Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor

Zhenyang Dai
daizy19@mails.tsinghua.edu.cn
Tsinghua University & Ant Group
China

Shuang Liu
ls123674@antgroup.com
Ant Group
China

Vilhelm Sjöberg[*]
vilhelm.sjoberg@certik.com
CertiK
USA

Xupeng Li[†]
xupeng.li@columbia.edu
Columbia University
USA

Yu Chen
yuchen@tsinghua.edu.cn
Tsinghua University
China

Wenhao Wang
wangwenhao@iie.ac.cn
Institute of Information Engineering, CAS
China

Yuekai Jia
jyk19@mails.tsinghua.edu.cn
Tsinghua University & Ant Group
China

Sean Noble Anderson[†]
sean.noble.anderson@protonmail.com
Portland State University
USA

Laila Elbeheiry[†]
lelbehei@mpi-sws.org
Max Planck Institute for Software Systems
Germany

Shubham Sondhi
sondhi.shubham11@gmail.com
CertiK
USA

Yu Zhang[†]
yu.zhang.yz862@yale.edu
Yale University
USA

Zhaozhong Ni
zhaozhong.ni@certik.com
CertiK
USA

Shoumeng Yan[*]
shoumeng.ysm@antgroup.com
Ant Group
China

Ronghui Gu
ronghui.gu@columbia.edu
Columbia University
USA

Zhengyu He
zhengyu.he@antgroup.com
Ant Group
China

Abstract

As trusted execution environments (TEE) have become the corner stone for secure cloud computing, it is critical that they are reliable and enforce proper isolation, of which a key ingredient is spatial isolation. Many TEEs are implemented in software such as hypervisors for flexibility, and in a memory-safe language, namely Rust to alleviate potential memory bugs. Still, even if memory bugs are absent from the TEE, it may contain semantic errors such as mis-configurations in its memory subsystem which breaks spatial isolation.

In this paper, we present the verification of the memory subsystem of a software TEE in Rust, namely HyperEnclave. We prove spatial isolation for the secure enclave though correct configuration of page tables for an early prototype of HyperEnclave. To formally model Rust code, we introduce a lightweight formal semantics for the Mid-level intermediate representation (MIR) of Rust. To make verification scalable for such a complex system, we incorporate the MIR semantics with a layered proof framework.

[*]Corresponding author.
[†]Work done during an internship at CertiK.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640398

CCS Concepts: • **Security and privacy** → **Virtualization and security**; **Logic and verification**; • **Software and its engineering** → **Formal methods**.

Keywords: Formal verification, Rust, trusted execution environments, extended page tables

ACM Reference Format:
Zhenyang Dai, Shuang Liu, Vilhelm Sjöberg, Xupeng Li, Yu Chen, Wenhao Wang, Yuekai Jia, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, Yu Zhang, Zhaozhong Ni, Shoumeng Yan, Ronghui Gu, and Zhengyu He. 2024. Verifying Rust Implementation of Page

Tables in a Software Enclave Hypervisor. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620665.3640398

1 Introduction

The Rust programming language. Rust is a programming language providing memory safety guarantees at compile time. The safety guarantees stem from its ownership and lifetime compiler checks, instead of runtime garbage collection. As a result, developers are starting to implement systems software in Rust, including operating systems (Redleaf [36], Redox [11], Tock [30]), hypervisors (Google's Chrome OS Virtual Machine Monitor crosvm [5], Diosix [6], Cloud Hypervisor [4]), and even the Linux kernel is adding support for Rust [8].

However, while Rust does reduce memory related bugs [17, 48], memory safety alone does not guarantee functional correctness or security. For example, an unprivileged attacker may try to bypass the memory isolation scheme of an operating system with carefully-crafted system call sequences. These vulnerabilities are subtle, difficult to find, and critical to system security. Beyond the capability of language guarantees, functional correctness and security are often tackled by formal verification [36]. Various tools have been developed for Rust verification [15, 28, 34], but so far no industrially deployed systems software written in Rust has undergone verification.

Trusted Execution Environments. Much software nowadays runs in a cloud environment where infrastructure such as operating systems is provided by the cloud vendor. This infrastructure has a higher privilege than the software that may contain sensitive data, and if compromised it presents a threat to privacy and security. To tackle this problem, trusted execution environments (TEEs) are used. They offer security mechanisms such as remote attestation to ensure that the right user software is loaded, and then run it in a separate domain called the 'Enclave' isolated from the privileged yet untrustworthy environment. Due to their critical role in ensuring privacy and security, the correctness of TEEs is vital. A bug in a TEE could result in arbitrary code execution [1] or leak of private information [2].

Currently the most prominent examples of TEEs are Intel SGX [41], ARM TrustZone [14], AMD SEV [12], AWS Nitro [13] and Arm CCA [32]. Except Nitro, they are all implemented by hardware and firmware. However, hardware and firmware solutions are difficult to evolve and audit. An alternative is to implement TEEs as hypervisors and use system virtualization to protect the application and its sensitive information [24].

A key ingredient of the correctness and security of TEEs is the correct construction of their paging subsystems. They are directly responsible for *spatial isolation*, which means the untrusted domains can neither peek into or overwrite the private memory of enclaves. In hypervisor based TEEs, spatial isolation is achieved by virtualizing memory mapping, namely nested paging. Nested paging adds another layer of address translation on top of those controlled by each domain. The hypervisor i.e. the TEE is responsible for the translation via a separate set of page tables called extended page tables (EPTs). Even if malicious domains try to access enclave memory, the hypervisor can ensure isolation by carefully setting up its address translation so that any guest physical address from untrusted domains is mapped outside enclave memory. Therefore, the correctness and reliability of the paging subsystem is vital.

Our work. In this paper, we aim to prove systems software written in idiomatic Rust. Specifically, we prove the correctness of the paging subsystem of an industrial hypervisor TEE, namely HyperEnclave [21, 24] developed by Ant Group. HyperEnclave is a software TEE that uses EPTs to ensure the spatial isolation of the enclave, and is written in Rust to minimize the possibility of memory bugs. HyperEnclave is not merely an academic project, but is already used in production for cloud services and has been released as an open source project. We verify a version of HyperEnclave from early 2022 which is not identical to the open source release in July 2023. Still, even in 2022 HyperEnclave was used in production.

Challenges and Solutions. Proving the correctness of HyperEnclave, even only for the memory subsystem, is nontrivial. It is a complicated project with 2130 lines of idiomatic Rust code using sophisticated language constructs, and it interacts with low-level hardware in various ways. It also consists of multiple components: physical memory allocation, page table entry manipulation and address space management.

The first challenge is to choose a suitable tool for verifying HyperEnclave, which is quite different from verifying a library function, as our proofs must be able to compose. Existing verification tools such as Prusti [15] or Verus [28] are mainly concerned with handling the language soundly instead of scaling proofs. We follow SeKVM [31, 42] and formulate the proof of HyperEnclave in a layered fashion, by dividing our proof into 15 layers that span from frame allocation to address space isolation. This makes the proof more scalable, because each proof layer only sees the specification of the layer below, not the implementing code. To handle the variety of language constructs used, we create a formalized semantics of Rust at the Mid-level IR (MIR) level and model HyperEnclave with it. MIR is much simpler than surface Rust while also maintaining memory safety.

Another challenge arises from the fact that HyperEnclave is written in idiomatic Rust, instead of code specifically written for verification. Many programming constructs such as pointers require specific techniques to verify, and are often avoided by code specifically written for verification. For example, CertiKOS uses array indices instead of pointers. Ideally, we would like to rewrite as little code as possible for HyperEnclave to make verification tractable, so we must adapt our proof techniques to the constructs that are present in the code.

In particular, MIR code uses pointers pervasively, especially self pointers (`&self` in Rust) and pointers to page table entries. These pointers can be passed through functions in multiple proof layers. This breaks encapsulation in the original layered proof approach, where data is only visible to the layer it belongs to. Other layers cannot directly see the data, and must use the interface functions exported by the owner layer. To ensure modularity while avoiding a total rewrite into a pointer-free HyperEnclave, we model these pointers as opaque handles that are only usable in the layer that owns their data, guaranteeing that cross-layer pointers do not break encapsulation by sharing memory. Moreover, we leverage Rust's memory safety guarantees to use an object-view memory model without aliasing or pointers forged from integers.

We offer the following contributions in our work:

- We design a modular framework *MIRVerif* which can verify idiomatic Rust systems code based on a lightweight semantics of MIR (Sec. 3).
- In particular, we devise ways to handle pointers while preserving the encapsulation property of layered proofs (Sec. 3.4).
- We verify the paging subsystem of a realistic, industrially deployed software TEE. We prove functional correctness (Sec. 4) and information-flow security (Sec. 5) of the most important code while remaining within a practical budget (Sec. 6).

2 Background

TEEs provide an isolated area called the *enclave* for the processing of sensitive code and data. The key features provided by TEEs include remote attestation which reports the state of the enclaves to a remote party in a trustworthy way, and memory isolation which prevents unauthorized access to the enclaves. In this paper, we focus on ensuring memory isolation for HyperEnclave. Remote attestation is out of our scope.

In this section, we present an overview of the HyperEnclave design focusing on the isolation mechanism, define our threat model, and describe the code modifications we made to HyperEnclave for verification.

2.1 Overview of HyperEnclave

As shown in Figure 1, HyperEnclave mainly consists of a trusted software layer running in the host mode, referred

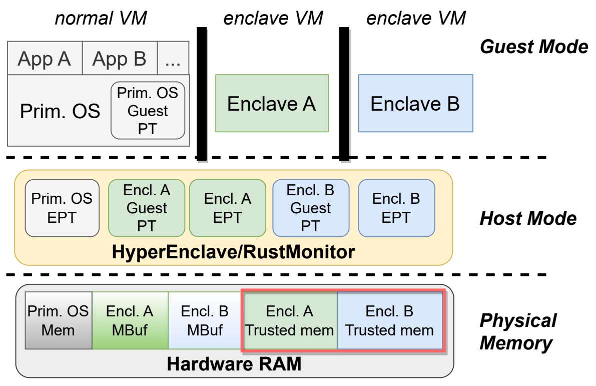

Figure 1. HyperEnclave architecture. The shadowed red box indicates dedicated secure memory. PT and EPT boxes stand for page tables. MBuf stands for marshalling buffer.

to as the *RustMonitor*[1], which is responsible for enforcing memory isolation. Similar to Intel SGX, user applications are split into an untrusted part and a trusted part. HyperEnclave creates a virtual machine called the *normal VM* to run the untrusted OS, referred to as the *primary OS*, and the untrusted parts of applications called the *App*s. It also manages multiple VMs for the trusted parts of the applications—these trusted parts are called *Enclaves*. Transitioning execution between the trusted and untrusted parts of applications is handled securely by RustMonitor. This architecture is different from traditional hypervisors where each VM is usually self-contained and runs separately from other VMs. Still, the isolation requirements are the same: the enclaves and the primary OS must be isolated and only through intended channels may they communicate.

The enclave life cycle is managed through a set of primitives from VMs to RustMonitor, implemented as emulation of privileged SGX instructions such as ECREATE, EADD and EINIT. To this end, a kernel module running in the primary OS provides similar functionalities by invoking RustMonitor through hypercalls, and exposes the functionalities to the applications by the ioctl() interfaces. By emulating the privileged SGX instructions, RustMonitor is responsible for the management of the enclave life cycle. Upon an enclave state transition, either synchronous or asynchronous enclave entry and exit, RustMonitor switches the virtual CPU (vCPU) mode by restoring the vCPU state, switching the guest page table (GPT) and the extended page table (EPT), and also flushing the corresponding TLB entries.

Memory Isolation. HyperEnclave enforces memory isolation with the hardware support for memory virtualization, specifically EPTs. During system boot, a portion of physical memory is reserved for HyperEnclave, which is used by RustMonitor and used as the secure memory of the enclaves.

[1]RustMonitor is only part of HyperEnclave, but in this paper we treat them as synonyms.

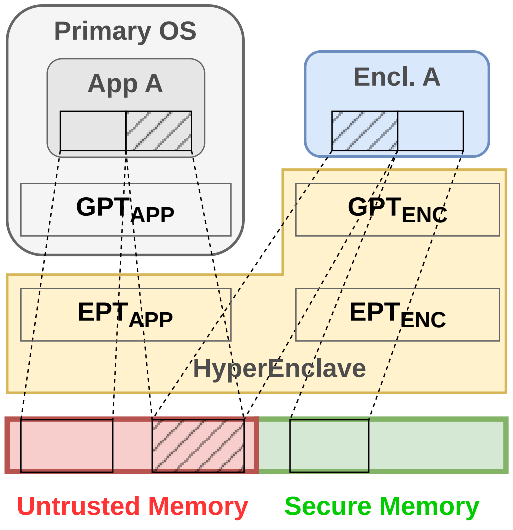

Figure 2. View of address translation. The hatched areas denote the marshalling buffer.

The reserved secure memory is managed by RustMonitor itself, while other memory which we call normal memory is managed and used by the primary OS. We borrow the SGX term and refer to the secure memory as Enclave Page Cache (EPC) memory. Most importantly, HyperEnclave creates separate EPTs for each enclave and the normal VM hosting the primary OS, and has itself rather than the untrusted OS manage these EPTs. To enforce isolation, HyperEnclave must take care in correctly setting up these EPTs and protect them from any malicious accesses.

Figure 2 shows the memory mappings of the applications within the normal VM and the enclaves. An enclave can only access its own protected memory and a *marshalling buffer* (explained below) shared with the untrusted application. The untrusted application and the primary OS are not allowed to access the EPC memory i.e. physical memory reserved for the enclaves or RustMonitor. For this purpose, RustMonitor maintains a data structure (i.e., Enclave Page Cache Map, EPCM) to store the EPC page states, and checks the correctness for memory allocation.

As shown in Figure 1 and 2, all the EPTs of the normal VM and the enclaves are managed by RustMonitor and cannot be exploited by the attacker directly. Furthermore, to prevent possible page table attacks [16, 46, 49] where the primary OS manipulates the enclave's page table, all enclaves' GPTs are managed by RustMonitor, while the GPTs of the primary OS and the Apps are still managed by the untrusted primary OS. With such design, the only memory region susceptible to attacks is the one allowed by the EPT of the guest OS, no matter how the untrusted OS modifies its own or its applications' GPT. Spatial isolation is achieved by ensuring that the aforementioned region is disjoint from accessible regions of enclaves, except for intended communication channels controlled and audited by RustMonitor i.e. marshalling buffers.

Marshalling Buffer. Unlike SGX, in HyperEnclave the enclave is not allowed to access the entire address space of the application. The address mappings of the application are controlled by the untrusted OS. Allowing the enclave to access the application's address space would give the untrusted OS a chance to manipulate address mappings of the enclave, known as "mapping attacks". To support passing data between the enclave and the application, a marshalling buffer in the application's address space is allocated from normal memory, and is shared with the enclave. The mappings of the marshalling buffer are fixed during the entire enclave life cycle. The marshalling buffer is the only channel where any data exchange between the application and the enclave is allowed.

2.2 Threat Model

The goal of HyperEnclave is to protect the code and data of enclaves from the potentially malicious primary OS. We do not consider the case where a buggy enclave voluntarily leaks data through its marshalling buffer. We assume the primary OS to be untrusted and possibly controlled by an adversary, with the following capabilities: (1) arbitrary memory access or malicious DMA to peek into or overwrite enclave memory; and (2) initiating hypercall sequences to try to tamper with the metadata within RustMonitor and subsequently trigger a hidden bug in memory management.

In both HyperEnclave and our verification, the Rust compiler and its safety guarantees are trusted. We also trust the verification tools, including Coq and the transpiler which converts MIR to Coq representations. We consider the underlying hardware as trusted. Similar to many other TEEs, denial of service attacks and side channel attacks are out of scope.

2.3 Retrofitting HyperEnclave to Verification

In HyperEnclave, some Rust language constructs are rarely used yet require significant effort for verification, particularly when working at the MIR level as we do. Therefore, we make some changes to the HyperEnclave code, but we prioritize keeping the modifications succinct. The changes we make are as follows:

1. We move large loop bodies into separate helper functions to make it easier to structure the Coq proofs (it can be confusing to keep track of where you are in the middle of the proof of a large function).
2. We replace closures with equivalent code in a few places. In HyperEnclave higher-order functions are only used to make the code slightly shorter, but in the generated MIR code an anonymous function turns into a separate named function and code to call it, so there is no point in using it.

3. We change `enums` with integer values into plain integer constants. When Rust code casts an enum value to an integer it generates a `discriminate` MIR instruction. By avoiding enums with specific values we can simply get rid of such `discriminate` instructions.
4. We hardcode some constants about the memory layout. The original code uses the Rust `lazy_static` crate to make these configurable. But then any function that uses a constant gets a block of code spliced in to initialize the static variable lazily, which can take a very simple function (which e.g. just does a single subtraction of an offset from an address) and turn it into a complicated block of code which is difficult to verify.

The performance impact of these changes should be minimal. At most we introduce one extra function call in some loops (and the loop occurs inside a hypercall to initialize a new enclave, so it is not called often).

3 The MIRVerif framework

The HyperEnclave memory isolation might fail for the two reasons:

1. Design errors—the intended setup of the page tables (which ranges are mappable, which pages can be aliased, etc) might not correctly isolate the enclaves.
2. Programmer errors—even if the design is flawless, the implementation of the hypervisor might contain errors such that the design is not correctly realized.

This leads to a natural way to decompose verification tasks: we give the program a high-level *functional specification*, in which its relevant interfaces—in this case, hypercalls—are described in terms of pure functions on system states, and we prove the high-level specification satisfies *security properties* and invariants. Then we separately prove that the actual code correctly implements the specification, and consequently the code satisfies security properties as well. The two parts are discussed in detail in section 5 and 4.

In this section we discuss our framework for proving functional correctness (conformance to a functional specification) of Rust code as shown in Figure 3.

We take a pragmatic approach to verification: it is rarely feasible and cost effective to verify a realistic piece of code *en masse*. Rather, in the CCAL-style of verification, functions are arranged in a dependency hierarchy, such that a correctness proof of a function in a high layer may depend on the correctness of a function in a lower layer. Every function verified reduces the attack surface of HyperEnclave. Ideally, we would like to shrink the attack surface by verifying all functions. We, however, take a more realistic approach, verifying only safety-critical functions; thereby balancing the trade-off between that additional security and available resources.

For functions that handle low-level architectural details or come from dependencies such as third party crates or

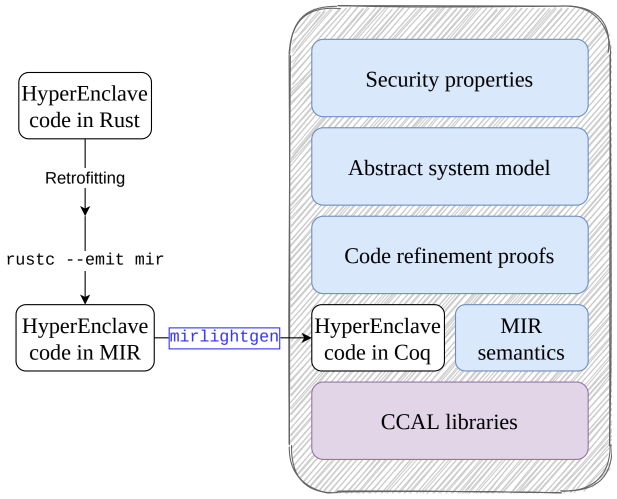

Figure 3. MIRVerif architecture. Elements in the hatched box are mechanized in Coq.

the standard library, we choose to declare them "trusted" and assume their correctness, including them in the trusted computing base. “Trusted” functions can later be pulled out and verified as more resources become available, or if we are led to believe that a particular function poses a higher risk than previously thought. This way we can tune the verification to the available resources.

In this paper, we declare low-level functions that go beyond the assumptions of our Rust semantics to be trusted. We also choose to declare some other functions as trusted in order to limit the scope of verification (see Sec. 4.2 for details).

3.1 Formalizing the Rust Semantics

To be able to verify Rust code (at least for the fragment used by HyperEnclave), we need a formal semantics for the Rust language. Although other authors have treated subsets of the language [3, 45], it is hard to integrate those works into the CCAL framework, so we developed our own semantics for MIR as a deep embedding in Coq.

The level of abstraction in MIR is comparable to the C programming language: there are still high-level expressions and statements, but the type system is much simpler than the surface Rust type system. In MIR, the compiler has typechecked the program and resolved all the trait functions, so, unlike surface Rust, the operational semantics are determined by the terms of the program and we do not need to model the type system. Similarly, the compiler has inserted “drop” statements at the end of the lifetimes of variables, defunctionalized lambda-expressions, and Rust “references” have been turned into pointers.

MIR programs are formatted as control flow graphs, where each labelled block consists of multiple statements followed by one “terminator”. We define the program syntax as a datatype in Coq (28 types of expressions and 11 statements/terminators are supported), and then define a small-step operational semantics in the style of CompCert [29], a verified compiler for C. We validated our semantics correctness by careful inspection of the Rust documentation.

3.2 Leveraging Language Guarantees for Verification

The MIR semantics are not very different from the C semantics used by the previous CCAL works, and we actually reuse part of their supporting definitions like arithmetic. However, the major difference is that Rust ensures memory safety and is free of dangling pointers, so our semantics are designed to exploit these properties to simplify verification.

Object-view Memory Model. In HyperEnclave, there are only a few unsafe type-casts which expose the low-level representation of data structures, all of which are within trusted code that need no proof. And the ownership rules in Rust enforce that pointers in safe code do not alias. Therefore, we can use a high-level representation of memory. We view the memory in a structured way, as collections of non-overlapping objects, different from the flat-array-of-bytes (or "blocks-of-flat-array-of-bytes") view in C. The C approach causes extra proof burden whenever a pointer is used because: (1) it needs to prove that each pointer points to a valid memory region; (2) it needs to prove that the types of the pointer and region match; and (3) in the case of aliasing, a single update could alter multiple values.

We take an object-view of memory and handle structs/enums as values rather than a block of contiguous memory, as in the following formula. In this way access to objects is made simpler, and is free from the above burden. A struct/enum object is represented with an integer discriminator and its list of fields, and our evaluation rules are concerned with projecting out fields directly, rather than resorting to complicated field offset logic.

value	:=	int	Integer values
		...	Other atomic values
		(int, list value)	Structs and Enums

Also, because we got rid of the flat-array view of memory, we propose using paths rather than integer addresses to locate objects. A path simply consists of an identifier with a list of integer indices, essentially the base object and a list of projections. For example the expression `foo.bar.1` will be modeled as `GlobalPath IDENT_foo [OFFSET_bar 1]`. The object-view of memory means that our proofs do not need to reason about how objects are laid out in memory. Assignment to memory creates a new (binary) relation, which we axiomatize as only changing at the assigned location.

Note that our model is only valid if there is no aliasing or cycles in memory, which is the case for Rust-allocated memory in HyperEnclave.

Memory Safety Implies Pointer Validity. Our Coq semantics for MIR does not model the Rust ownership type system, but we implicitly assume that it is correct by modeling the code as *never freeing* locally allocated objects in memory. In the actual code, the Rust compiler will deallocate a variable when the last reference to it goes out of scope, e.g. at the end of a function body, but this is treated as a no-op by us, similar to how one may specify the semantics of a language with garbage-collection (we still model the call to explicit "drop" functions). This means that functions can return pointers to locally allocated data, and the proofs never have to explicitly prove that a pointer is still valid. The surface Rust type system ensures that no pointer is used after an object is freed, so programs compute the same result under our semantics and the semantics with deallocations.

Lifting Local Variables. Another novelty lies in the treatment of local variables. Existing works (such as the Miri interpreter [43]) treat all MIR variables as lvalues which are allocated in memory. However, this adds an extra layer of indirection to all uses of variables, which we would like to avoid as much as possible to make proofs easier. Therefore, our MIR translator makes a distinction between "local" and "temporary" variables. Any variable that has its address taken is a local, while other ones are temporary, similar to e.g. the mem2reg pass in LLVM [9] . In the semantics there are two different operational rules: assignments to a local variable are written into the memory, while the values of temporary variables are kept in a "temporary environment" which only exists during the execution of the function. The net effect is a change in the point of execution where memory allocations takes place. In our semantics, a function which uses temporary variables will not itself modify the memory, but will just create values, which can be returned to a different function and be part of a bigger complex value, etc. Eventually the returned value will reach a function which assigns into a global variable or takes the address of a variable, and at that point, the value is written to memory – but until then, there are no side effects.

Unlike the previous two ways, lifting temporary values does not rely on guarantees ensured by the Rust language. In future work one could prove the soundness of the transformation, by proving that the program would eventually compute the same value if all the variables were treated as local. However, it is very helpful in practice by abstracting away the details of the Rust memory. In particular, the `memory` module of HyperEnclave has 77 Rust functions, but only 12 of them involves local variables. The other ones can be treated "functionally" as constructing and returning Rust values.

3.3 Importing MIR into Coq: `mirlightgen`

In order to import the Rust code into Coq so we can reason about it, we use a modified version of the standard Rust compiler `rustc` to pretty-print MIR into our Coq-based syntax. In other words, we take advantage of the fact that `rustc` can already print out the entire MIR representation of a program in human-readable form, and make it instead print the MIR representation out as an abstract syntax tree inside a Coq `.v` file. We call the tool `mirlightgen`, as a pun on Compcert's `clightgen` tool which plays a similar role. The result is a "big blob" of code. In order to verify it, we need to split it up into per-function code files, and order them into layers based on the call graph. This was done semi-manually with the aid of some ad-hoc scripts.

This design gives us high confidence that we are verifying the correct code, since we are verifying the same MIR code that the Rust compiler is operating on.

3.4 Integrating Rust Semantics with Layered Verification

MIRVerif is based on the the layered proof approach i.e. the CCAL framework because of its ability to compose and its modularity, and the previous successful uses of it in the CertiKOS [22] and SeKVM [31] projects. Yet the previous experience were in C and it has to be extended to handle Rust.

Background of the CCAL framework. The original CCAL framework is built on top of CompCert's formal operational semantics for C. It extended the C semantics to add a user-defined *abstract state* of the system undergoing verification, and views function executions as relations between abstract states, i.e. changes to the system state. We similarly add an abstract state to our MIR semantics. The abstract state encapsulates concrete memory to make the proof more modular, avoiding reasoning about memory in higher level proofs. The system will be divided into layers of functions depending on the caller-callee order. The design of HyperEnclave ensures that there are no functions from higher layers passed as callbacks to lower layers. Each C function will be proven against a specification in Coq, which is a functional specification that defines its behavior in terms of effects on the abstract state and the return value. These specifications usually have a type signature similar to `Args * AbsState -> Ret * AbsState`.

This approach makes use of data encapsulation in the program to enhance proof modularity, by making each variable in memory be "owned" by a particular layer. Functions inside that layer are verified with respect to the concrete memory semantics when they access the variable, but then the user supplies a refinement proof that the function behavior is equivalent to the specification in term of abstract state. Later, callers in higher layers do not need to know anything about the concrete memory representation.

3.4.1 Augmenting CCAL with Pointers.

Existing successes like CertiKOS and SeKVM were written with verification in mind from the beginning. As such, they avoid certain C idioms in order to make it easier to write proofs. Most conspicuously, they avoid the use of pointers and nearly all functions take integer arguments. For example, for thread identifiers CertiKOS always uses an integer index rather than a pointer to the thread control block; SeKVM accesses page table entries only through a one-line wrapper function and never actually dereferences the pointer.

By contrast, HyperEnclave is idiomatic Rust with a lot of object-oriented code, and uses of pointers are pervasive. Nearly every trait method comes with a self reference (compiled into a self pointer at MIR level), functions can take struct arguments via a pointer, and page table entries are accessed directly through pointers. A complete rewrite of HyperEnclave into pointer-free style certainly contradicts our goal, as the changes are expected to be minimal. This presents a new research problem: how can we do modular verification without leaking any details about the memory representation in a layer, if functions are receiving or returning pointers to that memory?

Our solution to this is to further extend the language semantics with new types of pointers which enforce module boundaries. The syntax of the language remains the same, but we specify the semantics as if there are different types of pointer values at runtime, and the semantics of a pointer read/write depends on what kind of pointer it is. When we write specifications for functions that take or return pointers, the type of pointer we use in the specification is is determined by how the pointer is used; there are three types as shown in Figure 4.

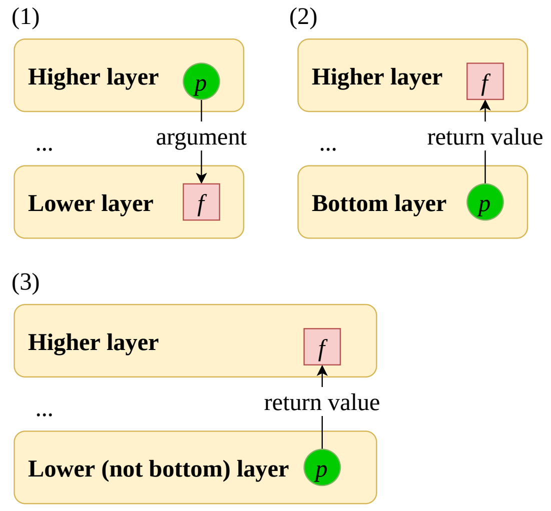

Figure 4. Pointer classification. The red box f indicates a function in that layer. The green circle p indicates a pointer whose pointee belongs to that layer.

Pointers passed to lower layers (1). In this case, a pointer is allocated by a function and passed as an argument to lower layers. This is the easiest case: there is no modularity concern because the caller is the owner of the object, so proofs about the caller are allowed to know about concrete representation. This is handled using normal pointers into concrete memory (in our case the path addresses described in Section 3.2). The specification of the function lower in the stack says that it expects the pointer to point to some object, and the correctness proof of the caller proves that it does.

Pointers from the bottom layer (2). However, sometimes the object-based concrete memory model is inadequate. In HyperEnclave one example is accessing the physical memory for the page tables. This is a security-critical feature and relies on low-level manipulations of individual bits of addresses, so (similar to SeKVM) we want to carefully specify a lowest layer where this memory area is represented as just a plain array of 64-bit words, so we can prove the functions did it right.

In SeKVM, that was done by rewriting the code to only access the page table through two particular "load/store" functions, and then providing (simple, trusted) specifications of what those functions do. However, we do not wish to do extensive rewrites of the HyperEnclave code, so the page table code that we need to verify contain pointer reads and writes that do not make sense in terms of the object-based view of memory.

Our solution is to add what we call *trusted pointers* to our language semantics. Instead of containing a memory path, trusted pointer values contain getter/setter functions that can access the abstract state, and the semantics of a pointer write is to call the setter function and update the state accordingly.

In particular, the abstract state type in our development contains the array representing physical memory. The few unsafe Rust functions that cast raw integers into pointers are ascribed specifications that return trusted pointers (containing load/store specifications), and the rest of the code can be verified in terms of those. We get the benefit of a load-store abstraction layer without having to rewrite the code.

Pointers from the lower (but not the bottom) layer (3). Another challenge is when pointers to objects allocated by lower layers are returned to a higher layer. This occurs very commonly in code using Rust traits, which are compiled into "self" pointers. A layer will define a struct type which is only allocated and should only be accessed through member functions defined in the same layer.

If we tried to specify these functions using concrete pointers, we would burden the caller with the details of exactly where these objects are stored in memory. Instead, we extend the semantics with *RData pointers*, where the payload inside the pointer value is just an identifier and a list of numerical indices. Our MIR semantics do not provide *any* way to read-/write through an RData pointer. Therefore, we can specify the intended semantics of the layer without any way for the clients to break encapsulation: the only thing that can be done with the pointer is passing it back to the layer that forged it.

Of course, it is not possible to verify the code of the methods with respect to this semantics, because the code does load and store through pointers. Instead, the functions are verified in the concrete Rust memory model, and then we do a refinement proof showing a simulation from the RData pointer specifications to the concrete memory semantics. In this way, we hide the concrete memory representation inside the layer, just like in the original CCAL approach.

4 Proving Functional Correctness

The preceding section outlines our lightweight MIR semantics and handling for language specific features. To actually prove HyperEnclave code, we need to use our semantics to model the actual Rust code, and prove that it conforms to our functional specification via refinement. The function specifications describe the intended behavior of each Rust function, and are the basis for higher level security proofs.

HyperEnclave is a quite large project accounting for over 5800 lines of code, and even only the memory subsystem is over 1000 lines of code. To make our proofs more scalable, we decompose the Rust code into layers of functions, such that each layer depends only on functions in lower layers. This approach enables functions to be proven in a modular way [23]. The proofs of higher-layer functions use specifications of those of lower layers, which encapsulates all the implementation details of the lower layer functions.

4.1 Functional Correctness Proofs for Page Tables

For the functional correctness proof, we verified the functions in the memory module of HyperEnclave, specifically those querying and updating the page tables. These functions walk the page tables for a virtual address, look up intermediate entries, allocate new intermediate frames by need, and ultimately retrieve or install a terminal entry. Entries are represented by plain 64-bit integers in the implementation, and each consists of two parts: a physical address and its associated flags. To update entries, we need to read or write to a specific physical address. As discussed in Sec. 3.4, we ascribe specifications in the trusted (bottom) layer to functions from these crates in terms of the abstract data. The abstract data contains a big flat array of integers representing the physical memory of the frame area, along with the specification for writing a page table entry to update the array.

In principle, we could end up with a single specification that views the page tables as a unstructured flat array of frames. However, proving invariants using the flat frame representation would be difficult, as walking a page table involves repeatedly following pointers and updating it. The flat representation does not rule out aliasing issues, which would happen if two entries pointed to the same intermediate page table or frame. With aliasing entries, installing a new mapping could be a non-local change potentially changing unrelated entries. Thus, we use two specifications for page tables: a *low spec* which is a flat representation as described previously, and a *high spec* which is a tree representation for use by the higher layers.

To define the tree-shaped high spec, first we abstract the 64-bit page table entries into a parameterized record, like in the code below.

```
Record PTE {content:Type} := mkPTE {
 addr_content: option (int64 * content);
 flags: list bool;
 unused_inv : addr_content = None
     -> (is_huge = false /\ is_present = false)
}.
```

Entries do not store an indirect index to the next page table, rather they contain the next page table directly (the `addr_content` field). Such nesting constitutes a tree-shaped view of page tables. If this is a terminal entry pointing to a possibly huge physical frame, `content` will be `unit` type. Otherwise, as page tables are just map from indices to entries, `content` will simply be a `ZMap`. For example level three page tables are defined as `l3pt := ZMap.t (option (@PTE (l2pt + unit)))`; options indicate possible absent entries; `l2pt` are type of level two pages tables and the plus sign indicates the (tagged) sum type.

To relate a low spec to a high spec, we use a refinement relation R over two abstract states d_1, d_2, which are states in tree and flat representation respectively. The relation $R\, d_1\, d_2$ holds if the page tables viewed as trees in d_1 agree in content with those viewed as flat memory in d_2. Defining R requires another relation $R_{\mathrm{pte}}\, p\, a$, which relates the `PTE` record p to the entry address a. If p is a terminal `PTE`, R_{pte} simply says `addr` and a should agree. Otherwise R_{pte} quantifies over page table indices and says that entry at each index should be recursively related (with R_{pte} itself) to a plus some offset. With R and R_{PTE}, we can finally prove a simulation from the flat representation to the tree representation, and that both are the same. This assures us to use the tree representation to simplify subsequent proofs.

Malformed Page Tables in the Wild. The code proofs, and in particular the refinement proof, shows that the page tables are correctly represented in physical memory. A simpler alternative would be to just write a model of the entry manipulation functions, perhaps operating directly on the tree-view representation of page tables, and ignore the Rust code and the flat-view completely. The question is: does our additional effort improve confidence?

In fact, during the development of HyperEnclave we did encounter a security bug related to this. In an earlier version, the page tables of the enclaves were not constructed from scratch; instead they were initialized by a "shallow copy" of the page table of the primary OS. The copy selected the relevant address ranges from the level-4 page table, but otherwise copied the existing entries. This is not secure, because HyperEnclave's page tables would then contain pointers to level-3 tables that are stored in physical memory controlled by the guest. Indeed, such a program would be impossible to prove in our setting, because we need to prove that the initial (empty) page table satisfies the relation R. If it is copied from arbitrary guest memory, there would be no way to prove that all the entries are inside HyperEnclave's frame area (the flat representation).

4.2 The Trusted Layer

At the very bottom of our layers is the *Trusted Layer*. It contains the specifications of functions that will not be verified, including the Rust standard library, 3rd party crates, and architecture-specific functions. More importantly, it also includes the primitives for interacting with the HyperEnclave global state, such as primitives that update page table entries. Rather than directly model the memory layout of the state, we define them in terms of abstract operations on state that are called from the specifications of trusted functions.

4.3 Code Proof

A benefit of CCAL is that is allows us to create 'sublayers' even when proving the code conforms to the specification. As refinement is transitive, we can insert a 'low spec' between the specification (now called the 'high spec') and the code. This partitions the proof into two parts when we have an intermediate spec, code proofs (from code to 'low specs') and refinement proofs (from 'low specs' to 'high specs'). For HyperEnclave we used these intermediate specifications only for the page tables, as we discussed in the previous section. Still, in either case we have to prove the code conform to some spec, be it 'high' or 'low'.

As in previous works in CCAL [22, 31], we reason about HyperEnclave code with our MIR operational semantics, and we prove that for any two initially related states, the effects as well as the return value of executing the HyperEnclave function (with MIR semantics) and executing its specification should agree.

Part of the reasoning is made automatic with custom Coq Ltac's [10], though we have to supply loop invariants manually.

4.4 Tuning Verification Coverage

As a practical matter, fully verifying the entire code base including language dependencies of a production system like HyperEnclave is a major challenge. It would be infeasible for a product release, for instance, to be held up while waiting for verification to finish. Instead, we rely on the separation of layers to verify the system piecemeal, from top to bottom, verifying as much of the code base as is feasible with our resources. Every function we verify is one fewer function in the trusted computing base, raising our confidence in the correctness of the whole. In this way verification can be amortized over time, with continuing work after release filling in gaps in the proof.

Rust is particularly suited for this approach because unverified functions in Rust have a lower impact than those in unsafe languages such as C. Undoubtedly, a proof with any holes at all is less reliable than a complete proof, and this is indeed a threat to our validity as we discuss in Sec. 6.1. Still, in C, leaving any piece of a program unverified risks undefined (memory) behavior due to the unrestricted use of mutable pointers, which might change any part of the system state in an unpredictable manner. By contrast, in Rust, as long as a function is memory-safe, a bug might result in a change in its return value or mutable parameters. Memory safety here essentially limits the range of blast whenever a bug is present in the unverified code, and ensures that unrelated code cannot tamper with each others' data.

5 Proving Security Properties for HyperEnclave

As discussed HyperEnclave must provide proper spatial isolation between enclaves and the untrusted primary OS, so that private data are not leaked or tampered with. We prove a number of invariants that hold during execution, then use these invariants to prove that HyperEnclave satisfies a *non-interference* property, one of an an important class of security property from the formal security literature [40].

Intuitively, a simple symmetric form of noninterference says that, if we divide the system into security principals, which are VMs in our case, each with permission to observe a subset of the system, no principal can deduce any information about a part of the system that it lacks permission to observe, or about actions taken by the other principals. This directly prohibits the untrusted OS from being privy to sensitive data, achieving confidentiality. A dual result is integrity, which implies the untrusted OS cannot tamper with sensitive data. If the untrusted OS were to overwrite private data that enclave operates on, the enclave would receive information about the OS, so the same non-interference theorem rules out this also.

5.1 System Transitions

To formally verify the noninterference property, we need to formally define an abstract model for how HyperEnclave operates, in the form of a transition system. Most importantly, this involves defining all possible steps that the primary OS or enclave can take, either through local computation or by making a hypercall into RustMonitor. In Coq, we define each step as a function mapping the abstract system state to a new one incorporating changes made by the step.

The local computation steps are considered nondeterministic, as HyperEnclave does not care about the exact computation happening inside each VM as long as it does not cause a VM exit. Instead we have two specifications, `mem_load` and `mem_store`, which model what happens when the VM accesses an arbitrary virtual address. The address is resolved using the current installed page table of either the primary OS or the enclaves. As part of these specifications we need a function representing the page table walk that the CPU performs; instead of manually writing this function in Coq (which we could get wrong), we actually use a corresponding page-walk function that is part of the `memory` module of HyperEnclave, which we have a verified Coq specification for. The load/store also has a special case for addresses in the marshalling buffer range (See Sec. 5.4). If the page walk succeeds, the physical memory and the current register set is updated.

The hypercalls are more interesting, since they are trapped into the RustMonitor. In the HyperEnclave implementation, we have the following hypercalls: `init` and `add_page`, which are called from the primary OS to create and initialize a new enclave; `enter` and `exit`, which transfer control into and out of a particular enclave. In our proofs we only model the first two, by inspecting the safe part of the Rust code and updating the abstract data accordingly. The other two hypercalls (`enter`/`enter`) do not manipulate the page table entries, and the point of them is mostly in the x86_64-specific details of installing a new register set etc. Proving them correct would be less informative about the memory isolation guarantees.

5.2 Page Table Invariants

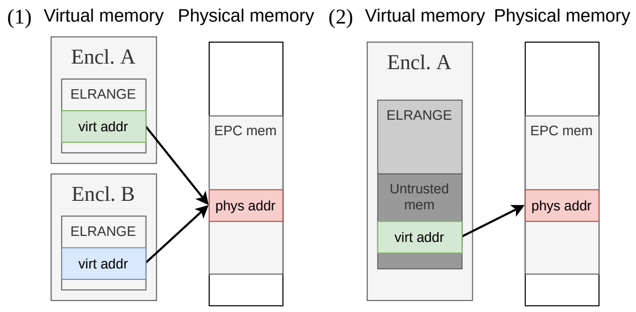

Figure 5. Example cases where a wrong page table design would lead to possible exploits. ELRANGE stands for the accessible virtual address space of an enclave.

The most important yet subtle part of our security proof is the invariants that the page table mappings must satisfy. Figure 5 illustrates some of the cases we must worry about. The first is a page alias, which happens when two different enclaves interfere with each other by having permission to access the same EPC page. More subtly, if a virtual address which appears to not be part of the enclave is actually mapped to a physical EPC address, an enclave may be fooled into corrupt its own EPC page when it believes it is writing normal memory.

Following the design of HyperEnclave, we formulate the following invariants for page table mappings:

- *ELRANGE (enclave linear range) memory isolation.* Two virtual addresses va_1 and va_2 that are in the ELRANGE (i.e. addresses used for EPC pages) of two different enclaves, must be mapped to different physical addresses, if there exist such mappings at all.
- *Marshalling buffer invariant.* If two virtual addresses va_1 and va_2 are translated to the same physical memory region by an page table and the page table of the primary OS, then va_1 and va_2 are in the marshalling buffer.
- *EPCM invariant.* All the page mappings in the page tables of enclaves correspond to an entry in the HyperEnclave's EPCM list (i.e., if HyperEnclave was asked to add a mapping, it did not forget to also record this in the metadata). This rules out covert mappings.
- *Enclave invariants.* Each enclave satisfies the following properties: a virtual address is mapped to a physical page in the EPC if and only if the virtual address is in the ELRANGE; the ELRANGE and the range of marshalling buffer are disjoint; and there are no huge pages in the page tables.

These invariants are stated in Coq in 106 lines of definitions, and we prove that the hypercalls preserve them.

The memory isolation invariant protects the enclave (EPC) pages from tampering. The page tables themselves are also protected, because they are allocated in a disjoint range of physical memory which is never in the range of a guest mapping.

5.3 Proof of Noninterference

To ensure confidentiality and integrity of enclaves, we need to ultimately prove noninterference. We use the invariants to prove an information-flow noninterference theorem. In other words, we prove that no matter what an enclave or the primary OS does, it cannot affect other enclaves in any way, except by explicit communication using the marshalling buffer.

As is standard for noninterference, we formalize the security property by defining a observation function $\mathcal{V}(p, \sigma)$ for a principal p on the current state σ. This is the subset of system state that an enclave/guest is allowed to see, e.g. the contents of its own memory. Two states σ_1 and σ_2 are said to be *indistinguishable* to a principal p if p's observations are identical, i.e. $\mathcal{V}(p, \sigma_1) = \mathcal{V}(p, \sigma_2)$. The noninterference theorem states that indistinguishability is preserved by the state transitions, as formalized by the following lemma.

Theorem 5.1. *Starting from two states σ_1 and σ_2 that are indistinguishable to a principal p, after the transition steps the final states σ_1' and σ_2' are still indistinguishable to p.*

The observation for a principal p includes: (1) the CPU's registers if p is the active (i.e. about to take the next transition) principal; (2) p's saved register context, (3) mappings in the page table owned by principal p, and (4) contents of the memory pages that are not shared with other principals. Even though the mapping of marshalling buffer is shared among principals, it is considered observable to both the enclaves and the host VM because the mapping is immutable once an enclave has been initialized. The contents of pages in the marshalling buffer are handled differently, as described in section 5.4.

Intuitively, Theorem 5.1 says that there is nothing p can do to learn any information that it is not supposed to have access to. For example, suppose there is some other enclave q which has some secret value in one of its EPC pages. Other enclaves' pages are not in the set of allowed observations, so from p's point of view, a state σ_1 where q's secret is 41 and a state σ_2 where the secret is 42 are indistinguishable. If there were security flaw in HyperEnclave that violated confidentiality, then p could run a program to somehow learn the secret value and load it into a register. But running such a program would take σ_1 to a state σ_1' where p's register contains 41, while it would take σ_2 to a state σ_2' where the register contains 42. p is allowed to observe its own registers, so σ_1' and σ_2' would *not* be indistinguishable for p, and so the theorem tells us that there is no such program.

Proof Structure. Following the approach used by SeKVM, we decompose the overall noninterference theorem to several step-wise security properties for each primitive.

Lemma 5.2. *Starting from an inactive state σ, if some other principal makes a CPU-local move to inactive state σ', then $\mathcal{V}(\sigma, p) = \mathcal{V}(\sigma', p)$.*

This lemma ensures the integrity of the private data belong to p because if other principals manage to modify p's data, the changes will be observed by p.

Lemma 5.3. *Starting from any two active, indistinguishable states σ_1 and σ_1', if p makes CPU-local moves to state σ_2 and σ_2', then the states are still indistinguishable.*

This lemma presumes p as a potential attacker and ensures the confidentiality of the private data belong to other principals because p's behavior cannot be affected by data from other principals. If there is data leakage from some other principal to p and p exploits the leakage, it will end up in distinguishable states, thus violating the lemma. However, this lemma relies on the condition that the starting states are indistinguishable given that p is active. Thus, we introduce the following lemma to guarantee that condition.

Lemma 5.4. *Starting from any two inactive, indistinguishable states σ_1 and σ'_1, if some other principal makes CPU-local moves to active states σ_2 and σ'_2, then the states are still indistinguishable.*

With rely-guarantee reasoning, the above lemmas are sufficient for the noninterference theorem: two runs of the machines can be decomposed into local and non-local steps in a suitable way. Similar to SeKVM, however, we leave this argument informal and only prove the above lemmas in Coq. In the end we have proven noninterference and it implies that the registers, memory, as well as mappings of enclaves are secure.

5.4 Sharing and Declassification with Data Oracles

Our model takes into account two important complicating factors: concurrency and shared memory. Memory is shared via the marshalling buffer, the pre-allocated memory area used to pass arguments and return values between enclaves and the normal VM. Since it is shared between security principals, it is excluded from the view relations that inform our noninterference property.

To model the fact that data in the marshalling buffer should be considered declassified, we apply the same approach as in SeKVM [31] and use a *data oracle*. Each execution is parameterized by an oracle (a stream of values) and we modify the semantics for memory load and memory store to treat the marshalling buffer separately. In particular, stores to the marshalling buffer are in effect ignored, so they never formally violate the noninterference theorem. Reads from the marshalling buffer are taken from the oracle. Because the theorem is proved for all possible oracles, including the one which returns the same values that were written by other guests, it still covers all possible code paths for the guests.

6 Evaluation

Proof Effort. Table 1 shows lines-of-code statistics. The Memory and Enclave modules of HyperEnclave consist of 2130 lines of code. The Coq development comprises around 15,100 lines of proof (as counted by `coqwc`), plus around 4,500 lines of specification, the MIRVerif framework, and a large amount of automatically generated Coq files for the imported MIR code and layer scaffolding.

The entire verification effort took around 3 person-years, of which 20% were spent on the verification framework, 30% on invariant-preservation and noninterference theorems, 10% on the page table refinement, and 40% on code proofs. (We wrote specifications interleaved with proofs, so we do not show separate efforts for them.)

The noninterference proofs took 6600 lines of proof, which seems comparable to other similar Coq projects.

For the code proofs using our new MIR semantics, we verified the code of 49 functions from the HyperEnclave Memory module (the full module is 77 functions and 1279

Component	Lines	Effort
HyperEnclave	5881	None
HyperEnclave undergone verification	2130	None
MIRVerif framework	3778	0.6py
Page table refinement proofs	4394	0.3py
Code specifications/models	2445	1.2py
Code proofs	4191	
Top-level specifications/models	2015	0.9py
Top-level proofs	6600	

Table 1. Code and proof statistics. py stands for person-years.

lines of Rust code), which are arranged in 15 layers. They compile into 3358 lines of mirlight code as counted by `coqwc -s`, while the proof script size (by `coqwc -r`) is 4191 lines, or 1.25 lines of proof per line of MIR.

For comparison, the SeKVM project used 4884 lines of proof script to verify 2260 lines of C code (counted by `cloc`), or 2.16 lines of proof per line of C. At the start of the project we hoped that the fact that the MIR code is compiler generated and quite verbose would make it less costly to verify than C, and as can be seen this is the case, but not enough to outweigh the expansion factor from the Rust-to-MIR compilation. It would be interesting to know how large the corresponding code would be if rewritten as a C program, since Rust can be quite concise (e.g. hiding a pattern match, if-, and return-statement behind a single question mark ? at the source level). Either way, the we see that this is practical for small amounts of carefully selected Rust code, as in this development, but not for large pieces of software. However, the overall CCAL methodology is agnostic about how the code proofs are done, so as the state-of-the-art of reasoning about Rust code improves it can be scaled up.

6.1 Threats to Validity

As our verification is focused on the memory subsystem, other unverified parts of HyperEnclave, such as handling of hypercalls, are the main threat to validity. Our page tables invariants would be debunked if unverified code somehow managed to modify the tables. This is not possible in safe Rust, because page tables are encapsulated in an owned memory of the page tables structures, and the memory-safety guarantees of Rust prohibit forging a mutable reference to them. Still, unverified unsafe code could bypass Rust's safety rules and modify the page tables.

To mitigate this threat, we manually checked the 105 unsafe blocks in HyperEnclave. The majority of them (74/105) are used to indirectly call unsafe functions, which includes constructing slices, manipulating state-save area and executing assembly. None of the blocks with raw pointer dereferences (13/105) involve page table memory. While this study does not prove them safe, combined with Rust's memory-safety it should increase our confidence that unverified code is unlikely to alter page tables.

Our code proofs rely on the soundness (but not completeness) of our lightweight MIR semantics. MIR is a relatively simple language, and we believe that we have accurately reflected the specification given in the documentation, but if our formalization is unsound, our proofs could be invalid. The proofs about manually written abstract models could be invalidated if we made a mistake transcribing the code.

An error in mirlightgen, or the ad-hoc layering scripts, could also cause HyperEnclave code to be mismodeled. Such errors are less likely, because in our subsequent proofs we verify the refinement between the code and our specification, during which process we examine each generated line.

Finally we trust the rustc compiler and its memory safety guarantees. The rustc compiler could go wrong to render the final object code incorrect, but its correctness is beyond our scope. Our proofs and security arguments are based on memory-safety, and would be invalid if Rust does not actually guarantee it. This is unlikely for a widely-used project like Rust, and also developers are currently formally modelling the type system of Rust [3].

7 Related Work

Verification of Separation Kernels. There have been previous works on kernel verification [23, 26] and hypervisor verification [36, 37, 44]. They have proven similar goals as ours, but they differ from HyperEnclave because they target general kernels or hypervisors rather than secure enclaves.

SeKVM [31, 42] verifies a reduced core of KVM, and ARM-CCA [32] verifies firmware for managing ARM Realms. We share similar techniques by also using the CCAL approach by structuring proofs in a layered fashion. These two systems are written in C avoiding pointers and do not incorporate language-provided guarantees into the proof. Komodo [20] is a verified TEE written in the Vale language that aims also to achieve isolation. The Vale language is designed for verification, and it allows verified assembly code that provides stronger guarantees. In contrast, verification of HyperEnclave has to deal with the intricacies of Rust and MIR.

Few system software written in Rust has been verified, possible due to the quick evolution of the language. RedLeaf [36] verifies an OS kernel in Rust automatically by use of SMT. They also pinpoint the possibility of aiding verification with language guarantees. RedLeaf is a small lab kernel compared to HyperEnclave, and global properties like isolation is not discussed.

Rust Verification. There are plenty of tools for formal verification and they achieved great success, but many of them do not target our language e.g. Frama-C [18] for C and CakeML for ML [27]. There are also automated verification tools targeting Rust, like Prusti [15], Verus [28] and Creusot [19] for SMT-based deductive verification, cargo-klee [34] for symbolic execution, or the Rust Model Checker [7] for model checking. We instead used CCAL-style proofs in an interactive proof assistant in order to be able to to decompose proofs into layers, and to have a general specification language to state complicated properties like noninterference.

There are also works to formalize the semantics of Rust, either as a practical verification tool or to investigate the metatheory of the Rust language. RustHornBelt [35] mechanizes the semantics in RustBelt [25] and is capable of verifying unsafe code. Besides practical verification, work has been done on the formalizing Rust. Patina [38] formalizes Rust with regard to ownership and borrowing, yet the formal semantics are not mechanized. KRust [45] presents an executable semantics a realistic subset of Rust in the K framework [39], but the only way to verify concrete programs is by the built in K proof system, which is less flexible than Coq. Oxide [47] formalizes the semantics of a large subset of Rust at source language level, and with a semantics tested against the Rust test suite. Many of these Rust semantics could be augmented with CCAL-style abstract state and be used for the code proofs, similar to our MIRlight semantics, so as those projects mature they may increase the productivity of future Rust system software verification.

The Spoq tool [33] automates part of the work of writing code-proofs for CCAL-style verification in C; similar techniques might improve the productivity of Rust system software verification too.

8 Conclusion

In this paper, we present `MIRVerif`, our verification framework of idiomatic Rust code with pointers, and use it verify the page table implementation of HyperEnclave. MIRVerif is based on the CCAL framework and extended to Rust, and we establish a lightweight formal semantics for MIR. During the proof we develop ways to handle Rust complexities like pointers, and we manage to rely on language-provided safety to simplify our proofs. In the end we prove that HyperEnclave faithfully implements spatial isolation formulated by our noninterference theorem.

9 Acknowledgements

We thank the anonymous reviewers, and our shepherd, Gustavo F. Petri, for their helpful feedback. This research was supported by Ant Group Research Intern Program, and the National Natural Science Foundation of China (grant numbers U19A2060).

References

[1] Cve - cve-2021-33478. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33478, 2021.

[2] Cve - cve-2021-33478. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30338, 2021.

[3] A PLT redex model of MIR and its type system. https://github.com/nikomatsakis/a-mir-formality, 2022.

[4] Cloud Hypervisor - Run Cloud Virtual Machines Securely and Efficiently. https://www.cloudhypervisor.org/, 2022.

[5] crosvm - the chrome OS virtual machine monitor. https://github.com/google/crosvm, 2022.

[6] Diosix hypervisor. https://diosix.org/, 2022.

[7] Kani rust verifier. https://github.com/model-checking/kani, 2022.

[8] Linus Torvalds: Rust For The Kernel Could Possibly Be Merged For Linux 5.20. https://www.phoronix.com/scan.php?page=news_item&px=Rust-For-Linux-5.20-Possible/, 2022.

[9] Llvm: lib/transforms/utils/mem2reg.cpp source file. https://llvm.org/doxygen/Mem2Reg_8cpp_source.html, 2022.

[10] Ltac — coq 8.15.2 documentation. https://coq.inria.fr/refman/proof-engine/ltac.html, 2022.

[11] Redox - your next(gen) os - redox - your next(gen) os. https://www.redox-os.org/, 2022.

[12] AMD Secure Encrypted Virtualization (SEV) | AMD. https://www.amd.com/en/developer/sev.html, 2023.

[13] Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/, 2023.

[14] Thaynara Alves and D. Felton. Trustzone: Integrated hardware and software security. 01 2004.

[15] 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.

[16] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[17] Shao-Fu Chen and Yu-Sung Wu. Linux kernel module development with rust. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–2, 2022.

[18] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, page 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022.

[20] 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.

[21] Ant Group. Hyperenclave open source release. https://github.com/HyperEnclave/hyperenclave, July 2023.

[22] 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, 2016.

[23] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. *SIGPLAN Not.*, 53(4):646–661, jun 2018.

[24] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[26] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, jun 2010.

[27] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014.

[28] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

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

[30] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799, 2021.

[32] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.

[33] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling Machine-Checkable systems verification in coq. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 851–869, Boston, MA, July 2023. USENIX Association.

[34] Marcus Lindner, Jorge Aparicius, and Per Lindgren. No panic! verification of rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 108–114. IEEE, 2018.

[35] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 841–856, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. Redleaf: Towards an operating system for safe and verified firmware. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 37–44, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Jan Nordholz. Design of a symbolically executable embedded hypervisor. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Eric C. Reed. Patina : A formalization of the rust programming language. 2015.

[39] Grigore Rosu. K: A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*, 2017.

[40] John Rushby. *Noninterference, transitivity, and channel-control security policies.* SRI International, Computer Science Laboratory Menlo Park, 1992.

[41] Matthias Schunter. Intel software guard extensions: Introduction and open research challenges. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, page 1, New York, NY, USA, 2016.

Association for Computing Machinery.

[42] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 866–881, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Miri team. Miri. https://github.com/rust-lang/miri, 2021.

[44] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *IEEE Symposium on Security and Privacy*, 2013.

[45] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51, 2018.

[46] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[47] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982*, 2019.

[48] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Trans. Softw. Eng. Methodol.*, 31(1), sep 2021.

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.