Formally Verifying Trusted Execution Environments with UCLID5

Pranav Gaddamadugu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-200
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-200.html

August 13, 2021

Acknowledgement

I would like to thank my advisor Sanjit Seshia, for his advice, encouragement, and patience since joining the Learn and Verify group. I owe a great deal to my collaborators Kevin Cheang and Dayeol Lee, without whom none of this work would have been possible. I would also like to thank Federico Mora and Elizabeth Polgreen, who have provided meaningful feedback and guidance over the years. Finally, I would like to thank my family and friends, who have been an invaluable source of support.

Formally Verifying Trusted Execution Environments with UCLID5

by Pranav Gaddamadugu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sanjit A. Seshia
Research Advisor

8/13/2021

(Date)

* * * * * * *

Professor David A. Wagner
Second Reader

8/13/2021

(Date)

Abstract

Formally Verifying Trusted Execution Environments with UCLID5

by

Pranav Gaddamadugu

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Hardware enclaves provide integrity and confidentiality guarantees of a remote execution on a machine owned by an untrusted third party using hardware-enforced isolation and attestation mechanisms. Although the sophistication of enclave designs has been increasing, few are formally verified. In this work, we use a formal tool, called UCLID5, to model and verify an abstract enclave platform. We then extend our standard model to encompass a novel extension of the Keystone enclave platform called Cerberus, which enables secure and efficient multiprocessing inside enclaves and reduces enclave initialization latency. We use UCLID5 to formally verify that Cerberus provides strong integrity and confidentiality guarantees to enclave programs.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Sanjit Seshia, for his advice, encouragement, and patience since joining the Learn and Verify group. I owe a great deal to my collaborators Kevin Cheang and Dayeol Lee, without whom none of this work would have been possible. I would also like to thank Federico Mora and Elizabeth Polgreen, who have provided meaningful feedback and guidance over the years. Finally, I would like to thank my family and friends, who have been an invaluable source of support.

Chapter 1

Background and Motivation

In this chapter, we familiarize the reader with a number of concepts associated with formal methods and secure program execution. The first section describes UCLID5, a formal verification tool with which we encode and verify a number of security properties. The second and third sections introduce trusted execution environments and motivate an extension to the Keystone enclave design. The fourth section introduces our verification methodology and our notion of security for enclave programs.

1.1 UCLID5 Primer

Formal methods for system design often require reasoning about both hardware and software. UCLID5 [24] is a software toolkit for formal modeling, specification, verification, and synthesis of computational systems. UCLID5 is an evolution of the earlier UCLID modeling and verification system [7]. At a high level, UCLID5 is designed with the following goals:

- Enable compositional modeling of finite and infinite state transition systems across a range of concurrency models and background logical theories.
- Verify a range of properties, including assertions, invariants, and temporal properties.
- Integrate modeling and verification with algorithmic and inductive synthesis.

Using UCLID5, a user can naturally model hardware as a transition system and verify properties using induction or bounded model checking. At the same time, a user can also model sequential software and verify assertions via Floyd-Hoare style program verification. The UCLID5 compiler translates a UCLID5 model into an SMT [3] specification and invokes a solver to run the verification query. This section provides a lightweight introduction to the modeling and verification techniques we use in this thesis. For a more detailed overview, we invite the reader to refer to the tutorial [1] or the original UCLID5 paper [24].

[1] https://github.com/uclid-org/uclid/blob/master/tutorial/tutorial.pdf

Figure 1.1: Model of the Fibonacci sequence as a transition system

```
1  module main {
2    var x, y : integer;
3
4    init {
5      x = 0;
6      y = 1;
7    }
8
9    next {
10     x' = y;
11     y' = x + y;
12   }
13
14   property ind_strengthen : x >= 0 && y >= 0;
15   property y_le_x : y <= x;
16
17   control {
18     v = induction;
19     check;
20     print_results;
21   }
22 }
23
```

Modeling Transition Systems

Figure 1.1 contains a UCLID5 model of the Fibonacci sequence as a transition system. In this toy problem, the primary property we wish to verify is that $y \leq x$. On line 1, we declare a `module` that encloses the verification context. On line 2, we declare two variables, `x` and `y`, both of which are integers. UCLID5 supports several types, including integers, bit-vectors, and arrays. Lines 4-7 contain the `init` block, which initializes the state of the transition system. Lines 9-12 contain the `next` block, which describes the transition relation of the transition system. Note that in the `next` block, we have variables followed by a single quotation; these primed variables refer to the state of the transition system after one step of the transition relation is executed. Lines 14, 15 contain properties over the transition system that we wish to verify. Finally, lines 17-21 contain a `control` consisting of a sequence of commands that drive the verification of our properties. In this model, we use induction to verify our properties. In this case, we must add strengthening invariants (e.g. line 14) to prove our desired property (e.g. line 15).

Figure 1.2: Model of array search as a sequential procedure

```
module main
{
  var numbers: [integer]integer;
  var tail : integer;

  procedure search(search_value: integer)
    returns (found : boolean)
    requires (tail >= 0);
    ensures (exists (idx : integer) :: idx >= 0 && idx < tail && numbers[
   idx] == search_value) <==> found;
  {
    var i : integer;

    i = 0;
    found = false;
    while (i < tail)
      invariant (i >= 0 && i <= tail);
      invariant (exists (idx : integer) :: idx >= 0 && idx < i && numbers[
   idx] == search_value) <==> found;
    {
      if (numbers[i] == search_value) {
        found = false;
      }
      i = i + 1;
    }
  }

  control {
    v = verify(search);
    check;
    print_results;
  }
}

```

Modeling Sequential Code

Figure 1.2 contains a UCLID5 model of linear search. On line 6, we declare our procedure `search`, which takes in `search_value` as a parameter. The procedure returns a boolean value, `found`, if `search_value` exists in `numbers`. In this problem, we are focused on verifying that the postcondition on line 9 is satisfied. The body of the procedure, on lines 11-23, contains the actual specification of the search code. When we specify the procedure, we must add invariants (e.g. lines 16,17) which allow us to reason about the behavior of the while loop. One of the main differences from the Fibonacci example is that we use the `verify` command instead of `induction` to check our postcondition. By invoking the `verify`, UCLID5 uses Floyd-Hoare style program verification to check our properties.

1.2 Trusted Execution Environments

In the last decade, a new paradigm of secure computing, *hardware enclave*, has been introduced and proliferated. Enclaves provide integrity and confidentiality guarantees for remote execution on a machine owned by an untrusted third party using hardware-enforced isolation and attestation mechanisms. These mechanisms allow an enclave to protect a user program even against highly privileged attackers such as compromised operating systems. Several studies have proposed various enclave designs on different platforms such as Intel SGX [14], Sanctum [10], and Komodo [11]. Such systems enable many secure applications including privacy-preserving machine learning [21], privacy-preserving smart contracts [9], and so on.

Keystone Security Monitor

Keystone [15] is an open-source enclave platform based on RISC-V. Keystone runs on open cores such as Rocket [2] and BOOM [8], or any other standard RISC-V core.

Similar to Sanctum [10], Keystone relies on a high-privilege software called security monitor (SM) for isolating the enclave memory (Figure 1.3). The SM runs in machine mode (M-mode), allowing it to use physical memory protection (PMP) to configure the access permissions of low-privilege software (e.g., OS or user) to certain memory regions. Keystone uses multiple PMP registers to dynamically control memory accessibility as shown in Figure 1.4. PMP controls the access permissions to a specified physical memory region by using a set of control status registers (CSR) in RISC-V. Each core may have 0-16 PMP registers, each of which consists of a configuration (`pmpcfg`) and an address register (`pmpaddr`) to define a *PMP entry*. As shown in Figure 1.4, the `pmpcfg` register defines the addressing mode and permission bits, and `pmpaddr` specifies the address range by encoding the address using a selected addressing mode. By default, PMP entries act as an allowlist, which means that the memory is inaccessible if none of the PMP entries is defined. PMP entries are statically prioritized, such that the lowest-numbered PMP entry that matches any byte of an access determines whether the access succeeds or fails.

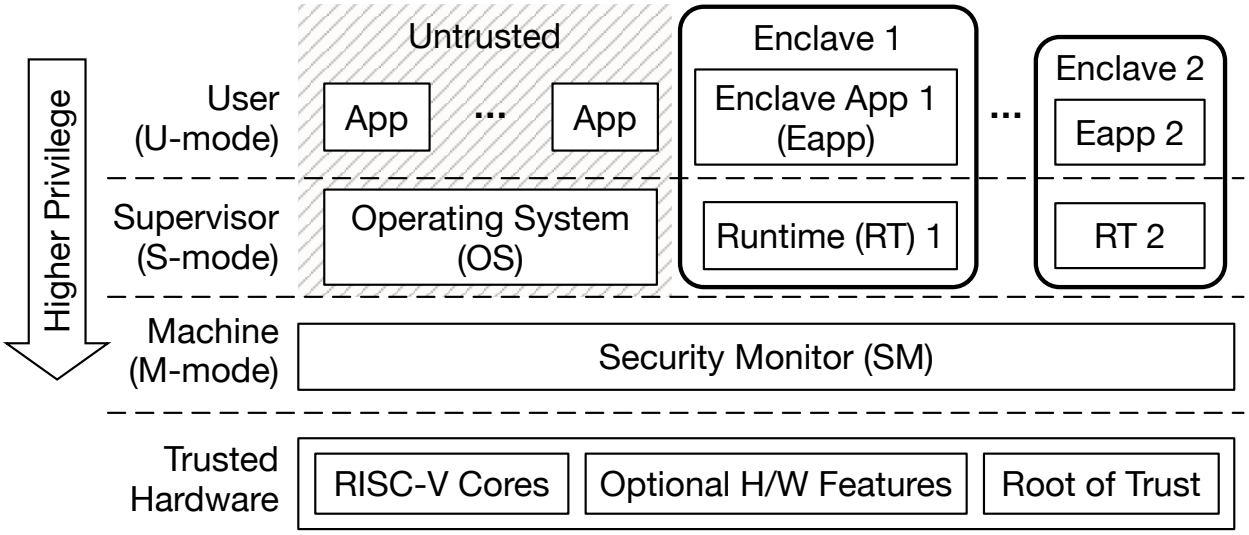

Figure 1.3: The overview of Keystone system.

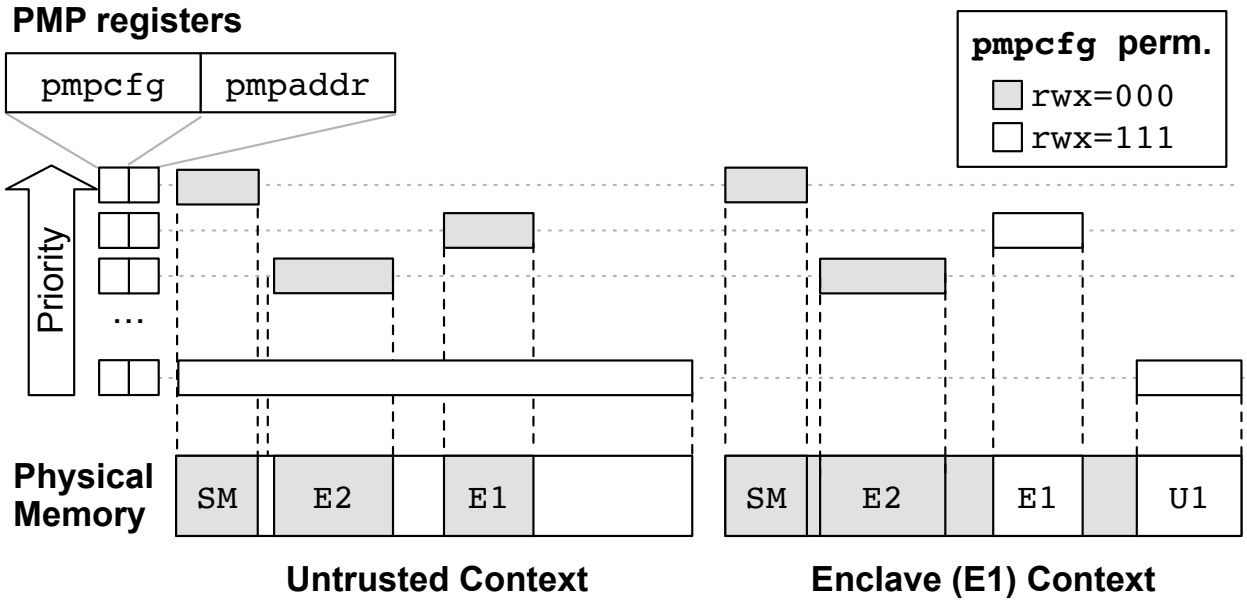

Figure 1.4: How Keystone uses RISC-V PMP for dynamic memory isolation. The SM uses a few PMP entries to guard its own memory (`SM`) and enclave memories (`E1`, `E2`). Upon enclave entry, the SM will reconfigure the PMP such that the enclave can only access its own memory (`E1`) and the untrusted buffer (`U1`).

When the system boots, the security monitor uses the first PMP register to protect its own memory region by setting all permission bits to zero and configuring the address to cover the entire image as well as the stack (Figure 1.4). Then, it sets the last PMP register to let the OS access the remaining part of the memory. Upon the creation of an enclave, the security monitor allocates an available PMP register to seal and isolate the enclave memory.

The security monitor implements memory isolation by switching the permission bits when the context changes. Before the enclave starts to run, the security monitor flips the permission bits in the enclave's PMP entry to allow computation on its isolated memory. In addition, the security monitor invalidates the last PMP register to deny the enclave access to the operating system's memory.

1.3 Supporting Novel Enclave Applications

Enclave platforms have a few limitations that make them hard to adopt, especially in the case of server applications. One of the limitations, which we highlight in this thesis, is that an enclave lacks the ability to clone itself. Cloning is one of the most common programming primitives implemented with process creation (e.g., `fork`, `vfork`, or `clone`) system calls. The address space of an enclave is always exclusive, which entirely limits such functionality. To the best of our knowledge, no enclave platform [14, 15] natively supports such functionality.

Admittedly, the advantages of cloning processes are less relevant these days. For example, concurrency in multi-process servers can be effectively handled by threads, and `fork` and `exec` could be replaced by `posix_spawn` [4]. Although this is true for normal applications, we argue that cloning is still very useful in the context of TEEs. In this section, we motivate our work by introducing a few example cases that need cloning of enclaves.

Multiprocessing

Although multithreading (MT) provides thread-level parallelism, it allows multiple threads to share data such that bytes written by one thread can be seen by another thread (Figure 1.5). This is undesirable if each thread handles data that is required to be confidential to each other. Moreover, one thread may break the entire program such that it affects the integrity of other threads. Thus, isolating the data of each task is necessary. As we will discuss in Related Work, task isolation can be implemented in many ways.

Snapshot and Rollback

We can use cloning as a security primitive, by checkpointing to a known machine state, and then resume execution from the state. Whenever the system falls into a stale state, we can rollback to the known state. Similar ideas have been introduced by many previous papers [18, 30, 5]. In addition, one can easily implement a stateless server enclave that clones

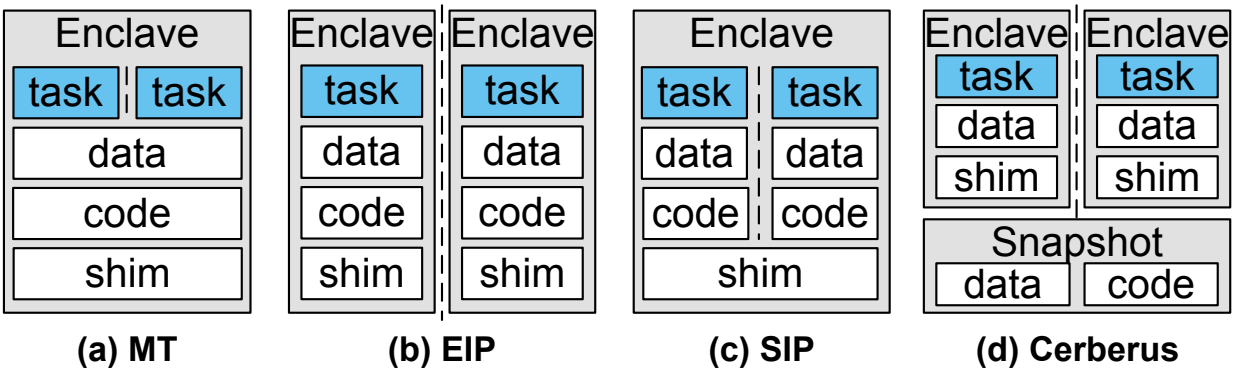

Figure 1.5: Comparing different task isolation mechanisms with enclaves: (a) Multithreading (MT), (b) Enclave-Isolated Process (EIP) [29, 20], (c) SFI-Isolated Process (SIP) [25],and (d) Cerberus (this work)

itself whenever there is a new request such that each of the enclaves starts from a known, secure state.

Cloning is also very useful in applications that want to take a snapshot of a virtual address space without having to block the entire application. Redis [23] uses `fork` to implement persistence. When Redis forks, the child process will be a snapshot of the in-memory database. The child then writes the database into the file without having to pause the parent. An efficient cloning mechanism should be also available if we run such applications in enclaves.

Related Work

A few frameworks such as Graphene [29] and Asylo [20], which are built on top of the enclave implementation, support enclave-isolated process (EIP) creation inside enclaves (Figure 1.5b). Both encapsulate a user application with a shim layer (e.g., LibOS) such that it runs inside an enclave. They implement process creation system calls by creating an enclave and transferring the entire state of the application to the new enclave. To be specific, when such system calls are invoked, ① the parent enclave coordinates with the host and creates an enclave with the same initial state; ② the parent attests the child enclave and exchanges a key to construct a secure channel; ③ the parent enclave takes a snapshot of the application memory, encrypts it, and transfers it over the secure channel; ④ the child enclave receives the snapshot, decrypts it, and restores the application memory.

Although this provides the functional correctness of cloning, it is suboptimal in terms of performance and resource usage for a few reasons. First, the creation time of an enclave linearly increases as the size of the enclave increases. When an enclave is created, all initial pages need to be copied into the protected physical memory. Then, the initial pages are *measured* with a cryptographic hash function. Thus, a larger initial enclave size will result

	MT	**EIP** [29, 20]	**SIP** [25]	**Cerberus***
Task Isolation	no isolation	enclave	MMDSFI	enclave
App. Pages	shared	isolated	isolated	write-isolated
Shim Pages	shared	isolated	shared	write-isolated
Init. Latency	very low	very high	low	low
Exec. Overhead	no	no	high	low

Table 1.1: Characteristics of different task isolation mechanisms

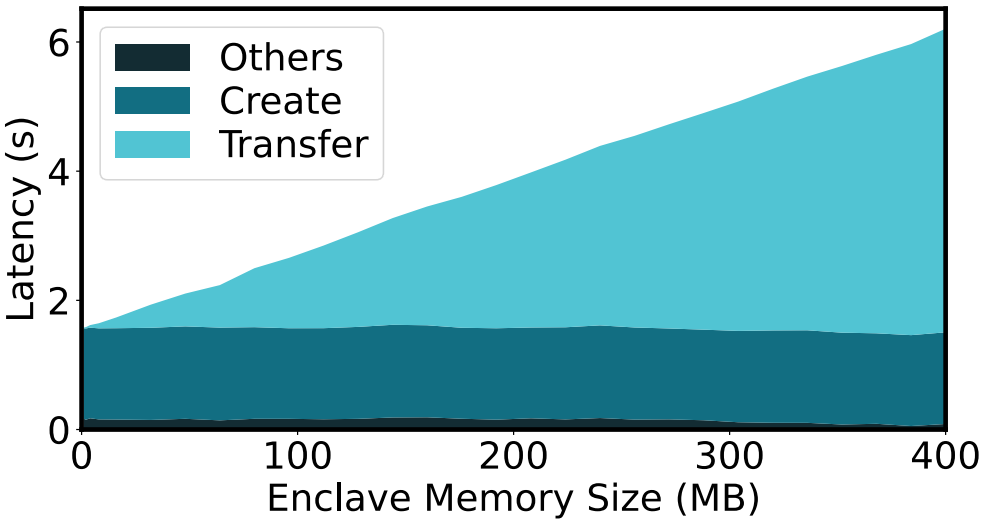

Figure 1.6: The latency breakdown of `fork()` system call in an enclave hosted by Graphene-SGX, running on an Intel i7-9750H processor. `fork()` was called after `malloc()` with different sizes where the initial enclave size was fixed to 512 MB.

in a longer latency. Second, the initialization time of the user application also linearly increases with the size of the active application memory. Because all allocated memory will be transferred via the encrypted channel, any large objects will make the latency prohibitively long. As an example, a neural network model in a deep learning application can easily go over a few MBs, which could take a few seconds to transfer. Figure 1.6 shows that the overall latency is an order of seconds and increases proportionally to the allocated memory size.

Shen et al. [25] address the problem by proposing Occlum, which enables multiprocessing inside the enclave using their intra-enclave isolation mechanism called MPX-based Multi-domain Software Fault Isolation (MMDSFI). The memory isolation within the enclave is enforced by SFI, which sanitizes every memory access instruction with Intel Memory Protection Extension (MPX). This allows Occlum to implement process creation within a single enclave. We compare SFI-isolated process (SIP) with others in Figure 1.5c. We highlight a

few limitations as shown in Table 1.1. First, the isolation relies solely on the implementation of MMDSFI, which was not formally specified and verified. Thus, it is hard to verify that Occlum provides the same or stronger isolation than EIP. Moreover, cache side-channel attacks are out-of-scope of Occlum, and it will require additional software mechanisms for cache side-channel attacks. Second, although the application data and code are isolated, tasks share the same LibOS, which means that they technically can interfere with each other. For example, stale global variables or cached file of LibOS may result in different behaviors for the same task. Third, since Occlum relies on binary instrumentation, it incurs about 36% execution overhead. In addition, Intel has removed MPX in 2019 and onward hardware, so Occlum will not be able to use the same mechanism in newer processors.

To address the limitations, we first formally model Cerberus and verify that it does not break the security properties of TEEs. As we will see, Cerberus inherits all isolation guarantees the TEE provides, because we still isolate each task with an enclave. Thus, if the enclave provides defense against cache side-channel attacks [15], enclaves in Cerberus will also have the defense. Furthermore, Cerberus has a different sharing model than the other isolation mechanisms. We define write-isolation as the requirement that write operations invoked by enclaves are isolated from each other. Cerberus' isolation mechanism allows enclaves to share data via reads while ensuring that writes are restricted to an enclave's own memory region, enabling a number of new enclave applications while also maintaining security. We prototype Cerberus in RISC-V Keystone, but also argue that a similar scheme can be applied to other TEEs such as Intel SGX. We show that Cerberus has very low initialization latency for creating a new task as well as very low execution overhead.

Example: Function-as-a-Service (FaaS)

Function-as-a-Service is a new cloud computing paradigm, which allows developers to write a function and deploy it on the server without having to manage the underlying server infrastructure. The function can be executed by external triggers such as HTTP requests, and the cost is charged based on the number of function invocations. The cloud provider is in charge of providing reliable and scalable servers that can execute the function.

As prior work [1, 6, 28] motivated, FaaS can also take advantage of the isolation of enclaves when each function invocation may process sensitive data from different users. Furthermore, the end user may want to attest the function before it provisions the secret data. For example, let us say a function analyzes a DNA sequence and returns whether the DNA is susceptible to a certain disease. The user does not want to disclose their DNA data to the cloud provider nor the service provider, but trusts the function implementation. The service provider can provision the function in an enclave, and allow the user to construct a secure channel to the enclave backed by remote attestation (e.g., RA-TLS [16]). However, the existing enclaves are inefficient to support FaaS workloads.

Initialization Latency As we discuss in Section 1.3, enclave initialization latency is prohibitive for cloud applications. This is because when the enclave is created, every initial

page needs to be copied into the protected memory region and then measured using a cryptographic hash function. For example, SGXv1 requires all heap pages to be initialized prior to entering the enclave. This increases the start-up latency up to a few seconds as the peak memory usage of the application increases. Trach et al. [28] try to address this by using dynamic memory allocation in SGXv2 [19]. However, dynamic memory allocation in SGXv2 does not remove the cost; it only delays the page initialization until the enclave actually uses the page. Keystone [15] also reports a few million cycles overhead just for hashing a single page with SHA3.

Memory Footprint Using enclaves in the FaaS workers significantly amplifies the memory footprint. In FaaS, many functions can share the identical language runtime (e.g., a specific version of Python), thus they can share some of the physical pages. This is not possible when each function is isolated in an enclave. Since physical memory is a limited resource, the throughput is limited by the memory usage of the entire language runtime, not by the memory usage of each function.

1.4 Trusted Abstract Platform and Secure Remote Execution

Formal verification of enclave platforms is crucial due to the strong threat model of hardware enclaves. In many cases, the security of an enclave system usually relies on the small trusted computing base (TCB) implemented with a combination of hardware and highly privileged software. Any bugs or vulnerabilities in the TCB could break the security not only of an enclave, but also of the entire system. For these reasons, several studies have formally verified either the abstract model [27] or the implementation [11] of enclave systems.

Previous work [27] uses formal methods to verify Intel SGX and Sanctum. The work introduces the Trusted Abstract Platform (TAP), which is a formal specification for hardware enclave systems. TAP is proven to provide confidentiality, integrity, and measurement against formally-defined adversary models. Using TAP as a basis of verification, a verification engineer can construct formal models of their enclave platform and show that they are refinements of the TAP model. In doing so, the security properties that hold over TAP also hold over the model of the enclave platform. Subramanyan et al. [27] use this technique to show that SGX and Sanctum satisfy a desired set of security properties. In this section, we dive deeper into the TAP model and the security properties it guarantees.

Trusted Abstract Platform

The Trusted Abstract Platform is an abstraction of an enclave platform. As in the paper by Subramanyan et al. [27], is modeled as a finite state transition system, $(\Sigma, T, init)$, with the set of states Σ, transition relation T, and initial states $init \in \Sigma$. Figure 1.2 contains a

summary of the state of the TAP model. Furthermore, Figure 1.3 shows the set of formally specified primitive operations that are sufficient to model enclave execution.

State Var.	Type	Description
`pc`	VA	Program counter
`regs`	$N \rightarrow W$	Architectural registers; map from natural numbers to words
`mem`	$PA \rightarrow W$	Memory; map from physical addresses to words
`addr_map`	$VA \rightarrow (ACL \times PA)$	Map from virtual addresses to permissions and physical addresses for current process
`cache`	$(Set \times Way) \rightarrow (B \times Tag)$	Cache: map from a tuple of cache sets and ways to valid bits and cache tags
`current_eid`	$\mathcal{E}_{id}$	Current enclave. `current_eid` = OS means that no enclave is being executed
`owner`	$PA \rightarrow \mathcal{E}_{id}$	Map from physical address to the enclave address is allocated to
`enc_metadata`	$\mathcal{E}_{id} \rightarrow \mathcal{E}_{\mathcal{M}}$	Map from enclave ids to metadata record type
`os_metadata`	$\mathcal{E}_{\mathcal{M}}$	Metadata record that stores a checkpoint of privileged software state

Table 1.2: TAP state variables

Secure Remote Execution

Users of any given enclave platform require that the platform executes their enclave program as intended. The Secure Remote Execution (SRE) property requires that the enclave platform preserves the semantics of the enclave program and guarantees that an adversary does not learn any more information that what is allowed. The semantics of an enclave e, denoted $[\![e]\!]$, is the set of finite or infinite execution traces, containing an execution trace for each input sequence, i.e. for each value of non-enclave memory and randomness at each step of execution. In prior work [27], this is formulated as follows:

Definition 1 *Secure Remote Execution of Enclaves. A remote platform performs secure execution of an enclave program e if any execution trace of e on the platform is contained within $[\![e]\!]$. Furthermore, the platform must guarantee that a privileged software attacker only observes a projection of the execution trace, as defined by the observation function obs.*

Subramanyan et al. decompose this property into three separate properties which entails stated below.

Operation	Description
`fetch`(v)	Fetch/read/write from/to virtual address v. Fail if v is
`load`(v)	not executable/readable/writeable respectively according to the
`store`(v)	`addr_map` or if `owner`[`addr_map`[v].PA] $\neq$ `current_eid`.
`get_addr_map`(v)	Get/set virtual to physical mapping and associated permissions
`set_addr_map`(v)	for virtual address v.
`launch`(e, m, x_v, x_p, t)	Initialize enclave e by allocating `enc_metadata`[e].
`destroy(e)`	Set `mem`[p] to 0 for each p such that `owner`[p] $= e$. Deallocate enclave `enc_metadata`[e].
`enter`(e), `resume`(e)	`enter` enters enclave e at entrypoint, while `resume` starts execution of e from the last saved checkpoint.
`exit`(), `pause()`	Exit enclave. `pause` also saves a checkpoint of `pc` and `regs` and sets `enc_metadata`[e].`paused` = `true`.
`attest`(e)	Return hardware-signed message with operand d and enclave measurement e: $d \| \| \mu(e) SK_p$.

Table 1.3: Description of TAP API.

- **Secure Measurement.** The platform must measure the enclave program to allow the user to detect any changes to the program prior to execution, i.e., the user must be able to verify that the platform is running an unmodified e.
- **Integrity.** The enclave program's execution cannot be affected by a privileged software attacker beyond providing inputs, i.e., the sequence of inputs uniquely determines the enclave's execution trace, and that trace must be allowed by the enclave's semantics $[\![e]\!]$.
- **Confidentiality.** A privileged software attacker cannot distinguish between the executions of two enclaves, besides what is already revealed by *obs*.

The model and associated proofs show that the specification guarantees Secure Remote Execution under three adversary models: M, MC, and MCP. These classes of adversaries differ by the state they are allowed to observe. Figure 1.4 details the differences between the various adversary models. The adversary can either try to tamper or observe the enclave by modifying its own state or invoking the enclave API.

Adversary	Observation
M	Allows adversary to observe the contents of all memory locations not private to the running enclave.
MC	Extends **M** to also observe whether locations not private to the running enclave are cached or not.
MCP	Extends **MC** to also observe the virtual to physical mappings and access permission bits for each address.

Table 1.4: Summary of adversary model observations

Chapter 2

Verifying Trusted Execution Platforms

In this chapter, we apply prior techniques for verifying trusted execution platforms towards a novel extension of the Keystone security monitor. We begin by replicating prior work on the Trusted Abstract Platform using UCLID5. Then, we describe Cerberus, a set of extensions to the Keystone security monitor that enable enclave programs to be cloned. Finally, we extend our Trusted Abstract Platform model with snapshot and clone functionality and verify that our augmented model preserves Secure Remote Execution.

2.1 Modeling a Trusted Abstract Platform with UCLID5

Prior work on TAP used Boogie to model the abstract system and generate a proof of secure remote execution. While Boogie demonstrates advantages when verifying sequential software, it falls short in modeling hardware systems. UCLID5, on the other hand, implements support for modeling both software and hardware by allowing users to describe sequential procedures and transition systems. Furthermore, UCLID5's focus on invariant synthesis makes it a suitable choice of modeling tool.

Extensions to UCLID5

In the process of porting the TAP models and proofs from Boogie to UCLID5, we implemented a set of features aimed at easier development and verification. Appendix A contains a subset of the TAP models that rely on the following features.

Importing Constants and Uninterpreted Functions

The Trusted Abstract Platform contains a number of submodules, each of which define the similar types, constants, and uninterpreted functions. Although type import across modules was already supported, we extended the UCLID5 language to allow for constant and function imports, greatly reducing redundancy across modules. We note that the current implementation does not currently support multiple inheritance and leave it as a future extension to the tool. Lines 4-6 in Appendix A, Listing A.2 demonstrate this feature.

Supporting Instance Procedure Calls

Preceding work proves Secure Remote Execution for TAP by decomposing the property into three sub-properties defining integrity, confidentiality, and measurement over two execution traces of the TAP. Replicating this result in UCLID5, implies instantiating two instances of the TAP model and requires support for invoking a procedure defined in an instance of a module. We implement support for this feature as well as the inclusion of module-level axioms in procedure verification, allowing us to verify properties over the state of multiple instances; furthermore, we note that adding this feature increases UCLID5's compatibility with 'object-oriented' software verification. In Appendix A, Listings A.2,3, we have a module of an abstract cache which is instantiated in the `abstract_cpu` model. The procedures in `abstract_cpu` invoke procedures in `abstract_cache`. This feature is useful as it allows us to build up and verify our models in a modular fashion. Moreover, this functionality meshes well with a future goal of implementing module-level assume-guarantee contracts in UCLID5.

Redefining the Implementation for the 'old' Operator

The 'old' operator, applied over a state variable, references the value of the variable before a procedure call or the value in the last step of the module. When a procedure with invariants containing the 'old' operator is invoked by another, the 'old' operator instead refers to the value of the variable before the calling procedure rather than the value of the variable before the procedure call. We implement a fix in UCLID5, by introducing fresh variables that take a snapshot of the state before each procedure call. In implementing this fix, the semantics of the 'old' operator within the context of procedure calls is identical to that of Boogie. For example, the procedure `store_va`, on line 134 in Appendix A, Listing A.3, is inlined at its call site. Note that the post conditions, e.g Line 134, Appendix A, Listing A.3, for this procedure contain references to the 'old' values of state variables. Without this modification in semantics, `old(cpu_mem)` would refer to the value of `cpu_mem` before the invocation of the calling function, which may result in the postcondition failing.

Description	Size			Verif. Time (s)
	#pr	#an	#ln	
TAP	38	472	2629	28.3
Integrity	1	63	255	326.3
Measurement	6	133	519	46.5
Confidentiality	3	211	1227	799
Total	**48**	**879**	**4630**	**1200.1**

Table 2.1: UCLID5 Models and Verification Metrics (Floyd-Hoare)

Description	Size			Verif. Time (s)
	#pr	#an	#ln	
TAP	22	204	1752	5
Integrity	12	145	985	26
Measurement	6	100	800	6
Confidentiality	8	200	1388	194
Total	**48**	**649**	**4925**	**231**

Table 2.2: BoogiePL Models and Verification Metrics (Floyd-Hoare)

Verification Results

We successfully translated TAP model and all proofs, created by Subramanyan et al. [27], in Boogie into UCLID5. [1] Apart from a few additional invariants and syntactic changes, the models are one-to-one. Tables 3.1 and 3.2 show a comparison between our verification effort in UCLID5 and prior work using BoogiePL. Note that pr, an, and ln denote the number of procedures, annotations, and lines of code. Note that number of annotation refers to the number of pre/post-conditions, assertions, and loop invariants written in the models. Generally, we observe that UCLID5 lends itself to code reuse, as shown by the lines of code required to express the Integrity and Measurement proofs. We also observe a slightly larger number of annotations written in the UCLID5 models. The primary point of comparison is in verification time. Boogie reports faster times, which could be due to optimization of queries to the SMT solver. Improving this is an area for future work on UCLID5. In particular, we could leverage work on MedleySolver [22] and in using different encodings than purely Floyd-Hoare style proofs, given UCLID5's capabilities in both concurrent and sequen-

[1] The source code can be found here: https://github.com/uclid-org/trusted-abstract-platform

tial modeling. We note that further investigation is required to explicitly characterize the differences between UCLID5 and BoogiePL, however this study offers a reasonable starting point.

Extending the Adversary Model

Existing work on the formal verification of secure enclaves proves Secure Remote Execution for three distinct parameterizations of a privileged software adversary. Prior work by Subramanyan et al. denote these parameterizations by M, MC, and MCP, each indicating the set of operations allowed to the adversary to tamper or observe the state of the enclave platform. Furthermore, it was shown that the Sanctum processor guarantees SRE under a MCP adversary, and Intel SGX guarantees SRE under a M adversary. While this model captures a large set of privileged software adversaries, it does not provide any guarantees against an adversary that can directly interact with the physical memory. Recent work has shown that sophisticated attacks can be successfully carried out against enclave platforms by such an attacker. We aim to capture this class of adversaries with a new paramaterization, denoted by M^*.

Adversary Observations

The observation function $obs_e^{M^*}(\sigma)$, for an enclave e and enclave platform state σ states that an M* adversary is able to observe the full range of physical addresses, with no restrictions on enclave memory. We formally define it as:

$$obs_e^{M*}(\sigma) = \lambda p.\sigma(\texttt{mem}[p])$$

Note that the observation function can be extended as in previous work [27] to define observation functions for M^*C and M^*CP adversaries.

Adversary Tampering

The tampering operations for M^* are given below.

(1) Unconstrained updates `pc` and `regs`.

(2) Loads and stores to memory with unconstrained address (va) and data ($data$) arguments, such that all platform protections are bypassed.

- $\langle op \rangle \leftarrow \texttt{fetch}(va)$
- $\langle \texttt{regs}[ri] \leftarrow \texttt{load}(va)$
- $\texttt{store}(va, data)$

(3) Modification of the adversary's view of memory by calling `get_addr_map` and `set_addr_map` with unconstrained arguments

- `set_addr_map`$(e, v, p, perm)$
- `get_addr_map`(e, v)

(4) Invocation of enclave operations with unconstrained arguments.

- Launch enclaves: `launch`(e, m, x_v, x_p, t)
- Destroy enclaves: `destroy`(e)
- Enter and resume enclaves: `enter`(e) and `resume`(e)
- Exit and interrupt enclaves: `exit` and `pause`

Note that the tamper operations can be extended as was in previous work to define tamper operations for M^*C and M^*CP adversaries.

Concluding Remarks

In this section we have shown that UCLID5 is a suitable verification tool for modeling and verifying an abstraction of trusted execution platforms. Furthermore, we have also extended the formalism of TAP's adversary models. At this point we have primarily replicated prior results, however, the transition to UCLID5 will play an important role in modeling further extensions to enclave platforms.

2.2 Cerberus: Secure and Efficient Cloning of Enclaves

Cerberus introduce two new operations, snapshotting and cloning, that enable performant and secure cloning of enclaves. An enclave can take a snapshot of itself in order to freeze its state, including enclave memory, general registers, control-and-status registers (CSRs), and the enclave metadata. Identical enclaves can then be quickly created from the snapshot. The new enclaves are still fully isolated with each other, but share the snapshot image with read-and-execute-only permission. When an enclave wants to write, it uses a copy-on-write mechanism to copy only the relevant pages over to its private memory. Newly written data is still confidentiality- and integrity-protected while enclaves can access the unmodified pages in the snapshot memory. Similar optimizations have been introduce to improve the performance of fuzzing [30] or security isolation [5]. In this thesis, we explain how we tailor the techniques to fit TEEs.

Threat Model

We note that Cerberus inherits threat model of the TEE, on which it is implemented. The program running inside the enclave will be confidentiality- and integrity-protected by the underlying isolation mechanism used by the enclave platform. To be specific, any private

data of one enclave must not be visible or modifiable by any software adversaries such as untrusted privileged software (e.g., the OS), or by other enclaves. The code of one enclave must not be modifiable by the software adversaries. Denial-of-service against the enclave is not defensible as in the TEE threat model. Thus, even if the enclave wants to clone, the OS can always refuse to cooperate.

However, multiple enclaves can share the same snapshot memory which contains the data or code before the enclave was cloned. The data is implicitly not confidential among these enclaves. However, snapshot memory is still invisible from the outside of these enclaves, for example a privileged software adversary or other enclaves that do not share the snapshot. We do not consider the application running in the enclave being vulnerable or malicious by itself. For example, an application can clone itself and dump the snapshot contents to the outside. This will trivially break the confidentiality of the snapshot memory.

We ensure that the enclaves are still *write-isolated*, which requires that any modification to the data from one enclave must not be visible to the others. However, enclaves are still allowed to read each other's data under a parent and clone relationship. Thus, any secret data needs to be provisioned after the enclave was cloned and attested. It is the enclave developer's responsibility to make sure that the snapshot does not contain any secret data that should not be visible by the cloned enclaves.

Since Cerberus inherits the threat model of the TEE, the enclaves are also resistant to cache side-channel attacks with the TEEs that support cache partitioning (e.g., [10, 15]). Our formal model assumes that each enclave has its own cache domain, and different enclaves may not share any cache lines even though physical addresses are shared. In the following section, we formally verify that that adding snapshot and clone functionality still retains the cache confidentiality guarantees in the previous formal model. We note that cache partitioning in Keystone [15] relies on the fact that the enclaves never share the same physical memory. Thus, the mechanism needs to be changed such that each enclave will always load the memory to a different region of the cache even though they access the same physical address. Although it can be simply implemented by taking the enclave ID into account when tagging a cache line, the actual implementation is out-of-scope of the paper.

Physical side-channel attacks [17] or physical tampering [13] are out-of-scope of this paper. We believe that those attacks can be mitigated with existing techniques such as on-chip memory, oblvious RAM [26], and memory encryption engine (MEE) [12].

Snapshot and Clone

`snapshot` is an irreversible operation that each enclave can call only once. Thus, once `snapshot` is called on an enclave, the enclave is permanently changed to a *snapshot enclave*. The snapshot enclave will freeze all the intermediate state of the enclave including the memory and the register contents. Since the TEE prohibits any modification to the state, the snapshot enclave cannot resume execution any longer. `snapshot` can be called only by the enclave itself. Thus, the initial enclave code needs to contain the code that calls `snapshot`. Since the enclave decides where it wants to freeze, a snapshot is inherently

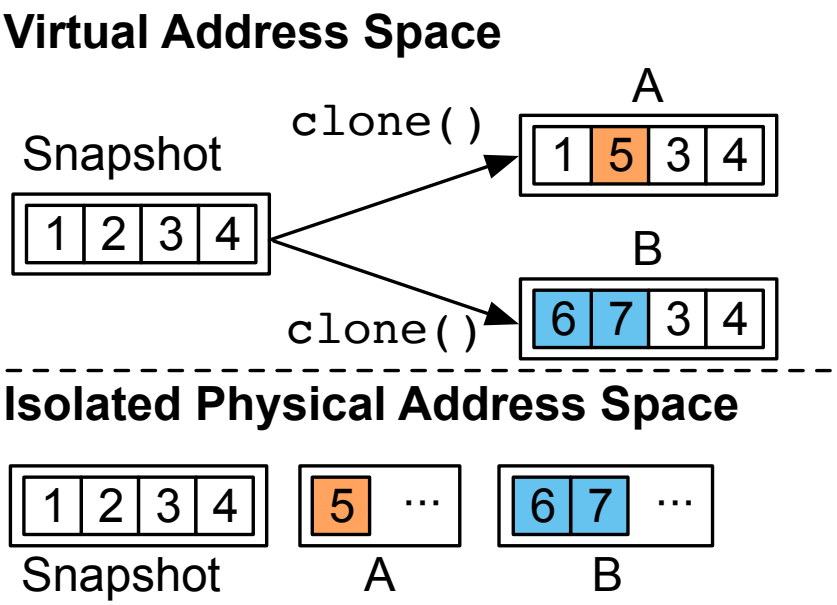

Figure 2.1: Address spaces for snapshots and clones

trusted if the enclave's initial state is trusted. This follows from the fact that enclaves with a valid attestation are deterministic with respect to enclave inputs. In fact, this is shown by the Secure Measurement property which is a component of Secure Remote execution.

`clone` then creates a fresh enclave with an empty isolated physical memory. The cloned enclave inherits every state of the snapshot including the virtual memory mapping. For that reason, `clone` does not need to copy the memory contents of the snapshot, saving a large number of cycles. `clone` can be called by the host. If the enclave wants to clone itself, it needs to coordinate with the host to create a new enclave with clone. This is natural given that all system resources are still managed by the OS. If the clone was due to a `fork` system call, we need to actually create two new enclaves: a parent and a child. After cloning from a snapshot enclave, the new enclaves can resume execution with read-and-execute-only permission to the snapshot memory. The TEE platform must enforce the permission so that the snapshot state is permanently not modifiable.

Figure 2.1 depicts a set of enclaves created by `snapshot` and `clone`. The figure shows three enclaves: Snapshot, `A`, and `B`. Note that `A` and `B` are clones of `Snapshot`. The top half shows the virtual address space of the enclaves, while the bottom half shows the memory regions that are unique to each of the enclaves. Cloned enclaves have access to certain certain physical addresses in their parent enclave, while also maintaining their own protected memory region.

We achieve write-isolation between clones by capturing any write instruction to the snap-

shot memory and perform copy-on-write, which we describe in the following subsection.

Copy on Write

Linux optimizes process cloning (i.e., `fork` and `clone`) with copy-on-write (CoW), which duplicates only the page table and postpones the actual page copy until either the parent or the child actually modifies the page. For a user process, CoW is based on the ability of the OS to manage the entire memory of user processes. Whenever the cloned process tries to write to any shared physical page, a page fault will occur, and the OS handler performs the actual copy and remaps the virtual address to the new page.

However, this is not possible with enclaves which are strictly isolated. Any page copy between enclaves may require an encrypted channel to be constructed after both enclaves attest each other. Moreover, the trusted component of the enclave platform usually has no control over the page tables. For example, in Intel SGX, page tables are entirely managed by the untrusted OS, whereas in other enclave platforms like RISC-V Keystone and Sanctum [10], page tables are managed by the enclave itself.

Cerberus instead relies on the write-access fault raised by the hardware when an enclave tries to write to a physical address without write permission. When the hardware exception occurs, the platform decides if it needs CoW by checking whether or not the faulting enclave is trying to access its snapshot memory. In Keystone, this is simply a machine trap, thus the security monitor can route the fault to the enclave. In SGX, this could be achieved by modifying hardware to raise a fault when enclave tries to write to its snapshot memory. Since there will be only one snapshot per enclave at most, the hardware can refer to the enclave metadata to see if it needs to raise the fault. Alternatively, the mechanism can be implemented in LibOS, by marking pages read-and-execute-only so that a page fault occurs.

The platform then allocates a new page in the private memory of the faulting enclave. Since there is no deallocation, the pages are just allocated starting from the beginning of the private memory. If there is no free memory, the enclave just fails to execute, and will be immediately destroyed. After the page is allocated, the platform copies the snapshot page into the new page, and remaps the virtual address of enclave to point to the new page. In Keystone, this can be done in-enclave by the supervisor-mode runtime. After CoW is finished, we resume execution of the enclave.

Nested Clone

We allow an enclave to call snapshot only once. Following that, any number of clones can be created from the snapshot, assuming that there exists enough free PMP entries to manage them. Furthermore, we do not allow cloned enclaves to invoke the `snapshot` operation and `clone` cannot be invoked on a cloned enclave. Although this only allows one layer of nesting, we found that this not only simplifies the model, but also fits most application requirements. We address extensions for arbitrary levels of nesting in Future Work.

2.3 Formally Verifying Cerberus

We wish to show that the addition of `snapshot` and `clone` to an enclave API does not violate the Secure Remote Execution property. To that end, we build upon our earlier work on TAP and use it to show that the desired integrity and confidentiality guarantees hold.

Extending the Platform Model

In this subsection we describe the changes made to the TAP model. We denote this new model TAP_C.

TAP State Variables

In our verification effort, we found it unnecessary to change most of the state variables of TAP for the extension of snapshot and clone. Most of our extensions are to the `enc_metadata` record, which is a map from enclaves IDs to metadata on the respective operating enclaves. The enclave metadata provided in the TAP abstraction was extended for reference counting and keeping track of enclaves that were snapshots. To accomplish this, Cerberus introduces several additional fields in the enclave metadata record for a given enclave e which are briefly described in Table 2.3. The `is_snapshot` field indicates whether e is a snapshot, which is initially false and set to true upon calling the snapshot(e) primitive. The `child_count` field is initially 0 and stores the number of child enclaves cloned from the enclave. The `parent` field indicates whether the enclave is a parent enclave, which is true if and only if the clone has been called on the enclave. Lastly, `wap_is_free` is a pool of free pages for the enclave, which is abstractly modeled as a map of addresses to boolean values, representing whether the address is free.

TAP Operations

As a result of introducing snapshot and clone to the TAP model in addition to changes in the enclave metadata record, the original TAP operations are also changed. We describe these changes in detail in Table 2.4.

Restricting Enclave Regions

In addition to augmenting the TAP API, we also make some notable to changes to platform's functionality. The most significant of which includes restricting enclaves to a single contiguous region of memory. In the original model [27], an enclave was able to specify exclusive physical addresses on initialization via an input parameter, `excl_paddr`, of type $PA \to B$, where PA denotes a physical address. This allowed enclaves to own disjoint regions of memory but prevented us from writing certain logical formulas comparing the number of exclusive physical addresses between enclaves; in other words, logical formula that required us to take the cardinality of a set.

State var.	Description
`entrypoint`	Enclave entrypoint.
`addr_map`	Virtual to physical mappings/permissions
`excl_vaddr`	Set of private virtual addresses.
`measurement`	Enclave measurement.
`pc`	Saved PC (in case of interrupt).
`regs`	Saved registers (in case of interrupt).
`paused`	Flag set only when enclave is interrupted.
`is_snapshot`	Flag indicating if the enclave is a snapshot.
`child_count`	Count of children cloned from this enclave.
`parent`	The parent enclave id if enclave was cloned.
`wap_addr_free`	A map of free physical addresses for the enclave.

Table 2.3: Fields of the enc_metadata. State variables introduced for Cerberus are below the middle line.

To accommodate these logical specifications, we modified the behavior of our TAP model to require users to provide an input parameter, `bounds`, of type (PA, PA) which denotes the inclusive upper and lower bounds of the memory region owned by an enclave. We then showed that our modified TAP model refines the one specified in [27], thus showing that the former model guarantees the security properties of the latter.

Modeling a Copy-on-Write Mechanism

Recall, from Section 3.2, that Cerberus uses the write-access fault as a way of invoking the CoW mechanism. In our TAP model, writes to memory are handled by the `abstract_cpu` in a procedure called `store_va`. Appendix B, Listing B.1 shows the modified `store_va`, which has been augmented with logic modeling copy-on-write. The new procedure allows an enclave to write to a virtual address, whose underlying physical address is owned by its parent. This logic is contained between lines 57 and 110. When a valid store to the parent region occurs, the CPU searches for a free address in `wap_addr_free`. If a free physical address is found, the data is stored to the new physical address and the virtual address mappings, otherwise, an out-of-memory fault is thrown.

The Adversary Model

The adversary model is largely unchanged compared to the original TAP model. To reiterate, TAP's adversary model is based on a privileged software attacker that consists of (i) the usual tamper relation describing how an attacker changes the platform state, and (ii) an observation function that describes a projection of states visible to the adversary. We

Operation	Description
`launch`(e, m, x_v, x_p, t)	Initialize enclave e by allocating `enc_metadata`[e]. Set `enc_metadata`[e].`is_snapshot = false`.
`destroy(e)`	Set `mem`[p] to 0 for each p such that `owner`[p] = e. Deallocate enclave `enc_metadata`[e] and decrement `enc_metadata`[`enc_metadata`[e].`parent`].`child_count` by 1.
`enter(`e`)`, `resume(`e`)`	`enter` enters enclave e at entrypoint, while `resume` starts execution of e from the last saved checkpoint. Returns invalid argument status if `enc_metadata`[e].`is_snapshot` is true.
`exit()`, `pause()`	Exit enclave. `pause` also saves a checkpoint of `pc` and `regs` and sets `enc_metadata`[e].`paused` = `true`. Returns invalid argument status if `enc_metadata`[e].`is_snapshot` is true.
`attest(`e`)`	The `attest` operation is unchanged from the original TAP model.
`snapshot(`e`)`	Set the writable bit `addr_map`[v].`ACL.write` to false for all virtual addresses v owned by enclave e and sets `enc_metadata`[e].`is_snapshot` to true.
`clone`(e_p, e_c)	If the parent enclave's field `enc_metadata`[e_p].`is_snapshot` is true and enclave e has enough free pages to clone e_p, then set `enc_metadata`[e_c].`parent` = e_p, increment the parent's child count `enc_metadata`[e_p].`child_count` by 1, and set `owner`[p] = e_c for each virtual address p allocated for the virtual address map for the child enclave e_c.

Table 2.4: Description of snapshot and clone in TAP_C, and changes to the original TAP API.

show that TAP_C maintains SRE over the various parameterized adversary models described in Table 1.4.

Verifying Secure Remote Execution and Beyond

We begin with some preliminary definitions. Let $\pi_i[j]$ be the state of the system in trace i at time j. Let $E_e(\sigma)$ be a projection of the system state σ containing only the state of enclave e. Concretely, E refers to `entrypoint`, `pc`, `regs`, `excl_vaddr`, and addr_map. Let *curr* be the currently executing enclave. Let I_e be the inputs to enclave e. Let O_e be the outputs of enclave e. Let A_e be adversary's state, defined by the adversary's observation function and untrusted system state. Let $I^P(\sigma)$ be the bits of non-determinism in a state σ. Let *obs* be a projection of the system state given by the adversary's observation function.

Equations (2) and (3) denote the formal specifications of integrity and confidentiality guaranteed by Secure Remote Execution. In the standard TAP model, e, e_1, and e_2 refer to enclaves created by the standard initialization process. However, since TAP_C allows for enclaves to be created via `clone`, we must also ensure that our confidentiality and integrity properties reflect this behavior. Note that the decomposition of the Secure Remote Execution property contains a Secure Measurement property; by default, this holds for TAP_C as the attestation scheme remains unchanged.

Before discussing the formal properties, we must first discuss the relationship between physical enclaves and enclave programs. Informally, a physical enclave is a protected region of memory managed by the enclave platform. The lifetime of a physical enclave begins upon an initialization request via the enclave API and ends on a destroy request via the enclave API. In our standard TAP model, the entirety of an enclave program is executed within one physical enclave. However, in TAP_C, an enclave program may execute across multiple physical enclaves. That is, an enclave launches, executes operations, and snapshots; then, a clone launches, executes, and finishes. This distinction is important, since the encoding of SRE in UCLID5 specifically applies to physical enclaves and enclave programs that span the duration of a single physical enclave. In this subsection, we address this difference by first proving SRE over physical enclaves and then proving additional properties which show that SRE holds for enclave programs spanning multiple physical enclaves.

SRE for Physical Enclaves

In the case of integrity, we show that for any two physical enclaves running the same enclave program, if their initial states are equivalent with respect to the projection E, their inputs are the same, and they run at the same time, then their state and outputs are the same, independent of whether or not the enclave was created as a clone. In the case of confidentiality, we show that the guarantees extend to cloned enclaves under various classes of software adversaries. We note that we use the same adversary models as [27]. Informally, these models consist of an adversary that can observe certain memory regions, an extension which can also observe a projection of cache state, and a further extension which can observe page table mappings and permission bits. The formal properties for integrity and confidentiality, respectively, are included in 2.3. Note that in 2.3, e refers to a physical enclave. We verify properties for physical enclaves that are created either by the `launch` or `clone` operation.

SRE for Enclave Programs

We emphasize that showing our enclave platform guarantees SRE for physical enclaves is not sufficient to guarantee that SRE holds for any enclave program. Rather, our enclave platform may violate SRE at the snapshot and clone boundary. For example, a privileged software adversary may be able to tamper with enclave platform to violate the integrity of the enclave program executing as a clone. In another case, the `snapshot` and `clone` operations may leak private data to an adversary.

$$
\begin{aligned}
&\forall \pi_1, \pi_2. \\
&(E_e(\pi_1[0]) = E_e(\pi_2[0])) \wedge \\
&\forall i.(curr(\pi_1[i]) = e) \iff (curr(\pi_2[i]) = e) \wedge \\
&\forall i.(curr(\pi_1[i]) = e) \Rightarrow I_e(\pi_1[i]) = I_e(\pi_2[i]) \Rightarrow \\
&(\forall i.E_e(\pi_1[i]) = E_e(\pi_2[i]) \wedge O_e(\pi_1[i]) = O_e(\pi_2[i]))
\end{aligned}
$$

$$
\begin{aligned}
&\forall \pi_1, \pi_2. \\
&(A_{e_1}(\pi_1[0]) = A_{e_2}(\pi_2[0])) \wedge \\
&\forall i.curr(\pi_1[i]) = curr(\pi_2[i]) \wedge I^P(\pi_1[i]) = I^P(\pi_2[i])) \wedge \\
&\forall i.curr(\pi_1[i]) = e \Rightarrow obs_{e_1}(\pi_1[i+1]) = obs_{e_2}(\pi_2[i+1])) \Rightarrow \\
&(\forall i.A_{e_1}(\pi_1[i]) = A_{e_2}(\pi_2[i]))
\end{aligned}
$$

Figure 2.2: Integrity and Confidentiality Properties proven over TAP_C

Therefore, in order to show that the platform provides SRE for any enclave program, we first show that the usage of snapshot and clone does not violate the integrity of an enclave program. In other words, for all physical enclaves, the corresponding clone must be initialized to an equivalent state, despite any operations invoked by the privileged software adversary. The formal property, which we call Integrity Preservation under Snapshot and Clone is shown in Figure 2.3. Note that e denotes the original physical enclave and e' denotes the cloned physical enclave. We then show that the usage of snapshot and clone does not violate confidentiality with respect to an adversaries that can observe unprotected memory regions, the cache, and page tables. The formal property, which we call Confidentiality Preservation under Snapshot and Clone is shown in Figure 2.4. Note that in the case of Confidentiality Preservation the adversary is running concurrently with enclave e and e', once it has been initialized. We then arrive at the following theorem.

Theorem 1 *An enclave platform, with snapshot and clone operations, that satisfies Secure Remote Execution for physical enclaves, Integrity Preservation under Snapshot and Clone, and Confidentiality Preservation under Snapshot and Clone also satisfies Secure Remote Execution for all enclave programs.*

Proof. Suppose we have a valid enclave program e that does not invoke snapshot and clone, then since e executes within a single physical enclave, SRE holds. Now suppose we have a valid enclave program e' that invokes snapshot and clone. Let i be the point in time when `snapshot` is invoked and let j be the point in time when `clone` is invoked. Note that

$$
\begin{aligned}
&\forall \pi. \\
&\forall k, j. \\
&(k > 0 \wedge j > k \wedge \\
&\ (curr(\pi[k]) = e) \wedge \\
&\ op(\pi[k]) = \texttt{snapshot}(e) \ \wedge op(\pi[j]) = \texttt{clone}(e, e')) \\
&\ \Rightarrow \\
&\ (E_e(\pi[k]) = E_{e'}(\pi[j]))
\end{aligned}
$$

Figure 2.3: Integrity Preservation under Snapshot and Clone

$$
\begin{aligned}
&\forall \pi. \\
&\forall k, j. \\
&(k > 0 \wedge j > k \wedge \\
&\ (curr(\pi[k]) = e) \wedge \\
&\ op(\pi[k]) = \texttt{snapshot}(e) \ \wedge op(\pi[j]) = \texttt{clone}(e, e')) \\
&\ \Rightarrow \\
&\ (obs(\pi[j]) = obs(\pi[j+1]))
\end{aligned}
$$

Figure 2.4: Confidentiality Preservation under Snapshot and Clone

$j > i$. It follows that for any time $t \leq i$ and $t > j$, SRE holds since e' has only been executing in a single physical enclave. By the preservation properties, we ensure that confidentiality and integrity holds for time $i < t \leq j$. Thus we have that SRE holds for the entirety of e''s execution.

Semantic Equivalence of Enclave Programs

Another property we would like to verify is semantic equivalence between enclave programs that use `snapshot` and `clone` and those that do not. Specifically, we show that, assuming the same inputs, an enclave program e that contains `snapshot` and `clone` operations is equivalent to an enclave program e' where the `snapshot` and `clone` operations have been removed. We verify this property under the presence of a privileged software adversary. Figure 2.5 contains the formal definition of semantic equivalence.

Figure 2.6 shows the two traces of the semantic equivalence property. The adversary's

$$
\begin{array}{l}
\forall k, j. \\
\forall \pi_1, \pi_2. \\
(k > 0 \wedge j > k \wedge \\
E_{e_1}(\pi_1[0]) = E_{e_2}(\pi_2[0]) \wedge \\
\forall i.(i < k) \Rightarrow ((curr(\pi_1[i]) = e_1) \iff (curr(\pi_2[i]) = e_2)) \wedge \\
\forall i.(i > j) \Rightarrow ((curr(\pi_1[i]) = e_1) \iff (curr(\pi_2[i]) = e_3)) \wedge \\
\forall i.(i > j) \Rightarrow ((curr(\pi_1[i]) = e_1) \Rightarrow (I_{e_1}(\pi_1[i]) = I_{e_2}(\pi_2[i]))) \wedge \\
\forall i.(i > j) \Rightarrow ((curr(\pi_1[i]) = e_1) \Rightarrow (I_{e_1}(\pi_1[i]) = I_{e_3}(\pi_2[i]))) \wedge \\
\forall i.(k \leq i \leq j) \Rightarrow (curr(\pi_1[i]) \neq e_1) \wedge \\
op(\pi_2[k]) = \texttt{snapshot}(e_2) \ \wedge op(\pi_2[j]) = \texttt{clone}(e_2, e_3)) \\
\ \Rightarrow \\
(\forall i.(i < k) \Rightarrow \\
\quad (E_{e_1}(\pi_1[i]) = E_{e_2}(\pi_2[i]) \ \wedge O_{e_1}(\pi_1[i]) = O_{e_2}(\pi_2[i])) \wedge \\
\forall i.(i > j) \Rightarrow \\
\quad (E_{e_1}(\pi_1[i]) = E_{e_3}(\pi_2[i]) \ \wedge O_{e_1}(\pi_1[i]) = O_{e_3}(\pi_2[i])))
\end{array}
$$

Figure 2.5: Semantic Equivalence under Snapshot and Clone

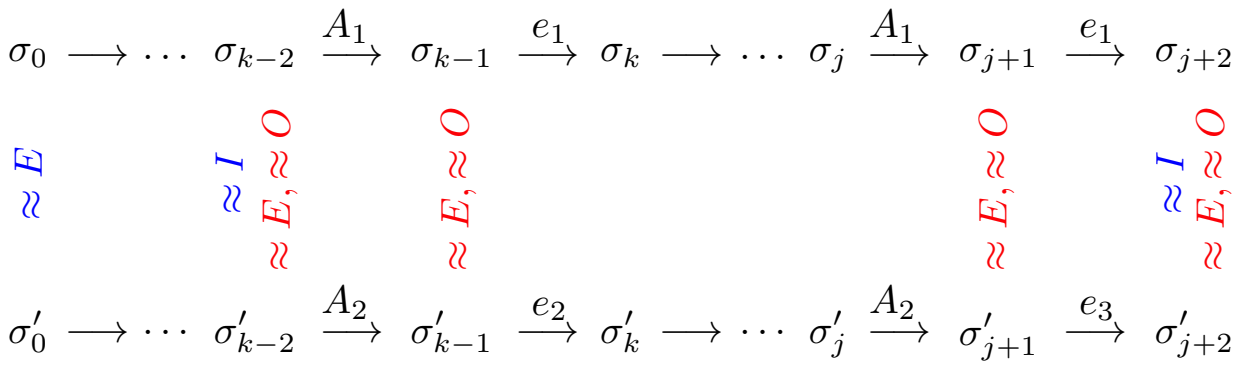

Figure 2.6: A trace diagram of the semantic equivalence property

steps are labelled A_1 and A_2 while the enclave's steps are labelled e_i. Note that e_1 corresponds to an enclave program that does not invoke `snapshot` and `clone`, and e_2 and e_3 correspond to the same enclave program before and after the snapshot and clone invocations respectively. Assumptions are annotated in blue, and proof obligations are shown in red. The enclaves' inputs are assumed to be the same in both traces; this is shown by the $\approx_I$ symbol. The initial states of e_1 and e_2 are assumed to be the same and this is shown by the $\approx_E$ symbol. The adversary's actions are defined by some function over the state of the enclave platform, and these actions may differ between the two traces. The semantic equivalence proof must show that the enclave's state and outputs do not differ despite this, even across snapshot and clone boundaries. These proof obligations are denoted by the red $\approx_E$ and $\approx_O$ symbols. We assume that the adversary executes for the same number of steps in both traces. This does not restrict the adversary's power as any attack that requires the adversary to execute for a different number of steps in the two traces can be simulated in our model by padding the adversary's shorter trace with the appropriate number of "no-ops." The theorem states that, given the above assumptions, enclave state and outputs are identical in the two traces at every step, before and after the snapshot/clone boundary.

Verification Results

Table 2.3 shows a number of metrics from the formal proof. Note that we #as refers to the number of assertions. This number is much larger than the number of annotations because the UCLID5 compiler expands the input model into a larger intermediate form. Single-Region TAP and Single-Region Refinement refer to the modified TAP model in which enclaves are restricted to single regions of memory. Overall, due to the increased complexity of our model, we verified a larger number of invariants than in the standard TAP models. Consequently, our reported verification times are also much higher. The source code for our formal proof can be found at https://github.com/uclid-org/trusted-abstract-platform.

Description	**Size**				**Verif. Time (s)**
	#pr	#an	#as	#ln	
Single-Region TAP	26	363	255	2273	21.3
Single-Region Refinement	1	44	117	460	35.4
Cerberus	56	785	738	4825	18.1
Preservation	3	323	448	365	52.3
Semantic Equivalence	1	215	1204	576	272.6
Integrity	2	287	1986	950	1147.4
Confidentiality	6	718	7236	3501	15612.5
Total	**95**	**2735**	**11984**	**12950**	**17159.6**

Table 2.5: UCLID5 Verification Metrics for TAP_C models

Chapter 3

Conclusion

In this report, we describe a set of techniques to model and verify trusted execution environments. We began by replicating prior work on the Trusted Abstract Platform using UCLID5, while also building additional features. We then describe the `snapshot` and `clone` extensions to the Keystone Enclave API. Finally, we extend the TAP model and verify that our extended enclave platform provides sufficient integrity and confidentiality guarantees for enclave programs. The findings from this report demonstrate and validate a number of techniques for using formal methods to reason about program security. In our work on TAP, we verified program security by reasoning about the enclave platform's behavior. We found that the flexibility afforded by UCLID5 makes it a strong candidate for future work at the intersection of formal methods and security.

3.1 Future Work

UCLID5

In this work, we used UCLID5 extensively to model and verify a number of security properties. While comparing the verification metrics of UCLID5 and BoogiePL, we noticed that at times Boogie reported faster verification times. Optimizing UCLID5's encoding to backend verification tools are one in which we may improve the overall run time. Furthermore, future work on invariant synthesis and assume-guarantee reasoning appears to be extremely promising in terms of affording the user flexibility while also reducing the verification time.

Cerberus and TAP

One limitation of Cerberus is that it is unable to handle nested cloning. While it simplifies the programming model, it also restricts users to a specific set of applications. Designing and formally verifying such a mechanism remains future work. Furthermore, we also aim to show that the source code of the Keystone security monitor refines our abstract model, ensuring that our formal guarantees are violated by the implementation.

Bibliography

[1] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. "S-faas: Trustworthy and accountable function-as-a-service using intel SGX". In: *CCSW*. 2019.

[2] Krste Asanović et al. "The Rocket Chip Generator". In: UCB/EECS-2016-17 (Apr. 2016).

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825. URL: https://doi.org/10.3233/978-1-58603-929-5-825.

[4] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. "A fork() in the road". In: *HotOS*. 2019.

[5] Andrea Bittau. "Toward Least-Privilege Isolation for Software". PhD thesis. UCL (University College London), 2009.

[6] Stefan Brenner and Rüdiger Kapitza. "Trust more, serverless". In: *SYSTOR*. 2019.

[7] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 78–92. DOI: 10.1007/3-540-45657-0_7. URL: https://doi.org/10.1007/3-540-45657-0%5C_7.

[8] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. June 2015.

[9] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution". In: *CoRR* abs/1804.05141 (2018). arXiv: 1804.05141. URL: http://arxiv.org/abs/1804.05141.

[10] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation." In: *USENIX Security Symposium*. 2016, pp. 857–874.

[11] 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*. ACM. 2017, pp. 287–305.

[12] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Cryptology ePrint Archive, Report 2016/204. 2016.

[13] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Cal, Ariel J. Feldman, and Edward W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys". In: *USENIX Security Symposium*. 2008.

[14] *Intel Software Guard Extensions*. `https://software.intel.com/sgx`.

[15] *Keystone Open-Source Secure Hardware Enclave*. `https://keystone-enclave.org/`.

[16] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. "Integrating remote attestation with transport layer security". In: *arXiv preprint arXiv:1801.05863* (2018).

[17] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. "An Off-Chip Attack on Hardware Enclaves via the Memory Bus". In: *USENIX Security*. 2020.

[18] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-Weight Contexts: An OS Abstraction for Safety and Performance". In: *OSDK*. 2016.

[19] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave". In: *HASP*. 2016.

[20] Jason Garms Nelly Porter. *Advancing confidential computing with Asylo and the Confidential Computing Challenge*. `https://cloud.google.com/blog/products/identity-security/advancing-confidential-computing-with-asylo-and-the-confidential-computing-challenge`. Feb. 2019.

[21] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. "Oblivious Multi-party Machine Learning on Trusted Processors". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. 2016.

[22] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. "MedleySolver: Online SMT Algorithm Selection". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 453–470. DOI: 10.1007/978-3-030-80223-3_31. URL: https://doi.org/10.1007/978-3-030-80223-3%5C_31.

[23] *Redis Persistence*. https://redis.io/topics/persistence. 2018.

[24] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *16th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2018, Beijing, China, October 15-18, 2018*. 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946. URL: https://doi.org/10.1109/MEMCOD.2018.8556946.

[25] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: *ASPLOS*. 2020.

[26] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *CCS*. 2013.

[27] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. "A formal foundation for secure remote execution of enclaves". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2435–2450.

[28] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. "Clemmys: Towards secure remote execution in FaaS". In: *SYSTOR*. 2019.

[29] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *ATC*. 2017.

[30] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing new operating primitives to improve fuzzing performance". In: *CCS*. 2017.

Appendix A

Abstract Cache and CPU Models

In this section, we present the cache and CPU models used in the TAP models. Note that they have been modified for readability.

```
module common {

//
// address types
//
type vaddr_t = bv32;


type wap_addr_t = bv22;
type word_t  = bv32;


// ------------------------------------------------------------------------ //
// memory.                                                                  //
// ------------------------------------------------------------------------ //
type mem_t = [wap_addr_t]word_t;


// ------------------------------------------------------------------------ //
// constants and functions for word_t                                       //
// ------------------------------------------------------------------------ //
const k0_word_t : word_t = 0bv32;


// ------------------------------------------------------------------------ //
// uarch state                                                              //
// ------------------------------------------------------------------------ //
type cache_set_index_t = integer;
type cache_way_index_t = integer;
type cache_tag_t       = integer;

const kmax_cache_set_index_t : cache_set_index_t;
```

```
33 const kmax_cache_way_index_t : cache_way_index_t;
34 axiom kmax_cache_set_index_t > 0;
35 axiom kmax_cache_way_index_t > 0;
36
37 define valid_cache_way_index(w: cache_way_index_t) : boolean = (
38   w >=0 && w < kmax_cache_way_index_t
39 );
40 define valid_cache_set_index(s: cache_set_index_t) : boolean = (
41   s >= 0 && s < kmax_cache_set_index_t
42 );
43
44
45 type cache_valid_map_t = [cache_set_index_t, cache_way_index_t]boolean;
46 type cache_tag_map_t   = [cache_set_index_t, cache_way_index_t]cache_tag_t
     ;
47
48 function paddr2set(pa : wap_addr_t) : cache_set_index_t;
49 function paddr2tag(pa : wap_addr_t) : cache_tag_t;
50
51 //
52 // registers
53 //
54 type regindex_t = integer;
55 type regs_t = [regindex_t]word_t;
56
57 //
58 // Page Tables (sort of: because we map addresses and not pages).
59 //
60 type addr_perm_t    = bv5;
61 type vaddr2bool_t   = [vaddr_t]boolean;
62 type excl_vaddr_t   = [vaddr_t]boolean;
63 type addr_valid_t   = [vaddr_t]addr_perm_t;
64 type addr_map_t     = [vaddr_t]wap_addr_t;
65
66
67
68 define tap_addr_perm_p(p : addr_perm_t) : boolean = p[0:0] == 1bv1;
69 define tap_addr_perm_a(p : addr_perm_t) : boolean = p[1:1] == 1bv1;
70 define tap_addr_perm_x(p : addr_perm_t) : boolean = p[2:2] == 1bv1;
71 define tap_addr_perm_r(p : addr_perm_t) : boolean = p[3:3] == 1bv1;
72 define tap_addr_perm_w(p : addr_perm_t) : boolean = p[4:4] == 1bv1;
73 define tap_addr_perm_v(p : addr_perm_t) : boolean
74 = tap_addr_perm_x(p) || tap_addr_perm_r(p) || tap_addr_perm_w(p);
75
76 // setters
77 define tap_set_addr_perm_p(p : addr_perm_t) : addr_perm_t = p[4:1] ++ 1bv1
     ;
78 define tap_set_addr_perm_a(p : addr_perm_t) : addr_perm_t = p[4:2] ++ 1bv1
      ++ p[0:0];
```

```
79 define tap_set_addr_perm_x(p : addr_perm_t) : addr_perm_t = p[4:3] ++ 1bv1
       ++ p[1:0];
80 define tap_set_addr_perm_r(p : addr_perm_t) : addr_perm_t = p[4:4] ++ 1bv1
       ++ p[2:0];
81 define tap_set_addr_perm_w(p : addr_perm_t) : addr_perm_t =           1bv1
       ++ p[3:0];
82
83
84 //
85 // enclave types
86 //
87 type tap_enclave_id_t                       = integer;
88
89
90 // what addresses are exlusive to an enclave?
91 type owner_map_t                            = [wap_addr_t]tap_enclave_id_t;
92
93
94 // enclave API call results
95 type enclave_op_result_t = enum {
96   enclave_op_success,
97   enclave_op_invalid_arg,
98   enclave_op_failed
99 };
100
101
102 //
103 // exceptions
104 //
105 type exception_t = enum {
106   excp_none,
107   excp_os_protection_fault,
108   excp_tp_protection_fault
109 };
110
111
112 //
113 // constants and functions for enclave ids
114 //
115 const tap_null_enc_id       : tap_enclave_id_t = 0;
116 const tap_blocked_enc_id    : tap_enclave_id_t = 1;
117 const tap_user_def_enc_id_1 : tap_enclave_id_t = 2;
118 const tap_user_def_enc_id_2 : tap_enclave_id_t = 3;
119 const tap_user_def_enc_id_3 : tap_enclave_id_t = 4;
120 const tap_user_def_enc_id_4 : tap_enclave_id_t = 5;
121 const tap_user_def_enc_id_5 : tap_enclave_id_t = 6;
122
123 define valid_enclave_id(id : tap_enclave_id_t) : boolean
124 = id != tap_null_enc_id       && id != tap_blocked_enc_id    &&
125   id != tap_user_def_enc_id_1 && id != tap_user_def_enc_id_2 &&
```

```
126   id != tap_user_def_enc_id_3 && id != tap_user_def_enc_id_4 &&
127   id != tap_user_def_enc_id_5;
128
129
130
131 } // end module common
```

Listing A.1: Common file

```
1  module abstract_cache {
2
3  type * = common.*;
4  const * = common.*;
5  function * = common.*;
6  define * = common.*;
7
8  var cache_valid_map  : cache_valid_map_t;
9  var cache_tag_map    : cache_tag_map_t;
10
11 procedure [noinline] init_cache()
12   ensures (forall (i : cache_set_index_t, w : cache_way_index_t) ::
13       (valid_cache_set_index(i) && valid_cache_way_index(w)) ==> !
     cache_valid_map[i, w]);
14   modifies cache_valid_map;
15 {
16   var ind : cache_set_index_t;
17   var way : cache_way_index_t;
18   ind = 0;
19
20   while (ind < kmax_cache_set_index_t)
21     invariant (forall (i : cache_set_index_t, w : cache_way_index_t) ::
22         (i >= 0 && i < ind && valid_cache_way_index(w)) ==> !
     cache_valid_map[i, w]);
23   {
24     way = 0;
25     while (way < kmax_cache_way_index_t)
26       invariant (forall (i : cache_set_index_t, w : cache_way_index_t) ::
27           ((i >= 0 && i < ind && valid_cache_way_index(w)) || (i == ind &&
     w >= 0 && w < way)) ==>
28           !cache_valid_map[i, w]);
29     {
30       cache_valid_map[ind, way] = false;
31       way = way + 1;
32     }
33     ind = ind + 1;
34   }
35 }
36
37 procedure [noinline] query_cache(pa : wap_addr_t, repl_way :
     cache_way_index_t)
38   returns (hit : boolean, hit_way : cache_way_index_t)
```

```
39    requires (valid_cache_way_index(repl_way));
40
41    // set hit status
42    ensures !hit <==> (forall (w : cache_way_index_t) ::
43                         valid_cache_way_index(w) ==>
44                           (!old(cache_valid_map)[paddr2set(pa) , w] ||
45                             old(cache_tag_map)[paddr2set(pa), w] !=
       paddr2tag(pa)));
46    ensures (hit <==> (exists (w : cache_way_index_t) ::
47                         valid_cache_way_index(w)              &&
48                         old(cache_valid_map)[paddr2set(pa), w] &&
49                         old(cache_tag_map)[paddr2set(pa), w] == paddr2tag(pa
       )));
50    // do replacement if necessary.
51    ensures (if (!hit)
52                then cache_valid_map[paddr2set(pa), repl_way]
53                else cache_valid_map[paddr2set(pa), hit_way]);
54    ensures (if (!hit)
55                then cache_tag_map[paddr2set(pa), repl_way] == paddr2tag(pa)
56                else cache_tag_map[paddr2set(pa), hit_way]  == paddr2tag(pa)
       );
57
58    // and no other lines are affected.
59    ensures !hit ==>
60            (forall (i : cache_set_index_t, w : cache_way_index_t) ::
61              (i != paddr2set(pa) || w != repl_way) ==> cache_valid_map[i, w
       ] == old(cache_valid_map)[i, w]);
62    ensures !hit ==>
63            (forall (i : cache_set_index_t, w : cache_way_index_t) ::
64              (i != paddr2set(pa) || w != repl_way) ==> cache_tag_map[i, w]
       == old(cache_tag_map)[i, w]);
65    ensures hit ==> (cache_valid_map == old(cache_valid_map) &&
       cache_tag_map == old(cache_tag_map));
66    modifies cache_valid_map, cache_tag_map;
67  {
68    var set : cache_set_index_t;
69    var tag : cache_tag_t;
70    var way : cache_way_index_t;
71
72    set = paddr2set(pa);
73    tag = paddr2tag(pa);
74
75    way = 0;
76    hit = false;
77    while(!hit && way < kmax_cache_way_index_t)
78      invariant (way >= 0);
79      invariant (way <= kmax_cache_way_index_t);
80      invariant hit ==> (cache_valid_map[set, hit_way] && cache_tag_map[set,
       hit_way] == tag);
81      invariant
```

```
82        (!hit <==>
83          (forall (w : cache_way_index_t) ::
84            (w >= 0 && w < way) ==>
85              (!cache_valid_map[set, w] || cache_tag_map[set, w] != tag)));
86    {
87      if (cache_valid_map[set, way] && cache_tag_map[set, way] == tag) {
88        hit = true;
89        hit_way = way;
90      }
91      way = way + 1;
92    }
93
94    if (!hit) {
95      cache_valid_map[set, repl_way] = true;
96      cache_tag_map[set, repl_way] = tag;
97    }
98  }
99
100
101 procedure set_cache_state(_cache_valid_map : cache_valid_map_t,
       _cache_tag_map : cache_tag_map_t)
102   ensures (cache_valid_map == _cache_valid_map);
103   ensures (cache_tag_map == _cache_tag_map);
104   modifies cache_valid_map;
105   modifies cache_tag_map;
106 {
107   cache_valid_map = _cache_valid_map;
108   cache_tag_map = _cache_tag_map;
109 }
110
111
112 control {
113   verif_init_cache  = verify(init_cache);
114   verif_query_cache = verify(query_cache);
115
116   check;
117   print_results;
118 }
119
120 }
```

Listing A.2: Abstract cache model

```
1  module abstract_cpu {
2
3  type * = common.*;
4  const * = common.*;
5  function * = common.*;
6  define * = common.*;
7
8
```

```
9  instance cache : abstract_cache();
10
11 //
12 // CPU state
13 //
14 var cpu_mem          : mem_t;
15 var cpu_regs         : regs_t;
16 var cpu_pc           : vaddr_t;
17 var cpu_enclave_id   : tap_enclave_id_t;
18 var cpu_addr_valid   : addr_valid_t;
19 var cpu_addr_map     : addr_map_t;
20 var cpu_owner_map    : owner_map_t;
21
22 //
23 // CPU Flags
24 //
25 var cpu_cache_enabled : boolean;
26
27
28
29
30 //
31 // CPU Procedures
32 //
33 procedure initialize_cache()
34   modifies cache;
35 {
36   call cache.init_cache();
37 }
38
39
40
41 procedure [inline] fetch_va(vaddr : vaddr_t, repl_way : cache_way_index_t)
42   returns (data : word_t, excp : exception_t, hit : boolean)
43   requires valid_cache_way_index(repl_way);
44   modifies cpu_addr_valid;
45   modifies cache;
46 {
47   var paddr : wap_addr_t;
48   var owner_eid : tap_enclave_id_t;
49   var hit_way : cache_way_index_t;
50
51   // default
52   data = k0_word_t;
53   hit = false;
54
55
56
57   // translate VA -> PA
58   if (!tap_addr_perm_x(cpu_addr_valid[vaddr])) {
```

```
59      excp = excp_os_protection_fault;
60    } else {
61      paddr = cpu_addr_map[vaddr];
62      // are we not allowed to access this memory
63      owner_eid = cpu_owner_map[paddr];
64      if (owner_eid != tap_null_enc_id && owner_eid != cpu_enclave_id) {
65        excp = excp_os_protection_fault;
66      } else {
67        // update accessed bit
68        cpu_addr_valid[vaddr] = tap_set_addr_perm_a(cpu_addr_valid[vaddr]);
69        // perform load
70        data = cpu_mem[paddr];
71        excp = excp_none;
72        // update cahce
73        if (cpu_cache_enabled) {
74          assert(valid_cache_way_index(repl_way));
75          call (hit, hit_way) = cache.query_cache(paddr, repl_way);
76        }
77      }
78    }
79  }
80
81  procedure [inline] load_va(vaddr : vaddr_t, repl_way : cache_way_index_t)
82    returns (data : word_t, excp : exception_t, hit : boolean)
83    requires valid_cache_way_index(repl_way);
84    ensures (!tap_addr_perm_r(old(cpu_addr_valid)[vaddr]) ||
85             (cpu_owner_map[cpu_addr_map[vaddr]] != tap_null_enc_id &&
      cpu_owner_map[cpu_addr_map[vaddr]] != cpu_enclave_id))
86              ==> (hit == false);
87    ensures (forall (p : wap_addr_t, w : cache_way_index_t) ::
88               (paddr2set(p) != paddr2set(cpu_addr_map[vaddr]) || w !=
      repl_way)
89                 ==> ((cache.cache_valid_map[paddr2set(p), w] == old(cache.
      cache_valid_map)[paddr2set(p), w]) &&
90                      (cache.cache_tag_map[paddr2set(p), w] == old(cache.
      cache_tag_map)[paddr2set(p), w])));
91
92    ensures (excp == excp_none && cpu_cache_enabled && !hit)
93                ==> ((cache.cache_valid_map[paddr2set(old(cpu_addr_map)[
      vaddr]), repl_way] == true) &&
94                     (cache.cache_tag_map[paddr2set(old(cpu_addr_map)[vaddr
      ]), repl_way] == paddr2tag(old(cpu_addr_map)[vaddr])));
95    ensures (!cpu_cache_enabled) ==> (cache.cache_valid_map == old(cache.
      cache_valid_map) && cache.cache_tag_map == old(cache.cache_tag_map));
96
97
98    modifies cpu_addr_valid;
99    modifies cache;
100 {
101   var paddr : wap_addr_t;
```

```
102   var owner_eid : tap_enclave_id_t;
103   var hit_way : cache_way_index_t;
104
105   // default
106   data = k0_word_t;
107   hit = false;
108
109
110
111   // translate VA -> PA
112   if (!tap_addr_perm_r(cpu_addr_valid[vaddr])) {
113     excp = excp_os_protection_fault;
114   } else {
115     paddr = cpu_addr_map[vaddr];
116     // are we not allowed to access this memory
117     owner_eid = cpu_owner_map[paddr];
118     if (owner_eid != tap_null_enc_id && owner_eid != cpu_enclave_id) {
119       excp = excp_tp_protection_fault;
120     } else {
121       // update accessed bit
122       cpu_addr_valid[vaddr] = tap_set_addr_perm_a(cpu_addr_valid[vaddr]);
123       // perform load
124       data = cpu_mem[paddr];
125       excp = excp_none;
126       // update cache
127       if (cpu_cache_enabled) {
128         call (hit, hit_way) = cache.query_cache(paddr, repl_way);
129       }
130     }
131   }
132 }
133
134 procedure [inline] store_va(vaddr : vaddr_t, data : word_t, repl_way :
      cache_way_index_t)
135   returns (excp : exception_t, hit : boolean)
136   requires valid_cache_way_index(repl_way);
137   ensures (excp != excp_none ==> cpu_mem == old(cpu_mem));
138   ensures (excp != excp_none) ==> (cpu_addr_valid == old(cpu_addr_valid));
139   ensures (excp == excp_none) ==>
140             (forall (va : vaddr_t) ::
141               (va != vaddr)
142                 ==> (cpu_addr_valid[va] == old(cpu_addr_valid)[va]));
143   ensures (excp == excp_none)
144             ==> (cpu_addr_valid[vaddr] == tap_set_addr_perm_a(old(
      cpu_addr_valid)[vaddr]));
145   ensures (!tap_addr_perm_w(old(cpu_addr_valid)[vaddr]) ||
146            (cpu_owner_map[cpu_addr_map[vaddr]] != tap_null_enc_id &&
      cpu_owner_map[cpu_addr_map[vaddr]] != cpu_enclave_id))
147           ==> (hit == false);
148   ensures (forall (p : wap_addr_t, w : cache_way_index_t) ::
```

```
149             (paddr2set(p) != paddr2set(cpu_addr_map[vaddr]) || w !=
    repl_way)
150               ==> ((cache.cache_valid_map[paddr2set(p), w] == old(cache.
    cache_valid_map)[paddr2set(p), w]) &&
151                    (cache.cache_tag_map[paddr2set(p), w] == old(cache.
    cache_tag_map)[paddr2set(p), w])));
152
153
154   ensures (!cpu_cache_enabled) ==> (cache.cache_valid_map == old(cache.
    cache_valid_map) && cache.cache_tag_map == old(cache.cache_tag_map));
155
156
157   modifies cpu_mem;
158   modifies cpu_addr_valid;
159   modifies cache;
160 {
161   var paddr : wap_addr_t;
162   var owner_eid : tap_enclave_id_t;
163   var hit_way : cache_way_index_t;
164
165   // default
166   hit = false;
167
168   //translate VA -> PA
169   if (!tap_addr_perm_w(cpu_addr_valid[vaddr])) {
170     excp = excp_os_protection_fault; ;
171   } else {
172     paddr = cpu_addr_map[vaddr];
173     // are we not allowed to access this memory
174     owner_eid = cpu_owner_map[paddr];
175     if (owner_eid != tap_null_enc_id && owner_eid != cpu_enclave_id) {
176       excp = excp_tp_protection_fault;
177     } else {
178       // update accessed bit
179       cpu_addr_valid[vaddr] = tap_set_addr_perm_a(cpu_addr_valid[vaddr]);
180       // perform store
181       cpu_mem[paddr] = data;
182       excp = excp_none;
183       // update cache
184       if (cpu_cache_enabled) {
185         call (hit, hit_way) = cache.query_cache(paddr, repl_way);
186       }
187     }
188   }
189   assert (forall (v : vaddr_t) ::
190             (v != vaddr) ==> (cpu_addr_valid[v] == old(cpu_addr_valid)[v])
    );
191 }
192
193 control {
```

```
194   fetch_va_verif = verify(fetch_va);
195   load_va_verif = verify(load_va);
196   store_va_verif = verify(store_va);
197
198
199   check;
200   print_results;
201 }
202
203 }
```

Listing A.3: Abstract cache model

Appendix B

Modification to `store_va`

In this section, we present the updated `store_va` in the Abstract CPU model that enables the copy-on-write mechanism.

```
1  procedure [inline] store_va(vaddr : vaddr_t, data : word_t, repl_way :
       cache_way_index_t)
2    returns (excp : exception_t, hit : boolean)
3    requires (cpu_enclave_id != tap_invalid_enc_id);
4    requires valid_cache_way_index(repl_way);
5    requires tap_enclave_metadata_parent[tap_null_enc_id] ==
       tap_invalid_enc_id;
6    requires (forall (p : wap_addr_t) :: cpu_owner_map[p] !=
       tap_invalid_enc_id);
7    requires (forall (eid : tap_enclave_id_t, pa : wap_addr_t) ::
       tap_enclave_metadata_wap_addr_free[eid][pa] ==> cpu_owner_map[pa] ==
       eid);
8    ensures (forall (eid : tap_enclave_id_t, pa : wap_addr_t) ::
       tap_enclave_metadata_wap_addr_free[eid][pa] ==> cpu_owner_map[pa] ==
       eid);
9
10   ensures (excp != excp_none ==> cpu_mem == old(cpu_mem));
11   ensures (excp != excp_none) ==> (cpu_addr_valid == old(cpu_addr_valid));
12   ensures (excp == excp_none) ==>
13             (forall (va : vaddr_t) ::
14               (va != vaddr)
15                 ==> (cpu_addr_valid[va] == old(cpu_addr_valid)[va]));
16   ensures (excp == excp_none)
17             ==> (cpu_addr_valid[vaddr] == tap_set_addr_perm_a(old(
       cpu_addr_valid)[vaddr]));
18   ensures (!tap_addr_perm_w(old(cpu_addr_valid)[vaddr]) ||
19            (cpu_owner_map[cpu_addr_map[vaddr]] != tap_null_enc_id &&
       cpu_owner_map[cpu_addr_map[vaddr]] != cpu_enclave_id && cpu_owner_map[
       cpu_addr_map[vaddr]] != tap_enclave_metadata_parent[cpu_enclave_id]))
20          ==> (hit == false);
21   ensures (forall (p : wap_addr_t, w : cache_way_index_t) ::
```

```
22              (paddr2set(p) != paddr2set(cpu_addr_map[vaddr]) || w !=
    repl_way)
23                ==> ((cache.cache_valid_map[paddr2set(p), w] == old(cache.
    cache_valid_map)[paddr2set(p), w]) &&
24                     (cache.cache_tag_map[paddr2set(p), w] == old(cache.
    cache_tag_map)[paddr2set(p), w])));
25
26
27    ensures (!cpu_cache_enabled) ==> (cache.cache_valid_map == old(cache.
    cache_valid_map) && cache.cache_tag_map == old(cache.cache_tag_map));
28
29
30    modifies cpu_mem;
31    modifies cpu_addr_valid;
32    modifies cache;
33    modifies cpu_addr_map;
34    modifies tap_enclave_metadata_wap_addr_free;
35  {
36    var paddr : wap_addr_t;
37    var owner_eid : tap_enclave_id_t;
38    var hit_way : cache_way_index_t;
39
40    // default
41    hit = false;
42
43    //translate VA -> PA
44    if (!tap_addr_perm_w(cpu_addr_valid[vaddr])) {
45      excp = excp_os_protection_fault; ;
46    } else {
47      paddr = cpu_addr_map[vaddr];
48      // are we not allowed to access this memory
49      // Allow a child to COW its parents memory
50      owner_eid = cpu_owner_map[paddr];
51      if (owner_eid != tap_null_enc_id &&
52         (owner_eid != cpu_enclave_id && owner_eid !=
     tap_enclave_metadata_parent[cpu_enclave_id]))
53      {
54        excp = excp_tp_protection_fault;
55      } else {
56          // Although this is included in the CPU model, this is actually
    managed by the SM.
57          if (owner_eid == tap_enclave_metadata_parent[cpu_enclave_id]) {
58              // Do COW
59              var found: boolean;
60              var new_paddr: wap_addr_t;
61              var paddr: wap_addr_t;
62              var wap_addr_free_map : [wap_addr_t]boolean;
63
64              // Sanity check
65              assert (cpu_enclave_id != tap_null_enc_id);
```

```
66
67              found = false;
68              paddr = k0_wap_addr_t;
69
70              // Need to find new physical address, by looking at the free
    pool for the current enclave
71              while (LT_wapa(paddr, kmax_wap_addr_t) && found == false)
72                  invariant ((found == false) ==>
73                      (forall (pa : wap_addr_t) ::
74                          (LT_wapa(pa, paddr) ==> (
    tap_enclave_metadata_wap_addr_free[cpu_enclave_id][pa] == false))));
75              {
76                  if (tap_enclave_metadata_wap_addr_free[cpu_enclave_id][
    paddr]) {
77                      found = true;
78                      new_paddr = paddr;
79                  }
80                  paddr = PLUS_wapa(paddr, k1_wap_addr_t);
81              }
82
83              if (found == false) {
84                  if (tap_enclave_metadata_wap_addr_free[cpu_enclave_id][
    paddr]) {
85                      found = true;
86                      new_paddr = paddr;
87                  }
88              }
89
90              wap_addr_free_map = tap_enclave_metadata_wap_addr_free[
    cpu_enclave_id] ;
91              wap_addr_free_map[new_paddr] = false;
92              tap_enclave_metadata_wap_addr_free[cpu_enclave_id] =
    wap_addr_free_map;
93
94              if (found == false) {
95                  // TODO: This means that we couldn't find any free
    physical memory
96                  excp = excp_out_of_memory_fault;
97              } else {
98                  assert (cpu_owner_map[new_paddr] == cpu_enclave_id);
99                  cpu_addr_map[vaddr] = new_paddr;
100                 // update accessed bit
101                 cpu_addr_valid[vaddr] = tap_set_addr_perm_a(cpu_addr_valid
    [vaddr]);
102
103                 // perform store
104                 cpu_mem[new_paddr] = data;
105                 excp = excp_none;
106                 // update cache
107                 if (cpu_cache_enabled) {
```

```
108                     call (hit, hit_way) = cache.query_cache(new_paddr,
    repl_way);
109                 }
110             }
111         } else {
112             // update accessed bit
113             cpu_addr_valid[vaddr] = tap_set_addr_perm_a(cpu_addr_valid[
    vaddr]);
114             // perform store
115             cpu_mem[paddr] = data;
116             excp = excp_none;
117             // update cache
118             if (cpu_cache_enabled) {
119               call (hit, hit_way) = cache.query_cache(paddr, repl_way);
120             }
121         }
122     }
123   }
124   assert (forall (v : vaddr_t) ::
125             (v != vaddr) ==> (cpu_addr_valid[v] == old(cpu_addr_valid)[v])
    );
126 }
```

Listing B.1: Common file