Data Enclave: A Data-Centric Trusted Execution Environment

Yuanchao Xu†, James Pangia‡, Chencheng Ye‖, Yan Solihin⋆, and Xipeng Shen‡
University of California, Santa Cruz†, North Carolina State University‡,
Huazhong University of Science and Technology‖, University of Central Florida⋆
yxu314@ucsc.edu, jpangia@ncsu.edu, yecc@hust.edu.cn, xshen5@ncsu.edu, Yan.Solihin@ucf.edu

***Abstract*—Trusted Execution Environments (TEEs) protect sensitive applications in the cloud with the minimal trust in the cloud provider. Existing TEEs with integrity protection however lack support for data management primitives, causing data sharing between enclaves either insecure or cumbersome. This paper proposes a new data abstraction for TEEs, *data enclave*. As a data-centric abstraction, data enclave is decoupled from an enclave's existence, is equipped with flexible secure permission controls, and crytographically isolated. It eliminates the hurdles for enclaves to cooperate efficiently, and at the same time, enables dynamic shrinking of the height of integrity tree for performance. This paper presents this new abstraction, its properties, and the architecture support. Experiments on synthetic benchmarks and three real-world applications all show that data enclave can help improve the efficiency of enclaves and inter-enclave cooperations significantly while enhancing the security protection.**

I. Introduction

Trusted Execution Environment (TEE) receives increasing interests in cloud computing as it offers a hardware-protected environment (i.e., an *enclave*) to host security-critical and privacy-preserving applications with minimal trust in the cloud provider [1]–[9]. TEEs isolate program execution and states from the underlying OS, hypervisor, I/O devices, and even individuals with physical access to the machine, providing extensive protection against various attacks.

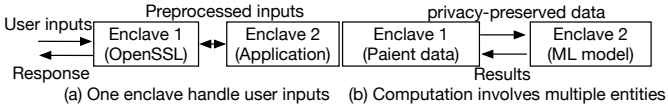

Fig. 1. Examples of uses of multi-enclave data sharing. (a) OpenSSL in nested enclave [10]. (b) Multi-party ML [11]

With TEE popularity, TEE is increasingly used to host increasingly complex code, including entire applications, beyond just security-critical cryptographic functions. With such uses, the inflexibility of current TEE designs becomes problematic [12]–[19]. A key issue is TEEs lack support for data management primitives, which leads to several problems. One problem is the complication of sharing data between enclaves. This is unfortunate because allowing multiple enclaves to share data has been demonstrated to improve security and support multi-entity computation model [10], [11], [20]–[23]. As recent studies showed that attackers could manipulate user inputs and exploit vulnerabilities in an enclave to leak its data [24]–[28], researchers proposed to isolate the code that handles user inputs or third-party libraries (e.g. OpenSSL) in a first enclave, while leaving computation in a second enclave [10], [20], [21](Fig. 1 (a)). By employing this isolation, attacks are limited to the enclave 1, effectively preventing significant data leaks from the enclave 2. In scenarios involving multiple entities, each entity is typically reluctant to share their data or code with others [11], [22], [23]. For instance, a hospital does not want to share patient data to a machine learning (ML) provider that performs diagnoses, and the ML provider does not want to share ML models with the hospital. Using multiple enclaves and privacy-preserving data sharing fulfills these requirements (Fig. 1 (b)).

An ideal data sharing should satisfy several criteria: it should be high-performing (with low overhead access latency to shared data), flexible (allowing users to specify sharing policy and sharers), and secure (providing authentication and uncompromising security). Existing solutions for data sharing satisfy at most one of these criteria. For example, sharing data via public (untrusted) memory [29] requires software encryption and decryption, which incurs high performance overheads and leaves the public memory vulnerable to tampering and replay attacks. Similarly, data sealing [30] allows an enclave to seal data for later use by another enclave from the same developer, but sealing and unsealing require additional encryption, decryption, and integrity verification. Elasticlave [29] permits an enclave's data to be memory-mapped to another enclave, providing low-overhead access to shared data. However, the sharing is not flexible. First, the enclave must specify the ID of the other enclave, which necessitates the other enclave to be instantiated prior to sharing and to pass its ID to the sharing enclave. Moreover, data does not exist independently beyond the enclave; hence, if the sharing enclave terminates, the data disappears. Finally, the security of data sharing is crucial since sharing data across enclaves introduces a new risk where vulnerabilities in one enclave can lead to compromising the other enclave via the shared data. This has not been addressed in prior works.

We argue that the root of the difficulties in managing data in enclaves is the absence of a good data abstraction. Current TEEs only define enclaves as process-centric abstractions akin to processes and threads outside of TEEs. Data has no existence beyond being a part of an enclave. In contrast, outside of TEEs, data abstraction is rich (files, memory maps, etc.), where data can be managed separately from processes. Thus,

in this paper, we propose a new data abstraction for TEEs which we refer to as *Data Enclave*. A data enclave supports several important features. First, data enclave's *existence* is decoupled from an enclave's existence, allowing flexible sharing. An enclave needs to *attach* it before use and can *detach* it after use. When detached, a data enclave continues its existence until deleted. A data enclave has read/write *permission* that can be set for a specific enclave. A data enclave also needs to be *cryptographically isolated* to protect against the same threats as enclaves, i.e. from system software and physical attacks. Hence, a data enclave employs memory encryption and integrity verification, supported with appropriate metadata (encryption counters, message authentication codes, and the integrity tree). The data enclave abstraction also supports primitives for efficiently and flexibly managing data sharing between enclaves.

With the new data-centric abstraction for TEEs, new capabilities are enabled or enhanced. First, primitives for data sharing between enclaves allow for highly efficient cooperation, enabling high performance sharing patterns. Second, data-centric abstraction allows programmers to define independent lifecycles and permissions for data pools while providing secure authentication. Third, detachment of data enclaves enables an optimization to shrink the height of integrity tree to improve enclave performance. Finally, although beyond the scope of this paper, data enclave abstraction can enable future new data management techniques which may include durability, fault tolerance, NUMA optimizations, etc.

The data enclave can be integrated with most TEEs. To demonstrate its advantages, we design it on top of Intel SGX. Our design features an authenticator for verifying attach requests, new instructions supporting sharing primitives, and memory controller logic that enforces access permissions while handling integrity trees for both data enclaves and SGX enclaves. This design accommodates independent lifecycles and dynamic, asymmetric permissions for cooperating enclaves on data enclaves, fitting the needs of various situations.

We model our data enclave abstraction using the Gem5 [31] simulator to evaluate its performance and security, on microbenchmarks, SPEC, and three real-world workloads. Overall, this paper makes the following contributions:

1) We propose a new data-centric abstraction for TEEs, the **Data Enclave**, which provides existence, permission, and isolation for data.
2) We propose a **scalable**, **flexible**, and **secure** data-sharing method for the Data Enclave that enables enclaves to cooperate in computation via shared data.
3) We introduce the **integrity tree attachment/detachment** mechanism to explore security-performance implications.
4) Our evaluation shows that the data enclave achieves a 1-3 orders of magnitude speedup in inter-enclave sharing latency and enhances application performance by 20-179% compared to scalable SGX.

II. Background

This section provides some background on Intel Software Guard Extensions (SGX) as an example TEE. SGX assumes the processor as the Trusted Computing Base (TCB) [32]–[34]. All off-chip resources, including the memory bus, physical memory, privileged software (e.g., OS or hypervisors), and applications, are vulnerable to unauthorized reads or writes by attackers.

SGX shields the execution of code and data in a hardware-protected environment called an *enclave*. In this threat model, SGX Enclaves employ counter mode encryption [35], [36] to provide data confidentiality. During encryption, the ciphertext is generated by XORing a plaintext block (from a written back cacheline) with a One Time Pad (OTP). A per-cacheline counter is incremented on each cacheline write to ensure temporal variation in the encrypted data.

SGX uses Message Authentication Codes (MACs) to provide data integrity, and the integrity tree to detect replay attacks. On every cacheline writeback, the MAC of this block is generated using a cryptographic hash function of the block content, address, counter, and a secure key [36]. On a last-level cache (LLC) miss, the MAC is recomputed and is checked against the stored MAC.

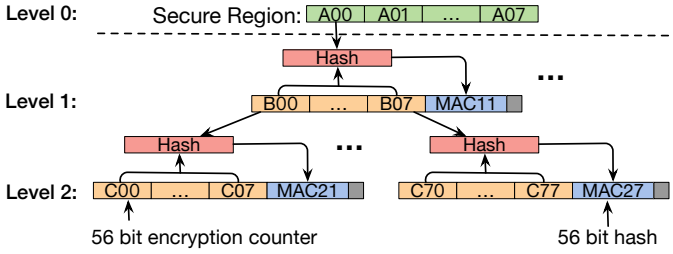

Fig. 2. Intel SGX integrity tree

Replay attacks involve the attacker overwriting the existing tuple {data, MAC, and encryption counter}, which may pass MAC verification. An integrity tree, a structure for ensuring the integrity of a large area of memory with limited on-chip storage, was proposed to detect such attacks [32], [37], [38], as shown in Figure 2. On each access to a counter (e.g., C00), its integrity can be verified by accessing its parent node counter at level 1 (B00) and recomputing the MAC (MAC21). The integrity of the level 1 node can be verified by using the counter at level 0, and so on. The integrity tree root is always stored in the trusted secure region. On a cacheline writeback, the counter for that block and all its ancestor counters are incremented, and corresponding MACs are updated.

The high-level design of Intel SGX is as follows (more details in SGX manuals [39]). SGX partitions the memory into *enclave page cache* (EPC) and non-EPC. When the application allocates a sensitive page, it is mapped in the EPC. SGX realizes its access logic through the handling of TLB misses; hence TLB flushes are needed when the application enters or exits the enclave [33]. A register in the core is added to keep the enclave ID that issues the TLB misses. In the memory

controller (MC), SGX introduces *Processor Reserved Memory Range Registers* (PRMRRs) to distinguish the memory ranges for normal memory and *processor reserved memory* (PRM). Memory Encryption Engine (MEE) in the MC accelerates encryption, MAC, and integrity tree computation.

III. Motivation and Design Principles

The current approaches [13], [29], [30] to supporting data sharing among enclaves are inadequate, as shown in Table I. We will discuss more details.

A classic strategy is SGX-provided *sealing* [30], which permits a new enclave to access sealed data created by either the same developer or the same enclave. Sealing and unsealing are expensive as they require additional encryption, decryption, and integrity verification. Sealing enables a data pool to maintain an independent lifecycle, but it does not allow programmers the flexibility to express enclave-wise sharing schemes or to modify them dynamically. Instead, sharing is restricted to enclaves created by the same developer.

Another approach is to rely on software encryption and decryption [29]. This approach employs public memory as a coordinator for data transmission between two enclaves. As depicted in Figure 3, client and server enclaves, operating with distinct encryption keys, exchange data via public, untrusted memory. Since the coordinator uses public memory, the sending enclave must encrypt data in software for confidentiality, append MACs for integrity, and then place it into the coordinator's memory. This software encryption differs from the hardware encryption used in enclave memory. After the data is copied into the destination enclave, it must be decrypted and verified in software.

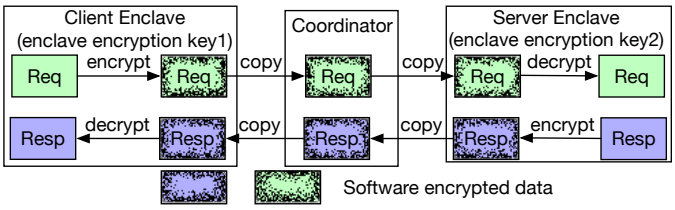

Fig. 3. Client-server data sharing in TEEs.

This data sharing method incurs a high overhead, as it requires multiple trusted/untrusted boundary crossings, resulting in multiple data copying (two for the request and two for the response) and software encryption (one pair for the request and one for the response). Its sharing flexibility is also low due to the lack of data-centric abstraction in TEEs. Current TEEs regard enclave data as part of an enclave, binding its lifecycle and permission to the enclave. This absence hinders programmers' ability to define sharing schemes between enclaves and data pools before enclave creation, define independent lifecycles for data pools, and set associated access permissions for enclaves. Another major weakness of this sharing approach is overlooking the potential emergence of new attack surfaces introduced by sharing. For example, a potential attack surface could allow malicious attackers to create an attacker-controlled enclave, posing as both client and server enclaves, to monitor or modify data-sharing between actual client and server enclaves. This attack is similar to man-in-the-middle attacks [40]. Therefore, robust and secure authentication mechanisms are indispensable to authenticate enclave identities to thwart such threats.

TABLE I
Comparison on different designs for data sharing.

Names	Performance	Flexibility	Security
Software encrypt-decrypt	**Low**	**Low**	**Low**
Elasticlave [29]	**High**	**Low**	**Low**
Sealing [30]	**Low**	**Medium**	**Medium**
Plug-In Enclave [13]	**Read-only code during initialization**		
Data Enclave (this paper)	**High**	**High**	**High**

Elasticlave [29] is a recent solution that allows data from one enclave to be shared with another enclave through a new `share` (X) system call. This call expresses that a consecutive Virtual Address (VA) memory in the caller's enclave is shared with another enclave whose ID is X. To avoid the overhead of encryption and decryption associated with moving data between enclaves, Elasticlave assumes all enclaves use the same encryption key. However, this assumption compromises security as it negates cryptographic isolation between enclaves. Moreover, it is incompatible with current TEEs, such as AMD SEV [41] and Intel TDX [42], which operate under the design that different enclaves employ distinct encryption keys. In terms of flexibility, Elasticlave does not decouple the lifecycle of the shared data and enclave, resulting in three issues: the caller's enclave must know the ID of the other enclave, the other enclave must be instantiated prior to sharing, and data ceases to exist if the caller's enclave terminates.

Finally, Plug-in Enclave [13] allows sharing, but only for read-only code and only during enclave initialization. This design does not support data sharing.

Addressing the shortcomings of prior solutions is essential for the practical adoption of enclaves when data sharing is needed. Specifically, an ideal solution should follow three principles: **high performance**, **flexible sharing**, and **uncompromising security**. The rest of this paper presents the solution we create based on these principles. To the best of our knowledge, this is the first proposal that offers secure data abstraction, decoupled from existing enclave while maintaining the strong security and efficiency.

IV. Overview

A. Data Enclave Abstraction

We highlight two important aspects of our data enclave abstraction: (1) dynamic and asymmetric permission management; (2) secure and flexible sharing primitives. We will refer to regular enclaves as "main enclaves" or simply as "enclaves", in contrast to our new abstraction,"data enclaves".

Our data abstraction is based on the concept of *data enclaves*. A data enclave wraps data that is managed as a single unit in terms of the life cycle, permission, sharing, and isolation. Each data enclave is associated with a *unique key* generated by the processor for encryption. After creation

(and until deletion), its lifetime is independent from the the creating enclave's lifetime. The attachment of data enclaves to main enclaves is N-to-M; a data enclave may be attached to multiple main enclaves, and a main enclave may have multiple data enclaves attached.

Data enclave offers two levels of permissions: the first level pertains to which enclave can attach it (*attach permission*), and the second level pertaining an enclave's ability to read or write a data enclave that it attaches (*access permission*). The attach permission is used for authentication in response to an attach(); the attach() fails if the authentication fails. The access permission is memory access permission made by an enclave to data in an attached data enclave. The access permission is restricted to at most the attach permission.

Dynamic and asymmetric permissions. Existing TEEs can only share data through the untrusted world (Figure 3). Our abstraction allows users to create independent data enclaves and allows main enclaves to have dynamic asymmetric attach permissions on them. Programmers can dynamically grant an attach permission to a main enclave for a data enclave, and revoke it later. Our abstraction also enables low-overhead data sharing: a main enclave can attach and directly access a data enclave after attaching it, eliminating the copying and software encryption through the untrusted world. Furthermore, the asymmetric attach permissions allow a data enclave to support concurrent accesses with different permissions. For example, to support a producer-consumer sharing pattern, write-only attach permission can be given to the producer, while read-only attach permission can be given to the consumer. Once attached by both, a data enclave is mapped to both the address space of the producer and the consumer.

While the attach permission allows for the accommodation of various computational and communicational patterns, the management of access permission facilitates efficient intra-enclave isolation. This mechanism can be utilized to dynamically enable or disable read/write permission, analogous to a protection domain in Intel's Memory Protection Keys (MPK) [43], but is applied at a data enclave granularity level. The change() instruction, as illustrated in Table II, provides such a capability. The ability to associate a data enclave with a protection domain helps prevent accidental disclosure of datum in a data enclave or accidental overwrite of a datum, just as MPK in providing intra-process isolation for a regular process [44].

Flexible, secure data sharing. Our data enclave abstraction allows each data enclave to have an independent lifecycle, thereby allowing programmers to manage the data enclave independently from enclaves. The data enclave allows users to express sharing schemes specifying which enclaves can attach a data enclave before or after launching main enclaves. We design a secure *measurement-based* authentication to verify enclaves' identities when they attach a data enclave. A main enclave can only attach the data enclave if this attachment is permitted in user-defined sharing schemes. By using measurements for authentication, an enclave can share a data enclave to future enclaves (even including the future instance of itself) that are not instantiated yet. Intel SGX relies on measurement for attestation, and here we use a similar mechanism for attach authentication.

TABLE II
DATA ENCLAVE INSTRUCTIONS AND THEIR SEMANTICS

Instructions	Semantics
create(size, deid)	Create a data enclave whose id is deid.
destroy(deid)	Destroy this data enclave.
grant(deid, M, P)	Grant P access permission of this data enclave to a main enclave whose measurement is M.
revoke(deid, M)	Revoke the access of this data enclave to a main enclave whose measurement is M.
attach(deid, P)	The caller main enclave has up to P access permission on this data enclave if authentication passes.
detach (deid)	Remove the caller main enclave access permission on this data enclave
change(deid, P)	Change the caller main enclave access permission on this data enclave (deid) to P.
update(S)	Update caller main enclave's measurement by hashing its measurement and string S.

Data enclave instructions. Table II lists data enclave instructions and their semantics. All instructions can only be executed within a main enclave. Excluding the create() instruction, all other instructions can be successfully issued by main enclaves that attached this data enclave. The update() instruction can be used to prevent attacks from replicating main enclaves. More details is in Section VII-E.

B. Data Enclave Usage Example

Figure 4 illustrates an example of using a data enclave to share data between two main enclaves. Data Enclave 1 has already been created. As shown in Figure 4 (a), the programmer initially sets up the client enclave and the server enclave, and defines sharing schemes: Data Enclave 1 can be shared with a main enclave whose measurement is X (Client Enclave); and a main enclave whose measurement is Y (Server Enclave). The respective permissions for these enclaves are read-write and read-only.

After that, computation of the two enclave begins (Figure 4 (b)). Client Enclave A's attach() can be executed successfully if its measurement is X. After that, its access permission is added to the data enclave local metadata. Server Enclave B also uses its measurement for attach() authentication, and the read-only permission is obtained if authentication passes.

We enable programmers to describe sharing schemes when they set up enclave computation, providing flexible data sharing. These schemes can be modified after computations starts. Throughout this sharing process, no data copy or additional software encryption/decryption is needed. Main enclaves can access data enclave 1's data after passing authentication, accessing data in-place. Programmers can define sharing schemes without requiring data enclaves to be part of an enclave. Authentication verifies the enclave's measurement as the unique identifier during attach(), thwarting potential attacks from attackers' enclaves.

C. Threat Model

We use the threat model similar to SGX [9], [10], [45], [46]. The TCB of our system contains the processor and

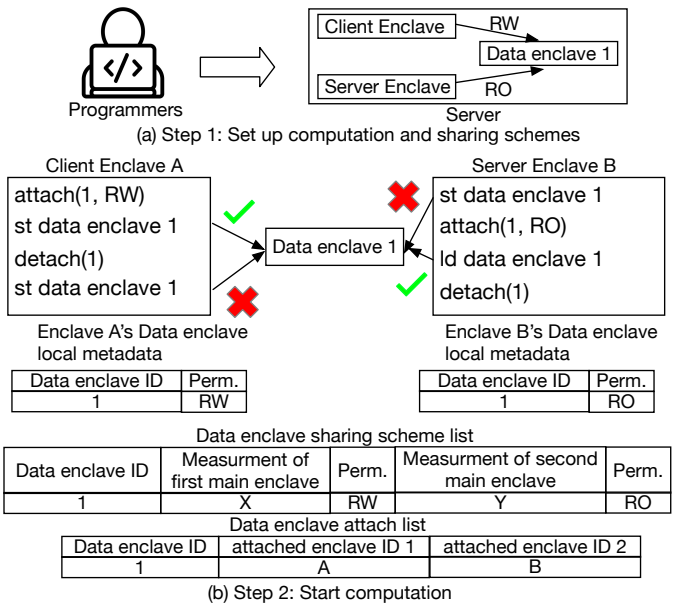

Fig. 4. Data enclave usage example.

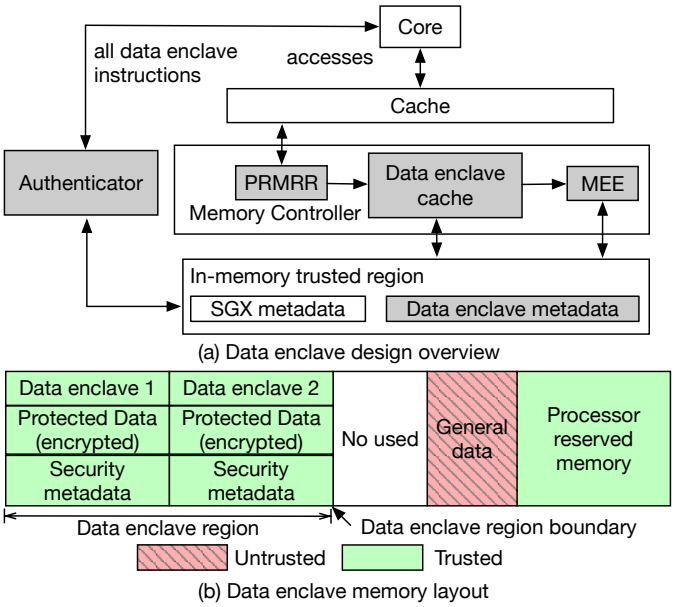

Fig. 5. (a) Data enclave architecture overview, grey parts are modified or new components. (b) Data enclave memory layout.

processor reserved memory. All other hardware (e.g., off-chip memory) and software (e.g., OS and hypervisors) are untrusted. An attacker may have full control of the privileged software (e.g., OS) and application that attempt to compromise data confidentiality and integrity. An adversary could perform a man-in-the-middle attack by intercepting the memory bus and replaying stale data values.

The difference between our threat model and the SGX threat model is that a main enclave can trust other main enclaves as some enclaves may collaboratively compute. This permits users to split an application into multiple collaborating main enclaves. A main enclave only shares data enclaves with other trusted main enclaves. Although important, side-channel attacks [47], [48], CPU bugs [49], and denial-of-service [50] are out of scope of this work.

V. Design

A. Data Enclave Design Overview

Figure 5 (a) provides the overview of data enclave design. Data enclave metadata records sharing information and data enclave runtime information (see Section V-B). We design a new authenticator to handle all data enclave instructions (Section V-C). The new data enclave cache in the memory controller (MC) enforces data enclave access logic (Section V-D). We modify the Memory Encryption Engine (MEE) logic to support integrity tree attaching (Section V-E).

Figure 5 (b) shows memory layout. We partition DRAM into the following regions: the Processor Reserved Memory (PRM) for SGX, the general data region, and the data enclave region. The data enclave region can grow or shrink to accommodate newly allocated data enclaves and release memory for deallocated data enclaves. This region-based design is beneficial for hardware to accelerate normal memory access checks (see Section V-D). All SGX metadata and data enclave metadata are stored in PRM. Each data enclave comprises a contiguous physical memory region.

B. Data Enclave Metadata

The data structures for data enclaves include global metadata and local per main enclave metadata, as shown in Figure 6. The first data structure, *data enclave sharing scheme list*, is a system-wide data structure that records user-defined sharing schemes. We enable programmers to describe sharing schemes when they set up enclaves.

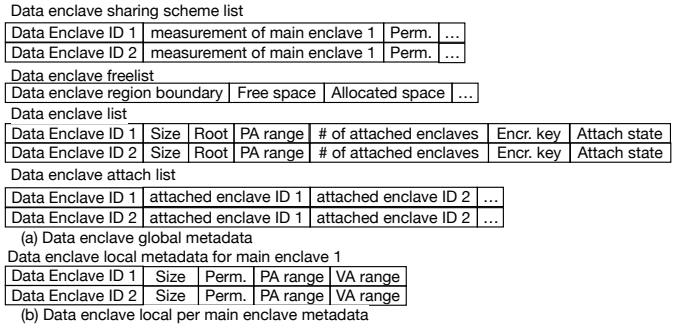

Fig. 6. Data enclave metadata

The second data structure, the *data enclave freelist*, a system-wide data structure that records the *data enclave region* boundary, allocated data enclaves, and free space in the region. The third data structure is the *data enclave list*, a system-wide data structure that records each data enclave information. Each entry is created upon a data enclave creation. The root and encryption key are sealed [30] in memory for confidentiality and integrity. The fourth data structure is the *data enclave attach list*, a system-wide data structure that records attach information of each data enclaves. Each data enclave can be attached to up to 64 main enclaves at the same time. The fifth data structure is the *data enclave local metadata*,a per main enclave data structure that records runtime information of attached data enclaves for a main enclave. Each entry is created upon a successful execution of attach().

C. Data Enclave Authenticator

Data enclave instructions in Table II can be executed in two ways: in hardware or in software. With the former approach, the hardware engine handles the execution of the instructions and manages data enclave metadata (state and sharing information) in hardware tables. However, this approach has a high hardware complexity (especially verification) because the instructions need to read (and modify) multiple metadata. Thus, we explore the latter option, by relying on a special type of enclave to execute the data enclave instructions in software, with data enclave metadata maintained as a software data structure in the special enclave. We refer to the special enclave as the *authenticator*.

We use the authenticator to verify and handle all data enclave instructions issued by main enclaves. The authenticator is implemented to behave like a *daemon main enclave* that is instantiated at system boot and alive for as long as the system is on. Special memory-mapped registers that are OS-inaccessible are added in order for the processor to communicate with the authenticator.

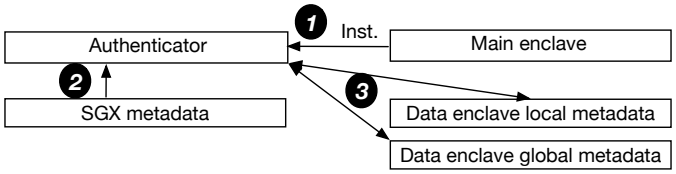

Fig. 7. Workflow of the authenticator

Figure 7 shows the workflow of the authenticator. Suppose a main enclave executes a data enclave instruction (say, to attach a data enclave). The instruction packet is written to a special register that triggers the authenticator, either by the authenticator polling on the register or an interrupt is triggered to switch in the authenticator. The authenticator then retrieves the type of instruction and parameters, and executes it ❶. The authenticator first retrieves the metadata (e.g., measurement) of the caller enclave ❷. According to data enclave metadata and caller enclave measurement, if this instruction passes authentication and is allowed, data enclave metadata is updated accordingly ❸. For our current design, an enclave's measurement includes its code and data memory. We use enclave's measurement as the unique type of an enclave to describe sharing schemes and perform authentication, because it is stored in the trusted computing base and cannot be overwritten. It is practically impossible for attackers to construct another enclave with the same measurement to pass authentication.

Data enclave instructions are not executed frequently, with the exception of change() that modifies a main enclave access permission on a data enclave. There are several works exploring hardware-based intra-enclave isolation designs [10], [51], [52]. Designing efficient intra-enclave isolation is not the focus of our paper, so we use the LIGHTENCLAVE [52] design for change().

D. Data Enclave Access Logic Design

Contrary to the SGX design, which only needs to differentiate between accesses to EPC memory or not. Data enclave design expands the SGX access logic to distinguish among three types of memory access: EPC memory, attached data enclave access, and non-enclave memory, as shown in Fig 8.

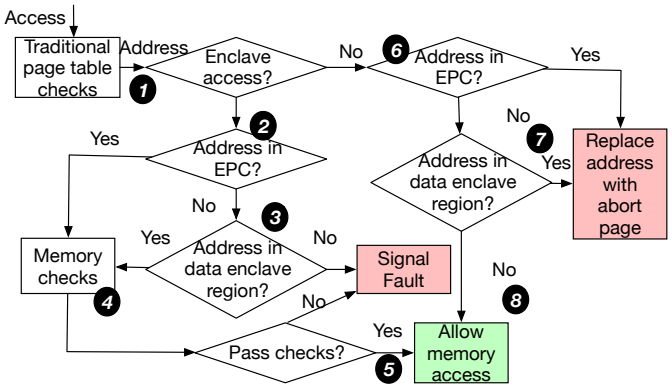

Fig. 8. Data enclave access logic

An access is compared against the PTE permission ❶. If the passed access is from an SGX enclave (i.e., main enclave), its address is compared against the EPC address range ❷ and the data enclave region address range ❸. If this address belongs to any of them, the memory checks are performed for this access. If not, a signal fault is triggered. Memory checks verify integrity by using the MAC and integrity tree verifications ❹. If both verifications pass, this access is allowed. If one of them is not passed, a signal fault is triggered ❺. If this access is not from an SGX enclave, its address is compared against the EPC address range ❻ and the data enclave region address range ❼. If this address belongs to any of them, the access address is replaced with an aborted page. If not, this access is allowed ❽.

To support this access logic, we design a data enclave cache and a data enclave register to record the current address range of the data enclave region in MC to handle TLB misses. Figure 9 illustrates the design and mechanism of the data enclave cache. Each entry has a 36-bit Page Frame Number (PFN) start, a 36-bit PFN end, a 10-bit data enclave ID, a 10-bit SGX enclave IDs, and a 2-bit permissions, and a 2-bit attach state.

The data enclave cache is used in the following way. A TLB miss with an address and an SGX enclave ID uses the address to compare against the PRM range register and the data enclave range register ❶. If the address is in the PRM range and the enclave ID is not 0, which indicates this access is from an SGX enclave and accesses the SGX enclave, the logic performs the checks and constructs the SGX enclave TLB entry if checks passed ❷. If the address is in the data enclave range and the ID is not 0, the address and the SGX enclave ID are compared against data enclave cache entries ❸. If a matching entry is not found, the SGX enclave ID and the address are used to search in memory metadata ❹. The

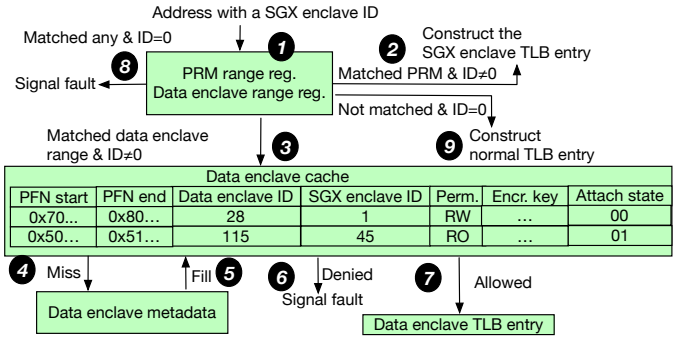

Fig. 9. Data enclave lookaside buffer design and workflow

constructed entry is added to the data enclave cache ❺. If the SGX enclave does not have permission to this data enclave, a fault is generated ❻. Otherwise, the access is allowed, and a data enclave TLB entry is constructed ❼.

For a TLB miss that does not come from an SGX enclave, its SGX enclave ID is 0. To optimize the non-SGX-enclave TLB miss latency in data enclaves, we use a data enclave range register to indicate the address range of the data enclave region. Because all non-SGX-enclave accesses cannot access any data enclave, we do not need to know which data enclave it tries to access by searching the data enclave cache and in-memory metadata. With this design, a TLB miss whose SGX enclave ID is 0 uses the address to compare against two range registers ❶. If the address matches with any of them, indicating that this TLB miss is issued by non-SGX-enclave but tries to access SGX enclaves or data enclaves, a signal fault is generated ❽. If the address does not match with any range registers, meaning that this TLB miss tries to access normal memory, the normal TLB miss handler proceeds to construct a normal TLB entry ❾.

For a last-level cache miss or cacheline write back, the physical address of the cacheline is compared against PFNs in the data enclave cache to retrieve the corresponding encryption key. The Advanced Encryption Standard (AES) encryption engine uses this encryption key to decrypt/encrypt the cacheline.

This design aligns with the existing Intel SGX design, which relies on the TLB to accelerate access checks. Due to the reliance on TLB checking, SGX enclave EENTER or EEXIT issues TLB range flushes for SGX enclave and data enclave region addresses. `Revoke()`, `attach()`, `detach()`, or `change()` changes the permission in metadata and flushes the corresponding data enclave address range in TLB.

E. Integrity Tree Attaching

Intel SGX stores integrity tree roots of enclaves in a limited number of MEE registers as part of the root of trust. When the number of enclave roots exceeds register capacity, the infrequently used roots may be sealed and moved to DRAM. In the data enclave design, many roots of data enclave integrity trees need to be placed into MEE root registers. Frequently sealing and spilling roots to memory could result in significant overheads.

A straightforward solution is to add more registers to accommodate more roots, like PENGLAI enclave [53]. The drawback is that the performance will reduce significantly due to register thrashing, which occurs when the working set of roots exceeds the number of registers. Given various application characteristics, determining a suitable number of registers to avoid thrashing is challenging. We propose attachable integrity trees to enable several integrity trees to form a single root integrity tree to mitigate register thrashing and balance performance and hardware costs.

1) Integrity tree attachment, detachment, and swapping mechanisms: We propose new primitives to attach, detach, and swap the data enclave's integrity tree to or from the main/SGX enclave integrity tree. In order to attach a data enclave's integrity tree to an SGX integrity tree, there needs to be room in the SGX integrity tree. We can increase the SGX integrity tree level to create a new root, and use the second level (root's child) to attach/detach data enclave integrity tree roots to the SGX integrity tree. This can continue until all the new root's children are taken. Beyond that, if there is another data enclave that needs to be attached, we have several options that can be considered (discussed in Section V-E3).

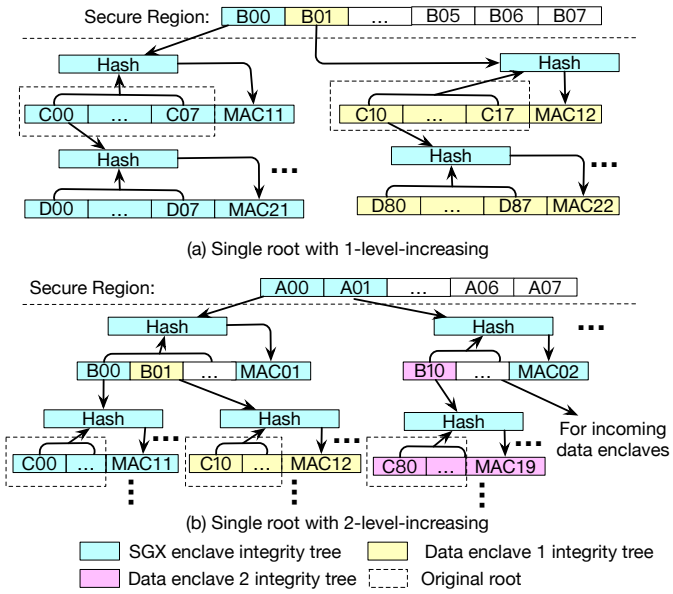

Fig. 10. Integrity tree attachment by adding one level (a) or two levels (b).

Figure 10(a) illustrates integrity tree attachment by adding one tree level. Let us first illustrate adding one tree level. In the figure, C00-C07 is the original SGX integrity tree root, while B00-B07 is the newly created root. We use the SGX integrity tree key to further hash it into a 56-bit B00. Then the new root value is created, with B00 computed from the old root, B01-B07 zeroed, and MAC value (MAC11) computed and stored in the node. Then, the new root is stored in the secure on-chip register, displacing the original root.

After a new SGX enclave root is created, the next step is to integrate the data enclave integrity tree by attaching it. The data enclave original root (C10-C17) is hashed into a 56-bit

string; then, it is put into B01. The MAC value (MAC12) is computed and stored in the node where the data enclave root stays. The SGX integrity tree reuses the existing integrity subtree of this data enclave (D80-D87 and etc.).

Therefore, with the mechanism discussed above, adding one level to the integrity tree accommodates 7 data enclave integrity trees. Figure 10 (b) shows an example that by adding two levels to the original SGX integrity tree, now we can accommodate $8 \times 8 - 1 = 63$ data enclave integrity trees. Generalizing, for an N-ary tree, adding M levels allows us to attach $M^N - 1$ data enclave integrity trees.

The steps of *integrity tree detachment* are as follows. Integrity tree detachment recalculates the hash (MAC12) and verifies data enclave root (C10-C17) integrity until the SGX enclave root. If both verifications pass, the root is sealed and written back to data enclave metadata. Then the data enclave's related nodes are cleaned. Specifically, in the one additional level case, cleaning only needs to set the hashed root (B01) as zero. In the two additional levels case, cleaning needs to set the hashed root (B01) as zero and recalculate the node's MAC (MAC01) and the related root value (A00).

Finally, in addition to the integrity tree attachment, we propose *integrity tree swap*. If there are more data enclaves to attach than the number of nodes in the integrity tree, instead of adding another level to the tree, we could alternatively swap an existing tree with a new one. With the *integrity tree swap* mechanism, we can detach one least recently used (LRU) data enclave root to its data enclave global metadata, and then attach another data enclave integrity tree in its place. Swapping allows us to keep a shallow tree and prioritize keeping hot data enclaves' integrity trees attached. Note that the integrity tree attachment status is not the same as the data enclave attachment status; with swapping, an attached data enclave may have its integrity tree detached.

There are three major benefits of the integrity tree attachment mechanism. (1) This mechanism allows dynamically attaching/detaching the integrity tree, reducing tree height to improve its update/verification performance. (2) This mechanism forms a single root integrity tree, requiring very small hardware modification to the existing SGX design to support it. (3) The integrity tree attachment or detachment only involves several nodes calculation and one memory access, incurring a smaller overhead than sealing.

Integrity tree attachment, detachment, and swapping only affect integrity trees. The access permissions of different SGX enclaves on data enclaves are enforced on handling TLB misses (see Section V-D).

2) Security Metadata Management: In our design, all data enclaves and SGX enclaves use fixed mappings between data blocks and security metadata. When attaching the integrity tree, the integrity tree of the SGX enclave might grow by 1 or 2 levels. The additional metadata resulting from this growth is housed in the SGX enclave's preallocated space. Each SGX enclave statically preallocates 4608 bytes to accommodate metadata stemming from a 2-level growth in the integrity tree. The locations to store additional metadata is also fixed for integrity tree growth by one level or two levels. This preallocated space is only used if the integrity tree is increased by tree attaching.

With integrity tree attachment, the Memory Controller (MC) needs information on how to retrieve security metadata from increase integrity tree. We utilize a 2-bit attach state in data enclave metadata to indicate this. A 00, 01, or 10 state indicates this data enclave's integrity tree is "not attached", "attached to an enclave tree with 1-level growth, and "attached to an enclave tree with 2-level growth", respectively.

3) Design Space of Integrity Tree Roots: With SGX design, MEE has registers for SGX enclave integrity roots, so these roots can be accessed efficiently. It is hard to envision how many registers are needed to accommodate data enclave's roots in different applications. We explore three designs to handle this problem. Figure 11 illustrates the three designs.

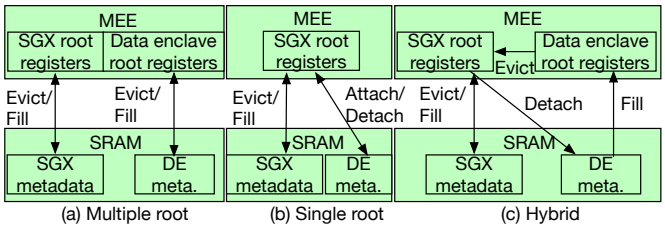

Fig. 11. Three designs of integrity tree roots: (a) Multiple root, (b) Single root, and (c) Hybrid. DE meta. represent data enclave metadata.

The multiple root design (Figure 11(a)) directly adds more root registers in the MEE for data enclaves to store the roots of data enclaves, which is similar to PENGLAI [53]. This design does not increase tree heights for both integrity trees. However, this design suffers from a significant performance drop from frequently evicting and filling roots between registers and memory when there is register thrashing.

On the other hand, the single root design (Figure 11(b)) leverages integrity tree attachment to reuse existing root registers in the SGX design. When a data enclave integrity tree needs to be accessed, this tree is attached to the SGX enclave integrity tree, and the integrity tree detachment is performed if necessary. This design does not need additional registers, while it needs to update one or two more nodes for an enclave write.

The hybrid design (Figure 11 (c)) balances hardware cost and performance by combining the two designs discussed above. When a data enclave integrity tree needs to be accessed, its root is placed in the data enclave root registers initially to achieve better performance. When data enclave root registers are used up, the LRU data enclave root is evicted (i.e., attached) to the SGX enclave integrity tree. We estimate the LRU data enclave by selecting the data enclave is not in data enclave cache. When the SGX integrity tree cannot accommodate a new data enclave root, the LRU data enclave root in the SGX integrity tree is detached and stored in data enclave metadata. In this design, we put hot data enclave roots in root registers and put warm data enclave roots in the SGX integrity tree, avoiding performance degradation of the multiple root design due to limited hardware resources and improving the single root performance.

The increased level could be made dynamically adjustable. Although important, this paper does not focus on profiling and optimizing this parameter. Our evaluation uses the fixed increased level.

F. Other Details

To allocate data in data enclaves, we implemented a simple region-based memory management library for programmers to specify the data enclave ID in the allocation/deallocation calls. As each data enclave space is allocated on creation, the library only needs to maintain a freelist for each data enclave to handle allocation/deallocation.

A data enclave can be swapped out entirely to storage, and then swapped to DRAM when it is attached. When a data enclave is swapped in, preference is given for swapping to the same Physical Address (PA) range that it was attached last; this allows its integrity to be verified on the fly, as the integrity tree root is always in the secure region. If a data enclave is swapped to a different PA range, its integrity is verified first, and then the data is re-encrypted with new PAs.

We assume 1024 as the maximum number of data enclaves, based on the length of the data enclave ID in hardware components. This constraint can be relaxed by extending the number of bits used for the enclave ID.

If two main enclaves attach a data enclave concurrently, the integrity tree of the data enclave is not attached to the integrity trees of any main enclaves. Its metadata update and verification are performed independently.

G. Space Costs

Our design adds the data enclave cache, a data enclave region boundary register and data enclave root registers in MC. Date enclave metadata is the in-memory data structure. The total on-chip storage introduced is 1984 bytes, which is much smaller than the counter cache. The single root design introduces 1040 bytes. The memory space includes the global, per SGX, and per data enclave metadata.

TABLE III
SPACE COST

New on-chip Components	Entry size (bytes)	# of entries	Size (bytes)
Data enclave cache	29.75	32	952
Data enclave root registers	64	16 (0)	1024
Data enclave region boundary	8	1	8

Memory space	Size (bytes)
Global data enclave metadata	32768
Per SGX metadata for data enclave	20992
Per data enclave metadata	256

VI. Evaluation Methodology

Simulator: To evaluate our designs, we built a cycle level simulator with Gem5 v20.1.0.5 [31]. Table IV shows the parameters. Integrity verification of a newly fetched block is overlapped with decryption and data use, similar to prior work [46], [54], [55]. We use separate metadata caches for counters, MACs, and integrity tree nodes. Attach and detach use a 1000-cycle latency as it needs to range flush TLB entries, in line with range flush cost in prior work [56].

TABLE IV
SIMULATOR PARAMETERS

Processor Configuration	
CPU	1 core, OOO, x86 64, 4.00GHz
L1 Cache	8-way, 64KB, 64B block, Access latency: 2 cycles
L2 Cache	512KB, 16-way, 64B block, Access latency: 20 cycles
L3 Cache	4MB, 32-way, 64B block, Access latency: 30 cycles
Memory and Memory Controller Configuration	
Counter Cache	128KB, 8-way, 64B block
MAC Cache	128KB, 8-way, 64B block
Integrity Tree Cache	128KB, 8-way, 64B block
Data enclave cache	Fully associative, 32 entries, access latency: 30 cycles
Root registers	Access latency: 8 cycles
MAC/Encryption/ Integrity tree latency	40 cycles for a cacheline
SRAM/Memory	DDR4 2400MHz, tRCD/tCL/tRP/ tRAS/tWR=14/14/14/32/15ns
Data enclave instructions latency	
attach()/detach() latency	1000 cycles

Workloads: We evaluate data enclave sharing using synthetic benchmarks and three real-world applications. To assess the performance impact of splitting data into multiple data enclaves without sharing, we conduct experiments on SPEC 2006 benchmarks and microbenchmarks.

We construct synthetic benchmarks for two sharing patterns: producer-consumer and client-server patterns. The synthetic benchmarks only include data sharing time and exclude any other application execution time. We evaluate IOZone [57], Recommender [58] and Redis [59] to show performance impacts for real-world applications in data sharing between enclaves. More details are in Section VII-B

We assess the impact of shrinking a large integrity tree into smaller ones by distributing data into different data enclaves using SPEC benchmarks. Each benchmark is executed within an SGX enclave that stores the code, local variables, and small heap data. We designate each heap object larger than 32KB as a data enclave.

In our microbenchmark evaluation, we investigate the impact of shrinking a large integrity tree on different numbers of total data enclaves and in-use data enclaves. Each microbenchmark runs within an SGX enclave that stores the code and local variables. We initialize 128 32MB data enclaves for each microbenchmark data structure. During execution, each microbenchmark performs update operations on a subset of these data enclaves (4, 8, 16, 32, 64) for 100K operations, representing varying numbers of in-use data enclaves. After 100K operations, the microbenchmark detaches the data enclaves of the current subset (i.e., working set) and attaches data enclaves of another subset. The five microbenchmarks are: Linked List (LL), AVL Tree (AVL), Hash Map (HM), B+tree (BT), and Red-black tree (RBT), which are also utilized in other studies [60], [61].

For synthetic benchmarks, we fast-forward the simulator before the data sharing part and then perform detailed simulation. For other benchmarks, we sample 10 billion cycles.

Schemes: We evaluate the following eight schemes. (1) SGXv1, which is SGX version 1 with 256MB EPC. (2)

Scalable SGX (ScaleSGX) is an idealized version of SGXv1 with a large enough PRM to accommodate all enclave data and metadata without page swapping. Both SGXv1 and Scalable SGX perform software measurement and software encryption/decryption when they put or fetch data from public memory. (3) Data enclave hybrid design with adding one level to the integrity tree (Hybrid1 or Data enclave) (4) Data enclave (single root) with one additional level to the integrity tree (SingleRoot1). (5) Data enclave with the multiple root design (MultipleRoot). (6) ElasticOneKey, which is Elasticlave [29] with one encryption key for all enclaves and shared data. (7) ElasticMultiKey, which is Elasticlave modified to use different encryption keys for different enclaves and shared data. (8) ElasticIntegrity, which adds MAC and integrity tree to ElasticMultiKey to provide the same protection level as the data enclave. We implement all Elastic schemes in Gem5 for comparison.

VII. Evaluation

A. Synthetic Benchmark Results

Figure 12 presents the latency results for both producer-consumer sharing and server-client sharing patterns. Data enclave latency is substantially lower than ScaleSGX due to in-place data access without necessitating software encryption/decryption or integrity verification. In the producer-consumer sharing model, the data enclave design achieves a speedup of 100X for 256 bytes and 2000X for 4096KB of data. Additionally, the latency of data sharing through data enclaves remains almost constant across all record sizes. ElasticOneKey has slightly higher overheads (4-10%) than data enclave due to page table manipulation. ElasticMultiKey needs decrypt and re-encrypt the shared data region; thus, it incurs 2.4 to 4.9× higher overheads than data enclave. Importantly, data enclave surpasses ElasticMultiKey's performance while providing a more flexible data abstraction with enhanced capabilities.

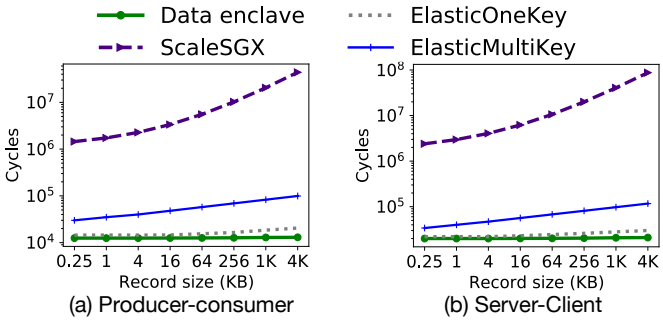

Fig. 12. Performance of the two data sharing patterns.

Figure 13 provides a breakdown of the client-server sharing pattern. For ScaleSGX, the majority of the "other" component is setting up the secure communication channel. When dealing with small record sizes, data copying and the other component consume most of the time. As the record size increases, the overhead of encryption/decryption and integrity verification become more dominant since their latency grows linearly with the request size. Data enclave instructions' latency represent the majority of the sharing time (between 89-94%).

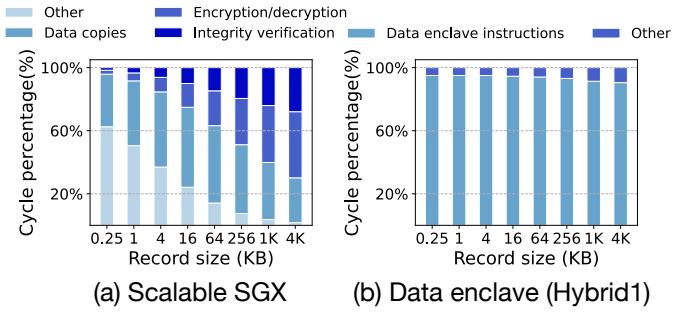

Fig. 13. The breakdowns of the client-server sharing pattern.

B. Real-world Application Results

IOZone (File I/O). IOZone [57] is a benchmark tool for file systems that generates and measures a variety of file operations. We place the IO request generator in an SGX enclave and the file system that handles the requests within another SGX enclave. A data enclave is utilized to transfer requests and data between the two enclaves. This evaluation represents the producer-consumer sharing pattern for writes and the client-server sharing pattern for reads.

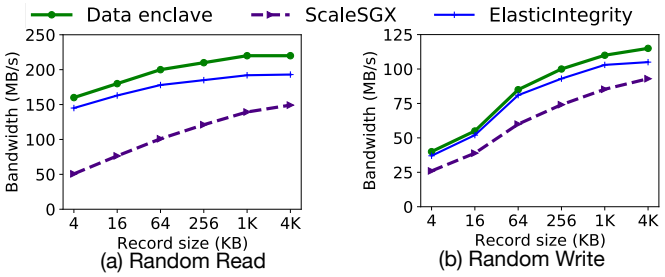

Fig. 14. The performance of read and write.

Figure 14 presents read and write bandwidth on different request sizes. Data enclave offers a 1.5-3.2× speedup over ScaleSGX on read. As the record size increases, the performance gap narrows since the data enclave performance approaches the performance bottleneck of storage. Data enclave exhibits a 1.2-1.6× speedup over ScaleSGX on writes due to the lower frequency of data exchange. ElasticIntegrity is 5-12% slower than data enclave due to additional encryption and decryption.

Recommender (product recommendation service). Recommender [58] is an open-source tool that uses collaborative filtering to suggest products. The tool builds a model based on a user's past behavior, and the behavior is extrapolated from other users to provide highly accurate suggestions. We retrofit the included benchmark to build the model stored in the server enclave. Each user has a data enclave (32MB) to store the personal history of purchases. Each user has its own function to generate new history (1KB/16KB/256KB) stored in its data

enclave. Each user's function is put in an SGX enclave. The server enclave fetches data from user data enclaves in turn.

This application represents the producer-consumer sharing model with multiple producers and one consumer. The server SGX enclave attaches all user data enclaves with read-only permission, while each user SGX enclave attaches its own data enclave with read-write permission.

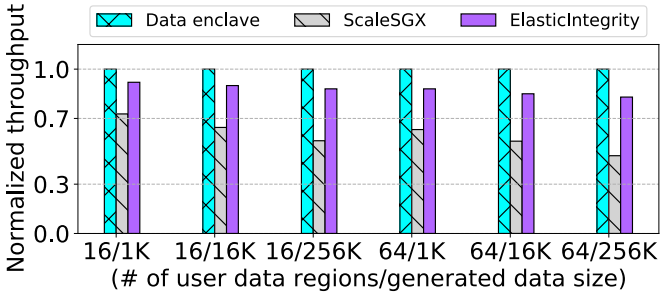

Fig. 15. Normalized throughput of Recommender over data enclave.

Figure 15 results show that the data enclave design outperform ScaleSGX in performance, achieving 1.4-1.8X speedup with 16 user data regions and 1.6-2.1X when increased to 64. Data enclave outperforms ElasticIntegrity by 9-14% for 16 user data regions and 14-21% when expanded to 64. Improved performance stems from the increased data exchanges between server and user SGX enclaves as user numbers grow and the smaller integrity tree size.

Redis (key-value store application). We build a key-value store server based on Redis [59], an open-source key-value store. The client code generate requests, and the Redis server code performs insert/update/search of requests. The server code and data are in one enclave, while the client code and data are in another enclave. The client submits requests to the data enclave request buffer, and the server processes these requests after every (1/4/16) request(s). Once processing is complete, the server places the results in the response buffer for the client to retrieve. ScaleSGX employs two public memory pools for server-client data exchange.

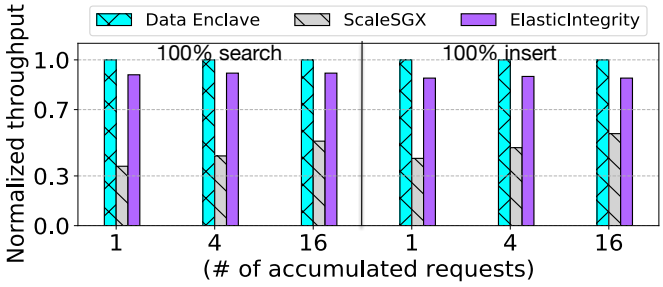

Fig. 16. Normalized throughput of Redis over data enclave.

Figure 16 demonstrates the data enclave design's 81-179% improvement over ScaleSGX and 8-12% over ElasticIntegrity. Greater speedup is shown with fewer accumulated requests due to ScaleSGX's and ElasticIntegrity's more sharing latency.

C. SPEC Results

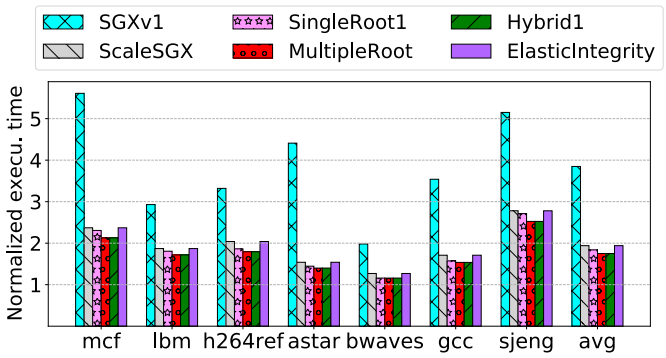

Fig. 17. Execution time of various schemes with SPEC benchmarks, normalized to the non-secure baseline.

Figure 17 shows the execution time of SPEC benchmark results, normalized to the insecure baseline. These benchmarks have 2-23 data enclaves, with an average of 10 data enclaves. SingleRoot1 reduces 5.3% overhead over ScaleSGX on average, because its tree height is almost the same as ScaleSGX. MultipleRoot further improves integrity tree performance by 10.4% from smaller integrity trees. Hybrid1 has almost the same performance as MultipleRoot, while it reduces the need for hardware registers in MEE. ElasticIntegrity has a similar performance as ScalaSGX. SGXv1 incurs more overhead than all schemes, because page swapping from/to EPC dominates its execution time.

D. Microbenchmark Results

Figure 18 shows microbenchmark results, with each figure representing a microbenchmark. The x-axis shows increasing data enclave numbers (working set size), and different lines illustrate various schemes. Each point indicates execution time normalized to a non-enclave system. Lower temporal locality is observed when multiple data enclaves are involved.

Several cross points reveal interesting performance trends. SingleRoot1 outperforms ScaleSGX with smaller working sets due to a reduced integrity tree height. However, when working sets exceed 8 data enclaves, SingleRoot1's performance falls behind ScaleSGX as integrity tree swapping overheads dominate. MultipleRoot outperforms ScaleSGX and SingleRoot1 for working sets of up to 16 data enclaves, given 16 root registers in MEE. When working sets surpass 16 data enclaves, performance declines due to frequent root spilling/restoring. Hybrid1 mitigates degradation by employing integrity tree attachment without extra hardware costs, improving ScaleSGX performance by 13.1% on average and outperforming all other schemes.

E. Security Analysis

For external-to-enclave threats, our design provides the same protection for data enclaves as SGX enclaves. We assume attackers could have full control of the privileged software (e.g., OS) and physical access to the machine. For the assurance of confidentiality, each data enclave is encrypted with

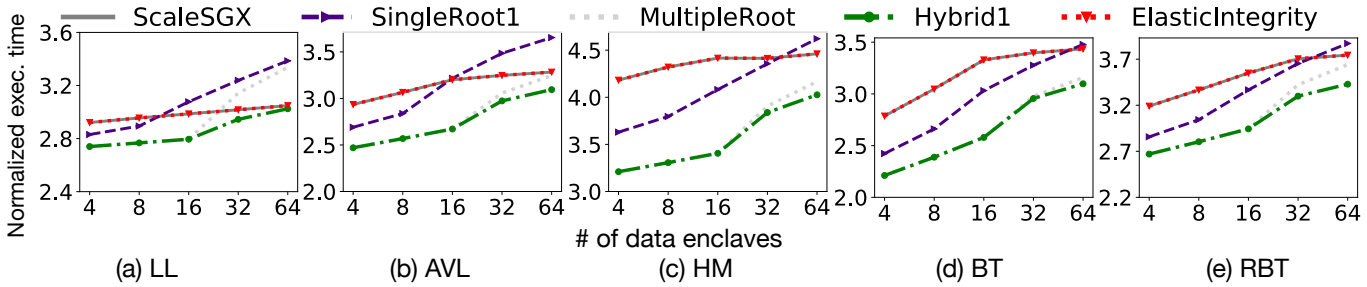

Fig. 18. Normalized execution time of multiple designs on different working set sizes over non-secure baseline.

a unique encryption key known exclusively to the hardware. Integrity is upheld via MACs and an integrity tree for each data enclave. Access control is strictly enforced through our architecture, prohibiting the OS or other SGX enclaves from accessing data or security metadata within a data enclave. The data enclave range register guarantees that only an SGX enclave can construct data enclave TLB entries. The data enclave cache and the in-memory metadata record which SGX enclave can access a data enclave.

Our design effectively prevents three potential risks associated with inter-enclave sharing. The first risk pertains to the attacker's potential initiation of no-plaintext attacks [62], which aims to disclose the victim enclave's data by comparing ciphertext in attacker-controlled data enclaves and victim enclaves. This threat is non-existent if the encryption seed relies on physical addresses, since it is enclave-specific. However, it is still possible that addresses are reused over time. Our design prevents this threat by employing distinct keys for data enclaves and main enclaves, thereby overcoming the vulnerability in the Elasticlave system [29], which assumes one encryption key for all enclaves. The second risk involves the attacker enclave's attach() to pre-existing data enclaves. Our design mitigates this by using the main enclave's measurement for authentication in attach(). Even though the attacker can launch main enclaves and try to attach data enclaves, measurements of main enclaves are securely calculated and stored in TCB and cannot be modified by attackers. It is practically impossible for attackers to create a main enclave that has the same measurement as the victim's main enclave.

The third potential risk from inter-enclave sharing is that attacker may attempt to create a fake main enclave (a replica) by replicating a main enclave's code and data. The attacker can use the replica to attach data enclaves. While the attacker is not able to modify code and data in a replica, with inter-enclave sharing, if the main enclave has non-idempotent operations (i.e., increase a value by 100) on data enclaves, the replica can corrupt data enclave content, e.g. by increasing this value by 100 again, making it different from intended. Such a risk is mitigated by the ability of main enclave measurement to include a secret value, such as a string.

Even though replicating code and data is very difficult, our design can mitigate this risk. As shown in Fig. 19, when the user describes the sharing scheme, the measurement is a hash value of code, data, and a secret ❶. After the initialization of

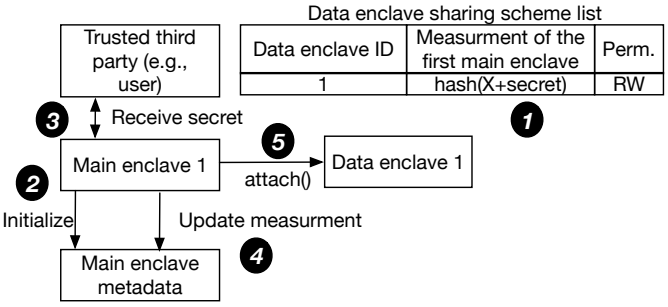

Fig. 19. Example of preventing attacks from replicating a main enclave.

Main Enclave 1, its measurement is the hash value of the code and data, represented as X ❷. After some computation in main enclave 1, it seeks to attach data enclave 1. It first establishes a secure communication channel with a trusted third party, such as the user, to receive a secret (e.g., a 256-byte string) ❸. The main enclave 1 executes the `update` instruction to update its measurement by hashing the current measurement X and the secret ❹. The main enclave can attach data enclave 1 and pass authentication by providing the measurement of code, data, and the secret ❺. Even though the attacker can replicate code and data to create a replica, the attacker has no knowledge of the secret. Therefore, its measurement does not allow attaching data enclave 1. Prior work has studied how to prevent the attacker enclave from pretending to be the user's enclave to build communication channel [63].

VIII. Related Work

We discussed closely related work in Section III. This section discusses other related work.

A rich set of work optimizes integrity trees. PENGLAI [53] introduces more integrity tree root registers to store sub-roots of the integrity trees, such that updates/verifications can stop at sub-roots. VAULT [46] uses a variable arity integrity tree to reduce the memory traffic of the integrity tree. Bonsai Merkle Trees are another type of integrity tree whose MACs only reside in leaf nodes [37]. Gassend et al. [64] proposed to cache tree nodes on the CPU to improve verification performance. Szefer et al. [65] propose a skewed tree that puts frequently accessed locations of memory to the nodes with a shorter path to the root to improve integrity tree performance.

One branch of prior efforts provides intra-enclave isolation. EnclaveDOM [51] and LIGHTENCLAVE [52] propose

combining memory protection keys (MPK) [43] to define domains and their access control for intra-enclave isolation. Nested enclave [10] proposes two-level enclaves where an inner enclave can access the outer enclave, but not vice versa.

Some TEEs do not include integrity protection. They extend the virtual machine/monitor to manage protected memory allocation, like AMD Secure Encrypted Virtualization (SEV) [41], [66], ARM TrustZone-based TEEs [67] and RISC-V-based TEEs [68]–[70]. IceClave [71] adds memory encryption and the integrity tree on top of Trustzone in ARM.

IX. Conclusion

The paper introduces a novel concept called *Data Enclaves* as a solution to the limitations of current TEEs in enabling secure data sharing between enclaves. The proposed architecture designs have minimal hardware requirements and support data enclave logic with attachable integrity trees. Our experimental results demonstrate that using data enclaves can provide a speedup of 1-3 orders of magnitude of data sharing latency over existing TEEs.

Acknowledgment

We thank all the anonymous reviewers whose feedback is helpful for improving the final version of the paper. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1900724, 2106629, CCF-1525609, CNS-1717425, CCF-1703487, and Office of Naval Research (ONR) under grant No. N00014-20-1-2750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[2] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[3] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[4] Chia-Che Tsai, Donald E Porter, and Mona Vij. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[5] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[6] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[7] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[8] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, 2020.

[9] Intel software guard extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Online; accessed Jun, 2022.

[10] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk 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. IEEE, 2020.

[11] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious {Multi-Party} machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, 2016.

[12] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[13] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 306–318. IEEE, 2021.

[14] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[15] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[16] Aakanksha Saha and Sonika Jindal. Emars: efficient management and allocation of resources in serverless. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 827–830. IEEE, 2018.

[17] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. Deadline-aware dynamic resource management in serverless computing environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 483–492. IEEE, 2021.

[18] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, 2018.

[19] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[20] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. {SGXJail}: Defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 353–366, 2019.

[21] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient multi-client isolation under adversarial programs. In *NDSS*, 2021.

[22] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, pages 94–108, 2021.

[23] Wenhao Wang, Yichen Jiang, Qintao Shen, Weihao Huang, Hao Chen, Shuang Wang, XiaoFeng Wang, Haixu Tang, Kai Chen, Kristin Lauter, et al. Toward scalable fully homomorphic encryption through light trusted computing assistance. *arXiv preprint arXiv:1905.07766*, 2019.

[24] Twelve malicious python libraries found and removed from pypi. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/. Online; accessed Jun, 2022.

[25] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 21:34–58, 2014.

[26] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[27] Marion Marschalek. The wolf in sgx clothing. *Bluehat IL (Jan 2018)*, 2018.

[28] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel sgx. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2019.

[29] Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves, 2020.

[30] Intel scalable sealing. https://sgx101.gitbook.io/sgx101/sgx-bootstrap/sealing. Online; accessed Jun, 2022.

[31] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[32] Shay Gueron. A memory encryption engine suitable for general purpose processors. *Cryptology ePrint Archive*, 2016.

[33] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[34] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[35] Helger Lipmaa, Phillip Rogaway, and David Wagner. Ctr-mode encryption. In *First NIST Workshop on Modes of Operation*, volume 39. Citeseer. MD, 2000.

[36] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[37] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196. IEEE, 2007.

[38] G Edward Suh, Dwaine Clarke, Blaise Gasend, Marten Van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 339–350. IEEE, 2003.

[39] Intel sgx development guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf. Online; accessed Jun, 2022.

[40] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE communications surveys & tutorials*, 18(3):2027–2051, 2016.

[41] Amd sev-snp. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Online; accessed Jun, 2022.

[42] Intel. Intel trust domain extensions (intel tdx). *https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html*, 2023.

[43] Intel. Intel 64 and ia-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm. Online; accessed 11 November, 2019.

[44] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 241–254, 2019.

[45] Intel scalable sgx. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html. Online; accessed Jun, 2022.

[46] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[47] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[49] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[50] Anthony D Wood and John A Stankovic. Denial of service in sensor networks. *computer*, 35(10):54–62, 2002.

[51] Marcela S Melara, Michael J Freedman, and Mic Bowman. Enclavedom: Privilege separation for large-tcb applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[52] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. A {Hardware-Software} co-design for efficient {Intra-Enclave} isolation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3129–3145, 2022.

[53] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the {PENGLAI} enclave. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 275–294, 2021.

[54] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. Poisonivy: Safe speculation for secure memory. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[55] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 454–465. IEEE, 2018.

[56] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287. IEEE, 2017.

[57] WilliamD NORCOTT. Iozone filesystem benchmark. *http://www.iozone. org/*, 2003.

[58] Recommender is a c library for product recommendations/suggestions using collaborative filtering (cf). http://ghamrouni.github.io/Recommender/index.html. Online; accessed Sept, 2022.

[59] Redis. https://redis.io/. Online; accessed February, 2022.

[60] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692. IEEE, 2020.

[61] Pengfei Zuo, Yu Hua, and Yuan Xie. Supermem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 479–492, 2019.

[62] Xiang Peng, Peng Zhang, Hengzheng Wei, and Bin Yu. Known-plaintext attack on optical encryption based on double random phase keys. *optics letters*, 31(8):1044–1046, 2006.

[63] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N Asokan. Migrating sgx enclaves with persistent state. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 195–206. IEEE, 2018.

[64] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 295–306. IEEE, 2003.

[65] Jakub Szefer and Sebastian Biedermann. Towards fast hardware memory integrity checking with skewed merkle trees. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2014.

[66] Amd sev. https://developer.amd.com/sev/. Online; accessed Jun, 2022.

[67] Arm trustzone for cortex. https://www.arm.com/technologies/trustzone-for-cortex-a#:~:text=Arm%20TrustZone%20technology%20offers%20an,trust%20based%20on%20PSA%20guidelines. Online; accessed Jun, 2022.

[68] 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, 2016.

[69] 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*, pages 287–305, 2017.

[70] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[71] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211, 2021.