Laboratory for Innovative Software
University of Pisa, Department of Computer Science

Project Report

Academic Year 2021-2022

Andrea Laretto	(619624)	–	a.laretto@studenti.unipi.it
Filippo Lari	(545736)	–	f.lari3@studenti.unipi.it
Vincenzo Palazzo	(619631)	–	v.palazzo1@studenti.unipi.it
Niccolò Piazzesi	(619619)	–	n.piazzesi@studenti.unipi.it
Antonio Strippoli	(625044)	–	a.strippoli1@studenti.unipi.it
Gabriele Tedeschi	(560469)	–	g.tedeschi6@studenti.unipi.it

Abstract

The aim of this project is to design, implement and analyze an `OCaml` simulator for the open-source *Sancus* platform [1], a small microprocessor that supports non-interruptible enclaved execution. Following the formal specification provided in [2], we **implemented** both Sancus^{H} and its extension supporting secure interruptibility $\textbf{Sancus}^{\textbf{L}}$. A **property-based testing** framework was employed to check for the **correctness** of the implementation, **model a formal proof** of the *full abstraction* between Sancus^{H} and $\textbf{Sancus}^{\textbf{L}}$ and experiment with different extensions.

1 Introduction

We developed an `OCaml` implementation of Sancus$^{\mathrm{H}}$ and **Sancus**$^{\mathbf{L}}$, two formal models of a secure microprocessor introduced in [2]. Both mechanisms support enclaved execution, an isolation mechanism with the aim of providing a trusted execution environment. Enclaved execution is designed to be resistant against a very powerful attacker that controls all other software on the platform, including privileged system software. In addition to that, the attacker in [2] is also able to schedule *cycle-accurate* interrupts with the help of an external I/O device; this powerful capability allows the attacker to break the confidentiality guarantees of the enclave if interrupts are implemented naively. In **Sancus**$^{\mathbf{L}}$ a novel padding mechanism is introduced, and is showed to prevents interrupt based attacks. Working in this scenario, our project implementation managed to experimentally verify the following characteristics of the Sancus model:

- **Sancus**$^{\mathbf{L}}$ does not introduce any new attack with respect to Sancus$^{\mathrm{H}}$, as formally proved in [2];
- Removing the entire padding scheme from **Sancus**$^{\mathbf{L}}$ allows an attacker to fully break the abstraction;
- Restricting the padding scheme to using just the pre-padding mechanism prevents some interrupt based attacks, but it is still possible for the attacker to break the enclave;

We conclude the introduction by briefly presenting the structure of the report.
Section 2 contains an overview of the project structure, the necessary dependencies and how to build and run the test suite. Section 3 discusses the main design choices and difficulties encountered while implementing the Sancus simulator. Section 4 presents the main testing mechanisms and attacks we designed to verify the correctness and security of the implementation. Finally, Section 5 concludes the report by summarizing our experience and our learning opportunities.

2 Overview

2.1 Installation and Quick Start

The project is publicly available on GitHub at the following link:

https://github.com/iwilare/lis-2022

In order to build and run all the tests, the following dependencies are required:

- `OCaml` $\geq$ 4.13.0
- dune
- QCheck

Dependencies can be installed in two different ways namely, via opam or **esy**.

2.2 Installing through opam

If opam is already installed, simply create a 4.13 switch and install all the required dependencies with the following commands:

```
$ opam switch create 4.13.0
$ opam switch set 4.13.0
$ eval $(opam env)
$ opam install dune qcheck
```

After that, run these to download the project and run the test suite:

```
$ git clone https://github.com/iwilare/lis-2022
$ cd lis-2022
$ dune test -f
```

The flag `-f` makes sure that the test suite is always executed. It is needed because by default, dune executes it only if the project has changed since the last build, similar to how **make** targets works.

2.3 Installing through Esy

Esy is a tool used to build and run `OCaml` projects in isolation. Esy can be installed through the nodejs package manager by running `npm install -g esy`. To download the project and execute the test suite run:

```
$ git clone https://github.com/iwilare/lis-2022
$ cd lis-2022
$ esy
$ esy test
```

2.4 Project structure

We strived to maintain low-coupling between files and make the project as modular as possible. The following is a tree overview of the main files and folders of the project, along with a brief description of their content:

`lib`: contains the implementation of the simulator.	
`config.ml`	Type definition for Configuration (to describe the internal state of the entire system) and functions definitions of related utilities.
`config_monad.ml`	Definitions for the underlying state + option monad for configurations.
`cpu_mode.ml`	Definitions for the unprotected/protected mode of the CPU.
`halt_error.ml`	Definitions of possible causes for halting errors.
`instr.ml`	Definitions for the assembly language instructions.
`interrupt_logic.ml`	Implementation of the interrupt on four security levels (**Sancus^{L}**, pre-padding only, no padding, Sancus^{H}).
`io_device.ml`	Definitions for IO devices and examples of devices used for tests, e.g., an empty device, a clock device, and an on-demand interrupt device.
`layout.ml`	Definitions for the memory layout (enclave/unprotected).
`mac.ml`	Definitions for the Memory Access Control of Sancus.
`memory.ml`	Definitions for memories and related utilities on addresses.
`native.ml`	Module signature conceptualizing basic operations and overflow on native low-level datatypes and implemented by in `types.ml`.
`register_file.ml`	Definition of register files, along with write and read operations.
`semantics.ml`	Execution of individual instructions both single-step and full runs.
`serialization.ml`	Encode/decode of the instructions and registers.
`types.ml`	Byte and Word operations and module, along with overflow definitions.
`test/qcheck`: contains property-based tests.	
`enclave_map_test.ml`	Tests to prove that the attacker is capable of fully mapping instructions timing of the enclave when no padding is employed.
`generators.ml`	Definitions of generators employed in the property-based tests. We wrote generators for the memory, registers, configurations, IO devices, and simple programs.
`memory_test.ml`	Memory related tests, e.g. getting and setting memory addresses.
`registers_test.ml`	Registers related tests, on getting and setting register values and flags.
`security_test.ml`	Non-interference related tests, where various attackers and different versions of Sancus are tested against such attacks.
`semantics_test.ml`	Tests on the correct semantics of all the main instructions in Sancus.
`serialization_test.ml`	Encode/decode related tests, e.g. to ensure that the composition of the two procedures is the identity.
`runner.ml`	QCheck2 collection of all the above tests.

3 Challenges and Choices

3.1 Understanding the paper

We started by delving deeper in the contents of the paper, which was an immediate follow-up to the lectures taught in-class about the work.
The entirety of the team started with a foundation of theoretical background, including inference rules and major concepts of compilers, which allowed us to grasp the notation and technicalities of the paper. Understanding the paper required us to go though the concepts and details multiple times: even minor technicalities turned out to be crucial after later mechanisms were fully understood and implemented. The opportunity to directly communicate with one of the authors turned out to be extremely useful, as it helped us to solve doubts and speed up the whole process.

3.2 Choosing the technology stack

We had a group discussion at the beginning of the project to agree on the stack of technologies to use. The previous experiences of each team member allowed us to better evaluate all possible choices and consider the tools we thought could ease development the most, with the final decision falling on `OCaml` and its testing library **QCheck**. Reasons for this choice are manifold:

- **Familiarity**: all the team members had previous experiences with `OCaml`, while only part of us was confident enough with other viable options like Rust or Haskell;
- **Features**: `OCaml` has been used in several compiler-related projects and shown to be an excellent choice for tools such as evaluators, typecheckers and parsers. Memory safety and its powerful type system proved to be extremely useful in speeding up development and making us feel confident with the robustness of the codebase.
- **Property-based testing**: we preferred QCheck over similar libraries such as JaneStreet Quickcheck because, in our experience, it provides easier primitives to define test generators and properties and has an equally thorough documentation.

3.3 Implementation details

At the beginning of the project, we went for a side-effectful architecture of the code, with the execution of the instructions altering the configurations and memories they were given. This was primarily motivated by code that was easier to write and structure, and we thought that implementing a state-based processor inevitably implied that also using side-effects ourselves would prove *effective*[1]. This was also motivated by a lack of tools and constructs to manage monads and pure computations in `OCaml`, which we initially deemed ineffective and rather cumbersome. However, as we approached an advanced state of development, we found ourselves to be struggling with some particularly hard to find bugs related to side effects, as well as problems that arose after developing our property-based tests; for example, we were not able to check and inspect failing initial configurations generated by tests, as they were modified by the execution.

Switching to monadic effects

At the same time, we discovered some new crucial capabilities in the `OCaml` syntax that would have allowed us to better manage monadic effects than we previously thought. We then chose to switch the entire codebase from a side-effectful approach to a completely pure one using a custom state monad, combining the effects of the option monad to model failure and halting. Since we had already encapsulated much of the behaviour in well-defined modules and definitions, we managed to adapt the entire codebase in approximately 2/3 hours. For simplicity, we chose not to employ external libraries to define and manage monadic definitions and compositions.
The main syntactical advancement we employed is through the use of a combination of *customly defined lets* and *let-punning*, which allowed us to write code such as the following:

```
(* lib/config.ml *)
...
let isr c = c.layout.isr_range.range_start
let pc c = c.r.pc
let sr c = c.r.sr
let sp c = c.r.sp
...
(* lib/interrupt_logic.ml *)
...
    let@ sp in (* Obtain the values of the registers at the current time *)
    let@ sr in
    let@ pc in
    let@ isr in (* Obtain the isr address of the current configuration *)
    (* Push PC and SR in memory *)
    mset Word.(sp - from_int 2) pc >>
```

[1]Pun intended.

```
    mset Word.(sp - from_int 4) sr >>
    (* Jump to the ISR *)
    rset PC isr >>
    rset SR Word.zero >>
    rset SP Word.(sp - from_int 4) >>
    reset_last_arrival_time >>
    advance_config 6
...
```

The idea: `let@` takes a standard getter function from configurations, such as `pc` or `isr`, and binds its variable name to the value returned by applying it to the current configuration. The implementation: `let@` implicitly uses the monadic accessor function `gets` combined with the `let*` syntax, which in turn is an alternative use of the bind operator `(>>=)` of the underlying monad.

3.4 Implementation and paper differences

Due to ease of implementation, we decided to opt for some minor differences between the codebase and the semantic rules. For example, we do not differentiate between exception configurations and halting, despite initially writing the related definitions: this was mainly to ease debugging and avoid executing too many useless tests, since reaching an exception configuration usually indicated that something was wrong with our tests.
Experimenting with the architecture both directly and through testing allowed us to identify some relatively minor issues in the original paper. For example, we noticed a loose definition of *enclave entry point*: if the entry point address is defined to refer to the first byte of the enclave, then only some instructions with specific sizes can be inserted at the beginning of the protected mode, since every byte of the instruction must be marked with the *executable* permission.
We noticed this issue thanks to our tests failing, which prompted us to verify step-by-step the execution of the processor and notice the problem. In order to solve this issue, we opted to define the entry point of the enclave as the first word of the region[2] since every instruction is at least two bytes long, and typically add a `NOP` instruction at the beginning of our enclaves.
In a similar fashion, we noticed that after strictly following the semantic definitions for the instructions we ended up with the `SUB` could be swapping the order of the operands, and we had to reference the manual to resolve our doubts.

[2]This notwithstanding, addresses are always aligned to words.

3.5 Work organization

At the beginning of the project, we had to deal with an initial bottleneck of setting up both the environment and the basic building blocks and definitions of our project, such as memories, register files, and configurations. Since the team size of 6 people was still relatively small, we did not split the project into separate modules with well-defined ”a priori” interfaces in order to parallelize the work: instead, we parallelized ”locally”, by organizing and splitting our work whenever we collectively constructed each new module. For example, our organization turned out to be particularly effective during the implementation of the instruction semantics (and their tests), where each member worked on different instructions at the same time; this method also gave us the advantage that each member was essentially knowledgeable in every section of the project, without compartmentalizing information and ”expertise”. Working all together at the same time on the same module also allowed us to proof-check each other’s code; furthermore, it better allowed us to juggle the multiple references and sources that we needed to repeatedly consult to implement the project correctly, e.g.the paper definitions, the `OCaml` and QCheck2 reference, and the Sancus manual.

4 Testing

Developing the testing environment proved to be the most challenging part of the project. While the implementation was in a sense ”guided” by the paper itself, testing it required us to think precisely what we were building and what properties we wanted to actually verify. In the end, we decided to follow an incremental approach to property-based testing: we first started by verifying the correctness of the implementation, and then we moved on testing some relevant security properties.

4.1 Semantic tests

To verify that our instruction semantics was correct and actually implemented the model from the paper, we wrote tests for each relevant module in the codebase. We checked that our memory, register and encoding operations were working correctly by defining some simple properties; for example, we checked memory operations to be involutive, meaning that putting a byte b or word w in memory and then retrieving it yielded the same starting value. A similar property was also verified for the encoding and decoding of instruction, both as pure lists of bytes and in-memory. For register instructions, we verified that the flags operations on the status register worked as expected.

After establishing the correctness of the elementary components of our codebase, we moved on to verify the actual semantics of CPU instructions. We checked that each instruction computes the right value and modifies the right registers or memory location. While these resemble more classical unit tests, using a property-based approach proved to be extremely valuable. The declarative random generation of values allowed us to easily generate edge cases that were problematic. The other big feature that greatly sped up bug detection was *shrinking*. QCheck automatically simplifies counterexamples from failing tests to a minimal size that still makes the test fail. This mechanism, combined with our customly-built debug printers, allowed us to easily understand what was failing in the test and how to fix the issue. We present a simplified snipped of code showing a semantic test for the **`MOV_LOAD`** instruction in Figure 1.

4.2 Security tests

At the beginning of the project, we collectively went through a phase of brainstorming to imagine how we could test security-relevant properties of our implementation, starting with the full abstraction property. We then quickly realized that testing full abstraction and contextual equivalence using property-based testing would prove extremely difficult in the generation of programs and contexts; even in systems where known attacks exists, generating programs and contexts completely at random would make the probabilities of actually testing non-trivial programs and properties infinitesimal. We therefore tried to focus on simpler tests and mechanisms that, despite having fixed attackers targeted at specific mechanisms, would retain a degree of randomness so that the robustness of a system to attacks and their variations could still be demonstrated.

Non-interference tests

Our first security-relevant test tries to show that by removing the padding mechanism in **Sancus$^{\mathbf{L}}$** we could break the enclave abstraction; i.e., given two externally indistinguishable enclave programs that require the same amount of clock cycles to execute, we could construct an attacker able to distinguish them and reveal if the two given enclaves (and, by extension, their secrets) are identical. We refer to this testing mechanism to non-interference test, since by starting from two identical low-level programs (the unprotected region of memory) we can end up in two different execution results. The two enclaves are generated by using simple linear instructions with no jumps, no memory interaction, no halting, but with different individual cycle lengths: this excludes the cases where the memory accidentally leaks its own secret by showing different total clock cycles while executing, and allows the timing to be independent from the starting values of the registers given as input. The equivalence of two low-level programs is then checked by comparing the value in the register R3.

```
let test_load_um =
  (* Define the predicate to be checked *)
  let property (c, r1, r2, unprotected_addr) =
    (* Set the address in r1 inside c *)
    ...
    (* Run MOV_LOAD with the config c, and check the two cases *)
    step_and_check_instruction (MOV_LOAD (r1, r2)) c
      ~predicate_ok:
        (* Property to test when execution is successful *)
        (fun c' ->
        let before_r1 = register_get r1 c.r in
        let after_r1 = register_get r1 c'.r in
        let unchanged_r1 = r1 == r2 || after_r1 = before_r1 in
        let changed_r2 = register_get r2 c'.r == memory_get before_r1 c.m in
        changed_r2 && unchanged_r1)
      ~predicate_halt:
        (* The processor should halt in no circumstance *)
        false
  in
  (* Generate all elements relevant to the test *)
  let gen =
    QCheck2.Gen.(Config.any_config_unprotected_no_mem () >>= fun config ->
    quad (pure config)          (* Starting configuration                *)
         Register.gp_register (* First register with the address       *)
         Register.gp_register (* Second destination register           *)
         (Memory.unprotected_address config.layout)) (* Loading address *) in
  in
  (* Combine property, printer and generator *)
  QCheck2.Test.make
    ~name:"MOV_LOAD (UM) changes the memory with the correct value from the register"
    ~print:(fun (c, r1, r2, _) -> printer_step (MOV_LOAD (r1, r2)) c)
    ~count:20000 gen property
```

Figure 1: Example of the semantic test for the MOV_LOAD instruction.

```
# <Attacker section>
    MOV <enclave_start>, R3
    JMP R3  # Immediately jump to the enclave
# <ISR section>
    IN R3   # R3 will be the distinguishing register
    HLT
# <Enclave section>
    # ...
    # <randomly generated instructions>
    # ...
    MOV <isr>, R3  # Jump to the ISR in case no interrupts were issued;
    JMP R3         # this can be the case of Sancus^H, with no interrupt logic.
```

Figure 2: Enclave and context code for the non-interference attacker A.

Interrupt-latency attack

The first attack scenario A is constructed as shown in Figure 2, with the control immediately given to the attacker. This is the simplest Nemesis-type attack, as in [3].
Crucially, the attacker context uses a clock-based IO device constructed as follows: each state outputs as read the current time since the start of the system. At a random time, it issues an interrupt and causes the attacker to exit the enclave. We present this automata in Figure 3. Since the enclave instruction begin executed at the given time must fully complete before jumping to the ISR, this allows the attacker to obtain different clock times depending on the length of the instruction.

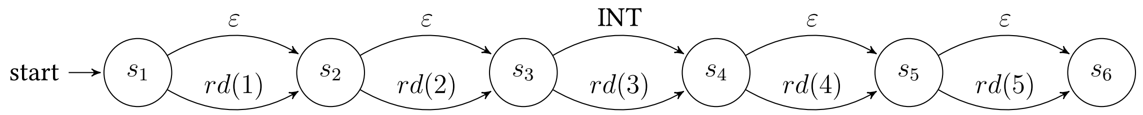

Figure 3: An example of a linear clock automata with interrupt delay = 3.

We identify with the name of **Sancus**$^{\mathrm{U}}$, or *Sancus unsafe*, the execution mode where the interrupt logic does not apply any padding mechanism. In **Sancus**$^{\mathrm{U}}$, attacker A is successful in breaking the enclave abstraction and making the QCheck test fail after some random tentatives. In the case of Sancus$^{\mathrm{H}}$and **Sancus**$^{\mathrm{L}}$, however, this test does not fail even under the stress of 3 million iterations.

Enclave map in Sancus^{U}

In order to show the attacker capabilities in the **Sancus**^{U} case, we also constructed an attacker that generates a *temporal map of the enclave*: i.e., the list of timings for each instruction executed inside the enclave. The attacker was constructed with great effort, due to the accuracy required to get the timings correctly, and we present the full code in Figure 5.

In order to assist the attack we need to construct a somewhat complex supporting IO device, presented in Figure 4 and that we refer to as *on-demand interrupt io device*. Contrary to the clock IO device, this automata is completely fixed and is not generated.

The idea of the attack is to raise an interrupt exactly at the first cycle of each instruction of the enclave. When we return to the ISR, we ask the IO device for the current time, and append it in the memory region of the attacker. Then, we ask the IO device to reset and tell it to issue the next interrupt after k cycles, during which we execute the **RETI** instruction and return to the enclave to repeat the mechanism. The delay $k = (5+2)+1$ is selected carefully to be the time required to restart the device, with an **OUT** instruction of 2 cycles, and to return to the enclave with the **RETI** of 5 cycles.

Interestingly, we also tested attacker A in the case of Sancus^{PP}, or *Sancus pre-pad*, which only applies the pre-padding mechanism; as we expected, the pre-padding policy prevents the attack.

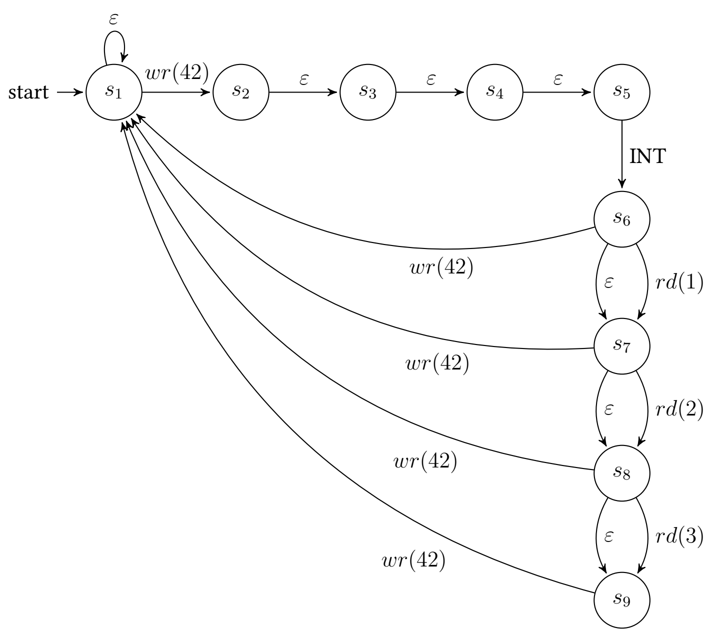

Figure 4: The on-demand interrupt automata with interrupt delay = 3.

```
# <Attacker section>
    MOV 42, R3             # Special value to start the interrupt countdown
    OUT R3                 # (2 cycles) Interrupt delay starting
    NOP                    # (1 cycles) Time padding
    MOV <enclave_start>, R3 # (2 cycles)
    JMP R3                 # (2 cycles)
    # Total cycles before first interrupt must be equal to OUT + RETI = 5 cycles
 attacker_out:
    # n bytes are reserved here as output for the attacker
# <ISR section>
    IN R3         # R3: length of the instruction + 6. The additional 6 cycles
                  # is given by the time taken to manage the interrupt.
    MOV 42, R4    # Special IO device value to stop and restart the device
    MOV 2, R6     # Constant to increment the memory address used as output
    MOV 6, R10    # Delay to subtract and get the actual instruction duration
    MOV <attacker_out> R5
    OUT R4        # Stop and reset the clock to state 1
    MOV @R5, R7   # R5: &output_addr, R7: output_addr
    SUB R10, R3   # Remove interrupt time of RETI
    MOV R3, 0(R7) # Instruction length stored at output_addr
    ADD R6, R7    # output_addr = output_addr + 2
    MOV R7, 0(R5) # output_addr + 2 written at &output_addr
    OUT R4        # Restart the clock, with delay occuring in k cycles
    RETI          # Return to the enclave
# <Enclave section>
    # ...
    # <randomly generated instructions>
    # ...
    HLT           # Terminate to signal the end of the enclave
```

Figure 5: Enclave and context code for the enclave timing map test.

```
# <Attacker section>
    MOV <enclave_start>, R3
    JMP R3  # Immediately jump to the enclave
  attacker_epilogue:
    IN R3   # Read the final IO device clock
    HLT     # Terminate
# <ISR section>
    RETI    # Immediately return to the enclave
# <Enclave section>
    # ...
    # <randomly generated instructions>
    # ...
    MOV <attacker_epilogue>, R3
    JMP R3  # Jump to the attacker epilogue
```

Figure 6: Enclave and context code for the attacker B with resume-to-end.

Resume-to-end attacks and post-padding

We then constructed a *resume-to-end* attacker B to also tackle the $\text{Sancus}^{\text{PP}}$ mechanism. This attack was also conducted on Sancus^{H} and **Sancus^{L}** to again ensure that they are immune to these attacks. The attack is presented in Figure 6, and the device used is the same as attacker A with a linear clock telling the time.
The idea behind the code is that, without a post-padding mechanism, it is possible to misalign the timings of the enclave and detect differences in the overall cycles. This is due to the fact that some enclaves will apply a greater pad than others, without balancing out the total cycles waited.

5 Conclusions

In this project we developed an `OCaml` implementation of the Sancus platform, together with the extensions supporting secure interruptible enclaves proposed in [2]. We implemented all the core features described in the paper and, using property-based testing, we checked the semantic of our implementation and verified that the security measures introduced to permit interrupts inside enclaves actually prevent possible leaks, with the **Sancus**$^{\mathrm{L}}$ programs being as secure as the Sancus$^{\mathrm{H}}$ ones.
Apart from a few first lectures on preliminary concepts, we appreciated the experimental nature of the course, which allowed the lectures to become non-frontal and gave us the independence to discuss the paper and work on the project. We scheduled weekly meetings with the professors and one of the authors of the paper to keep them updated and discuss doubts and implementation choices.
In the end, this project taught us some valuable skills. We remark in particular a few lessons that we took from this work:

- **Reading a technical paper.** We learned how a work of this size is structured, and effective ways to better read and understand what is presented in a scientific paper. For this part, the possibility of having a direct interaction with one of the authors proved fundamental.

- **Importance of testing.** While testing does not provide the same guarantees of a formal proof, it was an invaluable part of our development process. The immediate feedback given by tests guided us to the right solutions and easily spotted issues and potential bugs. Using a powerful approach such as property-based testing helped us increase both quality and speed of writing tests, leading us to write better code.

- **Working as a group.** We learned how to manage and organize a non-trivial project and effective ways to organize code, split tasks, and communicate in a big working group. Parallelizing the work turned out to be a difficult endeavour, but being in a team allowed us to cross-check and discuss together issues both in reading the paper and writing down our work.

5.1 Future work

Due to the time constraints of the project, we could only explore a minimal part of the target topic, and many improvements and extensions can be explored and developed. A possible extension could be to investigate and try to reproduce newly discovered bugs of Sancus [4], using property based testing to verify if it breaks full abstraction from Sancus$^{\mathrm{H}}$to **Sancus**$^{\mathrm{L}}$. Another possible extension could be to add additional features to the architecture itself (caches, pipeline, speculative execution, ...) and investigate the possible security implications of such features, in the same way interruptible enclaves were investigated in [2]. Other than these extensions, the section of our project that could see a major improvement is the generation of attackers in security relevant tests. Currently, we use a set of fixed ad hoc attackers that simply setup the environment and jump to a randomly generated enclave. Investigating the generation of random yet sensible attackers should be the top priority to generalize and extend our results.

References

[1] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., August 2013. USENIX Association.

[2] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Securing interruptible enclaved execution on small microprocessors. *ACM Trans. Program. Lang. Syst.*, 43(3):12:1–12:77, 2021.

[3] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 178–195, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *43rd IEEE Symposium on Security and Privacy (S&P)*, May 2022.