Semester Project Report

Verified Page Tables Manipulation

Michael PAPER
Supervised by Edouard BUGNION and Charly CASTES

June 15th, 2022

Contents

Contents **2**

1 Introduction **3**

2 Related work **3**

2.1 Verified systems 3

2.2 Verification tools 6

2.3 Finite interfaces 6

3 Problem statement **7**

3.1 Reminders on paging 7

3.2 The Serval design 7

4 Contribution **8**

4.1 A finite interface to manage page tables 8

4.2 Specification 10

4.3 Proofs 10

5 Discussion **11**

5.1 The attempt to use Prusti 11

5.2 The limitations of Serval 12

6 Conclusion **12**

7 References **13**

Acknowledgements

I would like to thank Edouard Bugnion for taking the time to discuss possible project topics with me and letting me choose to work what I was most interested in. I would also like to thank him for all our meetings where he taught me many things about the inner workings of virtual memory.

I would like to thank Charly Castes for his responsiveness through the whole semester and for giving me the right advices to make the project move forward.

Finally, I would like to thank all my friends who supported me while I kept blaming my computer for not being clever enough to do what I begged him to do, even though I was clearly the one not clever enough.

1 Introduction

Formal verification of a piece of software consists in specifying its expected behavior and to prove that its execution will always match the specification. For anything to be provable, some assumptions are always required, such as the correctness of the hardware for which the software has been developed, or the correctness of the tools used to prove the correctness of the software [21]. In general, if these assumptions are reasonable, formal verification of a software allows its users to have a high confidence in its correctness. If some security properties are formulated in the specification of this software, then its users can also be highly confident in the fact that these security properties are met by the software.

Privileged low-level software, such as monitors, hypervisors and operating systems, must usually be trusted by higher-level applications for their own security. This makes them interesting targets for formal verification.

The Data Center Systems Laboratory (DCSL) is currently working on a small privileged software handling negotiations between a hypervisor and an operating system, removing unnecessary access rights from the hypervisor, and allowing nested layers of trusted computing. Because it will be the most privileged software running on the machine, its security will be very critical, and its small code should make some parts of it amenable to formal verification.

The privileged mechanisms my project focused on was the allocation of physical memory and its mapping to virtual memory through the page tables. The aim of this library is to become a part of DCSL's project.

In section 2, I will explore previous work on formal verification of software systems. In section 3, I will explain how previous work influenced our design choices and what exactly we ended up trying to implement. In section 4, I will explain our attempts at implementing our design. Finally, I will conclude in section 6.

2 Related work

2.1 Verified systems

Prior work on unverified security monitors and small hypervisors mentions using formal verification to provide a stronger source of trust to their guests.

It is the case of TrustVisor [14], a hypervisor with many similarities to the project developed

by the DCSL in its goals of secure execution. Its authors claim in the abstract that its small code size makes it amenable to formal verification. However, no formal verification technique has been used for TrustVisor yet.

It is also the case of Keystone [10], a Trusted Execution Environment (TEE) implemented as a RISC-V monitor software. Keystone enclaves are full operating systems, running running their own applications. To provide strong safety properties, the authors propose to run the applications on top of the verified operating system seL4 [8] within those secure enclave. Furthermore, even though the Keystone security monitor is not verified, its authors claim that it is amenable to formal verification due to its small code size. Later, [16] wrote partial specifications to ensure the functional correctness of Keystone, the confidentiality of its enclaves, and the absence of undefined hardware behaviors through all executions. Using these partial specifications, the goal was to detect the presence of bugs, not their absence. Four bugs have been found that way, and have been fixed since. No complete formal specification and verification of Keystone has been achieved yet.

The first formally verified uniprocessor operating system was seL4 [8]. seL4 has evolved since its original publication, to integrate proofs of high-level security properties [19, 15] in addition to the original proofs of functional correctness and to verify the compiled executable against the verified source code [20]. The seL4 µ-kernel was implemented in C. A concrete specification of the C code was written in Haskell, and an abstract specification was written in Isabelle. The authors proved that the C code *refined* the concrete specification, which in turn refined the abstract specification. The authors claim that this 2-layer specification methodology was a significant net cost saver, yet for 8.7k lines of C and 600 lines of assembly, the proof was made of 200k lines of Isabelle script and took 20 person-years. The functional correctness proofs of seL4 relied at the time on the correctness of 1.2k LoC of boot and initialization, the unverified TLB/cache flushing mechanism, and, in its original publication, the C compiler. The high-level security proofs of seL4 were mostly derived from its abstract specification.

Another (almost) fully verified operating system is CertiKOS. Its design, uniprocessor, and multiprocessor implementations are presented in [5], [6] and [7] respectively. CertiKOS has roughly 10x more lines of proofs than lines of code, and it also relies on some unproven parts of the C code. Its proofs are written in Coq using a home-made version of ClightGen (called ClightGenX). I will present ClightGen in subsection 2.2.

Small pluggable pieces of systems have also been the targets of formal verification. For instance, Ipanema [11] is a Domain Specific Language (DSL) aimed at developing scheduling policies for Linux. Given a policy written in the DSL, the framework produces two outputs: a Linux scheduler module, and a WhyML code corresponding to the policy. The authors managed to implement work-conserving schedulers and prove that they were work-conservative with 2k lines of WhyML. FSCQ [2] is a file-system written in Haskell, proven to be crash-safe in Coq. Its design is inspired by the design of the file-system of xv6, and it required 10x more lines of code (including proofs). Later, some of the authors of FSCQ released GoJournal, a crash-safe verified file-system close to FSCQ but written in Go and supporting multi-threaded uses.

The proofs of correctness of those two systems rely on the correctness of a significant software stack. Ipanema's 41k lines of OCaml, the C compiler, WhyML and the Linux kernel must all behave correctly for the proofs to hold. In the case of FSCQ, the correctness of Coq, the Haskell runtime, and FUSE were part of the assumptions. It has been found 3 years after the publication of FSCQ that a bug in FUSE broke the crash-safety property of FSCQ.

Verve [23] and ExpressOS [13] are uniprocessor µ-kernels with some parts proven automatically. Verve's kernel contains a specified and verified "Nucleus", but the rest of the kernel is

not verified. The kernel must be built with the applications that will run on it, and application executables cannot be loaded dynamically. That way, Verve can avoid reasoning about virtual memory. The proofs of ExpressOS only express that it encrypts all private data of applications before sending it to the system services. This allows ExpressOS to prove its security without proving its functional correctness.

Komodo [4] is a verified uniprocessor ARM TrustZone monitor, smaller than a µ-kernel. Some parts of it are proven automatically, other proofs are hand-written.

Finally, the Hyperkernel [17] is a fully automatically proven x86 uniprocessor OS inspired by the design of xv6, delegating many responsibilities to user-space like an exokernel. For it to be automatically proven, its authors have had to design it with a finite interface. I will get back to what that means in subsection 2.3, and I will explain in details the virtual memory management unit of the Hyperkernel in section 4. Its verification framework is a python program designed to prove exactly the correctness of the LLVM IR produced by the Hyperkernel, and seems hardly portable to any other project. Its authors tried to write it in Rust, but found that the Rust memory model was hard to formalize and imposed many constraints that were not necessary for a single-threaded kernel. That is why they wrote the Hyperkernel in C.

It is worth noting that several years before verifying the whole µ-kernel, one of the authors of seL4 wrote about methods to prove the correctness of a virtual memory manager [9], and so did an author of CertiKOS [22].

Table 1 sums up the data of the projects presented in this section.

System	Lines of code	Lines of proof	TCB approximation
seL4 (as of [8])	9.3k	200k	Initialization, Isabelle, C compiler, TLB/cache flushes
CertiKOS (as of [7])	6.5k	50k more than [6]	Initialization, ELF loader, "*functions such as `memcpy` [...] because of a limitation arising from the CompCert memory model*"
Ipanema	41k for the DSL, 250-350 for the schedulers	2k	DSL, C compiler, Linux, WhyML
FSCQ	30k including proofs	Not specified in [2]	Haskell compiler and runtime, FUSE, Coq
Verve	Not specified in [23]	6k	Boogie/Z3, BoogieASM, Typed Assembly Language (TAL) checker, assembly, linker
ExpressOS	14.5k	0.5k	Dafny, Boogie, L4, C# compiler
Komodo	875 lines of assembly	4.4k lines of spec, 2.7k lines of Vale, 18.7 lines of proof	Dafny and an assembly printer
Hyperkernel	7.6k	1k	Z3, LLVM IR compiler, initialization, glue code ("e.g., *assembly for register save and restore*")

Table 1: Comparison of the formally verified systems presented in this section.

2.2 Verification tools

CompCert [12] is a C compiler written in OCaml and Coq, verified in Coq. It works by translating a C program to a semantically equivalent Clight representation, then compiling this Clight program into semantically equivalent machine code (after many translations into intermediate representations). CompCert comes with ClightGen, a tool to work, inside Coq, on the CFG of the Clight representation of a program generated by CompCert. That way, ClightGen allows the user to write proofs on C programs which will be proven to also hold on the binary produced by CompCert.

Prusti [1] is a framework to automatically prove properties of Rust programs. Using Rust's macros, a developer can state properties that should hold at a given point of the execution (similar to `assert` statements), as well as functions' pre- and post-conditions, and cross fingers hoping that Prusti will be able to prove that all assertions hold. Prusti models some parts of the complex memory model of Rust such as `Box`es. However, all of the Rust language is not fully supported by Prusti, proofs are done at the level of the source code, and the compilation is not verified.

Serval [16] is a framework developed by some of the authors of the Hyperkernel. It is written in Rosette, which is a module of the Racket programming language, which is a LISP dialect. Rosette adds constraint-solving to the Racket programming language. A developer can write an interpreter of a given machine code in Rosette (the authors provide such interpreters for LLVM byte-code, eBPF, x86 and RISC-V), then compile applications to this machine code representation using his usual toolchain (the artifacts contain examples written in C compiled with gcc). The developer can then write a specification for this program in Rosette in the form of constraints on the state of the Rosette interpreter. Serval will look in the debug symbols to match the symbols mentioned in the specification with the symbols of the program. Finally, Serval tries to verify that the constraints of the specification hold for all executions of the program, using the optimized constraint solver of Rosette.

Serval was used to retro-fit slightly modified versions of Komodo and of the uniprocessor implementation of CertiKOS into automatically verified RISC-V monitors, in 4 person-weeks each. Because they were retro-fitted to RISC-V monitors, they do not include any form of page tables manipulation, and rely only on RISC-V's Physical Memory Protection (PMP).

2.3 Finite interfaces

The papers of Serval [16] and of the Hyperkernel [17] both define the notion of finite interfaces as interfaces that can be implemented without unbounded loops or recursion. This is a key notion for automating functional correctness proofs of systems with symbolic evaluation tools, as finite interfaces allow for implementations for which symbolic evaluation will not be subject to state explosion.

In the Hyperkernel, this definition was used to design all of the system call interface. In the Serval paper, it has been used as a guiding principle when retro-fitting CertiKOS and Komodo.

The Hyperkernel paper gives the example of how the `dup` system call, which is implemented in xv6 as specified by POSIX, was "finitized". The `dup` POSIX system call is non-finite, since it requires the OS to look for the smallest unused file descriptor. A "finitized" version of the same system call would take the old file descriptor as a first argument and the new file descriptor as a second argument. It would delegate to the user the responsibility of finding a valid new file descriptor, and would only check the validity of the new file descriptor and update the file descriptor table accordingly.

3 Problem statement

In this section, we define the different goals that we set during my project, and why they were kept or discarded.

Two main aspects were kept unchanged from the beginning to the end of the project: proofs would have to be automated, and the library would be alone in charge of managing the allocation of physical memory and the mappings from virtual addresses to physical memory through the page tables. The main goal was to be able to prove that a page allocated by a single user of the library was never mapped to any other user's virtual address space.

Some automated proof tactics for Coq exist, but for now they do not perform well enough to be considered here. For instance, CoqHammer [3] only manages to prove 25% of the proofs of CompCert's lemmas, which have themselves already been carefully subdivided in easier sub-lemmas to ease proof readability and maintainability.

This ruled out using ClightGen as CertiKOS. As we have seen in section 2, there is no framework for developing automatically proven concurrent low-level software as of today. Because of Rust's runtime guarantees, if we had managed to prove that the Rust code of the library is correct, then there would have been no possible race conditions. For this reason, we chose to try Prusti first. I will explain where exactly problems arose and how we switched to using Serval.

We also considered using Prusti to specify and prove the correctness of our library. I will explain in more details the problems we encountered with Prusti and the design we tried to implement in subsection 5.1.

3.1 Reminders on paging

As a reminder, the mapping from virtual address to physical address performed by the hardware on a typical AMD64 processor uses hierarchical page tables, as illustrated by Figure 1. The CR3 register is set by the operating system and points to the root of the address space of a user, a level 4 page table. A virtual address is made of 48 bits. The 9 first bits of a virtual address represent the offset in which is written the physical address of the level 3 page table that will be used for the rest of the translation. The rest of the translation works the same way, using the 9 next bits of the address at every level, until 12 bits remain. At this point, the 12 bits are not used for the translation but to index a byte in the 4096 bytes long memory region starting at the physical address indicated by the level 1 page table entry.

Because every memory access performed by a user goes through this virtual-to-physical translation, a user can only access the memory that is in a leaf of this tree of page tables.

3.2 The Serval design

The Hyperkernel and Serval papers describe the concept of finite interfaces, which I explained in subsection 2.3, and claim that a system must have finite interfaces to be amenable to automated verification with symbolic evaluation. Therefore, we shaped the design of our interface for it to be finite. This made our design very close to the one of a security monitor or an exokernel, as it only cares about ensuring mechanisms, not any form of policy.

Through this new interface, a user can request to allocate a page that they have to specify, and they can manipulate the page table tree that corresponds to its virtual address space. Our library checks that allocations are legal and that mappings do not point to addresses that

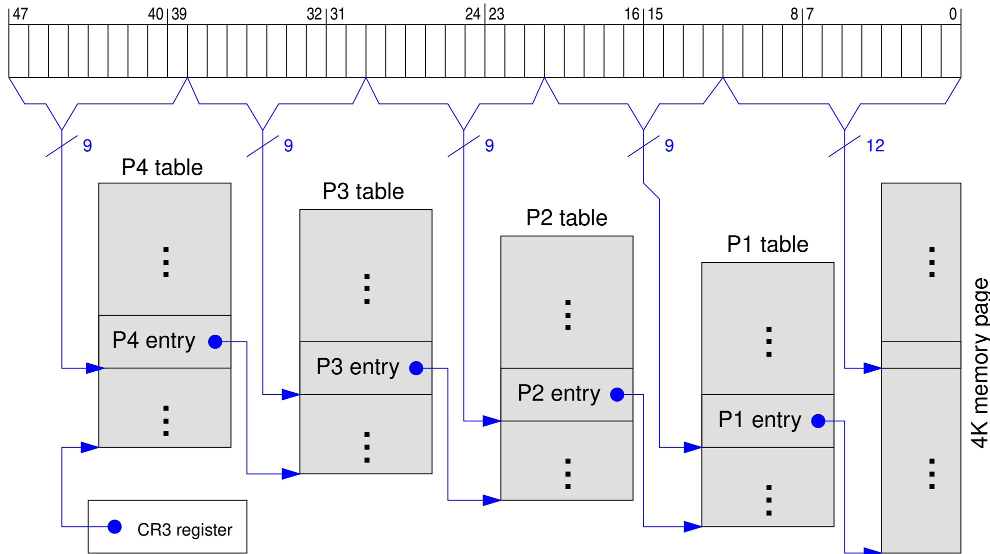

Figure 1: Page tables translate virtual addresses into physical addresses (image unmodified from [18]).

do not belong to this user.

The next chapter explains in more details this interface, its implementation, and its specification.

4 Contribution

4.1 A finite interface to manage page tables

To design the interface of our virtual memory manager, we looked closely at the virtual memory manager of the Hyperkernel, which has a finite interface. According to [17], a finite interface is an interface that allows an implementation free of unbounded loops and recursion. For instance, the `dup` POSIX system call is non-finite, since it requires the OS to look for the smallest unused file descriptor. A finitized version of the same system call would take the old file descriptor as a first argument and the new file descriptor as a second argument. It would delegate to the used the responsibility of finding a valid new file descriptor, and would only check the validity of the new file descriptor and update the file descriptor table accordingly.

In the Hyperkernel, the handling of the linked list of free pages is not verified, even though it is part of the memory allocator. The specification of the virtual memory manager states that if some properties of the state of the virtual memory hold before any call to the Hyperkernel, they will also hold after the call. However, there is no specification of when the calls requesting the allocation of a page should succeed or fail. This allows the Hyperkernel to use the free list as a hint to try to allocate a page. To the eyes of the verification framework, even if the free list is implemented perfectly, it can not be trusted, since the it is not specified, let alone verified. Taking the hint, the Hyperkernel will still verify that the page is indeed free and fail otherwise, and this is sufficient for the specification to hold.

The Hyperkernel needs to use this unverified data structure inside of the verified parts because it tries to choose a free page itself. In contrast, our approach was to let the users in charge of

finding free pages. This allowed us to implement a library free of unverified data structures, and therefore a specification free of seemingly unjustified possible allocation rejections. To help the user to choose a free page, we still provide two functions to manipulate a free list (one to add a page number to the free list, the other to remove a page from the free list). These functions are not specified at all with Serval.

The library maintains a fixed-size array containing the metadata of every page of the machine. It also maintains an array of as many 4096-bytes pages representing the whole memory of the machine. The interpretation of these pages is specified by the content of their corresponding meta-data structure. Finally, a global array containing the top-level page table corresponding to each user is initialized and never modified by any call to the library.

The modifications made to the page tables must all be checked not to map any page to an address space it does not belong to. Therefore all the modifications made to the page tables must be made through an interface defined by the library. Two types of operations can interest a user, either add or remove an entry from a page table. To create or destroy such a mapping, the rights of the page table and of its entry mut be checked against the caller.

To sum up, the functions of the library are the following:

```
1  void init_frames(void);
2  int allocate_frame(pn_t frame_number, page_type_t type, uint32_t permissions);
3  int free_frame(pn_t frame_number);
4  int map_page_table_l3_entry(pn_t l4e, uint16_t l4_offset, pn_t l3e);
5  int map_page_table_l2_entry(pn_t l3e, uint16_t l3_offset, pn_t l2e);
6  int map_page_table_l1_entry(pn_t l2e, uint16_t l2_offset, pn_t l1e);
7  int map_page_table_frame(pn_t l1e, uint16_t l1_offset, pn_t frame);
8  int unmap_page_table_entry(pn_t freepte, uint16_t l4_offset, pn_t pte_entry)
9  pn_t pick_free_frame(void);
10 void add_free_frame_to_free_list(pn_t frame_number);
```

As previously mentioned, the last two functions of the list are not verified.

All the verified functions perform basically the same way, performing many checks and returning an error if anything went wrong, then allocating or de-allocating a resource correctly if everything went right. As an example, here is the source code of the `free_frame` function.

```
1  int free_frame(pn_t frame_number)
2  {
3    /* You can't free a frame that does not exist */
4    if (frame_number >= NUMBER_OF_FRAMES)
5      return 1;
6    /* You can't free a frame if it's not yours */
7    if (frames_metadata[frame_number].owner != current)
8      return 1;
9    /* You can't free a frame that is still referenced by page table entries */
10   if (frames_metadata[frame_number].refcount > 0)
11     return 1;
12   /* You can't free a page table has valid entries */
13   if (frames_metadata[frame_number].type == PAGE_L4_ENTRY ||
14       frames_metadata[frame_number].type == PAGE_L3_ENTRY ||
15       frames_metadata[frame_number].type == PAGE_L2_ENTRY ||
16       frames_metadata[frame_number].type == PAGE_L1_ENTRY)
17     if (frames_metadata[frame_number].entry_count > 0)
18       return 1;
19
20   frames_metadata[frame_number].type = PAGE_FREE;
21   frames_metadata[frame_number].permissions = 0;
22   frames_metadata[frame_number].owner = 0;
23   return 0;
24 }
```

In the next section, I will explain in more details what verifications are done for the allocation and the mapping of pages, and what properties we managed to prove using Serval.

4.2 Specification

In this section, I will explain the structures of my code and go through each functionality of the library and explain how users are allowed to use them.

The data structure holding the metadata of a page is as follows.

```
struct frame_metadata {
  uint64_t address;       // address of the corresponding page
  pn_t next_free;         // next frame in the free list (if the page is free)
  page_type_t type;       // either FREE, FRAME, or PAGE_TABLE_L{1,2,3,4}
  uint32_t refcount;      // number of page table entries referencing this page
  uint32_t permissions;   // SHARED | READ | WRITE | EXEC
  pid_t owner;            // PID of the user who allocated the page
  uint16_t entry_count;   // number of valid entries (if the page is a PT)
}; /* struct size: 64+64+32+32+32+16+16 = 256b = 32B */
```

For a user to allocate a page using `allocate_frame`, this page only needs to currently have the type `PAGE_FREE` and the user must give any non-`PAGE_FREE` type to the freshly allocated page.

For a frame to be freed by a user, it must be non-free and owned by the same user.

For a mapping of the page Y to be created at offset o of the page table X, the user must be the owner of X, and Y must either be owned by him or be shared. Also, the offset o of X must already be an empty entry. Finally, if X is shared, then Y must also be shared.

4.3 Proofs

All of the properties I wrote with Serval are of the form "if the invariant holds before a call to one of the library's functions, it also holds after the call". These properties are formulated for the functions `allocate_frame`, `free_frame`, `map_page_table_*` and `unmap_page_table_entry`.

For the `init_frames` function, the specification is that, no matter if the invariant holds before its call or not, it should hold after the call. In other words, `init_frames` sets the global data structures in a coherent state. Unfortunately, the `init_frames` function iterates over all the frames of the machine, and is therefore subject to state explosion during its symbolic evaluation. As expected, reducing the number of frames of the machine makes the verification successful quickly. Otherwise, the verification takes too long.

I list below the invariants that Serval proved to always hold, separated in three categories for clarity.

Integrity

- Each page metadata's `address` field is non-null and unique
- All free pages and top-level page tables have a `refcount` field set to 0.

Typechecking

- All non-empty level 4 page table entries are level 3 page tables

- All non-empty level 3 page table entries are level 2 page tables
- All non-empty level 2 page table entries are level 1 page tables
- All non-empty level 1 page table entries are frames

Hereditary shared attribute

- All entries of a shared page table are shared pages
- All exclusive entries entries of an exclusive page table is owned by the same user as the page table itself.

As an example, this is what the property stating that all free pages have a `refcount` of 0 looks like, expressed in Racket with Serval.

```
1 (define-symbolic i (bitvector 64))
2 (define fm (symbol->block 'frames_metadata))
3 (forall (list i) (=> (bvult i (bv NUMBER_OF_PAGES PAGE_NUMBER_TYPE_SIZE))
4   (=> (equal? (mblock-iload fm (list i 'type)) PAGE_FREE)
5       (equal? (mblock-iload fm (list i 'refcount)) (bv 0 32)))))
```

The variable `fm` defined on line 2 uses Serval's parsing of the debug information from the binary to correspond to the memory block of the `frames_metadata` global array of the library. The function `mblock-iload` called in lines 4 performs a symbolic memory access to `frames_metadata`, first using the index `i` then using the struct field `type`.. This call would be translated to `frames_metadata[i].type` in C syntax.

The specification is made of 260 lines of Racket, while the whole library is made of 440 lines of C. The verification takes about 17 seconds on an Intel i7-10610U with a frequency of 4.9GHz.

5 Discussion

5.1 The attempt to use Prusti

The library we wanted to create presented the abstraction of Virtual Memory Areas (VMAs) to its user. It was made of the functions `create_vma`, `destroy_vma` and `alias_vma`. VMAs are exclusive to a single user unless `alias_vma` is called on them. With `create_vma`, a user could ask the library to be given a certain amount of memory and for it to be mapped at a specific range of addresses in its virtual address space. With `destroy_vma`, a user could ask for one of its VMAs to be removed from its address space, and the corresponding memory to be freed. `alias_vma` allowed a user to ask for a VMA residing in another user's address space to be mapped in their address space as well, and would only succeed if that VMA had the right permissions set when it was initialized.

Internally, the library used a global linked list of free pages, and all of the page tables. To help proof automation, we created one type for each kind of page. A page could either be free, a frame, or an ith level page table, for $1 \leq i \leq 4$.

We tried to develop this library two different ways. First, we developed the Rust library progressively trying to make Prusti accept our code at each step. What we did on this try never escaped of the scope of the parts of the Prusti tutorial that have already been released. Yet we ran into many issues, some regarding the Visual Studio plugin of Prusti,

others regarding the error reporting of Prusti. We sent those issues to the corresponding repositories, following advices taken from the Zulip group of Prusti. The plugin has been fixed, not the error reporting bug.

The other approach was to create the whole library, to add annotations, and only then to try to make it comply to Prusti. That way, we found many more limitations to Prusti. Surprisingly, some portions of `unsafe` code were not problematic for Prusti, but bitwise operations on integers appeared not to be supported by Prusti. A quick fix was to implement them as arithmetic operations, or to tell Prusti to "trust" that these pieces of code behaved as the corresponding arithmetic operations. As page table manipulation requires many bit-wise operations to compute page tables offsets based on virtual addresses, none of these options were completely satisfactory. Even after applying those fixes, a partially implemented version of the library had 14 more errors, most of them caused by unsupported Rust features. Furthermore, Prusti took more than 20 minutes to report those errors, even though they were simply unsupported features. At this point, we decided to give up on Prusti's promises of concurrency and its powerful automated theorem proving capabilities, and we decided to go for Serval's uniprocessor symbolic evaluation approach.

5.2 The limitations of Serval

I found Serval to be a very powerful symbolic evaluation tool. However, it remains a symbolic evaluation tool, and using it restricted out set of designs to finite interfaces. This issue is general to symbolic evaluation tools, and not restricted to Serval. I think a more ideal framework to reason automatically about more general systems would reason about the CFG of the functions considered generated by ClightGen. This would allow reasoning about more complex data structures (for instance Prusti works on the CFG of Rust programs and it can prove properties of linked lists), and this would remove from the TCB of the proofs the correctness of the debug information retrieved by Serval. Such an approach could take advantage of the properties of the Clight CFG to avoid using a full-fledged general Coq theorem prover like CoqHammer and use a more adapted automated theorem prover, like Prusti does. Unlike Prusti, as this approach would work with CompCert, no part of the compiler would be in the TCB of the proofs.

Another issue with Serval is that it is still in its early stages of development, and it lacks a complete documentation. The `address` field of `struct frame_metadata` is not necessary for the implementation of the C functions. It was needed to reason with Serval about `&pages[i]`. A good candidate Serval function that might have removed the need for this field is `mblock-resolve`, which is defined in the file `serval/lib/memory/mblock.rkt` of Serval, and seems to return the address of a memory block. Because Serval lacks a proper documentation, I was not able to understand its usage. It is worth noting however that both Racket and Rosette are both heavily documented.

6 Conclusion

At the beginning of this project, I have looked at many approaches taken by previous research projects trying to verify the functional correctness of systems formally. I found two approaches that seemed to be a fit with the system we wanted to develop: Prusti and Serval.

We designed reasonable interfaces that seemed to be coherent with the respective verification frameworks considered with regard to the algorithms they implied on their implementations. We did not manage to get a working version of the Prusti prototype, but we managed to finish

a proof of concept with Serval.

Using Serval, we were able to prove high-level invariants on the global data structures managed by the library. The library is still very rudimentary, and it still has several limitations. For instance, it only allows pages to be exclusive to their owner or to be shared with all of the other users. This certainly needs to be improved in order to allow the nested levels of confidential computing that the project being developed by the DCSL tries to enforce.

Possible extensions to this project could try to batch calls to the library in order to reduce the number of privilege-transitions done to, for instance, allocate a large number of pages.

7 References

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[2] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Łukasz Czajka. Practical proof search for coq by type inhabitation. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, page 28–57, Berlin, Heidelberg, 2020. Springer-Verlag.

[4] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 287–305, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: a certified kernel for secure cloud computing. In *APSys*, 2011.

[6] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 595–608, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[9] Gerwin Klein and Harvey Tuch. Towards verified virtual memory in l4. 2004.

[10] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema: Application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 293–304, New York, NY, USA, 2013. Association for Computing Machinery.

[14] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 143–158, USA, 2010. IEEE Computer Society.

[15] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[16] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Philipp Oppermann. Writing an OS in Rust (First Edition). https://os.phil-opp.com/edition-1/, 2022.

[19] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. Sel4 enforces integrity. In *Proceedings of the Second International Conference on Interactive Theorem Proving*, ITP'11, page 325–340, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 471–482, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, aug 1984.

[22] Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In *CPP*, 2012.

[23] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 99–110, New York, NY, USA, 2010. Association for Computing Machinery.