ELASTICLAVE: An Efficient Memory Model for Enclaves

Jason Zhijingcheng Yu[†] Shweta Shinde[*,‡] Trevor E. Carlson[†] Prateek Saxena[†]

[†]*National University of Singapore* [‡]*ETH Zürich*

Abstract

Trusted execution environments (TEEs) isolate user-space applications into secure enclaves without trusting the OS. Existing TEE memory models are rigid—they do not allow an enclave to share memory with other enclaves. This lack of essential functionality breaks compatibility with several constructs such as shared memory, pipes, and fast mutexes that are frequently required in data intensive use-cases. In this work, we present ELASTICLAVE, a new TEE memory model which allows sharing. ELASTICLAVE strikes a balance between security and flexibility in managing access permissions. Our implementation of ELASTICLAVE on RISC-V achieves performance overheads of about 10% compared to native (non-TEE) execution for data sharing workloads. In contrast, a similarly secure implementation on a rigid TEE design incurs 1-2 orders of magnitude overheads for these workloads. Thus, ELASTICLAVE enables cross-enclave data sharing with much better performance.

1 Introduction

Isolation, commonly through the use of the process abstraction provided by an OS, is a cornerstone for security. It allows us to isolate and limit software compromises to one *fault domain* within an application and is the basis for applying the design principle of privilege separation. In the last few years, user-level enclaves have become available in commodity CPUs that support TEEs. Conceptually, enclaves are in sharp contrast to processes in that they do not assume a trusted OS, promising a drastic reduction in the trusted computing base (TCB) of a fault domain. The enclaved TEE design is of fundamental importance to security because they offer a new isolation primitive for software.

We revisit one of the key abstractions provided by enclaved TEEs—their memory model. Several existing TEEs, including SGX [7, 19, 21, 24, 27, 31, 39, 40], follow what we call the *spatial isolation model*. In this model, the virtual memory of the enclave is statically divided into two types: *public* and *private* memory regions. These types are fixed throughout the lifetime of a region. When applied to enclaves, the spatial isolation model is a simple but rigid model that is insufficient for memory sharing. Its underlying principle breaks compatibility with the most basic of data sharing patterns where the enclave needs to compute privately on some data before making it public or sharing it externally. If we want to support memory sharing between enclaves on spatially isolated memory, we require additional *trusted coordinator* enclaves together with cryptographic secure message passing channels. Without these additional mechanisms, achieving secure shared memory is fraught with challenges in managing ownership and access rights of the shared region, as attack vectors like permission re-delegation [26], confused deputy [28], malicious races [14], and TOCTOU attacks [25] have shown.

In this work, we revisit the spatial isolation memory model adopted by modern TEEs. We propose a new memory model called ELASTICLAVE which allows enclaves to share memory across enclaves and with the OS, with more flexible permissions than in spatial isolation. While allowing flexibility, ELASTICLAVE does not make any simplistic security assumptions or degrade its security guarantees over the spatial isolation model. We view enclaves as a fundamental abstraction for partitioning applications in this work, and therefore assume that enclaves do *not* trust each other and can become compromised during their lifetime. ELASTICLAVE strikes a balance between flexibility and safety. Each enclave in the partitioned application has its separate view of the memory permissions. These permission views are *asymmetric* and *dynamically* adjustable, and can give rights of *exclusive* access to an enclave. Thus, with ELASTICLAVE, enclaves can selectively share memory while protecting it from faulty enclaves. Unlike approaches to managing access rights and delegations in non-TEE settings (e.g., in languages or OS design)—such as ownership transfer [38], static permission systems [6, 49], or fully dynamic permission delegation [18, 62, 63]—ELASTICLAVE *simplifies* security decision-making for coordinating enclaves and TEE implementations.

[*]Part of the research was done while at University of California, Berkeley.

It is possible to implement ELASTICLAVE on spatially isolated TEEs, as mentioned earlier, by using trusted coordinators and cryptographically secured public memory. The trusted coordinator design, in fact, is already used in many existing frameworks for Intel SGX [8, 12, 51, 52, 61]. However, in real applications, the performance overheads due to this design can be prohibitively high compared to those in native execution (in normal processes without enclaves). For example, emerging enclave-based function-as-a-service (FaaS) deployments that create frequent read-only copies of the entire enclave code exhibit the worst-case performance of this design [5]. In this paper, we show that ELASTICLAVE can be implemented in next-generation TEEs with much better performance, reducing the overheads to within *about* 10% *of the native*, for data workloads that otherwise incur 1–2 *orders of magnitude* overheads on a TEE that adopts the spatial isolation model. ELASTICLAVE completely eliminates the need for expensive data copy and encryption-decryption operations which are necessary for security in the spatial model but not in ELASTICLAVE.

We implement our design on RISC-V [9, 47] and evaluate its performance and hardware complexity impact of ELASTICLAVE using a cycle-accurate RTL simulator [32] on synthetic as well as real-world workloads. The benchmarking results confirm the following claims:

- *TCB / Hardware Implementation Simplicity.* The prototype implementation of ELASTICLAVE only includes a privileged security monitor that spans 7,000 LoC.
- *Data-independent Overheads on RISC-V.* The performance overheads are affected primarily by the number of enclave-to-enclave context switches. In contrast to spatial isolation, the overheads are *independent* of the size of shared data in a region. Further, the increased hardware register pressure due to ELASTICLAVE does *not* affect the critical path delay, i.e., the latency of the address translation, for all the synthesized RISC-V core configurations tested.

Contributions. This paper proposes a new memory model for enclaved TEEs called ELASTICLAVE. We offer a prototype implementation on RISC-V and show that ELASTICLAVE results in significantly better performance than the spatial isolation model with a modest hardware complexity impact.

2 Problem

TEEs provide the abstraction of enclaves to isolate components of an application which run with user-level privileges. We want to design an efficient memory model for TEEs that support memory sharing between enclaves. The TEE implementation is trusted and assumed to be bug-free. In our setup, a security-sensitive application is partitioned into multiple components. Each component runs in a separate enclave created by the TEE, which serves as a basic isolation primitive. We assume that enclaves are *mutually distrusting*, since an adversary may compromise them during their execution, e.g., due to software exploits. This assumption is of fundamental importance, as it captures the essence of why an application is partitioned to begin with.

Most TEEs available on commodity processors follow a memory model which we call the *spatial isolation* model [16, 19, 19, 21, 35, 40, 50, 57]. In this model, each enclave has two different types of non-overlapping virtual memory regions: (a) *Private memory* is exclusive to the enclave itself and inaccessible to the OS and all the other enclaves running on the system; (b) *public memory* is fully accessible to the enclave and the untrusted OS, which may share it with other enclaves.

The spatial isolation model embodies the principle of dividing trust in an *all-or-none* manner [52]. For each enclave, any entity is fully trusted to access the public memory, whereas the private memory is accessible only to the enclave itself. This principle is in sharp contrast to any form of memory sharing where an enclave needs to exchange data with the outside world, including with other enclaves. Although shared memory is not directly supported in the spatial isolation model, it can be *emulated* with a secure communication channel over untrusted memory. This style of emulation is commonly adopted in existing frameworks, which for example, work on Intel SGX [8, 52, 61]. This design requires multiple data copies and encryption-decryption operations (see Figure 3) to secure each enclave against attacks from the OS and other enclaves. Table 1 summarizes the overheads for three textbook data patterns and shows that spatial isolation can cause 1–2 orders of magnitude performance slowdown. Section 3.2 explains why these performance overheads are fundamental to spatial isolation.

2.1 Security Challenges in Memory Sharing

We discuss the potential design choices to enable secure memory sharing for enclaves. As an example, we consider a client-server setup where a frontend client enclave uses an in-memory key-value store (e.g., Memcached [41]) as a backend server. The server enclave e_1 owns the region r and wants to share it with the client enclave e_2. This way, each enclave can read the data written by the other, perform private computation on it, and write back the result to the shared memory. Concretely, the expected sequence of operations on r is: e_1 writes, e_2 reads, e_2 writes, and then e_1 reads. Consider the following threat: After sharing the region, e_1 and e_2 can become compromised at any point during the operations. We refer to such compromised enclaves as *faulty* and assume that they can behave arbitrarily.

Option 1: Static Permissions. The simplest design is to allow both enclaves to simultaneously access the region. Before execution, the enclaves are *statically* granted the maximum

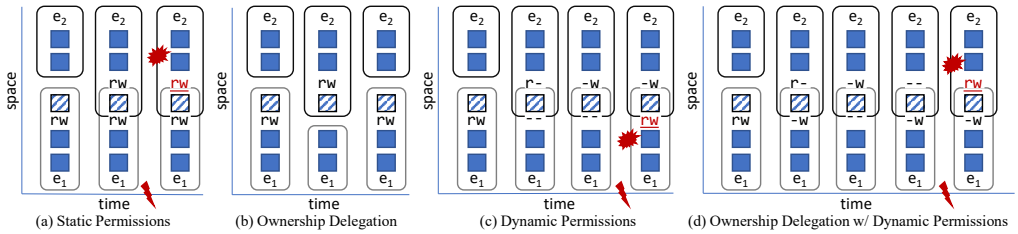


Figure 1

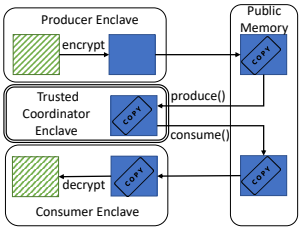


Figure 2

Figure 1: The striped boxes are regions of interest for sharing purposes. The labels around the striped boxes denote the the permissions of the corresponding enclaves. The thunderbolts indicate when an enclave become faulty. The underlined permission labels denote misuse. Figure 2: Producer-consumer pattern implemented with a trusted coordinator on spatial ShMem baseline. The double-lined box denotes the trusted coordinator enclave. The striped boxes represent unencrypted data.

required permissions for the region. This allows the enclaves to make their security decisions based on the static permission view before they start sharing. In our example, both e_1 and e_2 can be granted read-write permissions. Enclaves can perform in-place changes, thus removing the need for memory copies. However, if e_2 becomes faulty at any point, it can observe or tamper with the intermediate state of r when e_1 is performing its own operations. Figure 1(a) demonstrates this with detailed space-time sequences. In this design, neither e_1 nor e_2 can change the permissions of themselves or other enclaves.

Option 2: Ownership Delegation. An alternative is to dynamically transfer the ownership of the region. This way, the enclave that wants to perform an operation gains exclusive ownership to ensure no other enclave can interfere (i.e., read or tamper with intermediate states). After completion, the enclave can transfer the ownership to the next accessor. In our example, e_1 delegates the ownership to e_2, e_2 performs read-write, and transfers the ownership to e_1 (see Figure 1(b)). The exclusive access simplifies security reasoning. However, the faulty enclave can pass the access to other faulty enclaves while refusing to return control to the designated enclave. In addition, the design does not allow concurrent access, only one enclave can perform operations at any given time. Thus, the design restricts enclave functionality, such as that needed for thread synchronization.

Option 3: Dynamic Permissions. One way to support functionality while ensuring security is to allow owner enclaves to grant and revoke permissions at runtime. In our example, owner e_1 performs its write. Then it revokes its own permissions and grants permissions to e_2. As the owner of the memory region, e_1 can revoke the permissions of e_2, say, after e_2 has finished its read-write operations. Such dynamic changes allow fine-grained control and reduce the attack window from faulty enclaves. The design allows enclaves to have simultaneous access if they both have the permissions. Although this design may seem both efficient and secure, it has a subtle caveat. The owner enclave can dynamically change permissions and the non-owner enclaves do not have a fixed permission view. For instance, although e_2 can check if e_1 has revoked its permissions before beginning its own write operation, when e_2 is in the middle of a write operation, e_1 can regain its read/write permissions and interfere with the operations of e_2, as shown in Figure 1(c). In this way, faulty owner enclaves can leverage the inconsistent permission view to launch malicious TOCTOU attacks.

Option 4: Ownership Delegation with Dynamic Permissions. Naïvely combining the above mechanisms is not a safe solution. As a middle ground, consider the design that allows enclaves to make dynamic permission changes (as in Option 3) coupled with ownership delegation. This way, non-owner enclaves can gain temporary ownership to reliably control permissions without getting sidelined by the owner enclave. In our example, e_2 gains temporary ownership to perform read or write operations. Since e_1 is no longer an owner, it cannot change its own permissions to launch malicious TOCTOU attacks. This dynamicity brings complexity—the current owner can still change permissions at any time, making security decision-making difficult. In case of simultaneous access, only one enclave can be an owner, say, e_2. The other enclave, e_1, can get the current view and make a security decision to initiate an operation. However, the temporary owner e_2 can change permissions in the middle of such an operation to attack e_1, as shown in Figure 1(d). Thus, the added complexity does not improve security against malicious races.

2.2 Problem Formulation

None of the design options outlined above allows secure memory sharing while supporting application functionality at the same time. They are either too restrictive or too permissive. To this end, we ask the following research question: *Does there exist a minimal relaxation of the spatial model which allows memory sharing while retaining its security guarantees?*

Threat Model. We assume that the OS can be arbitrarily malicious. The target application is partitioned into enclaves, which share one or more regions of memory. Any subset of enclaves can become compromised, i.e., faulty during the execution.

We desire the following two security properties.

Property 1: Bounded Escalation. If an owner does not explicitly authorize an enclave e to access a region r with a said permission, e will not be able to make that access.

Property 2: Enforceable Serialization of Non-faulty Enclaves. If the application has a pre-determined sequence in which non-faulty enclaves should serialize their accesses, the accesses will either obey the sequence or be aborted. Specifically, consider any desired sequence of memory accesses $a_1, a_2, \ldots, a_n$ on a shared region and assume that all enclaves performing these accesses are non-faulty. The application author should be guaranteed that the accesses will follow the desired sequence, even in the presence of other faulty enclaves, or can be aborted safely. For instance, this property is sufficient to implement memory consistency models such as total memory ordering [44, 54] and sequential consistency [33, 55].

If a system satisfies the above two properties, it can securely execute the example (Figure 1). Property 1 eliminates re-delegation and escalation attacks by limiting permissions. Property 2 eliminates malicious races and TOCTOU attacks by enforcing a safe access sequence. At the same time, the properties admit concurrent accesses and do not encumber functionality. We will present a baseline design in Section 3.1 which achieves the above goal but with high performance overheads. Our main contribution is a novel design called ELASTICLAVE, which offers significantly better performance, and is explained in Section 4.2.

Assumptions. We assume that the TEE implementation, including both the hardware and the software components, is bug-free. Denial-of-service (DoS) attacks on shared memory are out of scope, since the OS needs to be able to reclaim memory at any time for legitimate management reasons. We assume that the developer correctly expresses the high-level application-specific security properties as low-level ELASTICLAVE permissions. ELASTICLAVE is not designed to address the scenario where an enclave wants to share sensitive data but cannot decide if the receiver enclave is faulty or not. In addition, ELASTICLAVE cannot protect an enclave that intentionally shares its data with a malicious enclave. Note that this limitation holds true for any form of sharing, including spatial isolation. Furthermore, our focus is on defining a memory interface—micro-architectural implementation flaws and side channels are out of scope. Defenses against attacks on the physical RAM or bus interfaces are orthogonal to our work.

3 Baseline Design with Spatial Isolation

We start with a solution that emulates a shared memory abstraction between two spatially isolated enclaves. We refer to this design as the *spatial ShMem baseline.* This solution can satisfy our security goals but has poor performance.

3.1 Emulating Shared Memory

Consider two enclaves that keep private copies of the shared data. Due to the restrictions of the spatial isolation model, the two enclaves cannot access each other's private data. The shared data must therefore either reside in the public memory accessible to both the enclaves, or rely on message-passing (e.g., via RPCs) which itself must use the public memory. Since the untrusted OS can always access data in the public memory, the spatial ShMem baseline requires employing a cryptographic secure channel over the public shared memory. Specifically, the two enclaves encrypt the data before copying it to public memory and decrypt the data after copying it to the enclave private memory. We call this mechanism a *secure public memory*. Let us assume that the cryptographic keys are pre-established securely by the enclaves and analyze the performance overheads.

A secure public memory is not sufficient for a shared memory abstraction in the spatial ShMem baseline. Concurrently executing enclaves may want to access data simultaneously, and such accesses may require serialization in order to maintain typical application consistency guarantees. Notice that reads and writes to the secure public memory involve encryption and decryption sub-steps, the atomicity of which is not guaranteed by the TEE. No standard synchronization primitives such as semaphores and `futexes`—which often rely on OS-provided services—are trustworthy in our threat model.

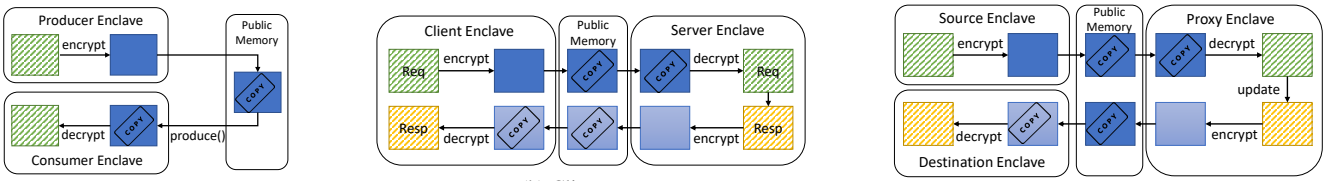
One simple way to serialize access in the spatial isolation model is to use a third enclave as a *trusted coordinator*. For achieving memory consistency, accesses to the shared memory are emulated by making RPCs to the trusted coordinator enclave. The coordinator enclave implements the shared memory by keeping its content in its private memory.

For example, to implement a shared counter, the trusted coordinator keeps the counter in its private memory, and the enclave which wants to access the counter can send messages to the trusted coordinator for reading or updating the counter.

We assume in the baseline that the trusted coordinator is not faulty or compromised. Without this assumption, the baseline would require further defenses to tolerate faulty coordinators (e.g., using BFT-based mechanisms). This will result in even larger performance overheads, since a compromised trusted coordinator can subvert the semantic correctness of the shared memory abstraction. It is straightforward to see that in the baseline design, enclaves never access the shared region directly. Thus, the trusted coordinator can maintain the permission view of each enclave and centrally enforce them, thus satisfying our desired security properties.

3.2 Illustrative Performance Costs

The spatial ShMem baseline is significantly more expensive than the original shared memory abstraction in a non-enclave (native) setting. We refer readers to Section 6 for the raw per-



(a) Producer-consumer (b) Client-server (c) Proxy

Figure 3: Data sharing patterns on the spatial ShMem baseline. The striped boxes represent unencrypted data.

formance costs of the spatial ShMem baseline over the native setting for data sharing, which can be 1-2 orders of magnitude higher. The prime culprit is the encryption-decryption operations and additional memory copies that are inherent in the implementation of the secure channel and the trusted coordinator. Recent work has reported such costs over hundreds of programs [8, 51, 52, 61]. We present three representative textbook patterns of data sharing that ubiquitously arise in real-world applications and illustrate why spatial isolation incurs such significant costs.

Pattern 1: Producer-Consumer. In this pattern, the producer enclave writes a stream of data to shared memory for a consumer enclave to read and process. Applications use this pattern for signaling the completion of sub-steps of a larger task, such as in batch processing scripts in web frameworks [1, 2]. For supporting this pattern with the spatial ShMem baseline, the producer sends the data to the trusted coordinator first, which maintains all the available data inside a queue and sends the data to the consumer enclave upon the latter's request. Both data transfers have to be protected by secure channels, which in total would involve 4 data copies and 2 pairs of encryption and decryption (and hence 4 computing operations per memory word). Figure 2 depicts the steps. Alternatively, the queue can be stored inside the private memory of the consumer enclave. The producer directly sends the data to the consumer enclave, which saves it inside a queue in its own private memory. The consumer enclave can then consume the data directly from this queue at a certain point in the future. This solution involves 2 data copies and 1 pair of encryption and decryption as shown in Figure 3a.

Pattern 2: Client-Server. As explained in our Section 2.1 example, a client enclave and a server enclave exchange data with each other with simultaneous reads and writes. For supporting this pattern, there will be at least 4 data copies—one in server private memory, one client private memory, and two for passing data between them via a public memory. Furthermore, the data is encrypted and decrypted twice, once for read and once for write. Therefore, it incurs 4 computing operations per memory word as shown in Figure 3b.

Pattern 3: Proxy. An application may serve as an intermediate proxy between a producer and a consumer. For example, consider a caching proxy in a web service that saves responses

	Pattern	Spatial			ELASTICLAVE
		Enc	Dec	Cpy	Instructions
1	Producer-Consumer	L	L	$2 \cdot L$	4
2	Client-Server	$2 \cdot L$	$2 \cdot L$	$4 \cdot L$	4
3	Proxy	$2 \cdot L$	$2 \cdot L$	$4 \cdot L$	4

Table 1: Data sharing overheads of spatial isolation vs. ELASTICLAVE. L: data size (memory words) in the shared region.

to frequent requests and serves them by modifying incoming requests (e.g., Nginx [4]). Proxy designs can be implemented as two instances of the producer-consumer pattern, where the proxy acts as the consumer for the first instance and the producer for the second. However, in practice, proxies often optimize by processing the shared data *in-place* without copying it across different queues. Such in-place memory processing is not compatible with the spatial memory model, and applications which originally use this pattern must incur additional memory copies. The data stream must reside in the public memory to be accessible to the proxy enclave, but at the same time, the proxy cannot operate on the public memory in-place, or else it would risk modifications by other enclaves or leaking secrets through intermediate states of the data. Therefore, there are at least 2 memory copies of the 2 original shared data contents, totaling 4 copies when supporting this pattern with the spatial ShMem baseline, as shown in Figure 3c. Similarly, the data needs to be encrypted and decrypted twice, leading to 4 computing operations per memory word.

4 ELASTICLAVE Design

We have established in Section 2.1 that relaxing the memory model poses subtle security challenges. On the other hand, retaining the rigid spatial isolation model leads to prohibitive performance overheads. Next, we present the ELASTICLAVE design that strikes a balance between those two extremes.

4.1 Overview

We highlight the importance of three key first-class abstractions in ELASTICLAVE that allow interacting enclaves to:

(a) have individual *asymmetric permission views* of shared memory regions, i.e., every enclave can have a local view of their memory permissions; (b) *dynamically* change these permissions as long as they do not exceed a pre-established maximum; and (c) obtain *exclusive* access rights over shared memory regions and transfer it atomically in a controlled manner to other enclaves.

As a quick point of note, we show that the above three abstractions are sufficient to drastically reduce the performance overheads highlighted in Section 3.2. In particular, Table 1 shows that ELASTICLAVE eliminates all the extra copies and encrypt/decrypt operations with a small constant number of instructions, whereas the spatial ShMem baseline requires operations linear in the size L of the shared data accessed. We will explain how it achieves such reduction in Section 4.3.

Our design works on the granularity of ELASTICLAVE *memory regions*, which correspond to non-overlapping physical memory and map to contiguous ranges of virtual memory addresses in the enclave. From the view of each enclave, an ELASTICLAVE memory region has four permission bits: standard *read*, *write*, *execute*, and a protection *lock* bit.

Each memory region has one enclave as its sole *owner*. Only the owner of a memory region has the privilege to share it with other enclaves. We call all the enclaves that are interested in accessing a memory region its *accessors*. The owner of a memory region is also one of its accessors. ELASTICLAVE provides three first-class abstractions: *asymmetry*, *dynamicity*, and *exclusivity* in the permission views of an enclave.

Asymmetric Permission Views. In the data patterns examined in Section 3.2, different enclaves require different permissions over the shared memory. For example, one enclave has read-only access whereas the others have write access. The spatial model is a *one-size-fits-all* approach. It does not allow enclaves to set asymmetric permissions for a public memory region securely—the OS can always undo any such enforcement that enclaves might specify via normal permission bits in the TEE. ELASTICLAVE allows different enclaves to specify their own set of permissions (or *views*) over the same shared region for the TEE to enforce. This directly translates to avoiding the costs of creating data copies into separate regions, where each region has a different permission.

Dynamic Permissions. In the spatial isolation model, if enclaves need different permissions over time on the same shared data, they have to create separate data copies. ELASTICLAVE eliminates the need for such copies and allows enclaves to change permissions over time. For example, in Pattern 1, when the producer enclave generates data it has read-write permissions, while the consumer enclave has no permissions. After that, the producer drops all its permissions, and the consumer enclave gains read-write permissions to process the data. This way, the enclaves avoid interfering with each other's operations on the shared region.

While enabling dynamic permissions, ELASTICLAVE does not allow enclaves to arbitrarily escalate their permissions over time. Only the owner can share a memory region with other accessors during the lifetime of the memory region. When the owner shares a memory region, it sets the *static maximum* permissions it wishes to allow for the specified accessor. Once set by the owner, this static maximum is fixed for a specified enclave. For the owner, the static maximum is the full permissions with all bits set. Accessors can escalate or reduce their own privileges dynamically, but if an accessor tries to exceed the static maximum at any given point in time, ELASTICLAVE delivers an exception to it.

Instruction	Permitted Caller	Semantics
`uid = create(size)`	owner of `uid`	create a region
`err = map(vaddr, uid)`	accessor of `uid`	map VA range to a region
`err = unmap(vaddr, uid)`	accessor of `uid`	remove region mapping
`err = share(uid, eid, P)`	owner of `uid`	share region with an enclave
`err = change(uid, P)`	accessor of `uid`	change permission to a region
`err = destroy(uid)`	owner of `uid`	destroy a region
`err = transfer(uid, eid)`	current lock holder	transfer lock to another accessor

Table 2: Summary of security instructions in ELASTICLAVE.

Acquiring and Transferring Exclusive Locks. ELASTICLAVE incorporates a special bit for each memory region called the *lock* bit. It serves as a synchronization mechanism among mutually distrusting enclaves. ELASTICLAVE ensures that at any time at most one accessor has the lock bit set, and we call this accessor the *lock holder* when it exists. As long as the lock holder exists, it is guaranteed to be the only accessor that can access the memory region—the permissions for all the other accessors, including the owner, are temporarily disabled. In this way, ELASTICLAVE guarantees that the lock holder has exclusive access to the region. A lock holder can choose to clear the lock bit (i.e., release the lock) without specifying the next holder or atomically transfer it to another accessor. Atomic transfers are useful for flexible but controlled transfers of exclusive access over regions. For example, in Pattern 3, the source holds the lock bit for exclusive access to the region when writing the request. Then, the source transfers the lock directly to the proxy. After the proxy updates the data with exclusive access, it transfers the lock to the destination. The two lock transfers ensure exclusive access for the intended accessor at any time. Note that transferring exclusive lock does not entail a transfer of ownership.

4.2 Design Details

As summarized in Table 2, the ELASTICLAVE model consists of seven instructions which operate on ELASTICLAVE memory regions. Each region is addressable with a *universal* identifier that uniquely identifies it in the global namespace and can map to different virtual addresses in different enclaves. Next, we explain the ELASTICLAVE design by walking through the typical life cycle of a region.

Owner's View. An enclave *e* can create a new memory region *r* with the `create` instruction. It takes the memory region size as input and returns a universal identifier (`uid`) of the newly created memory region. In this case, the enclave *e* will be the unique owner of the memory region *r* throughout the lifetime of *r*. The owner can then share the memory with any other enclave using the `share` instruction, specifying the `uid` of the memory region, the enclave ID (`eid`) of the other accessor, and the static maximum permissions allowed for that accessor. Once a memory region has been shared with a target enclave, the owner cannot change or revoke the static maximum permissions without destroying the entire region.

A memory region can be destroyed by its owner at any time with the `destroy` instruction. This instruction ends the lifetime of the memory region in all enclaves and ELASTICLAVE sends a signal to all its accessors. The OS can tear down an enclave at any time (similar to `EREMOVE` in Intel SGX) to reclaim the memory region it owns and protect the system from enclave-launched denial-of-service attacks.

Accessor's View. Before accessing a memory region, an accessor, including the owner, needs to use the `map` instruction to map it to a specified virtual address range. This can be done more than once, resulting in multiple mapped instances in the virtual address space, but access permissions to the memory region apply to all of them. The accessor can then use the `change` instruction to dynamically change the permissions of a memory region. This is allowed as long as the permissions are equal to or more restrictive than (i.e., subset of) the static maximum permissions for that accessor. For the owner of a memory region, the static maximum permissions are the full permissions. The owner specifies the static maximum permissions for other accessors via the `share` instruction. Changes to such permissions are local to each accessor and do not interfere with permissions of other accessors. Unlike the owner, a non-owner accessor cannot invoke `share` or `destroy` to share or destroy a memory region.

Permission Checks. The ELASTICLAVE TEE implementation enforces the permissions defined by enclaves in their local views. A permission bit is set to 1 if the corresponding memory access (read, write, or execute) is allowed, and 0 otherwise. For each memory access, the TEE hardware performs a permission lookup and decides whether to allow the access. Such permission checks complement the paging-based memory protection mechanism, i.e., access is allowed only when it passes both checks. This enables both enclaves and the OS to manage their own paging while enforcing the permissions defined by ELASTICLAVE.

ELASTICLAVE performs two categories of security checks: (1) availability check of the requested resources (e.g., memory regions and enclaves) to ensure that instructions will not operate on non-existing resources; and (2) permission checks of the caller to ensure that it has sufficient privilege for the requested instruction. Table 2 shows the permitted caller for each instruction. For example, only the owner of a region can invoke `share` and `destroy` instructions.

The `change` instruction is the interface for dynamically updating permissions of a shared region. ELASTICLAVE requires that the newly requested permissions (P) by an enclave fall within the limits of its static maximum permissions (max). Specifically, ELASTICLAVE checks that $P \subseteq max$. The lock bit can only be set to 1 in the local view of a single enclave (i.e., the lock holder) at any instance of time. When it is set for one enclave, ELASTICLAVE enforces the local permission bits for that enclave and guarantee that none of the other enclaves has any access to the region. Otherwise, ELASTICLAVE enforces the local permission bits for all enclaves.

Lock Acquire & Release. An accessor can attempt to acquire or release the lock by using the `change` instruction. It returns the accessor's modified permissions, including the lock bit that indicates whether the acquire/release was successful. ELASTICLAVE ensures that at any instance of time, only a single enclave is holding the lock. If any other enclave accesses the region or tries to issue a `change` instruction on the permissions of that region, these requests will be denied.

A lock holder can use the `change` instruction to release locks. However, there are situations where the lock holder wishes to explicitly specify which other enclave it intends to make the next lock holder of the lock. ELASTICLAVE allows the lock holder to invoke a `transfer` instruction which specifies the enclave ID of the next desired accessor. The next lock holder must have the memory region mapped in its address space for the transfer to be successful. A successful `transfer` instruction clears the lock bit in the source enclave permissions and sets that of the target enclave atomically. The other permission bits and the virtual address mappings of the shared memory region remain unchanged.

ELASTICLAVE Exceptions & Signals. ELASTICLAVE issues exceptions whenever an enclave attempts memory operations that violate any permission checks. ELASTICLAVE notifies enclaves about events that affect the shared memory region via asynchronous signals. ELASTICLAVE issues signals under two scenarios. First, when the owner destroys a memory region *r*, ELASTICLAVE will invalidate permissions granted to other enclaves since the memory region no longer exists. To prevent enclaves from continuing without being aware that the memory region can no longer be accessed, ELASTICLAVE will send signals to notify all accessors who had an active mapping (i.e., mapped and not yet unmapped) for the destroyed memory region. The second usage of signals is to notify changes on lock bits. Each time an accessor successfully acquires or releases the lock (i.e., using `change` or `transfer` instructions), ELASTICLAVE issues a signal to the owner. The owner can choose to mask such signals or to actively monitor the lock transfers. When a `transfer` succeeds, ELASTICLAVE notifies the new accessor via a signal.

Using ELASTICLAVE Interface. Enclaves can invoke the ELASTICLAVE instructions listed in Table 2. Non-enclave

code, including the host process and the OS, can also use the ELASTICLAVE interface, but is treated as a single entity. The interface uniquely identifies enclaves and memory regions with identifiers (`eids` and `uids`), which we implement as non-repeating integer values. We reserve the special `eid` 1 for referencing the untrusted code. Enclaves can exchange `eids` and `uids` using attestation and secure channels.

Compatibility with Spatial Isolation. It is easy to see that ELASTICLAVE is more expressive than the spatial isolation model, and hence keeps complete compatibility with designs that work in the spatial isolation model. Setting up the equivalent of the public memory is simple. The owner can create the region and share it with `rwx`– for all the entities. Private memory simply is not shared after creation by the owner.

Built-in Privilege De-escalation. In ELASTICLAVE, enclaves can self-reduce their privileges below the allowed maximum without raising any signals to other enclaves. This enables compatibility with other low-level defenses which enclaves may wish to deploy for their own internal safety. For example, enclaves can use it to make shared data non-executable or to write-protect shared security metadata.

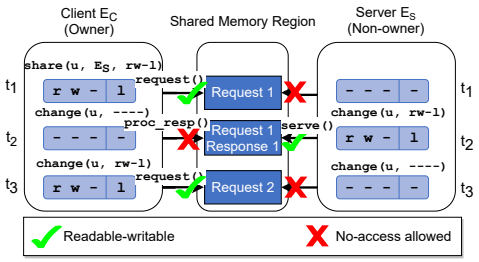
Achieving Our Security Goals. The two desirable security properties outlined in Section 2.2 immediately follow from the ELASTICLAVE interface. ELASTICLAVE ensures the first property of bounded escalation due to three design points:

(a) Only the owner can use `create` to change the set of enclaves that can access a region. Non-owner enclaves cannot grant access permissions to other enclaves since there is no permission re-delegation instruction in the interface.
(b) Each valid enclave that can access a region has its permissions bounded by an owner-specified static maximum.
(c) For each access or instruction (see Table 2), the identity of the accessor and the permission are checked to be legitimate.

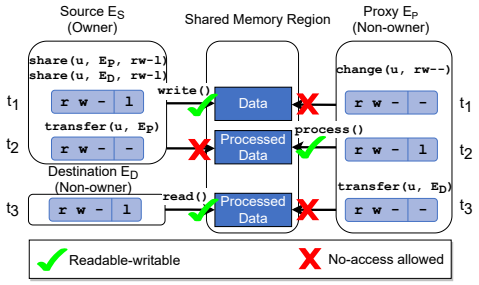
The second property of enforceable serialization of non-faulty enclaves is guaranteed by composing two ELASTICLAVE abstractions:

(a) For each access a_i by an enclave $e(a_i)$ in the predetermined sequence, the accessor can first acquire the lock to be sure that no other accessors interfere.
(b) When the accessor changes, say at access a_j, the current enclave $e(a_j)$ can safely hand over the lock to the next accessor $e(a_{j+1})$ by using the `transfer` instruction.

Faulty enclaves cannot acquire the lock and access the region at any intermediate point in this access chain. For example, in Pattern 3 (proxy), once the proxy enclave modifies the data in-place, simply releasing the lock is not *safe*. A faulty source enclave can acquire the lock before the destination does and tamper with the data. With the `transfer` instruction, the proxy can eliminate such attacks.



(a) Client-server



(b) Proxy

Figure 4: Data sharing patterns with ELASTICLAVE. `map` instructions are omitted. For the proxy pattern, we omit `change(u, rw--)` of the destination enclave at t_1.

4.3 Performance Benefits

We revisit the example patterns discussed in Section 3.2 to show that these patterns can be implemented with significantly lower costs (summarized in Table 1) with ELASTICLAVE.

Revisiting Pattern 1: Producer-Consumer. This pattern can be implemented in ELASTICLAVE with a queue in a shared memory region between the producer and the consumer enclaves. Each enclave acquires the lock before accessing the queue with exclusive write permissions. The consumer needs write permissions to update the queue head pointer. This way it can indicate the consumed locations that can be recycled by the producer. The exclusive access used by both enclaves prevents TOCTOU attacks and secret leakage through intermediate data. The overheads of each `enqueue` or `dequeue` operations are reduced to zero copies and no encryption-decryption. Both the enclaves can abuse their permissions. For example, only the producer should be able to update the queue data and the tail pointer, but a malicious consumer can try to modify them. This does not compromise security, however, because the consumer can only tamper with the content or the ordering of the data it is about to read. Similarly, only the consumer should update the head pointer, but a malicious producer can modify it too. Despite this, the producer can merely force the consumer to re-process or skip some queue elements, which it can already perform without abusing the permissions. Therefore, such permission abuse does not compromise the security of either enclave.

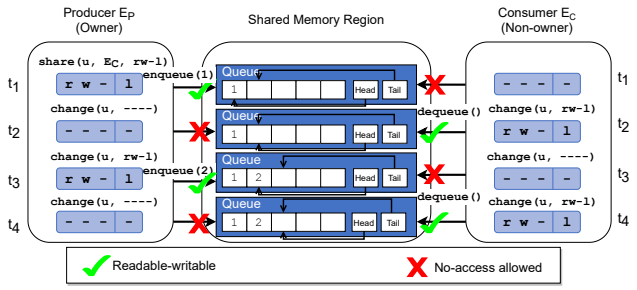


Figure 5: Producer-consumer pattern on ELASTICLAVE. `map` instructions are omitted. A queue (including both its data and the head and tail pointers) resides in a shared memory region. Enclaves use `change` to acquire exclusive writable permissions before producing or consuming data in the queue and update the head and tail pointers.

Revisiting Pattern 2: Client-Server. Client-server can exchange data via a client-owned memory region (Figure 4a), where the server has `rw` − `l` maximum permissions. The client writes the data and uses `change` to revoke its own permissions. The server gets exclusive read-write permissions with `change`, processes the data in-place, and revokes its permissions. The client then reads the response with exclusive read permissions. Thus, compared with spatial isolation, ELASTICLAVE eliminates data copies and encryption-decryption.

Revisiting Pattern 3: Proxy. In the proxy pattern, the three entities access the shared data in sequential order: first the source, then the proxy, then the destination. All three entities can hold the lock bit in this order to stop any faulty enclaves from accessing the shared memory where unintended. The ELASTICLAVE `transfer` instruction eliminates the attack windows when a lock is passed from one enclave to another. Specifically, it allows the source to atomically transfer the lock to the proxy, which then atomically transfers it to the destination. With these two `transfer` instructions, the proxy can be implemented without any extra copy of the shared data as shown in Figure 4b.

5 Implementation

We build a prototype implementation of ELASTICLAVE on Keystone, an open-source TEE for RISC-V platforms [34]. Keystone provides a Linux driver and an SDK to create, start, resume, and terminate enclaves. We do not use the original spatial isolation model in Keystone.

RISC-V Privilege Levels. RISC-V software runs on three different privilege levels: machine mode (`m-mode`), supervisor mode (`s-mode`), and user mode (`u-mode`). They form a privilege hierarchy similar to the protection rings on Intel x86, with `m-mode` being the most privileged and `u-mode` the least. `m-mode` software has full control over memory and hardware resources, and is not constrained by memory protection mechanisms such as paging. RISC-V TEEs [10, 24, 34] are usually implemented as slim `m-mode` security monitors. All the other software on the system, including the OS and user applications (including enclaves), resides in `s-mode` or `u-mode`.

RISC-V PMP. The physical memory protection (PMP) feature of RISC-V allows `m-mode` to restrict physical memory accesses of software at lower privilege levels (`s-mode` and `u-mode`). To use this feature, `m-mode` configures PMP entries, which are a set of registers in each CPU core. Each PMP specifies one contiguous physical address range and its corresponding access permissions. PMP is enforced alongside the classical page-based virtual memory system. For all `s-mode` and `u-mode` memory accesses, the hardware looks up PMP entries against their physical addresses after the address translation. Note that only `m-mode` can modify them.

RISC-V implementations are free to cache the PMP lookup results to optimize the performance. This choice concerns the micro-architectural implementation of RISC-V and is not part of the Elasticlave design or implementation. After each PMP update, the `m-mode` software needs to use the `sfence.vma` instruction in RISC-V to prevent the hardware from using stale PMP configurations. An example is the RocketChip implementation of RISC-V [9] which caches PMP lookup results in TLBs. In such a case, `sfence.vma` performs a TLB flush to keep PMP lookup results up-to-date. Interested readers can refer to the RISC-V standard specifications [47].

We implement ELASTICLAVE as an `m-mode` software security monitor, similar to other RISC-V TEEs [10, 24, 34]. The security monitor stores all the metadata about the memory regions, enclaves, static maximums, and permissions in the `m-mode` memory, which is protected by one reserved PMP entry. The OS (`s-mode`) and applications (`u-mode`, no matter in enclaves or not) cannot read or update them. Upon a context switch between an enclave and the untrusted OS, the security monitor looks up the permissions metadata maintained in the `m-mode` memory and loads it into the PMP registers.

When the enclave invokes an ELASTICLAVE instruction, the execution traps into `m-mode`. `s-mode` or `u-mode` software cannot change this control flow. After checking that the enclave is a permitted caller of the instruction (Table 2), `s-mode` performs the requested operations and updates the metadata and PMP when necessary.

ELASTICLAVE keeps three mappings in its implementation: (a) virtual address ranges to the corresponding mapped `uid` in each enclave; (b) `uid` and corresponding permission metadata (including permissions and static maximum permissions in each enclave and ownership); and (c) the effective physical address range to which each `uid` maps. Thus, when an enclave tries to access a virtual address, ELASTICLAVE performs a two-level translation: from a virtual address to a `uid` and subsequently to a physical address. Permission checks are performed by looking up the permission metadata tied to the `uid`. The `map` and `unmap` instructions only update mapping (a). The `transfer` and `change` instructions update mapping

(b). The `share` and `create` instructions update mappings (b) and (c). The `destroy` instruction removes data tied to the provided `uid` from all three mappings.

ELASTICLAVE relies on PMP entries to enforce the access permissions to memory regions. For each memory region, ELASTICLAVE sets up one PMP entry to protect its mapped physical address range, irrespective of the number of its accessors. Additionally, the security monitor and the OS each reserve one PMP entry for their own private data. Consider the case where each enclave has one private memory region and one shared memory region, each requiring a PMP entry. In this example, N PMP entries can support up to $(N-2)/2$ concurrent enclaves. When the enclave requests permission changes, the corresponding PMP entries must reflect the updates. In our multi-core implementation, changing permissions on one core sometimes requires adjusting the PMP entries on another core. For example, if core A is running code outside enclaves and core B is handling an enclave request to create a memory region, core A should not be able to access the new memory region. To handle such cases, ELASTICLAVE issues inter-processor interrupts to cores that need to adjust their PMPs. The core that initiated the permission change does not resume until all cores have finished the adjustment. When context-switching between enclaves, apart from the standard register save-restore, Keystone modifies PMP entries to disallow access to enclave private memory—this is because of its spatial isolation design. We modify this behavior to allow continued access to shared regions even when the owner is not executing. Our implementation of the exclusive lock bit, by atomically changing permissions, does not halt large portions of the system. It only halts enclaves that attempt to access a locked memory region.

The OS allocates physical memory in ELASTICLAVE, similar to Keystone. `m-mode` requests the OS to allocate or deallocate physical memory. For allocation, the OS returns the base of the physical memory and `m-mode` assigns the permissions and updates the PMPs. For deallocation, `m-mode` scrubs the memory region and informs the OS of its base and size.

To allow enclaves to capture exceptions, `m-mode` configures the exception delegation register (`medeleg`) to delegate access fault handling to `s-mode`. Signal delivery from `s-mode` to enclaves is immediate, i.e., the target enclave is immediately interrupted and upon return from `m-mode` jumps to a previously registered signal handler. Note that neither mechanism involves the untrusted OS.

6 Evaluation

We aim to answer the following questions in our experiments:

(a) How does the performance of ELASTICLAVE compare with the spatial ShMem model and native execution on RISC-V?

(b) What is the impact of ELASTICLAVE on software trusted computing base (TCB) and hardware complexity?

Benchmarks. We evaluate ELASTICLAVE and the spatial ShMem baseline (Section 3.1) on two types of benchmarks: (a) implementation of three data patterns with varying number of regions and size of data, thread synchronization workloads with controllable lock contention; (b) standard benchmarks for I/O (IOZone [29]), parallel computation (SPLASH-2 [13,56]), and CPU-intensive workloads (machine learning inference with Torch [59,60]). Table 3 provides a summary of the benchmarks details. We manually modify these programs to add ELASTICLAVE instructions, since we do not presently have a compiler targeting ELASTICLAVE. Due to the lack of compiler support, it is difficult to port larger real-world applications. However, to estimate performance improvements for larger applications, we profile two real-world applications: a web server that uses Lighttpd with Nginx and Memcached. We estimate the expected end performance improvements if they run on ELASTICLAVE from the profiled information.

Experimental Setup. We use a cycle-accurate, FPGA-accelerated simulation of RocketChip [9] with FireSim [32]. Each system consists of four RV64GC cores, 16 KB instruction and data caches, 16 PMP entries per core (unless stated otherwise), and a shared 4 MB L2 cache. For the evaluation of hardware complexity, we use a commercial 22 nm process with Synopsys Design Compiler version L-2016.03-SP5-2 targeting 800 MHz. We generate the L1 caches with commercial SRAM libraries and exclude the L2 cache.

6.1 Performance of ELASTICLAVE

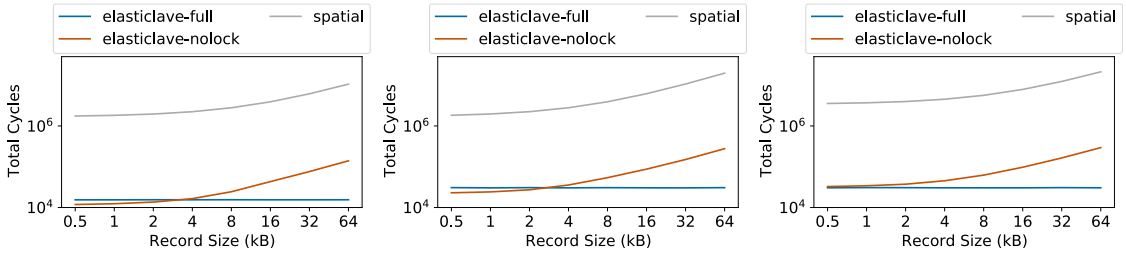
Synthetic Benchmark: Data-Sharing Patterns. We construct synthetic benchmarks for the three patterns. Since we are interested solely in data sharing overheads, we exclude any application-specific data processing in those benchmarks. We set up two enclaves for producer-consumer and client-server and three enclaves for the proxy pattern. We compare the (a) full ELASTICLAVE support as described in Section 4.2 (ELASTICLAVE-full); (b) ELASTICLAVE without the lock permission bit design (ELASTICLAVE-nolock); and (c) spatial isolation which transfers data through secure public memory. Figure 6 shows the performance for the three patterns. Figure 7 further shows the breakdown for the proxy pattern.

Observations: ELASTICLAVE-full performs better than spatial, especially for larger record sizes. We observe a 60× speedup for 512 bytes and 600× for 64 KB record sizes. ELASTICLAVE-full eliminates copies and the only costs are due to security instructions. Thus, its overheads do not increase with the data size, unlike spatial. ELASTICLAVE-nolock is slower than ELASTICLAVE-full. As shown in Figure 7, it incurs larger overheads that increase with the data size because it does not fully eliminate data copying.

Synthetic Benchmark: Thread Synchronization. We implement a common workload for locks between threads running in separate enclaves, such that none of the enclaves

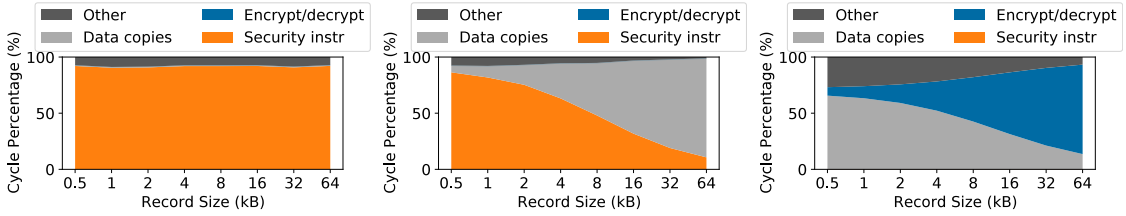
Benchmark	Shared Data Size	Data Sharing	Spatial	ELASTICLAVE
		Synthetic Benchmarks		
Client-Server	512 bytes – 64 KB	`C->S; S->C`	`C->U->S; S->U->C`	`r1 (C): S (r--l); r2 (A): S (rw-l)`
Producer-Consumer	512 bytes – 64 KB	`P->A`	`P->U->C`	`r (P): C (rw-l)`
Proxy	512 bytes – 64 KB	`S->P->D`	`S->U->P->U->D`	`r (S): P (rw-l), D (rw-l)`
Thread Synchronization	16 bytes	`A<->B`	`A<->U<->C<->U<->B`	`r (A): B (rw--), O (r---)`
		Real-World Benchmarks		
File I/O (IOZone)	4 KB – 1 MB	`A->O; O->A`	`A->U->O; O->U->A`	`r (A): O (r--l); r(A): O (rw-l)`
Parallel Computation (SPLASH-2)	16 MB	`A<->B`	`A<->U<->C<->U<->B`	`r (A): B (rw--)`
ML Inference	3 KB – 262 KB	`O->A`	`O->U->A`	`r (A): O (rw-l)`

Table 3: Summary of evaluated benchmarks. `S` (server/source), `C` (client), `P` (proxy), `D` (destination), `A` and `B` are user applications. `O` is the operating system. `U` is the untrusted memory. Arrows indicate the flow of data. For ELASTICLAVE, the shared memory regions are specified in the `region (owner): accessor1 (maxperm1), accessor2 (maxperm2), ...` format.



(a) Producer-consumer (b) Client-server (c) Proxy

Figure 6: Performance of the three data-sharing patterns. ELASTICLAVE achieves constant overheads irrespective of the shared data size, around two orders of magnitude lower than those of spatial, which increase with the shared data size.



(a) ELASTICLAVE-full (b) ELASTICLAVE-nolock (c) spatial

Figure 7: Performance breakdown for the proxy pattern. When the shared data size grows, copying and encryption-decryption claim a growing percentage of overheads of spatial, which ELASTICLAVE eliminates at a constant cost from security instructions.

trust the OS. For ELASTICLAVE, we implement locks with simple spinlocks (ELASTICLAVE-spinlock) and `futexes` (ELASTICLAVE-futex). For spinlocks, we keep the lock state in a shared region which is inaccesssible to the OS. For `futexes`, the untrusted OS has read-only access to the lock states, which allows enclaves to sleep while waiting for locks and be woken up by the OS when locks are released. This form of sharing corresponds to the one-way isolation described in Section 3.2, where the OS has read-only permissions. For spatial, we implement a dedicated trusted coordinator enclave to manage the lock states. Enclaves communicate with it through secure public memory for lock acquisition and release. To characterize the amount of work a thread does while holding the lock, we set up an empty loop between lock acquisition and lock release in our benchmarks. We adjust the number of iterations and report the number of cycles averaged over an iteration.

Observations: ELASTICLAVE-futex and ELASTICLAVE-spinlock perform better than spatial, especially when the work between lock acquisition and lock release is small, i.e., the lock is acquired and released often. For larger work amounts, the time spent waiting for the lock is smaller than the time spent acquiring and releasing the lock. Here, the three settings have comparable performance. In addition, ELASTI-

CLAVE-futex achieves up to 1.5× improvement in CPU-time performance over ELASTICLAVE-spinlock, despite having no observable advantage in terms of real-time performance (wall-clock latency).

Real-World Benchmark 1: File I/O. IOZone [29] makes frequent file I/O calls from the enclave into the untrusted host process. Here, the data pattern between the OS and the enclave corresponds to either the producer-consumer pattern (for write) or the client-server pattern (for read). For spatial, the OS and the enclave communicate via the public memory. Since the OS is one of the communicating entities, secure public memory is not needed. For ELASTICLAVE, the enclave passes data to the OS via a shared memory region. Figures 8a and 8b show the write and read bandwidth.

Observations: Even though spatial does not use secure public memory for communication, ELASTICLAVE achieves a higher bandwidth than spatial when the record size grows above a threshold (16 KB). The bandwidth increase reaches as high as 40% for the writer workload and around 50% for the reader workload when the record size is sufficiently large.

Real-World Benchmark 2: Parallel Computation. We adapt seven SPLASH-2 workloads to a two-enclave setting by placing the data in a dedicated memory region which is then shared across the enclaves. For spatial, a trusted coordinator maintains the shared data. The enclave runtime traps and emulates load/store instructions that operate on the memory region by converting them to RPCs. In ELASTICLAVE the enclaves directly store the data in a shared memory region. Figure 9 shows their performance comparison. Since we were not able to run `libsodium` inside the enclave runtime, we did not use encryption-decryption when copying data to and from secure public memory for spatial in this experiment. Therefore, the actual overheads in a secure implementation would be higher than reported here. Thus, even with cryptographic accelerators (e.g., AES-NI) to speed up spatial, ELASTICLAVE benefits from zero-copies and outperforms spatial.

Observations: ELASTICLAVE overheads are independent of the data size and are 2-3 orders of magnitude lower than those of spatial for data-intensive workloads.

Real-World Benchmark 3: ML Inference. We run four machine learning models for image classification [59] that involve minimal data sharing. Each of the inference models runs with a single enclave thread. In ELASTICLAVE, we use a shared memory region to load input images and write results, as in the client-server pattern. The three settings exhibit similar performance (Figure 12).

Observations: ELASTICLAVE does not impact the performance of CPU-intensive programs with minimal data sharing.

Summary. Compared to spatial, ELASTICLAVE improves I/O-intensive workload performance up to 600× (data size > 64 KB) and demonstrates 50% higher bandwidth. For shared-memory benchmarks, it gains up to a 1000× speedup.

Comparison with Native. ELASTICLAVE only adds 10% overheads compared to native (traditional Linux processes without enclave isolation) for a range of benchmarks (Figure 9, 10, 12, and 11). The performance of ELASTICLAVE is comparable to that of native for frequent data sharing over large record sizes (Figures 8a and 8b).

6.2 Impact on Implementation Complexity

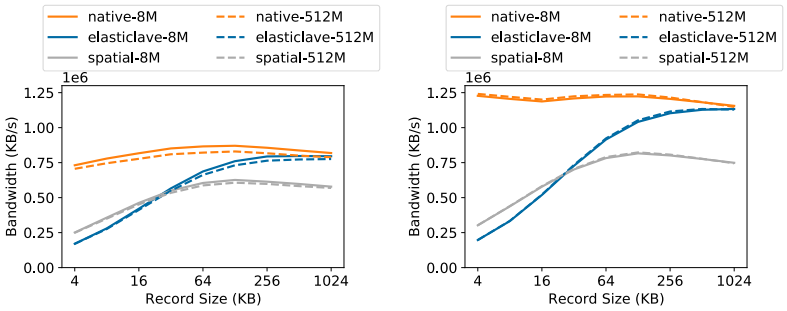
TCB. The ELASTICLAVE TCB is 6814 LoC, including 3085 LoC for implementing the ELASTICLAVE interface in `m-mode`, and 3729 LoC that uses the interface in an enclave. Figure 4 gives a detailed TCB breakdown.

Context Switches. Context switching between enclaves and the OS involves PMP changes. Thus, the overheads may change with the number of PMP-protected memory regions. To empirically measure this, we record the percentage of cycles spent on context switches in either direction for a workload that never explicitly switches out to the OS. Therefore, all context switches from the enclave to the OS are due to interrupts. The percentage overheads increase linearly with the number of memory regions but is negligibly small: 0.1% for one memory region and 0.15% for four memory regions.

TLB Flushes. Since we prototype ELASTICLAVE on RocketChip, which caches PMP lookup results in TLBs, it is necessary for the security monitor to perform a TLB flush on a core every time permissions on it are changed. This could incur extra overheads. However, note that such overheads are not inherent to ELASTICLAVE, but are incurred in Keystone as well whenever an enclave is created or destructed. Moreover, TLB flushes only occur on cores that require permission changes, which can happen only when the enclave running on the core is invoking the `change` instruction, or when a region is locked, unlocked, created, or destructed. Only lock and region creation/destruction operations incur updating additional cores that are affected by the change.

Hardware Critical Path Delay. The only impact of ELASTICLAVE on RISC-V hardware is on the increased PMP pressure (i.e., each memory region requires at least one PMP entry). To determine the critical path of the hardware design and examine if the PMP entries are on this path, we push the design to a target frequency of 1 GHz[1]. We measure the latency of the global critical path of the whole core and that of the critical path through the PMP registers. With this, we compute the slack, which is the desired delay minus the actual delay. A larger slack corresponds to a smaller actual delay in comparison to the desired delay. We find that the slack through PMP is significantly better than the global critical path. With 16 PMP entries, the slack through PMP is −44.1 picoseconds compared to −190.1 picoseconds for the global critical path. In other words, the PMP would allow for a higher

[1]We set the frequency higher than 800 MHz (which is what we have for our successful synthesis) to push the optimization limit of the hardware design so we can find out the bottleneck of the hardware design.



(a) IOZone Writer (b) IOZone Reader

Figure 8: IOZone Bandwidth for 8M and 512M byte files.

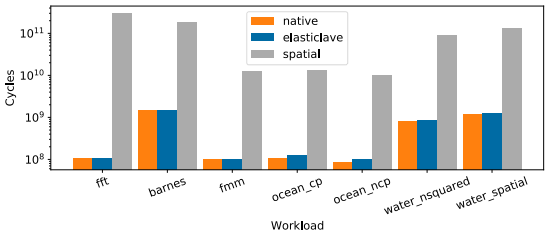


Figure 9: SPLASH-2 wall-clock time in cycles.

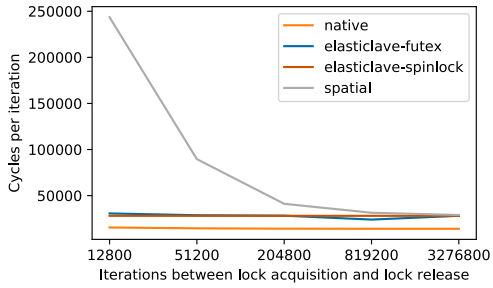


Figure 10: Synthetic Thread Synchronization Performance.

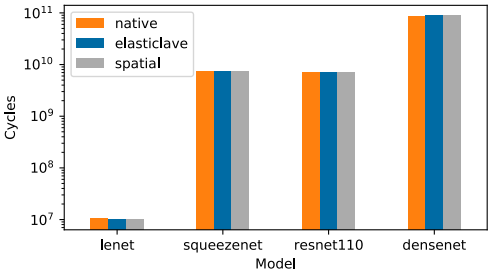


Figure 12: Cycles spent running each ML model.

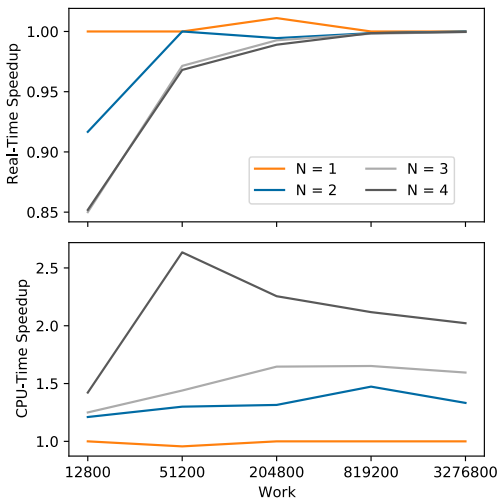


Figure 11: ELASTICLAVE-futex vs. ELASTICLAVE-spinlock.

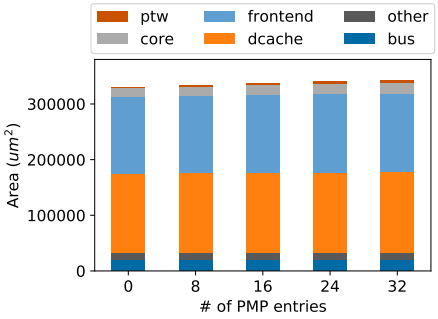


Figure 13: RocketChip Area vs. numbers of PMP entries.

clock speed, but the rest of the design prevents it. Thus, the number of PMP entries is *not the bottleneck of the timing of the hardware design*. We also tested that PMPs are not on the critical path for 8 and 32 PMP settings as well (details elided due to space). As a direct result, the number of PMP entries does not create a performance bottleneck for any instruction (e.g., load/store, PMP read/write) in our tests.

Area. To explore how the increased PMP pressure brought by our prototype ELASTICLAVE implementation may increase chip area requirements, we synthesize RocketChip with different numbers of PMP registers and collect the area statistics. The range we explore goes beyond the limit of 16 in the standard RISC-V ISA specification. Figure 13 exhibits the increase in the total area with increasing numbers of PMP entries. Starting with no PMP entries, every 8 additional PMP entries incurs an increase of 1% in the total area. This increase

Function	ELASTICLAVE Privileged TCB	Enclave Runtime
`uid` management	1070	0
Permission enforcement	574	0
ELASTICLAVE instruction interface	219	82
Argument marshaling	0	88
Wrappers for ELASTICLAVE interface	0	1407
Miscellaneous	960	1869
Total	3085	3729

Table 4: Breakdown in LoC of ELASTICLAVE TCB.

Case study	I. proxied web server				II. KV store
Transfer	**li→ng**	**ng→li**	**li→py**	**py→li**	**mc↔client**
Data size	2049.6MB	1.4MB	10.6MB	2051.5MB	1157.7MB
Frequency	250276	21004	185956	670786	2002017
Spatial Isolation					
Copy	4099.1MB	2.8MB	21.2MB	4103.1MB	2315.5MB
Enc-Dec	4099.1MB	2.8MB	21.2MB	4103.1MB	2315.5MB
ELASTICLAVE					
Instr	500552	42008	371912	1341572	4004034

Table 5: Profiling results of applications along with projected costs in spatial isolation and ELASTICLAVE.

is equivalent to 2.3% of the L1 data cache area.

6.3 Profiling Real Applications

We present profiling results to demonstrate how the performance improvements of ELASTICLAVE apply to large real-world applications. We use two common multi-process setups: (1) Lighttpd web server with Nginx as its SSL/TLS termination. The Lighttpd instance communicates (through FastCGI) with a Python process to generate dynamic HTTP responses. We generate the workload with single-threaded wrk maintaining 50 connections for 180 seconds and requesting 100 KB dynamically generated random resources.
(2) Memcached serving as an in-memory key-value store for another application (e.g., web server). It receives updates or queries from outside and sends back the results. We run it with YCSB workload A (one million records).

As shown in Table 5, both setups incur large inter-process data transfers. In the first setting, data transfers are especially intensive from Lighttpd to Nginx and from python to Lighttpd, both reaching 2 GB. The second setting also involves transferring over 1 GB of data. In both settings, spatial isolation requires copying data twice as well as a pair of encryption-decryption, leading to around 8 GB and 2 GB of data copying and encryption-decryption operations respectively. ELASTICLAVE eliminates those operations at the cost of 2 extra security instructions per data transfer, which totals around 2×10^6 and 4×10^6 instructions respectively. To estimate the performance improvement, we calculate the average record sizes of the data transfers, which are around 4 KB and 0.5 KB respectively. By comparing them against the benchmark results for the producer-consumer pattern (Figure 5), we conclude that ELASTICLAVE is able to bring 2 orders of magnitude performance improvement over spatial isolation for data sharing in both application settings. Through profiling we find that even in the native Linux environment, the applications spend significant amounts of time on data sharing (9.2% for Lighttpd and 23.1% for Memcached). On spatially isolated TEEs, data sharing will take an even much larger portion of time due to the need for extra encryption-decryption.

7 Related Work

ELASTICLAVE draws attention to a single point in the TEE design space, namely the memory model and its impact on memory sharing. The predominant model, *spatial isolation*, is adopted by Intel SGX [40], TrustZone [7], AMD SEV [30,31], and so on. ELASTICLAVE explains the conceptual drawbacks of this model and offers a relaxation that enables better performance. Intel SGX v2 follows the spatial isolation design, with the exception that permissions and sizes of private regions can be changed dynamically [39, 64]. Thus, it retains the all-or-none trust division between enclaves as in v1.

A range of TEE designs have shown the promise and feasibility of enclave TEEs [15, 19, 22, 35, 53, 57]. Several works have proposed security and compatibility improvements over the original TEE design [16, 17, 24, 27, 34, 39, 45, 64]. They allow for better security, additional classes of applications, better memory allocation, and hierarchical security protection. Nevertheless, they have not explicitly challenged the assumptions and still adhere to the spatial isolation model.

Our temporal memory model may seem similar to mechanisms in hypervisors and microkernels—for example, as used in page-sharing via EPTs [20, 65], IOMMU implementations for memory-mapped devices such as GPUs or NICs [37]. The key difference is in the trust model. Hypervisors [11] and microkernels [36] are entrusted to make security decisions on behalf of VMs. In TEEs, the privileged software is untrusted and enclaves make their own security decisions.

Emerging proposals such as Intel TDX [58], Intel MK-TME [42], Intel MPK [43], and Donky [48] enable hardware-enforced domain protection. However, they protect entire virtual machines or groups of memory pages. Notably, they extend fast hardware support to protect physical memory of a trust domain (e.g., from a physical adversary) but still adhere to the spatial isolation model.

Similar to ELASTICLAVE, pass-by-reference avoids copying large amounts of data and is efficient (e.g., compared to pass-by-value). It does so by allowing functions to access the same data simultaneously. However, it is often implemented with static permissions (Option 1 in Section 2.1) and is prone to attacks. POSIX and System V allow processes to set up shared memory regions (e.g., for IPC) albeit with static permissions. Thus, importing these interfaces as-is to TEEs is insufficient. For example, the POSIX interface is unsafe because it ties access permissions to file descriptors

and any process can re-share the descriptors with other processes without the owner's knowledge. The owner cannot stop such unwanted sharing, impose a local permission view, or forcefully destroy a region. In summary, the ELASTICLAVE interface is more secure than pass-by-reference and POSIX in the context of an enclaved TEE.

8 Discussion

Current Limitations. The RISC-V specification limits the number of PMP registers to 16 [47]. Since each PMP entry protects one memory region, this fixes the maximum number of simultaneous regions across all enclaves. Future RISC-V implementations can increase the number of PMP entries, orthogonal to ELASTICLAVE [3, 46]. Keystone, the basis of our prototype implementation, supports page swapping of the private memory. Swapping shared memory, however, has two major challenges. First, it is necessary to adjust the accessibility of the evicted page to accommodate the incoming data that is being swapped into the memory. For this reason, an enclave has to notify the security monitor before swapping shared memory. This way, the security monitor can adjust the PMP configuration accordingly. Second, when swapping in or out a shared page, the security monitor must inform all of its accessors so they can adjust their page tables. This implies that the security monitor needs to directly handle data swapping, otherwise an enclave may swap in shared data without the knowledge of other enclaves. Our current implementation does not support swapping the shared memory of an enclave. We plan to address these challenges in future work.

SGX-Based Implementation. The ELASTICLAVE design is not specific to Keystone or RISC-V and can be integrated into other TEE implementations, including Intel SGX, Intel TDX, AMD SEV, ARM TrustZone, and ARM Realms. For concreteness, we discuss the specific changes required to adopt ELASTICLAVE into Intel SGX since its memory management model is different. As in the Keystone-based implementation, an SGX-based implementation needs to maintain the permission metadata. However, unlike Keystone, SGX tracks the virtual address mappings of enclaves. This is because in SGX, the OS manages virtual address mappings and would be able to manipulate them to launch attacks if they were not monitored. To detect such attempts, SGX keeps its own mappings and performs checks for each enclave access. Specifically, SGX stores the reverse address mappings (from physical addresses to virtual addresses) along with access permissions in Enclave Page Cache Maps (EPCMs) [23, 40]. It only supports one virtual address per physical address and fixes the accessor to be the owner enclave. To support ELASTICLAVE in SGX, we can extend these data structures to allow multiple (enclave, virtual address, permissions) tuples per physical address.

Implementing these changes in the SGX design can be non-trivial and different from implementing our prototype system. First, the SGX implementation of ELASTICLAVE will have to involve the OS for operations that change enclave page tables. For instance, after `map` or `unmap`, the hardware will have to send the new mappings to the OS. Second, to enforce the ELASTICLAVE memory protection, the implementation must perform checks based on the aforementioned extended data structures for every memory access. The ease of such enforcement and the corresponding performance ramifications may not be the same as in our RISC-V design. Lastly, SGX includes a Memory Encryption Engine (MEE) to protect EPC (Enclave Page Cache) pages from physical attacks. Since the cryptographic keys used are not enclave-specific, the MEE can directly protect ELASTICLAVE shared memory regions without modifications. However, it is important to ensure that all shared memory regions are fully within the EPC and hence covered by the MEE. The above discussions offer a starting point for adapting SGX to ELASTICLAVE. We acknowledge that they are not comprehensive and a concrete implementation merits a separate paper.

9 Conclusion

We present ELASTICLAVE, a new TEE memory model for enclaves to selectively and temporarily share memory with other enclaves and the OS. We demonstrate that ELASTICLAVE eliminates expensive data copy and encryption-decryption operations securely. Our ELASTICLAVE prototype on RISC-V offers 1 to 2 orders of magnitude performance improvements over the existing spatial isolation model.

Acknowledgments

We thank our shepherd Jethro Beekman and the anonymous reviewers for their feedback. We thank Aashish Kolluri, Burin Amornpaisannon, Dawn Song, Dayeol Lee, Jialin Li, Li-Shiuan Peh, Roland Yap, Shiqi Shen, Yaswanth Tavva, Yun Chen, and Zhenkai Liang for their feedback on improving earlier drafts of the paper. The authors acknowledge the support from the Crystal Center and Singapore National Research Foundation (SOCure grant NRF2018NCR-NCR002 www.green-ic.org/socure). This material is in part based upon work supported by the National Science Foundation under Grant No. DARPA N66001-15-C-4066 and Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Singapore National Research Foundation, and Crystal Center.

Availability

ELASTICLAVE implementation and benchmarks are available at https://github.com/jasonyu1996/elasticlave.

References

[1] FastCGI Specification | FastCGI –. https://web.archive.org/web/20160119141816/http://www.fastcgi.com/drupal/node/6?q=node/22, 1996.

[2] CGI – Common Gateway Interface. https://www.w3.org/CGI/, 2009.

[3] Heimdallr: Scalable enclaves for modular applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021.

[4] NGINX Docs | NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/, 2021.

[5] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service Using Intel SGX. CCSW'19, 2019.

[6] AppArmor. https://apparmor.net/, 2021.

[7] Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *OSDI*, pages 689–703. USENIX Association, 2016.

[9] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[10] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with customizable and resilient enclaves. *CoRR*, abs/2010.15866, 2020.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[12] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283. USENIX Association, 2014.

[13] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[14] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[15] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure execution. *IBM, IBM Research Division, RC25287 (WAT1205-070)*, pages 1–9, 2012.

[16] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *MICRO*, pages 42–56. ACM, 2019.

[17] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[18] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 319–327. ACM Press, 1994.

[19] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor, 2007.

[21] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 2–13, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *ICS*, 2011.

[23] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[24] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[25] Drew Dean and Alan J. Hu. Fixing races for fun and profit: How to use access(2). In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[26] Adrienne Porter Felt. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, August 2011. USENIX Association.

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

[28] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[29] Iozone filesystem benchmark. http://iozone.org.

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

[31] David Kaplan, Jeremy Powell, and Tom Woller, 2016.

[32] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *ISCA*, pages 29–42. IEEE Computer Society, 2018.

[33] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

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

[35] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177. ACM Press, 2000.

[36] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[37] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: overhead-free IOMMU protection for networking. In *ASPLOS*, pages 301–315. ACM, 2018.

[38] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[39] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.

[41] memcached – a distributed memory object caching system. https://memcached.org/.

[42] Intel releases new technology specification for memory encryption. https://software.intel.com/content/www/us/en/develop/blogs/intel-releases-new-technology-specification-for-memory-encryption.html?wapkw=tme.

[43] Intel® 64 and ia-32 architectures software developer manual, 2018.

[44] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[45] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh. Nested enclave: Supporting fine-grained hierarchical isolation with sgx. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

[46] Penglai-Enclave (IPADS). https://github.com/Penglai-Enclave, 2021.

[47] The risc-v instruction set manual: Volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa.

[48] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *USENIX Security Symposium*, 2020.

[49] SELinux Wiki. http://selinuxproject.org/page/Main_Page, 2021.

[50] Software guard extensions programming reference rev. 2. http://software.intel.com/sites/default/files/329298-002.pdf, October 2014.

[51] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Shweta Shinde, Jinhua Cui, Satyaki Sen, Pinghai Yuan, and Prateek Saxena. Binary compatibility for SGX enclaves. *arXiv*, 2009.01144, 2020.

[53] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. Podarch: Protecting legacy applications with a purely hardware tcb. *National University of Singapore, Tech. Rep*, 2015.

[54] Pradeep S Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–41. Springer, 1992.

[55] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[56] The parsec benchmark suite. https://parsec.cs.princeton.edu/license.htm, 2020.

[57] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171. ACM, 2003.

[58] Intel® trust domain extensions. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[59] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *CoRR*, abs/1810.00602, 2018.

[60] Torch | scientific computing for luajit. http://torch.ch, 2020.

[61] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[62] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 29–46. USENIX Association, 2010.

[63] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37. IEEE Computer Society, 2015.

[64] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel® software guard extensions (intel® SGX) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *SOSP*, 2011.