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
cathylu@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

ABSTRACT

Hardware enclaves rely on a disjoint memory model, which maps each physical address to an enclave to achieve strong memory isolation. However, this severely limits the performance and programmability of enclave programs. While some prior work proposes enclave memory sharing, it does not provide a formal model or verification of their designs. This paper presents Cerberus, a formal approach to secure and efficient enclave memory sharing. To reduce the burden of formal verification, we compare different sharing models and choose a simple yet powerful sharing model. Based on the sharing model, Cerberus extends an enclave platform such that enclave memory can be made immutable and shareable across multiple enclaves via additional operations. We use *incremental verification* starting with an existing formal model called the Trusted Abstract Platform (TAP). Using our extended TAP model, we formally verify that Cerberus does not break or weaken the security guarantees of the enclaves despite allowing memory sharing. More specifically, we prove the Secure Remote Execution (SRE) property on our formal model. Finally, the paper shows the feasibility of Cerberus by implementing it in an existing enclave platform, RISC-V Keystone.

KEYWORDS

enclaves, memory sharing, trusted execution environments; formal methods; formal verification; security; programming languages; computer architecture; Keystone; RISC-V; secure remote execution;

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.

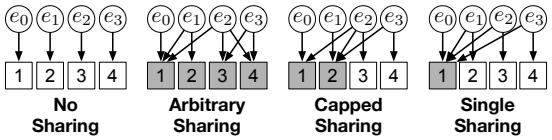

Figure 1: Memory sharing models with varying flexibility. Gray (and white) boxes indicate shareable (and non-shareable) physical memory regions, and circles indicate enclaves. An edge from an enclave to physical memory is an *access relation* stating that an enclave can access the memory it points to. The figure only depicts cases where the number of memory regions m is the same as that of enclaves n, but m can be greater than n in practice.

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 $O(mn)$.

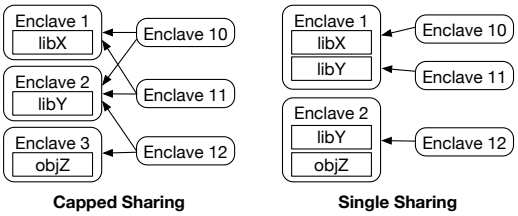

Figure 2: Difference between capped- and single-sharing models in use cases. `libX`, `libY`, and `objZ` are the large libraries or objects that enclaves want to share. Enclave 10 and 11 relies on `libX` and `libY`, while Enclave 12 relies on `libY` and `objZ`.

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 are 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

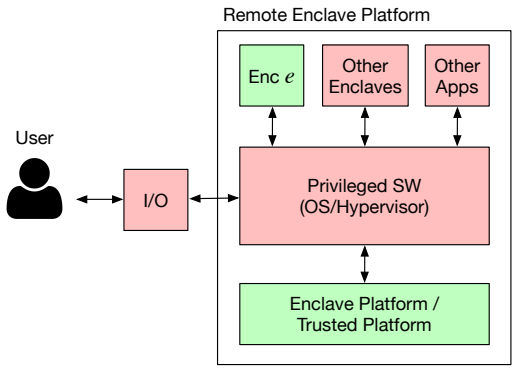

Figure 3: A user provisions their (protected) enclave e in the remote enclave platform isolated from untrusted software. Green/red boxes indicate trusted/untrusted components.

of an enclave, and `Clone` creates a logical duplication of an enclave. We make `Snapshot` 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 [57] 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 TAP_C.

4.1 Threat Model

Our extension follows the typical enclave threat model where the user's enclave program e 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 TAP_C Model Assumptions

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

- TAP_C 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 TAP$_C$ 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 $M = \langle S, I, \rightsquigarrow \rangle$ that is always in some state denoted by $\sigma \in S$. Alternatively, σ 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 $\rightsquigarrow \subset S \times S$. We write $(\sigma, \sigma') \in \rightsquigarrow$ to mean a valid transition of the platform from σ to σ'. An execution of the platform therefore emits a (possibly infinite) sequence of states $\pi = \langle \sigma^0, \sigma^1, ... \rangle$, where $(\sigma^i, \sigma^{i+1}) \in \rightsquigarrow$ for $i \in \mathbb{N}$. We write $\pi^i = \sigma^i$ interchangeably, but will usually write π^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 $init_e(\sigma) : S \rightarrow Bool$. We describe the set of variables V and enclave state $E_e(\sigma)$ for TAP$_C$ in the following Section §4.3.2.

4.3.2 TAP$_C$ State Variables. Each of the variables V in TAP$_C$ 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. $\Delta_{rf} : \mathbb{N} \rightarrow W$ is a register file that is a map† from the set of register indices (of natural numbers) $\mathbb{N}$ to the set of words W. $\Pi : PA \rightarrow W$ 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 a_{PA} and access permissions $a_{perm} : VA \rightarrow ACL$, where ACL is the set of read, write, and execute permissions. ACL can be defined as the product $VA \rightarrow Bool \times Bool \times Bool$, where $Bool \doteq \{true, false\}$ and the value of the map corresponds to the read, write, and execute permissions for a given virtual address index‡. $e_{curr} : \mathcal{E}_{id}$ represents the current enclave that is executing. $\mathcal{E}_{id} = \mathbb{N} \cup \{\mathcal{OS}\} \cup \{e_{inv}\}$ is the set of enclave IDs represented by natural numbers and a special identifier $\mathcal{OS}$ representing the untrusted operating system. We reserve the identifier e_{inv} 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 $valid(e_{id}) \doteq e_{id} \neq e_{inv} \wedge e_{id} \neq \mathcal{OS}$ 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 σ. 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.

State Var.	Type	Description
pc	VA	The program counter.
Δ_{rf}	$\mathbb{N} \rightarrow W$	General purpose registers.
Π	$PA \rightarrow W$	Physical memory.
a_{PA}	$VA \rightarrow PA$	Page table abstraction; virtual to physical address map.
a_{perm}	$VA \rightarrow ACL$	Page table abstraction; virtual to their permissions.
e_{curr}	$\mathcal{E}_{id}$	Current executing enclave ID (or $e_{curr} = \mathcal{OS}$ if the OS is executing).
o	$PA \rightarrow \mathcal{E}_{id}$	Map from physical addresses to the enclave that owns it.
$\mathcal{M}$	$\mathcal{E}_{id} \rightarrow \mathcal{E}_M$	Map of enclave IDs to enclave metadata. $\mathtt{emd}[\mathcal{OS}]$ stores a checkpoint of the OS.

Table 1: TAP$_C$ State Variables V.

Lastly, each enclave e has a set of enclave metadata $\mathcal{M}$, which is a record of variables described in Table 2. We abuse notation and write $\mathcal{M}^{pc}[e]$ to represent the program counter $\mathcal{M}^{pc}$ of e in the record stored in the metadata map $\mathcal{M}$. We use the enclave index operator $[\cdot]$ similarly for the other metadata fields defined in Table 2 to refer to a particular enclave's metadata. $\mathcal{M}^{EP}[e]$ is the entry point of the enclave that the enclave e starts in after the `Launch` and before `Enter`. $\mathcal{M}^{AM}_{PA}[e]$ is the virtual address map of the enclave program. $\mathcal{M}^{AM}_{perm}[e]$ is the map of address permissions for each virtual address. $\mathcal{M}^{EV}[e]$ is the map from virtual addresses to Boolean values representing whether an address is allocated to the enclave. $\mathcal{M}^{pc}[e]$ is the current program counter of the enclave. $\mathcal{M}^{regs}[e]$ is the saved register file of the enclave. $\mathcal{M}^{paused}[e]$ 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 TAP$_C$. We introduce the remaining four metadata variables required for Cerberus in Section §4.4, which are additional state variables in TAP$_C$ 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 $\mathcal{M}^{EP}[e]$, $\mathcal{M}^{AM}_{PA}[e]$, $\mathcal{M}^{AM}_{perm}[e]$, $\mathcal{M}^{EV}[e]$, $\mathcal{M}^{pc}[e]$, $\mathcal{M}^{regs}[e]$, and the projection of enclave memory $\lambda v \in VA.ITE(\mathcal{M}^{EV}[e][v], \Pi[\mathcal{M}^{AM}_{PA}[v]], \perp)$. In the last expression, $\lambda v \in VA.E$ 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. $\perp$ 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 $\mathcal{M}^{AM}_{PA}$.

*We write v: T to mean variable $v \in V$ has type T

†of type $L \rightarrow R$, where the index type is L and value type is R.

‡We use $\doteq$ to mean *by definition* to differentiate between the *equality* symbol $=$.

State Var.	Type	Description of each field
$\mathcal{M}^{EP}$	VA	Enclave entrypoint.
$\mathcal{M}^{AM}_{PA}$	$VA \rightarrow PA$	Enclave's virtual address map.
$\mathcal{M}^{AM}_{perm}$	$VA \rightarrow ACL$	Enclave's address permissions.
$\mathcal{M}^{EV}$	$VA \rightarrow Bool$	Set of private virtual addresses.
$\mathcal{M}^{pc}$	VA	Saved program counter.
$\mathcal{M}^{regs}$	$\mathbb{N} \rightarrow W$	Saved registers.
$\mathcal{M}^{paused}$	$Bool$	Whether enclave is paused.
$\mathcal{M}^{IS^\dagger}$	$Bool$	Whether the enclave is a snapshot.
$\mathcal{M}^{CC^\dagger}$	$\mathbb{N}$	Number of children enclaves.
$\mathcal{M}^{RS^\dagger}$	$\mathcal{E}_{id}$	Enclave's root snapshot.
$\mathcal{M}^{PAF^\dagger}$	$PA \rightarrow Bool$	Map of free physical addresses.

Table 2: Records of TAP$_C$ $\mathcal{E}_M$ enclave metadata. † indicates additional state variables added to support `Snapshot` & `Clone`.

4.3.3 ***Enclave Inputs and Outputs.*** Communication between an enclave e and external processes for a given state σ are controlled through e's inputs $I_e(\sigma)$ and its outputs $O_e(\sigma)$. $I_e(\sigma)$ 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 $\mathcal{O}$, in which the execution of an operation is deterministic up to its input $I_e(\sigma)$ and current state $E_e(\sigma)$. This means that given the same inputs $I_e(\sigma)$ 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 $\mathcal{O}_{base} \doteq \{$`Launch`, `Destroy`, `Enter`, `Exit`, `Pause`, `Resume`, `AdversaryExecute`$\}$. TAP$_C$ extends the base set with two additional operations: $\mathcal{O} \doteq \mathcal{O}_{base} \cup \{$`Snapshot`, `Clone`$\}$. We use the predicate $curr(\sigma) = e$ to indicate that enclave e, which may be adversary controlled, is executing at state σ and $curr(\sigma) = \mathcal{OS}$ 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 $\mathcal{A}$ 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 turn 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 `AdversaryExecute` in $\mathcal{O}_{base}$ 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.

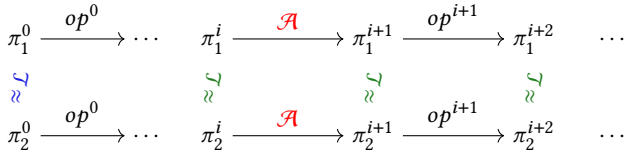

Figure 4: Illustrating the execution of two traces of the platform in the secure measurement, integrity and confidentiality proofs. Proof obligations for each property are checked as indicated by $\approx_{\mathcal{L}}$ and equal initial condition indicated as $\approx_{\mathcal{L}}$. op^i indicates enclave execution of an operation from $\mathcal{O}$ at step i and $\mathcal{A}$ indicates an adversary execution.

The semantics of the model allows the adversary to make these changes whenever it is executing. We allow all tampered states to be unconstrained in our models, which means they can take on any value. This type of adversary tamper function over-approximates what the threat model can change and is typically referred to as a havocing adversary [19, 57].

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 $obs_e(\sigma) \doteq O_e(\sigma) \doteq \lambda p \in PA.ITE(\sigma.o[p] \neq e, \sigma.\Pi[p], \perp)$. 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 † symbol. $\mathcal{M}^{IS}[e]$ is a Boolean valued variable indicating whether or not `Snapshot` has been called on the enclave e. $\mathcal{M}^{CC}[e]$ 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`. $\mathcal{M}^{RS}[e]$ is a reference to the root snapshot of e if one exists, and $\mathcal{M}^{PAF}[e]$ 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.

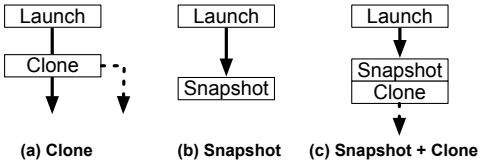

Figure 5: `Clone`, `Snapshot`, and `Clone`-after-`Snapshot`. The dotted arrow means an enclave newly-created by `Clone`.

`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 $e^p_{id} \in \mathcal{E}_{id}$ to clone, the enclave ID of the child enclave $e^c_{id} \in \mathcal{E}_{id}$ and a set of physical addresses assigned to the child enclave $x_p \subset PA$. The assigned physical addresses are marked as free (i.e., $\mathcal{M}^{PAF}[e_c][p] = true, \forall p \in x_p$), so that the parent's memory can be copied to them. The child enclave e_c with corresponding enclave ID e^c_{id} is used to create a clone of the parent e_p such that $E_{e_p}(\sigma) = E_{e_c}(\sigma)$. In other words, the virtual memory of both enclaves are equal. We write $E_{e_p}(\sigma_0)$ 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 $E_{e_p}(\sigma_0)$, 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 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 σ when `Clone` is called.

$$\begin{aligned} &\sigma.e_{curr} = \mathcal{OS} \wedge valid(e^p_{id}) \wedge active(e^p_{id}, \sigma) \quad \wedge \\ &valid(e^c_{id}) \wedge \neg active(e^c_{id}, \sigma) \quad \wedge \\ &e^c_{id} \neq e^p_{id} \quad \wedge \\ &\forall p \in PA.p \in x_p \Rightarrow \sigma.o[p] = \mathcal{OS} \quad \wedge \\ &sufficient\_mem(\sigma.o) \end{aligned} \tag{1}$$

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 $\Pi[\mathcal{M}^{AM}_{PA}[e_p][v]]$ to $\Pi[p]$, and update the page table of e_c such that $\mathcal{M}^{AM}_{PA}[e_c][v] = p$. 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, $sufficient\_mem : PA \rightarrow \mathcal{E}_{id} \rightarrow Bool$, *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 TAP_C 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 σ.

$$\begin{aligned} &valid(\sigma.e_{curr}) \wedge active(\sigma.e_{curr}, \sigma) \quad \wedge \\ &\neg\sigma.\mathcal{M}^{IS}[e] \wedge \neg valid(\sigma.\mathcal{M}^{RS}[\sigma.e_{curr}]) \end{aligned} \tag{2}$$

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 $\mathcal{M}^{IS}[e]$ 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 $\mathcal{M}^{CC}[e_p]$ by 1, and sets the root snapshot of e_c (i.e., $\mathcal{M}^{RS}[e_c]$) to either e_p or e_p's root snapshot $\mathcal{M}^{RS}[e_p]$ 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 reproving SRE on TAP_C. As

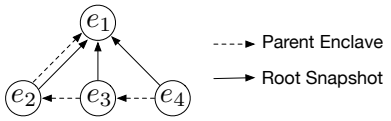

Figure 6: Parent-child relationship and root snapshot-child relationship of four enclaves in Cerberus. Enclave e_1 is a snapshot and the parent enclave of e_2, which is the parent of e_3, which is the parent of e_4. Despite the nested parent relationship, the root snapshot of e_2, e_3, and e_4 are e_1.

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 $\pi = \{\sigma_0, \sigma_1, ...\}$ be a possibly unbounded-length sequence of platform states and $\pi' = \{\sigma'_0, \sigma'_1, ...\}$ be the subsequence of π containing all of the enclave executing states (i.e. $\forall i \in \mathbb{N}.curr(\sigma'_i) = e$). Then the set $[[e]] = \{\langle I_e(\sigma'_0), E_e(\sigma'_0), O_e(\sigma'_0)\rangle, ... | init_e(E_e(\sigma_0))\}$ 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 $[[e]]$. In addition, the platform must guarantee that a privileged software attacker can only observe a projection of the execution trace defined by *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_1 and e_2 in their initial states are the same if and only if the enclaves have identical initial enclave states. μ 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.

$$\forall \sigma_1, \sigma_2 \in S.\big(init(E_{e_1}(\sigma_1)) \wedge init(E_{e_2}(\sigma_2))\big) \Rightarrow \big(\mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2)\big) \tag{3}$$

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.

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&E_{e_1}(\pi_1^0) = E_{e_2}(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e_1) \iff (curr(\pi_2^i) = e_2) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e_1) \Rightarrow I_{e_1}(\pi_1^i) = I_{e_2}(\pi_2^i)\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.E_{e_1}(\pi_1^i) = E_{e_2}(\pi_2^i) \wedge O_{e_1}(\pi_1^i) = O_{e_2}(\pi_2^i)\Big)\end{aligned} \tag{4}$$

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 $E_e(\pi_1^0)$, formalized as Eq. (5).

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&E_e(\pi_1^0) = E_e(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \iff (curr(\pi_2^i) = e) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \Rightarrow I_e(\pi_1^i) = I_e(\pi_2^i)\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.E_e(\pi_1^i) = E_e(\pi_2^i) \wedge O_e(\pi_1^i) = O_e(\pi_2^i)\Big)\end{aligned} \tag{5}$$

`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 π_1 and π_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_{e_1}, the adversary should not learn more than what's provided by the observation function *obs* and hence its state $A_{e_1}(\sigma)$ and $A_{e_1}(\sigma)$ 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.

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&A_{e_1}(\pi_1^0) = A_{e_2}(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = curr(\pi_2^i) \wedge I_{e_1}(\sigma_1^i) = I_{e_2}(\sigma_2^i)) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \Rightarrow obs(\pi_1^{i+1}) = obs(\pi_2^{i+1})\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.A_{e_1}(\pi_1^i) = A_{e_2}(\pi_2^i)\Big)\end{aligned} \tag{6}$$

`Snapshot` alone clearly does not affect the confidentiality of the enclave. `Clone` 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 TAP_C. Although the following list is not exhaustive[§], it provides a summary of the difference between the invariants in the base TAP model and the TAP_C model and explains what precisely makes the other sharing models more difficult to verify. The invariants are typically over the two traces π_1 and π_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 π hold for both traces π_1 and π_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 $\pi^i.\mathcal{M}^{RS}[e]$.

$$\begin{aligned}\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}.\Big(&valid(e) \quad \Rightarrow \\ &(\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = e \quad \vee \\ &\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = \pi^i.\mathcal{M}^{RS}[e])\Big)\end{aligned} \tag{7}$$

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 $mapped_e(\pi[i].a_{PA}[v])$[¶] either belongs to e itself or the root snapshot $\pi^i.\mathcal{M}^{RS}[e]$.

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.

$$\begin{aligned}\forall e \in &\mathcal{E}_{id}, v \in VA, \forall i \in \mathbb{N}. \\ &\Big((valid(e) \wedge active(e, \pi^i) \wedge mapped_e(\pi^i.a_{PA}[v])) \Rightarrow \\ &(\pi^i.o[\pi^i.a_{PA}[v]] = e \quad \vee \\ &\pi^i.o[\pi^i.a_{PA}[v]] = \pi^i.\mathcal{M}^{RS}[e])\Big)\end{aligned} \tag{8}$$

Lastly, a memory that is marked free for an enclave e is owned by that enclave itself, represented by Eq. (9).

$$\forall e \in \mathcal{E}_{id}, p \in PA, \forall i \in \mathbb{N}.\Big(\pi^i.\mathcal{M}^{PAF}[e][p] \Rightarrow \pi^i.o[p] = e\Big) \tag{9}$$

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

$$\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}.(valid(e) \Rightarrow \pi^i.\mathcal{M}^{RS}[e] \neq e) \tag{10}$$

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 e_{inv}.

$$\begin{aligned}\forall e \in &\mathcal{E}_{id}, \forall i \in \mathbb{N}. \\ &\Big((valid(e) \wedge active(e, \pi^i) \wedge \pi^i.\mathcal{M}^{IS}[e]) \Rightarrow \\ &\pi^i.\mathcal{M}^{RS}[e] = e_{inv}\Big)\end{aligned} \tag{11}$$

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

$$\begin{aligned}\forall e \in &\mathcal{E}_{id}, \forall i \in \mathbb{N}. \\ &\Big((valid(\pi^i.\mathcal{M}^{RS}[e]) \wedge active(\pi^i.\mathcal{M}^{RS}[e], \pi^i)) \Rightarrow \\ &(\pi^i.\mathcal{M}^{IS}[\pi^i.\mathcal{M}^{RS}[e]] \quad \wedge \\ &\pi^i.\mathcal{M}^{CC}[\pi^i.\mathcal{M}^{RS}[e]] > 0)\Big)\end{aligned} \tag{12}$$

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

$$\forall i \in \mathbb{N}.(\neg \pi^i.\mathcal{M}^{IS}[\pi^i.e_{curr}]) \tag{13}$$

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

[§]We refer the reader to the formal models in the UCLID5 code for the complete list of inductive invariants in full detail.

[¶]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_{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 $\mathcal{M}^{IS}$, $\mathcal{M}^{CC}$, $\mathcal{M}^{RS}$, and $\mathcal{M}^{PAF}$. All implementations are available at https://github.com/cerberus-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 DC1s_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.

Model/Proof	Size				Verif.
	#pr	#fn	#an	#ln	Time (s)
TAP Models					
`TAP`	43	14	225	2100	140
`Integrity`	2	0	52	525	285
`Mem. Conf`	3	0	44	838	342
TAP$_C$ Models					
`TAP`	45	16	466	3689	1380
`Integrity`	2	0	109	937	934
`Mem. Conf`	3	0	119	1307	944

Figure 7: Model Statistics and Verification Times

7.1 Verification Results

The TAP$_C$ model and proofs can be found at https://github.com/cerberus-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 TAP$_C$. 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 SMTLIB2.0 and verify them 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

[‖]For one of the properties, we found that Z3 would get stuck but CVC4 didn't.

```c
int main() {
  char* buf = malloc(SIZE);
  clock_t start = clock();
  if (!fork()) {clock_t end = clock();} // child
  else { return; }                      // parent
}
```

Figure 8: C code to measure `fork` latency

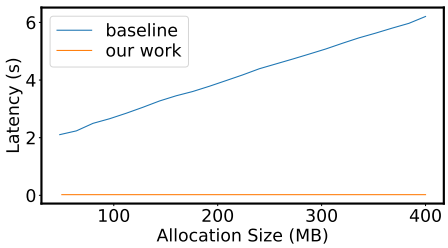

Figure 9: The latency of `fork` with respect to the size of the allocated memory.

heavily influenced our decisions and we strongly encourage the use of our sharing model.

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.

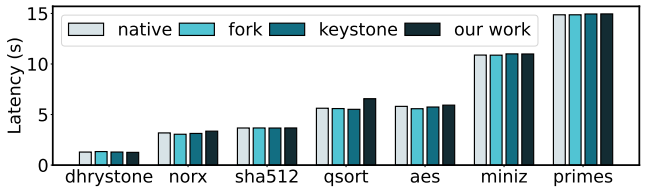

Figure 10: Computation Overhead on RV8. native: native execution of the original RV8, fork: native execution of the modified RV8 with `fork`, keystone: enclave execution of the original RV8, and our work: enclave execution of the modified RV8 with Cerberus.

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 MiBs). 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 $I_{e_1}(\sigma), I_{e_2}(\sigma)$ to the enclaves and platform operations controlled by the untrusted OS, all of the adversary controlled enclaves' state $E_e(\sigma)$, where e is not the protected enclave (i.e., e_1, e_2), and the adversary state $A_{e_1}(\sigma), A_{e_2}(\sigma)$. 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 untrusted inputs $I_{e_1}(\sigma), I_{e_2}(\sigma)$. 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 $E_{e_1}(\sigma), E_{e_2}(\sigma)$. 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 TAP_C, where it is actually a part of memory Π. 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

[1] [n.d.]. AWS SageMaker. https://aws.amazon.com/pm/sagemaker.
[2] [n.d.]. Fission.io. https://fission.io/.
[3] [n.d.]. Huggingface. https://huggingface.co/.
[4] [n.d.]. OpenFaaS. https://www.openfaas.com/.
[5] [n.d.]. Ray Serve. https://www.ray.io/ray-serve.
[6] 2013. ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone.
[7] 2017. RV8 Benchmark. https://github.com/michaeljclark/rv8-bench.
[8] 2020. HiFive Unleashed. https://www.sifive.com/boards/hifive-unleashed.
[9] 2021. Gramine. https://github.com/gramineproject/gramine.
[10] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. 2021. Chancel: efficient multi-client isolation under adversarial programs. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.
[11] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[12] Krste Asanović Andrew Waterman. 2021. The RISC-V Instruction Set Manual Volume II: Privileged Architecture. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[13] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22) (Lecture Notes in Computer Science)*. Springer. http://www.cs.stanford.edu/~barrett/pubs/BBB+22.pdf

[14] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLIne, Bart Jacobs, and Rustan Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO 2005* (fmco 2005 ed.). Springer Berlin Heidelberg. https://www.microsoft.com/en-us/research/publication/boogie-a-modular-reusable-verifier-for-object-oriented-programs/

[15] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. 2007. *The Undecidability of First-Order Logic* (5 ed.). Cambridge University Press, 126–136. https://doi.org/10.1017/CBO9780511804076.012

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. of USENIX Security Symposium*.

[17] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[18] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proc. of USENIX Anual Technical Conference (ATC)*.

[19] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 288–303. https://doi.org/10.1109/CSF.2019.00027

[20] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. 2017. The Cogent Case for Property-Based Testing. In *Proc. of Workshop on Programming Languages and Operating Systems (PLOS)* (Shanghai, China). 7 pages. https://doi.org/10.1145/3144555.3144556

[21] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2008.7

[22] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086.

[23] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of USENIX Security Symposium*.

[24] Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[25] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proc. of Symposium on Operating Systems Principles (SOSP)*.

[26] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[27] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. 2021. Towards Formal Verification of State Continuity for Enclave Programs. In *Proc. of USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity21/presentation/jangid

[28] David Kaplan. 2017. AMD SEV-ES. http://support.amd.com/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf.

[29] David Kaplan, Jeremy Powell, and Tom Woller. 2016. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. of Symposium on Operating Systems Principles (SOSP)*.

[31] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[32] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. 2022. Expressing Information Flow Properties. *Foundations and Trends in Privacy and Security* 3, 1 (2022), 1–102. https://doi.org/10.1561/3300000008

[33] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proc. of the Twelfth European Conference on Computer Systems (EuroSys)*.

[34] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. 2019. Sanctorum: A lightweight security monitor for secure enclaves. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

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

[36] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. of European Conference on Computer Systems (EuroSys)*.

[37] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. of Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (Dakar, Senegal). 23 pages.

[38] Mingyu Li, Yubin Xia, and Haibo Chen. 2021. Confidential Serverless Made Efficient with Plug-in Enclaves. In *Proc. of International Symposium on Computer Architecture (ISCA)*.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. of USENIX Security Symposium*.

[40] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).

[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP*.

[42] Marcela S Melara, Michael J Freedman, and Mic Bowman. 2019. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245* (2019).

[43] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*.

[44] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Serval: Scaling Symbolic Evaluation for Automated Verification of Systems Code. In *Proc. of Symposium on Operating Systems Principles (SOSP)*.

[45] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proc. of USENIX Security Symposium*.

[46] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[47] Nelly Porter and Jason Garms. 2019. 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.

[48] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB - A Secure Database using SGX. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[49] C. R. Reddy and D. W. Loveland. 1978. Presburger Arithmetic with Bounded Quantifier Alternation. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '78)*. Association for Computing Machinery, New York, NY, USA, 320–325. https://doi.org/10.1145/800133.804361

[50] John Rushby. 1982. Proof of Separability: A Verification Technique for a Class of Security Kernels. In *Proc. 5th International Symposium on Programming (Lecture Notes in Computer Science, Vol. 137)*. Springer-Verlag, Turin, Italy, 352–367.

[51] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. *IEEE Access* 9 (2021), 83067–83079. https://doi.org/10.1109/ACCESS.2021.3087421

[52] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proc. of Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[53] Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating Modeling, Verification, Synthesis and Learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 1–10. https://doi.org/10.1109/MEMCOD.2018.8556946

[54] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proc. of ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*.

[55] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[56] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[57] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security*

(CCS).

[58] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proc. of ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming; Software* (Indianapolis, Indiana, USA). 18 pages. https://doi.org/10.1145/2509578.2509586

[59] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[60] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[61] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proc. of Symposium on Operating Systems Principles (SOSP)*.

[62] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[63] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *ISCA*.

[64] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[65] Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. 2022. Elasticlave: An Efficient Memory Model for Enclaves. In *Proc. of USENIX Security Symposium*.

[66] S. Zdancewic and A.C. Myers. 2003. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 29–43. https://doi.org/10.1109/CSFW.2003.1212703

[67] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proc. of the Sixteenth European Conference on Computer Systems (EuroSys)*.