Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing

Dayeol Lee∗
University of California, Berkeley
dayeol@berkeley.edu

Kevin Cheang∗
University of California, Berkeley
kcheang@berkeley.edu

Alexander Thomas
University of California, Berkeley
alexthomas@berkeley.edu

Catherine Lu
University of California, Berkeley
cathyhu@berkeley.edu

Pranav Gaddamadugu
University of California, Berkeley
pranavsaig@berkeley.edu

Anjo Vahldiek-Oberwagner
Intel Labs
anjovahldiek@gmail.com

Mona Vij
Intel Labs
mona.vij@intel.com

Dawn Song
University of California, Berkeley
dawnsong@berkeley.edu

Sanjit A. Seshia
University of California, Berkeley
sseshia@berkeley.edu

Krste Asanović
University of California, Berkeley
krste@berkeley.edu

1 INTRODUCTION

The hardware enclave [6, 23, 25, 28, 29, 36, 41] is a promising method of protecting a program [18, 45, 47, 48] by allocating a set of physical addresses accessible only from the program. The key idea of hardware enclaves is to isolate a part of physical memory by using hardware mechanisms in addition to a typical memory management unit (MMU). The isolation is based on a disjoint memory assumption, which constrains each of the isolated physical memory regions to be owned by a specific enclave. A hardware platform enforces the isolation by using additional in-memory metadata and hardware primitives. For example, Intel SGX maintains per-physical-page metadata called the Enclave Page Cache Map (EPCM) entry, which contains the enclave ID of the owner [22]. The hardware looks up the entry for each memory access to ensure that the page is accessible only when the current enclave is the owner.

However, the disjoint memory assumption also significantly limits enclaves in terms of their performance and programmability. First, the enclave needs to go through an expensive initialization whenever it launches because the enclave program cannot use shared libraries in the system nor clone from an existing process [38]. Each initialization consists of copying the enclave program into the enclave memory and performing measurements to stamp the initial state of the program. The initialization latency proportionally increases depending on the size of the program and the initial data. Second, the programmer needs to be aware of the non-traditional assumptions about memory. For instance, system calls like fork or clone no longer rely on efficient copy-on-write memory, resulting in significant performance degradation [18, 47].

A few studies have proposed platform extensions to allow memory sharing of enclaves. Yu et al. [65] proposes Elasticlave, which modifies the platform such that each enclave can own multiple physical memory regions that the enclave can selectively share with other enclaves. An enclave can map other enclaves’ memory regions to its virtual address space by making a request, followed by the owner granting access. Elasticlave improves the performance of enclave programs that relies on heavy inter-process communication (IPC). Li et al. [38] proposes Plug-In Enclave (PIE), which is an extension of Intel SGX. PIE enables faster enclave creation by introducing a shared enclave region, which can be mapped to another enclave by a new SGX instruction EMAP. EMAP maps the entire virtual address space of a pre-initialized plug-in enclave. PIE improves the performance of enclave programs with large initial code and read-only data (e.g., serverless workloads). Although the prior work shows that memory sharing can substantially improve performance, they do not provide formal guarantees about security.

∗Both authors contributed equally to the paper.
Unsurprisingly, the disjoint memory assumption of enclaves is crucial for the security of the enclave platforms. Previous studies [25, 44, 51, 57] formally prove high-level security guarantees of enclave platforms such as non-interference properties, integrity, and confidentiality based on the disjoint memory assumption. However, to our best knowledge, no model formally verifies the security guarantees under the weakened assumption that the enclaves can share memory.

Practical formal verification requires choosing the right level of abstraction to model and apply automated reasoning. Verification on models that conform to the low-level implementation [44] or source-level code [11, 20, 30, 54, 56] is often platform-specific in that it only provides security guarantees to those implementations and thus does not apply generally. If one seeks to verify that a memory-sharing approach on top of a family of enclave platforms is secure, it is not easy to reuse verification efforts for specific implementations. We seek an approach that is incremental and also applicable to existing platforms.

Moreover, there are many ways one could design a memory-sharing model, each varying in complexity and flexibility. Complex models can provide more flexibility to optimize the applications for performance, but this often comes at the cost of increasing the complexity of formal verification. However, if memory sharing is too restrictive, it also becomes hard for programmers to leverage it for performance improvements. Thus, we seek a simple sharing model with a balance between flexibility and ease of verification.

To this end, this paper presents Cerberus, a formal approach to secure and efficient enclave memory sharing. Cerberus chooses single-sharing model with read-only shared memory, which allows each enclave to access only one read-only shared memory. We show that this design decision significantly reduces the cost of verification by simplifying invariants, yet still provides a big performance improvement for important use cases. We formalize an enclave platform model that can accurately capture high-level semantics of the extension and formally verify a property called Secure Remote Execution (SRE) [57]. We perform incremental verification by starting from an existing formal model called Trusted Abstract Platform (TAP) [57] for which the SRE property is already established. Finally, the paper shows the feasibility of Cerberus by implementing it in an existing platform, RISC-V Keystone [36]. Cerberus can substantially reduce the initialization latency without incurring significant computational overhead.

To summarize, our contributions are as follows:

- Provide a general formal enclave platform model with memory sharing that weakens the disjoint memory assumption and captures a family of enclave platforms.
- Formally verify that the modified enclave platform model satisfies SRE property via automated formal verification.
- Provide programmable interface functions that can be used with existing system calls.
- Implement the extension on an existing enclave platform and demonstrate that Cerberus reduces enclave creation latency.

## 2 Motivation and Background

### 2.1 Use Cases of Memory Sharing in Enclaves

Many programs these days take advantage of sharing their memory with other programs. For example, shared libraries allow a program to initialize faster with less physical memory than static libraries because the operating system can reuse in-memory shared libraries for multiple processes. Similarly, sharing large in-memory objects (e.g., an in-memory key-value store) can be shared across multiple processes. Running a program inside an enclave disables memory sharing because of the disjoint memory assumption. This section introduces a few potential use cases of memory sharing in enclaves to motivate Cerberus. Memory sharing can significantly improve the performance of enclave programs that require multiple isolated execution contexts with shared initial code and data.

**Serverless Workloads.** Serverless computing is a program execution model where the cloud provider allocates and manages resources for a function execution on demand. In the model, the program developer only needs to write a function that runs on a language runtime, such as a specific version of Python. Many serverless frameworks [2, 4] reduce the cold-start latency of the execution with pre-initialized workers containing the language runtime. As described earlier by Li et al. [38], the workers will suffer from an extremely long initialization latency (e.g., a few seconds) when they run in enclaves, as the language runtimes are typically a few megabytes (e.g., Python is 4 MB). Because the difference between worker memories (e.g., heap, stack, and the function code) can be as small as a few kilobytes, a large amount of initialization latency and memory usage can be saved by sharing memory.

**Inference APIs.** Machine learning model serving frameworks [1, 3, 5] allow users to send their inputs and returns the model’s inference results. Serving different users with separate enclaves will have a longer latency as the model size increases. As of now, the five most popular models in Huggingface [3] have a number of parameters ranging from a few hundred million to a few hundred billion, which would occupy at least hundreds of megabytes of memory. Sharing memory will drastically reduce the latency and memory usage of such inference APIs in enclaves.

**Web Servers.** Multi-processing web servers handle requests with different execution contexts while sharing the same code and large objects. For example, a web server or an API server that provides read access to a large object (e.g., front-end data or database) will suffer from long latency and memory usage running in an enclave. If enclaves can share a memory, they can respond with lower latency and smaller memory usage.

### 2.2 The Secure Remote Execution Property

As mentioned earlier, much of the prior work identifies integrity and confidentiality as key security properties for enclave platforms. As a result, we aim to prove a property that is at least as strong as these two, which is the SRE property. To provide intuition behind the property, the typical setting for an enclave user is that the user wishes to execute their enclave program securely on a remote enclave platform. The remote platform is largely untrusted, with an operating system, a set of applications, and other enclaves that...
may potentially be malicious. Thus it is desirable to create a secure channel between the enclave program and the user in order to set up the enclave program securely. Consequently, in order to have end-to-end security, we need to ensure that the enclave platform behaves in the following three ways:

- The measurement of an enclave on the remote platform can guarantee that the enclave is set up correctly and runs in a deterministic manner
- Each enclave program is integrity-protected from the untrusted entities and thus executes deterministically,
- Each enclave program is confidentiality-protected to avoid revealing secrets to untrusted entities.

These three behaviors manifest as the secure measurement, integrity, and confidentiality properties as defined in Section §5 and are ultimately what we guarantee for our platform model extended with Cerberus.

### 2.3 Formal Models of Enclave Platforms

Prior work has formally modeled and verified enclave platform models for both functional correctness and adherence to safety properties similar to the SRE property. While verification at the source code level (e.g., Komodo [25]) provides proofs of functional correctness and noninterference of enclaves managed by a software security monitor, existing verification efforts are often closely tied to the implementation, making it difficult to apply existing work to our extension. A binary- or instruction-level verification (e.g., Serval [44]) on the other hand, focuses on automating the verification of the implementation. Working with binary-level models is often difficult and tedious because the binary often lacks high-level program context (e.g., variable names). This paper aims to verify the enclave memory sharing on general enclave platforms. Thus, binary-level verification is a non-goal of this paper, while it can complement the approach by verifying that a given implementation refines our model at the binary level.

The **Trusted Abstract Platform model** [57] is an abstraction of enclave platforms that was introduced with the SRE property. The SRE property states that an enclave execution on a remote platform follows its expected semantics and is confidentiality-protected from a class of adversaries defined along with the TAP model. This property provides the end-to-end verification of integrity and confidentiality for enclaves running on a remote platform. It has also been formally proven that the state-of-the-art enclave platforms such as Intel’s SGX [22, 41] and MIT’s Sanctum [23, 34] refine the TAP model and hence satisfy SRE against various adversary models. To our best knowledge, the TAP is the only model for formal verification that has been used to capture enclave platforms in a general way. The level of abstraction also makes it readily extensible. For these reasons, this paper extends the TAP model.

### 3 DESIGN DECISIONS

Several design decisions were made in our approach to conform to our design goals. The memory sharing model and interface designs are crucial for modeling, verification, and implementation. This section discusses the details of how we chose to design the memory-sharing model and interface.

#### 3.1 Writable Shared Memory

Some programs use shared memory for efficient inter-process communication (IPC), which requires any writes to the shared memory to be visible to the other processes. Elasticlave [65] allows an enclave to grant write permissions for a memory region to the other enclaves such that they can communicate without encrypting or copying the data. However, the authors also show that such writable shared memory requires the write permission to be dynamically changed to prevent interference between enclaves. As formal reasoning on memory with dynamic permission will introduce a non-trivial amount of complexity, we leave this direction as future work. Thus, Cerberus does not support use cases based on IPC or other mutable shared data. Similarly, PIE [38] also only enables read-only memory sharing among enclaves.

#### 3.2 Memory Sharing Models

Fig. 1 shows four different memory-sharing models with varying levels of flexibility. We discuss the implications for the implementation and the feasibility of formal verification for each model. For this discussion, we use the number of access relations between enclaves and memory regions as a metric for the complexity of both verification and implementation.

- **No Sharing.** We refer to the model that assumes the disjoint memory assumption as the no-sharing model, which is implemented in state-of-the-art enclaves [25, 36, 41]. The no-sharing model strictly disallows sharing memory and assigns each physical address to only one enclave. As a result, the number of access relations is \(O(\max(m, n)) = O(m)\), where \(m\) is the number of physical memory regions, and \(n\) is the number of enclaves. Thus, implementations with no-sharing will require metadata scaling with \(O(m)\) to maintain the access relations. For instance, each SGX EPC page has a corresponding entry in EPCM, which contains the owner ID of the page. The no-sharing model has been formally verified at various levels [25, 44, 57].

- **Arbitrary Sharing.** One can completely relax the sharing model and allow any arbitrary number of enclaves to share memory (as in Elasticlave). We refer to this sharing model as the arbitrary-sharing model. In this case, the number of access relations between enclaves is \(O(mn)\). Consequently, arbitrary sharing requires metadata scaling with \(O(mn)\).
Capped Sharing. To achieve scalability in the number of access relations, one can constrain the sharing policy such that each enclave can only access a limited number of shared physical memory regions. We refer to this sharing model as the capped-sharing model. In Fig. 1, capped sharing shows an example where each enclave is only allowed to access at most two additional shared physical memory regions. As an example, PIE [38] introduces a new type of enclave called plug-in enclave, which can be mapped to the virtual address space of a normal enclave. This reduces the number of relations to $O(kn + m)$, where $k$ is the number of shared physical memory regions that are allowed to be accessed by an enclave.

Single Sharing. The single-sharing model is a special case of the capped sharing with $k = 1$. Thus, the model only allows enclaves to access the shared memory regions of a particular enclave. Single sharing reduces the complexity to $O(m)$.

3.3 Formal Verification of Sharing Models

Formally verifying arbitrary- or capped-sharing models is challenging due to the flexibility of the models. Verifying security properties such as SRE requires reasoning about safety properties with multiple traces and platform invariants with nested quantifiers. In our experience, modeling an arbitrary number of shared memory would add to this complexity. For example, one inductive invariant needed to prove SRE on TAP is that if a memory region is accessible by an enclave, the region is owned by the enclave. To allow an arbitrary number of memory regions to be shared, the invariant should be extended such that it existentially quantifies over all relations, for example, stating that the owner of the memory is one of the enclaves that shared their memory with the enclave (See §5 and Eq. (8) for details). The encoding of the invariant in TAP uses first-order logic with the theory of arrays and, in general, is not decidable [15]. As a result, the introduction of this quantifier further complicates the invariant. Despite the limiting constraint in capped sharing, a formal model capturing any arbitrary limit $k$ would still require modeling an arbitrary number of the shared memory as in the arbitrary sharing scheme and face the same complication.

In contrast, the single-sharing model significantly reduces the efforts of formal reasoning and implementation. First, the formal reasoning no longer requires the complex invariant because the memory accessible to an enclave either belongs to the enclave itself or only another enclave that is sharing memory. Second, the implementation becomes much simpler as it requires only one per-enclave metadata to store the reference to the shared memory. The platform modification also becomes minimal as it only checks one more metadata per memory access.

Despite its simplicity, the single-sharing model can still improve the performance of programs by having all of the shared contents (e.g., shared library, initial code, and initial data) in a shared enclave. Figure 2 depicts the difference between capped- and single-sharing models. With the capped-sharing model, each shareable content can be initialized with a separate enclave, allowing each enclave to map up to $k$ different enclave memory regions (i.e., Plug-In enclaves in PIE). Single-sharing model only allows each enclave to map exactly one other enclave, leaving potential duplication in memory when heterogeneous workloads have shared code (e.g., libY). We claim that the benefit of the model’s simplicity outweighs the limitation, as the single sharing does not have notable disadvantages over capped-sharing when there is no common memory among heterogeneous workloads.

3.4 Interface

Enclave programs need interface functions to share memory based on the sharing model. Elasticlave and PIE introduce explicit operations to map or unmap the shareable physical memory region to the virtual address space of the enclave. For example, Elasticlave requires an enclave program to explicitly call map operation to request access to the region, which will be approved by the owner via share operation. Similarly, PIE allows an enclave to use EMAP and EUNMAP instructions to map and unmap an entire plug-in enclave memory to the virtual address of the enclave.

Elasticlave and PIE allow an enclave program to map shareable physical memory regions to its virtual address space. However, there are a few downsides to the approaches. First, the programmers must manually specify which part of the application should be made shareable. In most cases, the programmers must completely rewrite a program such that the shareable part of the program is partitioned into a separate enclave memory. Second, a dynamic map or unmap requires local attestation, which verifies that the newly-mapped memory is in an expected initial state. Thus, the measurement property of a program relies on the measurement property of multiple physical memory regions.

Cerberus takes an approach similar to a traditional optimization technique, which clones an address space with copy-on-write, as in system calls like clone and fork. This approach fits Cerberus use cases where the shareable regions include text segments, static data segments, and dynamic objects (e.g., a machine learning model). In general, programmers expect such system calls to copy the entire virtual address space of a process — no matter what it contains — to a newly-created process. A similar interface will allow the programmers to write enclave programs with the same expectation. Also, such an interface will not require additional properties or assumptions on measurements of multiple enclaves. Since the initial code of an enclave already contains when to share its entire address space, the initial measurement implicitly includes all memory contents to be shared.

To this end, Cerberus introduces two enclave operations, which are Snapshot and Clone. Snapshot freezes the entire memory state.
We first introduce a threat model in Section §4.1 that is consistent with the current state-of-the-art enclave threat models. Our extension follows the typical enclave threat model where the enclave program is only callable from the enclave itself, allowing the enclave to decide when to share its memory. The adversary can call Clone any time, which does not break the security because it can be viewed as a special way of launching an enclave (See §4.4.1). When the adversary calls Clone on an existing enclave, a new enclave is created and resumes with a copy-on-write (CoW) memory of the snapshot. Thus, any changes to each of the enclaves after the Clone are not visible to each other. The following sections formally discuss the sharing model and the interface of Cerberus.

4 FORMAL MODEL

We first introduce a threat model in Section §4.1 that is consistent with these goals and the current state-of-the-art enclave threat models. Then, we list and justify our assumptions in Section §4.2, introduce our formal models of the platform and adversary based on these assumptions in Section §4.3 and then introduce the two new operations Snapshot and Clone of Cerberus in Section §4.4.

In section §5, we use these formal models to define the SRE property, which is a critical security property used to prove that enclaves executing in the remote platform are running as expected and confidentially. These properties are then formally verified using incremental verification on TAP. In other words, our formal models extend the TAP model introduced by Subramanyan et al. [57]. While SRE has been proven on the extended TAP model, Cerberus design weakens the disjoint memory assumption to allow memory sharing. In addition, it is not immediately clear that the two additional operations clearly preserve SRE. Thus, we prove that SRE still holds under our extended model with the operations. For the rest of the literature, we refer to the original formal platform model defined by Subramanyan et al. [57] as TAP and our extended model as TAPc.

4.1 Threat Model

Our extension follows the typical enclave threat model where the user’s enclave program is integrity- and confidentiality-protected over the enclave states (e.g. register values and data memory owned by the enclave program) against any software adversary running in the remote enclave platform. The software adversaries of an enclave include the untrusted operating system, user programs, and the other enclaves as shown in Figure 3.

With Cerberus, enclaves may share data or code that were common between enclaves before the introduction of the Clone. We assume that the memory is implicitly not confidential among these enclaves with shared memory. However, each enclave’s memory should not be observable by the operating system or other enclaves and applications. We ensure that the enclaves are still write-isolated, which means that any modification to the data from one enclave must not be observable to the other enclaves, even to the enclave that it cloned from. Thus, any secret data needs to be provisioned after the enclave is cloned. It is the enclave programmer’s responsibility to make sure that the parent enclave does not contain any secret data that can be leaked through the children.

We do not consider the program running in the enclave to be vulnerable or malicious by itself. For example, a program can generate a secret key in the shared memory, and encrypt the confidential data of the child with the key. This would break confidentiality among children enclaves write-isolated from each other because the children will have access to the key in the shared memory. We do not consider such cases, but this could be easily solved by having programs load secrets to their memory after they have created the distrusting children.

Since our main goal is to design a generic extension, we also do not consider any type of side-channel attack or architecture-specific attack [16, 17, 31, 35, 39, 40, 43, 52, 59, 60, 62, 64] in this paper. We leave side-channel resilient interface design as future work. We note that since the base TAP model has also been used to prove side-channel resiliency on some enclave platforms [23, 36], it is not impossible to extend our proofs to such adversary models. Denial of service against the enclave is also out of the scope of the paper; this is consistent with the threat models for existing state-of-the-art enclave platforms.

A formal model of the threat model is described in more detail in Section §4.3.5 after the formal definition of the platform.

4.2 TAPc Model Assumptions

Below, we summarize a list of assumptions about the execution model of TAPc: that we make for the purpose of simplifying and abstracting the modeling. These assumptions are consistent with the adversary model described above:

- TAPc inherits the assumptions and limitations of TAP [57], which include assuming that every platform and enclave operation is atomic relative to one another, assuming the DRAM is trusted, no support for demand paging, assuming a single-core and single-process model, and assuming properties of cryptographic functions used for measurement.
- If an enclave operation returns with an error code, we assume that the states of the platform are entirely reverted to the state prior to the execution of that operation.
- State continuity of enclaves is out-of-scope in our models, consistent with prior work TAP [57], and can be addressed using alternative methods [27, 46].
- The memory allocation algorithm (e.g. for copy-on-write) is deterministic given that the set of unallocated memory is the same. This means that given any two execution sequences of a platform, as long as the page table states are the same,
the allocation algorithm will return the same free memory location to allocate.

Next, we introduce our formal models describing the platform which extends the existing TAP model with Snapshot and Clone under these assumptions.

### 4.3 Formal TAPc Platform Model Overview

As mentioned, a user of an enclave platform typically has a program and data that they would like to run securely in a remote server, isolated from all other processes as shown in Figure 3. Such a program can be run as an enclave \( e \). The remote server provides isolation using its hardware primitives and software for managing the enclaves, where the software component is typically firmware or a security monitor. This software component provides an interface for the enclave user through a set of operations, denoted by \( O \), for managing \( e \). The goal is to guarantee that this enclave \( e \) is protected from all other processes on the platform and running as expected. For the purpose of understanding the proofs, we refer to the enclave we would like to protect as the protected enclave \( e \). We make this distinction to differentiate it from adversary-controlled enclaves.

#### 4.3.1 Platform and Enclave State

The platform can be viewed as a transition system \( M = (S, I, \rightarrow) \) that is always in some state denoted by \( \sigma \in S \). Alternatively, \( \sigma \) can be viewed as an assignment of values to a set of state variables \( V \). The platform starts in an initial state in the set \( I \) and transitions between states defined by a transition relation \( \rightarrow \subseteq S \times S \). We write \( (\sigma, \sigma') \in \rightarrow \) to mean a valid transition of the platform from \( \sigma \) to \( \sigma' \). An execution of the platform therefore emits a (possibly infinite) sequence of states \( \pi = \langle \sigma^0, \sigma^1, \ldots \rangle \), where \( (\sigma^i, \sigma^{i+1}) \in \rightarrow \) for \( i \in \mathbb{N} \). We write \( \pi^i = \sigma^i \) interchangeably, but will usually write \( \pi^i \) whenever referencing a specific trace. When an enclave is initially launched, it is in the initial state prior to enclave execution, which we indicate using the predicate \( \text{init}(\sigma) : S \rightarrow \text{Bool} \). We describe the set of variables \( V \) and enclave state \( E_e(\sigma) \) for TAPc in the following Section 4.3.2

#### 4.3.2 TAPc State Variables

Each of the variables \( V \) in TAPc are shown in Table 1. \( pc : VA \) is an abstraction of the program counter whose value is a virtual address from the set of virtual addresses \( VA \). \( AM : \Pi \rightarrow PA \) is a register file that is a map \( ^1 \) from the set of register indices (of natural numbers) \( \Pi \) to the set of words \( PA \). \( PA : VA \rightarrow PA \) is an abstraction of memory that maps the set of physical addresses \( PA \) to a set of words. We write \( \Pi[a] \) to represent the memory value at a given physical address \( a \in PA \). A page table abstraction defines the mapping of virtual to physical addresses \( AP \) and access permissions \( ap_{\text{perm}} : VA \rightarrow ACL \), where ACL is the set of read, write, and execute permissions. ACL can be defined as the product \( VA \rightarrow \text{Bool} \times \text{Bool} \times \text{Bool} \), where \( \text{Bool} = \{\text{true}, \text{false}\} \) and the value of the map corresponds to the read, write, and execute permissions for a given virtual address index. \( \epsilon_{\text{curr}} : \mathcal{E}_{\text{id}} \) represents the current enclave that is executing. \( \mathcal{E}_{\text{id}} = \mathcal{N} \cup \{OS\} \cup \{e_{\text{nao}}\} \) is the set of enclave IDs represented by natural numbers and a special identifier \( OS \) representing the untrusted operating system. We reserve the identifier \( e_{\text{nao}} \) to refer to the invalid enclave ID which can be thought of as a default value that does not refer to any valid enclave. For the ease of referring to whether an enclave is valid and launched, we define the predicate \( \text{valid}(e_{\text{id}}) = e_{\text{id}} = e_{\text{nao}} \lor e_{\text{id}} = OS \) that returns whether or not an ID is a valid enclave ID. The active predicate returns true for an enclave \( e \) whenever it is launched or cloned and not yet destroyed in state \( \sigma \). \( o \) is a map that describes the ownership of physical addresses, each of which can be owned by an enclave (with the corresponding enclave ID) or the untrusted operating system.

Lastly, each enclave \( e \) has a set of enclave metadata \( M \), which is a record of variables described in Table 2. We abuse notation and write \( M^pc[e] \) to represent the program counter \( M^pc \) of \( e \) in the record stored in the metadata map \( M \). We use the enclave index operator \([.] \) similarly for the other metadata fields defined in Table 2 to refer to a particular enclave’s metadata. \( M^ep[e] \) is the entry point of the enclave that the enclave \( e \) starts in after the Launch and before Enter. \( M^am[e] \) is the virtual address map of the enclave program. \( M^perm[e] \) is the map of address permissions for each virtual address. \( M^ev[e] \) is the map from virtual addresses to Boolean values representing whether an address is allocated to the enclave. \( M^{pc}[e] \) is the current program counter of the enclave. \( M^{exp}[e] \) is the saved register file of the enclave. \( M^{pause}[e] \) is a Boolean representing whether or not the enclave has been paused and is initially false at launch.

These variables were introduced in the base TAP model and are unmodified in TAPc. We introduce the remaining four metadata variables required for Cerberus in Section 4.4, which are additional state variables in TAPc that are not defined in the base TAP model.

The state \( E_e(\sigma) \) is a projection of the platform state to the enclave state of \( e \) that includes \( M^ep[e] \), \( M^am[e] \), \( M^{exp}[e] \), and the projection of enclave memory \( \lambda v \in VA.ITE(M^ev[e][\sigma][\Pi], M^am[e]\Pi) \). In the last expression, \( \lambda \) is the usual lambda operator over the set of virtual addresses \( v \) and expression body \( E \), \( ITE(c, expr_1, expr_2) \) is the if then else operator that returns \( expr_1 \) if condition \( c \) is true and \( expr_2 \) otherwise. \( \bot \) is the constant bottom value which can be thought of as a don’t-care or unobservable value. This projection of memory represents all memory accessible to enclave \( e \), including shared memory and memory owned by enclave \( e \) as referenced by the virtual address map \( M^am[e] \).

<table>
<thead>
<tr>
<th>State Var.</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>VA</td>
<td>The program counter</td>
</tr>
<tr>
<td>( \Delta_f )</td>
<td>( \Pi \rightarrow W )</td>
<td>General purpose registers.</td>
</tr>
<tr>
<td>( \Pi )</td>
<td>( PA \rightarrow W )</td>
<td>Physical memory.</td>
</tr>
<tr>
<td>( ap )</td>
<td>( VA \rightarrow PA )</td>
<td>Page table abstraction; virtual to physical address map.</td>
</tr>
<tr>
<td>( ap_{\text{perm}} )</td>
<td>( VA \rightarrow ACL )</td>
<td>Page table abstraction; virtual to their permissions.</td>
</tr>
<tr>
<td>( \epsilon_{\text{curr}} )</td>
<td>( \mathcal{E}<em>{\text{id}} \rightarrow \mathcal{E}</em>{\text{id}} )</td>
<td>Current executing enclave ID (or ( e_{\text{id}} = OS ) if the OS is executing).</td>
</tr>
<tr>
<td>( o )</td>
<td>( PA \rightarrow \mathcal{E}_{\text{id}} )</td>
<td>Map from physical addresses to the enclave that owns it.</td>
</tr>
<tr>
<td>( M )</td>
<td>( \mathcal{E}<em>{\text{id}} \rightarrow \mathcal{E}</em>{M} )</td>
<td>Map of enclave IDs to enclave metadata. ( \text{emsp}[OS] ) stores a checkpoint of the OS.</td>
</tr>
</tbody>
</table>

Table 1: TAPc: State Variables \( V \).
Table 2: Records of TAP

<table>
<thead>
<tr>
<th>State Var.</th>
<th>Type</th>
<th>Description of each field</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{VA}$</td>
<td>VA</td>
<td>Enclave entrypoint</td>
</tr>
<tr>
<td>$M_{MAM}$</td>
<td>VA</td>
<td>Enclave’s virtual address map</td>
</tr>
<tr>
<td>$M_{Perm}$</td>
<td>VA $\rightarrow$ PA</td>
<td>Enclave’s address permissions.</td>
</tr>
<tr>
<td>$M_{EV}$</td>
<td>VA $\rightarrow$ Bool</td>
<td>Set of private virtual addresses.</td>
</tr>
<tr>
<td>$M_{pc}$</td>
<td>VA</td>
<td>Saved program counter</td>
</tr>
<tr>
<td>$M_{regs}$</td>
<td>$\mathbb{N} \rightarrow W$</td>
<td>Saved registers.</td>
</tr>
<tr>
<td>$M_{paused}$</td>
<td>Bool</td>
<td>Whether enclave is paused</td>
</tr>
<tr>
<td>$M_{IS}$</td>
<td>Bool</td>
<td>Whether the enclave is a snapshot</td>
</tr>
<tr>
<td>$M_{R_SS}$</td>
<td>$\mathbb{N}$</td>
<td>Number of children enclaves.</td>
</tr>
<tr>
<td>$M_{RP}$</td>
<td>$\mathbb{E}_{id}$</td>
<td>Enclave’s root snapshot.</td>
</tr>
<tr>
<td>$M_{PAF}$</td>
<td>PA $\rightarrow$ Bool</td>
<td>Map of free physical addresses.</td>
</tr>
</tbody>
</table>

4.3.3 Enclave Inputs and Outputs. Communication between an enclave $e$ and external processes for a given state $\sigma$ are controlled through $e$’s inputs $L_e(\sigma)$ and its outputs $O_e(\sigma)$. $L_e(\sigma)$ includes the arguments to the operations that manage enclave $e$, areas of memory outside of the enclave that the enclave may access and an untrusted attacker may write to, and randomness from the platform. $O_e(\sigma)$ contains the outputs of enclave $e$ that are writable to by $e$ and accessible to the attacker and the user.

4.3.4 Platform and Enclave Execution. An execution of an enclave $e$ is defined by the set of operations from $O$, in which the execution of an operation is deterministic up to its input $L_e(\sigma)$ and current state $E_e(\sigma)$. This means that given the same inputs $L_e(\sigma)$ and enclave state $E_e(\sigma)$, the changes to enclave state $E_e(\sigma)$ is deterministic. The set of operations for the base TAP model is $O_{base} \equiv \{\text{Launch}, \text{Destroy}, \text{Enter}, \text{Exit}, \text{Pause}, \text{Resume}, \text{AdversaryExecute}\}$. TAP$_C$ extends the base set with two additional operations: $O \equiv O_{base} \cup \{\text{Snapshot, Clone}\}$. We use the predicate $curr(\sigma) = e$ to indicate that enclave $e$, which may be adversary controlled, is executing at state $\sigma$ and $curr(\sigma) = OS$ to indicate that the operating system is executing.

4.3.5 Formal Adversary Model. In our model, untrusted entities such as the OS and untrusted enclaves are represented by an adversary $A$ that can make arbitrary modifications to state outside of the protected enclave $e$, denoted by $A_e(\sigma)$. Consistent with the base TAP model, the untrusted entities and protected enclave $e$ takes turns to execute under interleaving semantics in our formal TAP$_C$ model, as illustrated in Figure 4. Under these semantics, the adversary is allowed to take any arbitrary number of steps when an enclave is not executing. Likewise, the protected enclave is allowed to take any number of steps when the adversary is not executing without being observable to the adversary.

Conventionally, we define an adversary with an observation and tamper function that describes what the adversary can observe and change in the platform state during its execution to break integrity and confidentiality. The execution of the adversary is the operation $A_{AdversaryExecute}$ in $O_{base}$ during which either the tamper or observation functions can be used by the adversary. Figure 4 describes these two functions for our model.

Tamper Function. The tamper function is used to model these malicious modifications to the platform state by the adversary and is defined over $A_e(\sigma)$ which includes any memory location that is not owned by the protected enclave $e$ and page table mappings.

Observation Function. The adversary’s observation function is denoted $obs_e(\sigma)$. In our model, we allow the adversary to observe locations of the memory that are not owned by the protected enclave $e$, described by the set $obs_e(\sigma) \equiv O_e(\sigma) \approx L \in PA.TE(\sigma \cdot o[p] \not= e, \sigma.P[\cdot], \bot)$. Intuitively, $obs_e$ is a projection of the platform state that is observable by the adversary whose differences should be excluded by the property. For example, if the same enclave program operating over different secrets reveals secrets through the output, that is a bug in the enclave program and we do not protect from this. The adversaries may try to modify or read the enclave state during the lifetime of the enclave.

Under this threat model, we prove that the TAP$_C$ model still satisfies the SRE property described in Section 5.

4.4 The Extended Enclave Operations

Cerberus is the extension of enclave platforms with two new operations Snapshot and Clone to facilitate memory sharing among enclaves. Intuitively, Snapshot converts the enclave executing the operation into a read-only enclave and Clone creates a child enclave from the parent enclave being cloned so that the child enclave can read and execute the same memory contents as the parent at the time of clone.

This extension requires four new metadata state variables that are indicated in Table 2 with the $\dagger$ symbol. $M_{IS}[e]$ is a Boolean valued indicating whether or not Snapshot has been called on the enclave $e$. $M_{CC}[e]$ is the number of children $e$ has, or, in other words, the number of times a clone has been called on the enclave $e$ where $e$ is the parent of Clone. $M_{R_SS}[e]$ is a reference to the root snapshot of $e$ if one exists, and $M_{PAF}[e]$ is a map of addresses that have been assigned to $e$ but are not yet allocated memory.

We now define the semantics of the two new operations introduced in Cerberus.

4.4.1 Clone. Clone creates a clone of an existing logical enclave such that there exist two enclaves with identical enclave states.
Clone alone provides a functionality similar to fork and clone system calls, no matter whether the platform enables memory sharing. More concretely, the Clone takes in three arguments: the ID of the existing parent enclave \( e^p \), and a set of physical addresses assigned to the child enclave \( x_P \). The assigned physical addresses are marked as free (i.e., \( MPAF [e^p][p] = true \), \( \forall p \in x_P \)), so that the parent’s memory can be copied to them. The child enclave \( e_c \), with corresponding enclave ID \( e^c \), is used to create a clone of the parent \( e^p \) such that \( E_{e^c} (\sigma) = E_{e^p} (\sigma) \). In other words, the virtual memory of both enclaves are equal. We write \( E_{e^c} (s_0) \) to denote the initial state of the parent such that \( init (E_{e^p} (\sigma)) \).

We view the Clone as a special way of creating an enclave; instead of starting from the initial enclave state \( E_{e^p} (s_0) \), we start from an existing enclave \( e^p \), which is effectively identical to creating two enclaves with the same initial state and then executing the same sequence of inputs up until the point clone was called.

To prevent the malicious use of clones, we require the condition Eq. 1 to hold during state \( \sigma \) when Clone is called:

\[
\begin{align*}
\sigma.e_{curr} &= OS \land valid(e^p) \land active(e^p, \sigma) \land \\
valid(e^c) \land \sim active(e^c, \sigma) \land \\
e^c \neq e^p \land \\
\forall p \in PA. p \in x_P \Rightarrow \sigma.o[p] = OS \land \\
sufficient_mem(\sigma.o)
\end{align*}
\]

This condition states that the Clone succeeds if and only if the operating system (and hence not an enclave) is currently executing, the parent is a valid and active enclave, the child enclave ID is valid but it doesn’t point to an active enclave, both the parent and child enclave IDs are distinct, all physical addresses in \( x_P \) are owned by the OS (and thus can be allocated to the enclave), and there is sufficient memory to be allocated to the enclave.

If the condition passes, Clone copies all of the data in the virtual address space of \( e^p \) to \( e_c \) to ensure write isolation. For each virtual address \( v \) mapped by \( e^p \) (mapped), Clone first selects a physical address \( p \) owned by \( e_c \), copies the contents from \( \Pi [MAM[e^p]](v) \) to \( \Pi[p] \), and update the page table of \( e_c \) such that \( MAM[e^c][v] = p \). This can be implemented in the platform itself (i.e., the security monitor firmware in Keystone) or in a local vendor-provided enclave (i.e., similar to the Quoting Enclave in Intel® SGX).

In Eq. 1, \( sufficient_mem : PA \rightarrow E_{id} \rightarrow Bool \). \( sufficient_mem \) can be viewed as a predicate that determines whether there is enough memory to copy all data. \( sufficient_mem \) is modeled abstractly in the TAPC model to avoid an expensive computation to figure out whether there is enough memory.

Clone is only called from the untrusted OS because it requires the OS to allocate resources for the new enclave. Thus, if an enclave program needs to clone itself, it needs to collaborate with the OS to have it call Clone on its behalf. As the newly-created enclave is still an isolated enclave, the SRE property on both parent and child enclaves should hold even with a malicious OS.

4.4.2 Snapshot. Clone by itself still requires copying the entire virtual memory to ensure isolation. To enable memory sharing, Snapshot makes the caller enclave \( e \) to be an immutable image (Figure 5b). After calling Snapshot, \( e \) becomes a special type of enclave referred to as a snapshot enclave or the root snapshot of its descendants. \( e \) is no longer allowed to execute at this point because all of its memory becomes read- or execute-only. On the other hand, \( e \) can be cloned by Clone, where the descendants of \( e \) are allowed to read directly from the \( e \)'s shared data pages. Any writes from the descendants to physical addresses \( p \in PA \) owned by \( e \) (i.e., \( \sigma.o[p] = e \)) trigger copy-on-write (CoW). This scheme ensures that the descendant enclaves are still write-isolated from each other.

Like Clone, Snapshot has a success condition described in Eq. (2). The condition checks that the current executing enclave is valid \( valid(\sigma.e_{curr}) \) and active \( active(\sigma.e_{curr}, \sigma) \), \( e \) is not already a snapshot and the enclave cannot have a root snapshot in the current state \( \sigma \).

\[
valid(\sigma.e_{curr}) \land active(\sigma.e_{curr}, \sigma) \land \\
\neg \sigma.MIS[e] \land \neg valid(\sigma, MRS[\sigma.e_{curr}])
\]

If Snapshot is called successfully in a state that satisfies this condition, \( e \) is marked as a snapshot enclave. In the formal model, the metadata state \( MIS[e] \) is to true.

4.4.3 Clone after Snapshot. In order to make Clone work with Snapshot, Clone additionally increments \( e_p \)'s child count \( M^{CC}[e_p] \) by 1, and sets the root snapshot of \( e_c \) (i.e., \( M^{RS}[e_c] \)) to either \( e_p \) or \( e_p \)'s root snapshot \( M^{RS}[e_p] \) if it has one.

With the single-sharing model, arbitrarily nested calls of Clone should still keep only one shareable enclave. As shown in Fig. 6, there will be only one root snapshot \( e_1 \), whose memory is shared across all the descendants. This means that even though cloning can be arbitrarily nested, the maximum height of the tree representing the root snapshot to the child enclave is one.

To maintain the same functionality, the virtual address space of the parent and the child should be the same right after Clone. Thus, a descendant enclave memory will diverge from the shared memory when the descendant writes. Unfortunately, there is no better way than having Clone copy the diverged memory from the parent to the child. This is a limitation of Cerberus because the benefit of sharing memory will gradually vanish as the memory of the descendant diverges from the snapshot. However, we claim that Cerberus is very effective when the enclaves mostly write to a small part of the memory while sharing the rest. It is the programmer’s responsibility to optimize their program by choosing the right place to call Snapshot.

5 FORMAL GUARANTEES

To recap, one of our goals is to prove that our extension applied to an enclave platform does not weaken the high-level security property SRE. We accomplish this by re-proving SRE on TAPC. As
per Theorem 3.2 [57] (restated below as Theorem 5.2), it suffices to show that the triad of properties – secure measurement, integrity, and confidentiality – hold on TAP_C to prove SRE. In this section, we formally define SRE, the decomposition theorem, and each of the properties along with informal justification as to why they hold in the TAP_C model against the adversary described in Section §4. Each of these properties has been mechanically proven on the base TAP model [57] without Snapshot and Clone, and in our work, we extend these proofs to provide the same guarantees for the memory adversary on the extended TAP_C model. For brevity, we leave out some of the model implementation details and refer the reader to the GitHub repository for the proofs. We also provide a list of additional inductive invariants required to prove the properties in the TAP_C model at the end of this section.

While there are several flavors of non-interference properties, the following properties are based on the observational determinism (OD) [21, 66] definition of non-interference generalized for traces of concurrent systems. OD commonly shows up in several formalisms of confidentiality and integrity including the classic work on separability by Rushby [50] among other work [19, 26, 32, 57]. At a high level, OD states that if the initial states are low-equivalent and low inputs are the same, all states including intermediate states must also be low-equivalent (i.e., observationally deterministic functions of the low state/inputs). Whereas the classic non-interference property has an obligation [66] to prove termination and does not reason about intermediate states. We find OD to be more appropriate because we desire to show that every state of execution is observationally deterministic and indistinguishable.

### 5.1 Secure Remote Execution

**Definition 5.1 (Secure Remote Execution).** Let \( \pi = \{\sigma_0, \sigma_1, \ldots\} \) be a possibly unbounded-length sequence of platform states and \( \pi' = \{\sigma_0', \sigma_1', \ldots\} \) be the subsequence of \( \pi \) containing all of the enclave executing states (i.e. \( \forall i \in \mathbb{N}. \text{curr}(\pi_i') = e \)). Then the set \( \mathcal{E} = \{I_e(\sigma_0), E_e(\sigma_0'), O_e(\sigma_0'), \ldots | \text{init}(E_e(\sigma_0))\} \) describes all valid enclave execution traces and represents the expected semantics of enclave \( e \). A remote platform performs SRE of an enclave program \( e \) if any execution trace of \( e \) on the platform is contained within \( \mathcal{E} \). In addition, the platform must guarantee that a privileged software attacker can only observe a projection of the execution trace defined by \( \text{obs} \).

To prove SRE, the following theorem from prior work [57] allows us to decompose the proof as follows.

**Theorem 5.2.** An enclave platform that satisfies secure measurement, integrity, and confidentiality property for any enclave program also satisfies secure remote execution.

**Secure Measurement.** In any enclave platform, the user desires to know that the enclave program running remotely is in fact the program that it intends to run. In other words, the platform must be able to *measure* the enclave program to allow the user to detect any changes to the program prior to execution. The first part of the measurement property stated as Eq. (3) requires that the measurements \( \mu_e(1) \) and \( \mu_e(2) \) of any two enclaves \( e \) and \( e' \) in their initial states are the same if and only if the enclaves have identical initial enclave states. \( \mu \) is defined to be the measurement function that the user would use to check that their enclave \( e \) is untampered with in the remote platform.

\[
\forall \sigma_1, \sigma_2 \in \mathcal{S}. (\text{init}(E_e(\sigma_1)) \land \text{init}(E_e(\sigma_2))) \implies \mu_e(\sigma_1) = \mu_e(\sigma_2) \tag{3}
\]

The second part of measurement ensures that the enclave executes deterministically given an initial state. This is formalized as Eq. (4), which states that any two enclaves \( e_1 \) and \( e_2 \) starting with the same initial states, executing in lockstep and with the same inputs at each step, should have equal enclave states and outputs throughout the execution. Together, these properties help guarantee to the user that their enclave is untampered with.

\[
\forall \sigma_1, \sigma_2. \left( E_e(\sigma_1) = E_e(\sigma_2) \quad \land \quad \forall i \in \mathbb{N}. \text{curr}(\sigma_i') = e \implies (\text{curr}(\sigma_i') = e_i) \quad \land \right)
\]

\[
\left( \forall i \in \mathbb{N}. E_e(\sigma_i') = E_e(\sigma_i') \land O_e(\sigma_i') = O_e(\sigma_i') \right) \implies \right)
\]

With the addition of the Clone and Snapshot, the measurement of enclaves does not change for two reasons. Eq. (3) is satisfied because the measurement of a child is copied over from the parent, and has an equivalent state as the parent. In addition, because each enclave child executes in a way that is identical to the parent without Clone, the child enclave \( e_c \) is still deterministic up to the inputs \( I_{e_c}(\sigma) \).

**Integrity.** The second property, integrity, states that the enclave program’s execution cannot be affected by the adversary beyond the use of inputs \( I_e \) at each step and initial state \( E_e(\sigma_i') \), formalized as Eq. (5).

\[
\forall \sigma_i. \left( E_e(\sigma_i') = E_e(\sigma_i') \quad \land \quad \forall i \in \mathbb{N}. \text{curr}(\sigma_i') = e \implies (\text{curr}(\sigma_i') = e_i) \quad \land \right)
\]

\[
\left( \forall i \in \mathbb{N}. E_e(\sigma_i') = E_e(\sigma_i') \land O_e(\sigma_i') = O_e(\sigma_i') \right) \implies \right)
\]

Clone creates a logical copy of the enclave whose behavior matches the parent enclave had it not been cloned and thus clone does not affect the integrity of the enclave. Snapshot freezes the enclave state and thus does not affect the integrity vacuously because the state of \( e \) after calling snapshot does not change until its destruction.
Confidentiality. Lastly, the confidentiality property states that given the same enclave program with different secrets represented by \(e_1\) and \(e_2\) in traces \(\pi_1\) and \(\pi_2\) respectively, if the adversary starts in the initial state \(A_{e_1}(\pi_1[0])\) and the protected enclave(s) \(e_1\) (and \(e_2\)) is operated with a (potentially malicious) sequence of inputs \(I_\ell\), the adversary should not learn more than what’s provided by the observation function \(obs\) and hence its state \(A_{e_1}(\sigma)\) and \(A_{e_2}(\sigma)\) should be the same. The fourth line of Eq. (6) requires that any changes by the protected enclave \(e\) do not affect the observations made by the adversary in the next step. This is to avoid spurious counter-examples where secrets leak through obvious channels such as the enclave output which is a bug in the enclave program as explained in Section 4.3.5.

\[
\forall \pi_1, \pi_2. \left( A_{e_1}(\pi_1) = A_{e_2}(\pi_2) \land \forall i \in N. (\text{curr}(\pi_1) = \text{curr}(\pi_2) \land L_{\pi_1}(\sigma_i) = L_{\pi_2}(\sigma_i)) \land \forall i \in N. (\text{curr}(\pi_1) = \pi) \Rightarrow obs(\pi_i^{\pi_1}) = obs(\pi_i^{\pi_2}) \right)
\]

(6)

Snapshot alone clearly does not affect the confidentiality of the enclave. C1one on the other hand also does not affect confidentiality because it creates a logical duplicate of an enclave. Had the adversary been able to break the confidentiality of the child \(e_c\), it should have been able to break the confidentiality of the parent \(e_p\) because both should behave in the same way given the same sequence of input.

5.2 Cerberus Platform Invariants

We describe a few key additional platform inductive invariants that were required to prove the SRE property on TAPc. Although the following list is not exhaustive\(^5\), it provides a summary of the difference between the invariants in the base TAP model and the TAPc model and explains what precisely makes the other sharing models more difficult to verify. The invariants are typically over the two traces \(\pi_1\) and \(\pi_2\) in the properties previously mentioned. However, there are single-trace properties, and unless otherwise noted, it is assumed that single-trace properties defined over a single trace \(\pi\) hold for both traces \(\pi_1\) and \(\pi_2\) in the properties.

Memory Sharing. We first begin with the invariants related to the memory-sharing model. As explained earlier, allowing the sharing of memory weakens the constraint that memory is strictly isolated. This means that the memory readable and executable by an enclave can either belong to itself or its root snapshot. This is true for the entrypoints of the enclave and the mapped virtual addresses. These are described as Eq. (7) and Eq. (8) respectively.

Eq. (7) states that all enclaves have an entrypoint that belongs to \(e\) itself or its snapshot \(\pi^e.\lambda MRS[e]\).

\[
\forall e \in E_{id}. \forall i \in N. (\text{valid}(e) \Rightarrow (\pi^e.\lambda M^{EP}[e] = e \lor \pi^e.\lambda M^{EP}[e] = \pi^e.\lambda M^{RS}[e]))
\]

(7)

Eq. (8) states that every enclave \(e\) whose physical address \(p \in PA\) corresponding to virtual address \(v \in VA\) in the page table that is mapped \(\text{mapped}(\pi^e.\lambda ap_p[v])\)\(^6\) either belongs to \(e\) itself or the root snapshot \(\pi^e.\lambda M^{RS}[e]\).

To illustrate the potential complexity of the capped and arbitrary memory sharing models, the antecedent of this invariant would need to existentially quantify over all the possible snapshot enclaves that that own the memory as opposed to the current two (the enclave itself or its root snapshot). This would introduce an alternating quantifier[49] in the formula, making reasoning with SMT solvers difficult.

\[
\forall e \in E_{id}. \forall \ell \in VA. \forall i \in N. \left( (\text{valid}(e) \land \text{active}(\pi^e) \land \text{mapped}(\pi^e.\lambda ap_p[v])) \Rightarrow (\pi^e.\lambda M^{RS}[e]) \lor e \right) 
\]

(8)

Snapshots. The next invariants relate to snapshot enclaves. First, the root snapshot of an enclave is never itself as follows:

\[
\forall e \in E_{id}. \forall i \in N. (\text{valid}(e) \Rightarrow \pi^e.\lambda M^{RS}[e] \neq e)
\]

(9)

Snapshots also do not have root snapshots Eq. (11). This invariant reflects the property that the root snapshot to ancestor enclave relationship has a height of at most 1. This is stated as all enclaves that are snapshots have a root snapshot reference pointing to the invalid enclave ID \(i_{inv}\).

\[
\forall e \in E_{id}. \forall i \in N. (\text{valid}(e) \land \text{active}(\pi^e) \land \pi^e.\lambda M^{RS}[e] \Rightarrow \pi^e.\lambda M^{RS}[e] = i_{inv})
\]

(11)

Next, if an enclave has a root snapshot that is not invalid (i.e. \(\pi^e.\lambda M^{RS}[e] \neq i_{inv}\), then the root snapshot is a snapshot and the child count is positive. This is represented as Eq. (12).

\[
\forall e \in E_{id}. \forall i \in N. (\text{valid}(\pi^e) \land \text{active}(\pi^e) \land \pi^e.\lambda M^{RS}[e] \Rightarrow (\pi^e.\lambda M^{RS}[e]) \lor \pi^e.\lambda M^{CC}[\pi^e.\lambda M^{RS}[e] > 0])
\]

(12)

The last notable invariant says that the currently executing enclave cannot be a snapshot as described in Eq. (13).

\[
\forall i \in N. (\neg \pi^e.\lambda M^{EP}[\pi^e.\lambda \text{cur}])
\]

(13)

We conclude this section by noting that coming up with the exhaustive list of inductive invariants for TAPc took a majority of the verification effort.

\(^5\)We refer the reader to the formal models in the UCLID5 code for the complete list of inductive invariants in full detail.

\(^6\)mapped is a function that returns whether a physical address is mapped in enclave \(e\) and is equivalent to the valid function in the CCS’17 paper [57]
6 IMPLEMENTATION IN RISC-V KEYSTONE

To show its feasibility, we implement Cerberus on Keystone [36]. Keystone is an open-source framework for building enclave platforms on RISC-V processors. In Keystone, the platform operations $O_{\text{base}}$ are implemented in high-privileged firmware called security monitor. We implemented additional snapshot and Clone based on our specifications. All fields of the enclave metadata are stored within the security monitor memory. We extended the metadata with the variables corresponding to $M^{TS}$, $M^{CC}$, $M^{RS}$, and $M^{PAF}$. All implementations are available at https://github.com/chererus-ccs22/TAPC.git.

The implementation complies with the assumptions of the model described in §4.2. First, Keystone enclave operations are atomic operations, and it updates the system state only when the operation succeeds. Second, we implement a deterministic memory allocation algorithm for copy-on-write, by leveraging Keystone’s free memory module.

For memory isolation, Keystone uses a RISC-V feature called Physical Memory Protection (PMP) [12], which allows the platform to allocate a contiguous chunk of physical memory to each of the enclaves. When an enclave executes, the corresponding PMP region is activated by the security monitor. We implemented the weakened constraints (i.e., Eq. (8)) by activating the snapshot’s memory region when the platform context switches into the enclave.

In the model, the platform would need to handle the copy-on-write. In Keystone, an enclave can run with supervisor privilege, which allows the enclave to manage its own page table. This was very useful when we prototype this work because the platform does not need to understand the virtual memory mapping of the enclave. Letting the enclave handle its own write faults does not hurt the security because the permissions on physical addresses are still enforced by the platform. One implementation challenge was that the enclave handler itself would always trigger a write fault because the handler requires some writable stack to start execution. We were able to implement a stack-less page table traverse, which allows the enclave to remap the page triggering the write fault without invoking any memory writes. The final copy-on-write handler is similar to the on-demand fork [67].

7 EVALUATION

Our evaluation goals are to show the following:

- **Verification Results**: Our incremental verification approach enables fast formal reasoning on enclave platform modifications.
- **Start-up Latency**: The Cerberus interface can be used with process-creation system calls to reduce the start-up latency of enclaves
- **Computation Overhead**: Our copy-on-write implementation does not incur significant computation overhead.
- **Programmability**: Cerberus provides a programmable interface, which can be easily used to improve the end-to-end latency of server enclave programs.

Throughout the performance evaluation, we used SiFive’s FU540 [8] processor running at 1 GHz and an Azure DC1x_v3 VM instance with an Intel® Xeon® Platinum 8370C running at 2.4 GHz to run Keystone and SGX workloads respectively. Each experiment was averaged over 10 trials.

7.1 Verification Results

The TAPC model and proofs can be found at https://github.com/cererus-ccs22/TAPC.git.

**Porting TAP from Boogie to UCLID5.** One other contribution of this work includes the port of the original TAP model from Boogie [14] to UCLID5 [53]. UCLID5 is a verification toolkit designed to model transition systems modularly, which provides an advantage over the previous implementation written in the software-focused verification IR Boogie. We also find that UCLID5 is advantageous over other state-of-the-art tools [13, 24, 37, 58] because of modularity and because it provides flexibility in modeling systems both operationally and axiomatically. This effort took three person-months working approximately 25 hours a week to finish.

**Verifying TAPC.** The modeling and verification took roughly three person-months to write the extensions to the TAP model and verify using a scalable approach. We note that this time is substantially less than it would have taken to rebuild the model from scratch without an existing abstraction.

Fig. 7 shows the number of procedures $\#pn$, number of (uninterpreted) functions $\#fn$, number of annotations $\#an$ (which include pre- and post-conditions, loop invariants, and system invariants), the number of lines of code $\#ln$. The last column shows the verification time which includes the time it took UCLID5 to generate verification conditions and print them out in SMTLIB format and verify using Z3/CVC4. The time discrepancy between the original proofs [57] and the ones in this effort can be explained by the way we generate all the verification conditions as SMTLIB on disk before verifying as a way to use other SMT solvers. We also use UCLID5 instead of Boogie [14]. We note that the number of lines for Snapshot and Clone is 1110, which means only 489 lines were used to extend the existing TAP operations and platform model.

Despite the added complexity, each operation for each proof took only a few minutes to verify individually as shown in the last column of Fig. 7. This demonstrates that our incremental verification methodology is practical and consequently reduces the overall time to verify additional operations at a high level.

As evident by the results, we emphasize that the single-sharing model is practical to formally encode and verify. We also confirm that introducing invariants with alternating quantifiers and existential quantifiers in our models degraded the verification time and would likely do the same for alternative models. These attempts

![Figure 7: Model Statistics and Verification Times](image-url)

For one of the properties, we found that Z3 would get stuck but CVC4 didn’t.
We measure the computation overhead incurred by CoW invocation. This is because we are not copying any of the parent’s memory allocation. Thus, copy-on-write memory accesses are triggered during the computation depending on the memory usage.

7.2 Start-up Latency

To show the efficacy of Cerberus interface, we implement fork and clone system calls based on Cerberus. When the system calls are invoked in the enclave program, it calls Snapshot to create an immutable image and cooperates with the OS to clone two enclaves from the snapshot using Clone. We compare the latency of fork on two different platforms: SGX-based Graphene [18] (now Gramine Linux Foundation project [9]) and RISC-V Keystone [36] with Cerberus. Fig. 8 shows the program that calls fork after allocating memory with SIZE.

The baseline (Graphene-SGX) latency increases significantly as the allocation size increases (Figure 9). With a 400 MB buffer, it takes more than 6 seconds to complete. Also, each of the enclaves will take 400 MB of memory at all times, even when most of the content is identical until one of the enclaves writes. With Cerberus, the latency does not increase with respect to the allocation size. This is because we are not copying any of the parent’s memory including the page table. It only took 23 milliseconds to fork on average, with a standard deviation of 16 microseconds.

7.3 Computation Overhead

We measure the computation overhead incurred by CoW invocation. In order to see the overhead for various memory sizes and access patterns, we use RV8 [7] benchmark. RV8 consists of 8 simple applications that perform single-threaded computation. We omit bitint as we were not able to run it on the latest Keystone, because of a known bug on their side. Since RV8 does not use the fork system call, we have modified RV8 such that each of them forks before the computation begins. Note that all of the application starts with allocating a large buffer, so we inserted fork after the allocation. Thus, copy-on-write memory accesses are triggered during the computation depending on the memory usage.

As you can see in Figure 10, the average computation overhead of copy-on-write memory over Keystone was only 3.9%. The worst overhead was 19.0% incurred in qsort, which uses the largest memory (about 190 MiB). We argue that the benefit of cloning an enclave is small for such workloads that have a large buffer that is not shared across enclaves.

7.4 Programmability

To show the programmability of Cerberus interface, we showcase how server programs can leverage memory sharing to improve their end-to-end performance.

Although Snapshot or Clone are not directly related to fork or clone, their behavior maps well with Snapshot and Clone. For example, those system calls create a new process with exactly the same virtual memory, which can be mapped to Clone and optimized by Snapshot. Thus, we provided two co-authors with the modified fork and clone that use Cerberus interface and asked to make the server programs leverage memory sharing.

An author modified darkhttpd, a single-threaded web server, to fork processes to handle new HTTP requests inside the event loop. This allowed darkhttpd to serve multiple requests concurrently and continue listening for new requests. We measure the latency of an HTTP request using wget to fetch 0.5 MB of data. The resulting program incurs only a 2.1x slowdown over the native (non-enclave) execution, in contrast to a 33x slowdown in corresponding Intel SGX implementation (the exact same program ran with Graphene). The 2.1x overhead is mainly due to the slow I/O system calls, which is a well-known limitation of enclaves [36, 63].

Another author implements a simple read-only database server application using Sqlite3, which is a single-file SQL library that supports both in-memory and file databases. The resulting program serves each query with a fresh child created by fork. We measure the latency of 1,000 SELECT queries served by separate enclaves. The resulting program incurs a 36x slowdown over the native execution, compared to a 262x slowdown in corresponding Intel SGX implementation. SGX overhead is much worse than in Darkhttpd because there is more data to copy over (the entire in-memory database). The 36x slowdown is mainly due to limited concurrency in Keystone: since Keystone implements memory isolation with a limited number of PMP entries, it can support only up to 3-4 concurrent enclaves. This is not an inherent limitation of Cerberus.
Both authors did not have any difficulties in allowing enclaves to share a memory, because they were already familiar with the expected behavior of the system calls. However, they did not have any knowledge of the codebase of Darkhttpd nor Sqlite3 prior to the modification. Darkhttpd required modification of less than 30 out of 2,900 lines of code, which took less than 10 person-hours, and Sqlite3 consists of 103 lines of code, which took less than 20 person-hours. This shows that the Cerberus extension can be easily used to improve the end-to-end performance of server programs.

8 DISCUSSION

Low-Equivalent States. Our security model contains the notion of low states as in standard observational determinism type properties, even though they are not explicitly stated in section § 5. Instead, the low states are constrained to be equal in the antecedent of the implication of the properties. The traditional non-interference or observational determinism properties most closely resemble the confidentiality Eq. 6. In this property, the low states include the inputs \( L_1(i), L_2(\sigma) \) to the enclaves and platform operations controlled by the untrusted OS, all of the adversary controlled enclaves’ state \( E_2(\sigma) \), where \( i \) is not the protected enclave (i.e., \( e_1, e_2 \)), and the adversary state \( A_1(\sigma), A_2(\sigma) \). The idea is to prove that under the same sequence of adversarial controlled inputs, the adversary cannot differentiate between the two traces which have the same enclave (i.e., \( e_1, e_2 \)) with differing high data in memory. Similarly, the low states of the integrity property (Eq. 5) includes the unhindered inputs \( L_1(i), L_2(\sigma) \). However, instead of constraining the adversary inputs to be the same, we want to show that an enclave executes deterministically regardless of the state outside the enclave. As a result, the remaining low states include the protected enclaves \( E_1(\sigma), E_2(\sigma) \). Lastly, secure measurement can be viewed as a form of integrity proof and contains the same low states as the integrity property.

Performance Comparison with Previous Work. The evaluation section does not make direct performance comparisons with previous work such as PIE, which is based on x86. However, based on our calculations, Cerberus’s overhead is on par with PIE. For example, PIE incurs about 200ms startup latency on serverless workloads [38] whereas Cerberus incurs 23ms on clone system calls. We analyze that Cerberus is faster mainly because it leverages Keystone’s ability to quickly create an enclave with zero-filled memory without measurement, which SGX does not support. PIE’s copy-on-write introduces 0.7-32ms overhead on serverless function invocations taking 144-1153ms (the paper did not provide relative overhead over native execution), which can be roughly translated into less than a few percent of overhead, which is similar to Cerberus.

Verifying the Implementation. This paper does not verify the implementation of Cerberus in Keystone. Unsurprisingly, any discrepancy between the model and the implementation can make the implementation vulnerable. In particular, the enclave page table is abstracted as enclave metadata in TAP and TAPC, where it is actually a part of memory II. Cerberus in Keystone does not create any security holes because the page table management is trusted (the enclave manages it). However, this does not mean that we can apply the same argument to the other implementations. To formally verify the implementation, we can construct the model for Keystone implementation and do the refinement proof to show that the model refines the TAP model as described by Subramanyan et al. [57]. We leave this as future work.

In-Enclave Isolation. Instead of modifying the platform, a few approaches [10, 33, 42, 55] use in-enclave isolation to create multiple security domains within a single enclave. However, security guarantees of such solutions rely on the formal properties of not only the enclave platform, but also the additional techniques used for the isolation. For example, the security of software fault isolation (SFI) [61] based approaches [10, 55] depends on the correctness and robustness of the SFI techniques including the shared software implementation and the compiler, which should be formally reasoned together with the enclave platform. Thus, such approaches will result in a significant amount of verification efforts.

9 CONCLUSION

We showed how to formally reason about modifying the enclave platform to allow memory sharing. We introduce the single-sharing model, which can support secure and efficient memory sharing of enclaves. We also proposed two additional platform operations similar to existing process-creation system calls. In order to formally reason about the security properties of the modification, we defined a generic formal specification by incrementally extending an existing formal model. We showed that our incremental verification allowed us to quickly prove the security guarantees of the enclave platform. We also implemented our idea on Keystone open-source enclave platform and demonstrated that our approach can bring significant performance improvement to server enclaves.

10 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the Qualcomm Innovation Fellowship, by Amazon, by Intel under the Scalable Assurance program, and by RISE, ADEPT, and SLICE Lab industrial sponsors and affiliates. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

REFERENCES


