58), while for the second module the CPU jumps to any other location (`A_EP` is chosen to be different from any other jump out address!) and is instructed to jump to `A_LOOP` (line 59). When *term* does not hold, the first module diverges, while for the second module the CPU jumps to a location in unprotected code and it is instructed to jump to `A_HALT` (line 65). Figures 12d and 12e may help the intuition.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \land \beta' = \varepsilon$. Analogous to the previous case.
- *Otherwise.* No other cases may arise, as noted in Property A.21.

Finally, the algorithm returns a device with the set of states Δ, the initial state 0 and the transition function built as just explained.

Property A.21. *Let* $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, $\overline{\beta}, \overline{\beta}'$ *be distinguishing traces of* $\mathcal{M}_M$ *and* $\mathcal{M}_{M'}$ *originated by some context* C^L *and let term and term' be any pair of booleans, then*
$\mathcal{D} = \textsc{BuildDevice}(\overline{\beta}, \overline{\beta}', joutd, joutd', term, term', C^L) \neq \bot$ *and* $\mathcal{D}$ *is an I/O device.*

Proof. We first show that BuildDevice never returns $\bot$ when $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces. For that, let $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$, and note that the only cases for which $\bot$ is returned are the following:

- *Case* $\beta = \beta' = \bullet$. Since $\beta \neq \beta'$ by hypothesis, this case never happens.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ *and* $\beta' = \texttt{jmpIn?}(\mathcal{R}')$ *(or vice versa).* This case never happens due to Proposition A.7.
- *Case* $\{\bullet, \texttt{jmpIn?}(\mathcal{R})\} \ni \beta \neq \beta' \in \{\bullet, \texttt{jmpIn?}(\mathcal{R}')\}$. Roughly, this means that the *same* context performed two different actions upon observation of the same trace ($\overline{\beta}_s$). Formally, we know by hypothesis that for the context $C^L = \langle \mathcal{M}_C, \mathcal{D}^L \rangle$

$$\mathcal{D}^L \vdash \mathrm{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s}{}^{*} c_1$$

$$\mathcal{D}^L \vdash \mathrm{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s}{}^{*} c_2.$$

with $c_1 \vdash_{mode} \mathsf{UM}$ and $c_2 \vdash_{mode} \mathsf{UM}$. Property A.20 guarantees that $c_1 \overset{U}{\approx} c_2$, thus by Property A.19 the same observable must originate from both c_1 and c_2, but that is against the hypothesis that $\beta \neq \beta'$.

Finally, it is easy to see that $\mathcal{D}$ returned by BuildDevice is an actual device. Indeed, its set of states Δ is finite (the algorithm always terminates in a finite number of steps and each step adds a finite number of state); its initial state 0 belongs to Δ; no *int*? transitions are ever added and a single $rd(w)$ transition outgoes from any given state: thus the transition relation respects the definition of I/O devices. □

The following property states that the context built by joining together the results of the two algorithms above is a distinguishing one:

Property A.22. *Let $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$; let $C^L = \langle \mathcal{M}_C, \mathcal{D}^L \rangle$; let*

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s}{}^* c_1' \xRightarrow{\beta} c_1$$

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s}{}^* c_2' \xRightarrow{\beta'} c_2$$

be such that $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}_e$ distinguishing traces of $\mathcal{M}_M$ and $\mathcal{M}_{M'}$; and let

$$term \iff \mathcal{D}^L{}_I \vdash c_1' \rightarrow^* \text{HALT}$$

$$term' \iff \mathcal{D}^L{}_I \vdash c_2' \rightarrow^* \text{HALT}.$$

If $(\mathcal{M}_C, joutd, joutd') = $ BuildMem$(\overline{\beta}, \overline{\beta}')$, $\mathcal{D} = $ BuildDevice$(\overline{\beta}, \overline{\beta}', joutd, joutd', term, term')$ and $C^H = \langle \mathcal{M}_C, \mathcal{D} \rangle$, then $C^H[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$ and $C^H[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{H}}$ (or vice versa).

Proof. Assume wlog that $C^L[\mathcal{M}_M]\Downarrow^{\mathsf{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{L}}$. By Lemma 6.5

$$C^H[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C^H{}_I[\mathcal{M}_M]\Downarrow^{\mathsf{L}} \quad \text{and} \quad C^H[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}} \iff C^H{}_I[\mathcal{M}_{M'}]\Downarrow^{\mathsf{L}}$$

It suffices thus proving that $C^H{}_I$ distinguishes $\mathcal{M}_M$ and $\mathcal{M}_{M'}$, i.e., $C^H{}_I[\mathcal{M}_M]\Downarrow^{\mathsf{L}}$ and $C^H{}_I[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{L}}$ or vice versa.

We show by induction on the length $2n + 1$ of $\overline{\beta}_s$ that if

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s}{}^* c_1'$$

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s}{}^* c_2'$$

then $\exists \overline{\beta}_s'$ s.t.

$$D^H{}_I \vdash \text{INIT}_{C^H{}_I[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'}{}^* c_3 \text{ and}$$

$$D^H{}_I \vdash \text{INIT}_{C^H{}_I[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'}{}^* c_4 \text{ with } \overline{\beta}_s' \approx \overline{\beta}_s \text{ (see Definition A.4).}$$

Note that the length of $\overline{\beta}_s$ must be odd as a consequence of Properties A.20 and A.19 and no $\bullet$ appears in it since otherwise it would mean that $\overline{\beta} = \overline{\beta}'$.

- *Case* $n = 0$. Then, $\overline{\beta}_s$ is `jmpIn?`$(\mathcal{R})$. Thus, Algorithm 1 guarantees that the current instruction is `IN pc` (at address `A_EP`) and its execution leads to address `A_JIN` (by Algorithm 2) and the same `jmpIn?`$(\mathcal{R})$ is observed starting from both $\text{INIT}_{C^H{}_I[\mathcal{M}_M]}$ and $\text{INIT}_{C^H{}_I[\mathcal{M}_{M'}]}$ and also $\overline{\beta}_s' \approx \overline{\beta}_s$.

- *Case $n = n' + 1$.* If

$$\mathcal{D}^L \vdash \mathrm{INIT}_{\mathcal{C}^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_1'' \wedge \mathcal{D}^L \vdash \mathrm{INIT}_{\mathcal{C}^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_2'''$$
$$\Downarrow$$
$$D^H{}_{I} \vdash \mathrm{INIT}_{\mathcal{C}^H{}_{I}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_3' \wedge D^H{}_{I} \vdash \mathrm{INIT}_{\mathcal{C}^H{}_{I}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_4' \ \wedge \overline{\beta}_s''' \approx \overline{\beta}_s'' \text{ (IHP)}$$

then

$$\mathcal{D}^L \vdash \mathrm{INIT}_{\mathcal{C}^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_1'' \xRightarrow{\overline{\beta}''}{}^{*} c_1' \wedge \mathcal{D}^L \vdash \mathrm{INIT}_{\mathcal{C}^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}''}{}^{*} c_2'' \xRightarrow{\overline{\beta}''}{}^{*} c_2'$$
$$\Downarrow$$
$$D^H{}_{I} \vdash \mathrm{INIT}_{\mathcal{C}^H{}_{I}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_3' \xRightarrow{\overline{\beta}'''}{}^{*} c_3 \wedge D^H{}_{I} \vdash \mathrm{INIT}_{\mathcal{C}^H{}_{I}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_4' \xRightarrow{\overline{\beta}'''}{}^{*} c_4 \ \wedge \overline{\beta}_s''' \cdot \overline{\beta}''' \approx \overline{\beta}_s'' \cdot \overline{\beta}''.$$

Note that it must be that $\overline{\beta}'' = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \cdot \texttt{jmpIn?}(\mathcal{R}')$ by Proposition A.7 and because we never observe $\bullet$ in the common prefix. By (IHP) and Property A.11 we have $c_1'' \overset{P}{\approx} c_3'$ and $c_2'' \overset{P}{\approx} c_4'$. Thus, by Properties A.9 and A.8, it must be that $\texttt{jmpOut!}(\Delta t'; \mathcal{R})$ is observed when starting in c_3' and $\texttt{jmpOut!}(\Delta t''; \mathcal{R})$ is observed when starting in c_4' (for some $\Delta t'$ and $\Delta t''$). By definition of coarse-grained traces, each of the computations above is generated by fine-grained trace in the form (we write _ to denote a generic configuration):

$$\mathcal{D}^L \vdash \_ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_1'' = c_1^{(0)} \xRightarrow{\alpha_1^{(0)}} \cdots \xRightarrow{\alpha_1^{(n_1-1)}} c_1^{(n_1)} \xRightarrow{\texttt{jmpOut!}(k_1^{(n_1)}; \mathcal{R})} c^{(n_1)+1} \xRightarrow{\xi \cdots \xi \texttt{jmpIn?}(\mathcal{R}')}{}^{*} c_1'$$
$$\mathcal{D}^L \vdash \_ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_2'' = c_2^{(0)} \xRightarrow{\alpha_2^{(0)}} \cdots \xRightarrow{\alpha_2^{(n_2-1)}} c_2^{(n_2)} \xRightarrow{\texttt{jmpOut!}(k_2^{(n_2)}; \mathcal{R})} c^{(n_2)+1} \xRightarrow{\xi \cdots \xi \texttt{jmpIn?}(\mathcal{R}')}{}^{*} c_2'$$
$$D^H{}_{I} \vdash \_ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_3' = c_3^{(0)} \xRightarrow{\alpha_3^{(0)}} \cdots \xRightarrow{\alpha_3^{(n_3-1)}} c_3^{(n_3)} \xRightarrow{\texttt{jmpOut!}(k_3^{(n_3)}; \mathcal{R})} c_3^{(n_3+1)}$$
$$D^H{}_{I} \vdash \_ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_4' = c_4^{(0)} \xRightarrow{\alpha_4^{(0)}} \cdots \xRightarrow{\alpha_4^{(n_4-1)}} c_4^{(n_4)} \xRightarrow{\texttt{jmpOut!}(k_4^{(n_4)}; \mathcal{R})} c_4^{(n_4+1)}.$$

Thus, due to Property 6.1 and by hypothesis, it holds that $\Delta t = \sum_{i=0}^{n_1} \gamma(c_1^{(i)}) + (11 + \texttt{MAX\_TIME}) \cdot |\mathbb{I}_{\alpha_1^{(0)} \cdots \alpha_1^{(n_1)}}| = \sum_{i=0}^{n_2} \gamma(c_2^{(i)}) + (11 + \texttt{MAX\_TIME}) \cdot |\mathbb{I}_{\alpha_2^{(0)} \cdots \alpha_2^{(n_2)}}|$. Also, since by (IHP) and Properties A.20 and A.19 it follows that $c_1^{(0)} = c_1'' \overset{U}{\approx} c_2'' = c_2^{(0)}$, we know $|\mathbb{I}_{\alpha_1^{(0)} \cdots \alpha_1^{(n_1)}}| = |\mathbb{I}_{\alpha_2^{(0)} \cdots \alpha_2^{(n_2)}}|$ (by Property A.18) and thus $\sum_{i=0}^{n_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2} \gamma(c_2^{(i)})$. Moreover, by (IHP) and Property A.11, we get $c_1^{(0)} = c_1'' \overset{P}{\approx} c_3' = c_3^{(0)}$ and $c_2^{(0)} = c_2'' \overset{P}{\approx} c_4' = c_4^{(0)}$. Now, as a consequence of Properties A.3, A.9 and A.8 we know that $\Delta t' = \sum_{i=0}^{n_3} \gamma(c_3^{(i)}) = \sum_{i=0}^{n_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2} \gamma(c_2^{(i)}) = \sum_{i=0}^{n_3} \gamma(c_3^{(i)}) = \Delta t''$. By (IHP) and since the first observable after c_3' and c_4' is the same, by Property A.20 it follows $c_3^{(n_3+1)} \overset{U}{\approx} c_4^{(n_4+1)}$. Thus, due to Property A.19, we get that the same coarse-grained observable $\texttt{jmpIn?}(\mathcal{R}''')$ is observed after $c_3^{(n_3+1)}$ and $c_4^{(n_4+1)}$. Finally, $\mathcal{R}'''$ is equal to $\mathcal{R}'$ since after any $\texttt{jmpOut!}(\cdot; \cdot)$ a `IN pc` instruction is executed and its execution leads to address `A_JIN` (by Algorithm 2) that performs $\texttt{jmpIn?}(\mathcal{R})$, and the thesis follows.

Since we proved that

$$D^H{}_{\not I} \vdash \mathrm{INIT}_{C^H{}_{\not I}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}'_s}{}^{*} c_3 \text{ and}$$
$$D^H{}_{\not I} \vdash \mathrm{INIT}_{C^H{}_{\not I}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}'_s}{}^{*} c_4$$

we also have that $c_3 \overset{U}{\approx} c_4$ by Properties A.20 and A.19.

Let $D^H{}_{\not I} \vdash c_3 \xRightarrow{\overline{\beta}_3}{}^{*} c''_3$ and $D^H{}_{\not I} \vdash c_4 \xRightarrow{\overline{\beta}_4}{}^{*} c''_4$, with $\overline{\beta}_3$ and $\overline{\beta}_4$ either empty or made of a single observable (either $\bullet$ or $\texttt{jmpOut!}(\cdot;\cdot)$, since no difference cannot be observed upon $\texttt{jmpIn?}(\cdot)$ as observed above). By exhaustive cases on β and β' we have:

- *Case* $\beta = \bullet$ *and* $\beta' = \texttt{jmpOut!}(\Delta t''';\mathcal{R}'')$. Note that, since $term \iff \mathcal{D}^L{}_{\not I} \vdash c'_1 \rightarrow^{*} \mathrm{HALT}$ and $c'_1 \overset{P}{\approx} c_3$ (by Properties A.10 and A.11), we get $term \iff \mathcal{D}^H{}_{\not I} \vdash c_3 \rightarrow^{*} \mathrm{HALT}$ by Property A.8 and since neither $\mathcal{D}^L{}_{\not I}$ nor $\mathcal{D}^H{}_{\not I}$ raise any interrupt. Thus, by definition of $\mathcal{D}^L$ (cf. Algorithm 2) the context C^H distinguishes the two modules.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t''';\mathcal{R}'')$ *and* $\beta' = \varepsilon$. Similar to the previous case (with *term'* in place of *term*).
- *Case* $\beta = \texttt{jmpOut!}(\Delta t''';\mathcal{R}'')$ *and* $\beta' = \texttt{jmpOut!}(\Delta t''';\mathcal{R}''')$ *with* $\mathcal{R}'' \neq \mathcal{R}'''$. Since $c'_1 \overset{P}{\approx} c_3$ and $c'_2 \overset{P}{\approx} c_4$, it must be that $\overline{\beta}_3 = \texttt{jmpOut!}(\Delta t^{v};\mathcal{R}'')$ and $\overline{\beta}_4 = \texttt{jmpOut!}(\Delta t^{vi};\mathcal{R}'')$. Thus, by Algorithms 1 and 2, C^H distinguishes the two modules.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t''';\mathcal{R}'')$ *and* $\beta' = \texttt{jmpOut!}(\Delta t^{iv};\mathcal{R}'')$. In this case it holds that $\overline{\beta}_3 = \texttt{jmpOut!}(\Delta t^{v};\mathcal{R}'')$ and $\overline{\beta}_4 = \texttt{jmpOut!}(\Delta t^{vi};\mathcal{R}'')$ with the same timings of the instructions (by Property 6.1). Since $c_3 \overset{U}{\approx} c_4$, the two times must differ one from each other otherwise, by the counterpositive of Property A.17, we would get $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. Again, by definition of Algorithms 1 and 2, one computation converges and one diverges, hence C^H distinguishes the two modules.

□

A.9 Proofs and additional definitions of Section 7

Property A.23. *Let c and c' be configurations such that $c, c' \vdash_{mode}$ UM. If $c \overset{U}{\approx} c'$ then $c \overset{L}{=} c'$.*

Proof. Since $c, c' \vdash_{mode}$ UM, the property follows directly from Definitions A.3 and 7.1. □

Lemma 7.3. *If $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.*

Proof. Assuming contextual equivalence in Sancus$^{\mathrm{L}}$ and that:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c \rightarrow \mathrm{HALT} \ \wedge\ \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \rightarrow^{*} c' \rightarrow \mathrm{HALT},$$

our goal is to prove that $c \overset{L}{=} c'$. From contextual equivalence it follows that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. By Lemma 6.11 we also know that for some c'':

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c \ \wedge\ \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c''.$$

Proposition A.6 and Property A.20 guarantee that $c \overset{U}{\approx} c''$. Then, since $c \rightarrow \mathrm{HALT}$, it must be $c'' \rightarrow \mathrm{HALT}$. For that and by determinism of the operational semantics of Sancus$^{\mathrm{L}}$ we have that $c' = c''$ and $c \overset{U}{\approx} c'$, which by Property A.23 implies $c \overset{L}{=} c'$. □

Theorem 7.4. *If $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.*

Proof. Since $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$, by Theorem 6.3 we also have that $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ and Lemma 7.3 concludes the proof. □

Theorem 7.6.

(1) If $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$ and (2) if $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$

Proof.

(1) Lemma 7.3 guarantees that $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$. We now set out to show that $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$ is implied by $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$ in our setting. Indeed, by definition of SSNI we can assume (wlog) that $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \rightarrow \mathrm{HALT}$, i.e., $C[\mathcal{M}_M]\Downarrow^{\mathsf{L}}$. By hypothesis it also follows that $C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{L}}$. For that and by definition of ISNI, it then follows $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

(2) By definition of SSNI it follows that for any C if $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$, then $C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}}$ and viceversa, i.e., $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}}$ which coincides with the definition of $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$. □

Theorem 7.10. *Let $\mathcal{M}_P$ be a module program, then*

(1) if $\forall \mathcal{M}_D, \mathcal{M}_{D'}.\ (\mathcal{M}_P \blacktriangleleft \mathcal{M}_D) \simeq^{\mathsf{L}} (\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'})$, then $\vdash^{\mathsf{L}}_{\mathrm{USSNI}} \mathcal{M}_P$; and

(2) if $\vdash^{\mathsf{H}}_{\mathrm{USSNI}} \mathcal{M}_P$, then $\forall \mathcal{M}_D, \mathcal{M}_{D'}.\ (\mathcal{M}_P \blacktriangleleft \mathcal{M}_D) \simeq^{\mathsf{H}} (\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'})$.

Proof.

(1) From the hypothesis and by definition of $\simeq^{\mathsf{L}}$, for any c:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]} \rightarrow^* c \rightarrow \mathrm{HALT} \Rightarrow \exists c'.\ \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'}]} \rightarrow^* c' \rightarrow \mathrm{HALT}$$

and vice versa.

Now let $\hat{c}$ and $\hat{c}'$ be a pair of configurations respecting the implication above. Note that both $\hat{c} \vdash_{mode} \mathsf{UM}$ and $\hat{c}' \vdash_{mode} \mathsf{UM}$ hold, since unprotected mode configurations are the only ones from which HALT is reachable in a single step. For the thesis to hold, it suffices to show that $\hat{c} \overset{L}{=} \hat{c}'$. Since $\mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]} \vdash_{mode} \mathsf{UM}$, $\mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]} \overset{U}{\approx} \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'}]}$, repeated applications of Property A.19 guarantee that

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]} \overset{\overline{\beta}}{\Longrightarrow}{}^* \hat{c} \rightarrow \mathrm{HALT} \wedge \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* \hat{c}' \rightarrow \mathrm{HALT}.$$

By Property A.20 it holds that $\hat{c} \overset{U}{\approx} \hat{c}'$, thus easily $\hat{c} \overset{L}{=} \hat{c}'$ because of Property A.23.

(2) By definition of USSNI it follows that for any C if $C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_D]\Downarrow^{\mathsf{H}}$, then $C[\mathcal{M}_P \blacktriangleleft \mathcal{M}_{D'}]\Downarrow^{\mathsf{H}}$ and viceversa by symmetry, i.e., $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}}$ which coincides with the definition of $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$. □

Property A.24.

(1) If $\mathcal{D} \vdash c \overset{\overline{\beta}}{\Longrightarrow}{}^* c'$ then $\exists! t, K.\ \mathcal{D} \vdash c \twoheadrightarrow^t_K c' \wedge \overline{\beta} \propto K$.

(2) If $\mathcal{D} \vdash c \twoheadrightarrow^t_K c'$ then $\exists \overline{\beta}.\ \mathcal{D} \vdash c \overset{\overline{\beta}}{\Longrightarrow}{}^* c' \wedge \overline{\beta} \propto K$.

where

$$\overline{\beta} \propto K \textit{ iff } |\overline{\beta}| = \begin{cases} K & \overline{\beta} \neq \overline{\beta}' \cdot \bullet \\ K+1 & o.w. \end{cases}$$

Proof.

(1) By determinism, there is a single computation χ from c to c' generating $\overline{\beta}$. From uniqueness of χ and by Definition 7.12, one gets existence and uniqueness of t and K.

(2) Directly follows from Definition 7.12 and Figure 9. □

Lemma 7.14.

(1) if $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ *and* *(2) if* $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$

Proof.

(1) Assuming contextual equivalence in Sancus$^{\mathsf{L}}$ and that:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^{t}_{K} c,$$

our goal is to prove that $c \overset{L}{=} c'$. From contextual equivalence it follows that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. By Lemma 6.11 we also know that for some c'':

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^{*} c \implies \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^{*} c''.$$

Proposition A.6 and Property A.20 guarantee that $c \overset{U}{\approx} c''$. By Property A.24.(1) there exist unique t and K such that

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^{t}_{K} c$$

and

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^{t}_{K} c''$$

thus by determinism of operational semantics of Sancus$^{\mathsf{L}}$, we have that $c'' = c'$ and $c \overset{U}{\approx} c'$, which by Property A.23 implies $c \overset{L}{=} c'$.

(2) Suppose that $C[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}}$ and $C[\mathcal{M}_{M'}]{\not\Downarrow}^{\mathsf{H}}$. But then they cannot be $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ (since HALT is in relation just with itself), which contradicts the hypothesis. □

Theorem 7.17. *The following relations are equivalent:*

(1) $\mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$ *(2)* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ *(3)* $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ *(4)* $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$

Proof. We only prove $(1) \iff (2)$; The other equivalences follow from Theorem 6.3.

- $(1) \Leftarrow (2)$. Since $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, by Lemma 6.11 we know that:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^{*} c \iff \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^{*} c'.$$

Thus, $\forall C. WTr(C[\mathcal{M}_M]) = WTr(C[\mathcal{M}_{M'}])$ as requested.
- $(1) \Rightarrow (2)$. Easy. □