A Novel Memory Management for RISC-V Enclaves

Haonan Li[1,2,*], Weijie Huang[1,2,*], Mingde Ren[2,3], Hongyi Lu[1,2], Zhenyu Ning[1,2,✉], Heming Cui[3], Fengwei Zhang[2,1*†]
{lihn,huangwj2018,12150069,12132884}@mail.sustech.edu.cn,
ningzy@sustech.edu.cn,heming@cs.hku.hk,zhangfw@sustech.edu.cn
[1]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology*
[2]*COMPASS Lab, Department of Computer Science and Engineering, Southern University of Science and Technology*
[3]*The University of Hong Kong*

ABSTRACT

Trusted Execution Environment (TEE) is a popular technology to protect sensitive data and programs. Recent TEEs have proposed the concept of enclaves to execute code processing sensitive data, which cannot be tampered with even by a malicious OS. However, due to hardware limitations and security requirements, existing TEE architectures usually offer limited memory management, such as dynamic memory allocation, defragmentation, etc. In this paper, we present Ashman—a novel software-based memory management extension of TEE on RISC-V, including dynamic memory allocation, migration, and defragmentation. We integrate Ashman into a self-designed TEE and evaluate the performance on a real-world development board. Experimental results have shown that Ashman provides memory management functions similar to native user applications while ensuring enclave security without modifying hardware.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; *Software security engineering*.

KEYWORDS

Memory Management, Enclave, RISC-V

ACM Reference Format:
Haonan Li[1,2,*], Weijie Huang[1,2,*], Mingde Ren[2,3], Hongyi Lu[1,2], Zhenyu Ning[1,2,✉], Heming Cui[3], Fengwei Zhang[2,1]. 2021. A Novel Memory Management for RISC-V Enclaves. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP '21), October 18, 2021, Virtual, CT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3505253.3505257

1 INTRODUCTION

It has been proved that OSs are not trustworthy for running sensitive applications. Therefore, to resolve the urgent needs of trusted computing, researchers and developers have proposed the concept of *Trusted Execution Environment* (TEE) [1, 2, 6, 8, 12–14, 22, 24, 28–30, 32]. Existing TEEs often offer enclaves to support applications running isolated from the host OS [2, 8, 13, 14, 22, 24, 29]. Consequently, users can easily develop enclave applications for trusted computing or services. Intel SGX [2] provides a specific Software Development Kit (SDK) for developers to create applications. However, the functionalities of enclave applications are limited in SGX; for example, dynamic memory allocation is unavailable in SGXv1 [11, 17, 26]. Although SGXv2 [25] proposes a new mechanism to support dynamic memory allocation, the lack of accessibility [21] reduces its usability. Other existing TEEs offer a set of runtime libraries [22] or even microkernels [8, 20] to allow standard *Executable and Linkable Format* (ELF) files running as enclave applications. Unfortunately, although the developers can build enclave applications without using specific SDKs, they still cannot provide full functionalities as a native application (i.e., the application running on the host OS). One of the most significant differences is memory management. Because enclave memory isolation is an essential requirement for TEEs, existing TEEs [14, 22] rely on hardware primitives (e.g., *TrustZone Address Space Controller* (TZASC) [23] on Arm and *Physical Memory Protection* (PMP) [3] on RISC-V) to ensure the memory isolation. However, these hardware primitives can only provide isolation for a limited number of segments of contiguous physical memory regions. As a result, these TEEs cannot provide flexible and efficient memory management since they depend on hardware primitives.

Nevertheless, continuous memory regions are expensive for enclave applications due to memory fragmentation, which refers to the discontinuity between newly requested memory and the previously allocated ones. Hence, TEEs have to use additional hardware primitive configurations to protect the discontinuous memory regions. As a result, once hardware primitives run out, the TEE cannot protect new memory regions. To address this challenge, Keystone [22] enforces the enclave application to request for adjacent memory only. However, in this case, the available memory for enclave applications is then affected by its neighbors. For example, if an enclave application's memory area is completely surrounded by two other ones, it would not be able to request new memory. Moreover, the hardware primitives also limit the number of concurrent enclave applications. For example, Keystone needs to assign a set of PMP entries to each enclave application, so the number of concurrent enclave applications is limited by the total number of PMP entries [22]. SANCTUARY [14], for another example, also has a limitation on the maximum number of running enclaves due to the hardware constraint of TZASC.

*Both authors contributed equally to this paper.
†Zhenyu Ning is the corresponding author.

HASP '21, October 18, 2021, Virtual, CT, USA

ACM ISBN 978-1-4503-9614-1/21/10…$15.00
https://doi.org/10.1145/3505253.3505257

To address the memory management restriction, Penglai [16], for example, modifies hardware ranter than utilize generic hardware primitives to protect enclave memory. Nevertheless, significant modifications to the hardware reduce its deployability. Furthermore, our goal is to construct an extension for existing TEEs, and we expect our solution to be based on generic hardware primitives without hardware modifications.

In this paper, we propose a flexible and efficient memory management mechanism named Ashman as an extension for TEEs on RISC-V. Based on the standard hardware primitives that protect the enclave memory regions, Ashman provides memory allocation and defragmentation algorithms for TEEs to manage memory. We implement Ashman on top of our self-designed TEE Coffer [7] with 411 *lines of code* (LoC). Then we evaluate Ashman on an Allwinner D1 Nezha development board [10] and QEMU emulation. The experiment shows that Coffer can support more than 100 concurrent enclaves with at least 149% improvement in memory utilization.

We summarize our contributions as:

- We break the constraint on the number of concurrent enclaves due to the hardware-based memory protection mechanism. The experiment shows that Ashman supports hundreds of concurrent enclaves without hardware modifications.
- We significantly increase the memory utilization of enclaves. Random high-pressure memory allocation tests illustrate that Ashman improves the utilization rate by at least 149%.
- We propose new algorithms of memory allocation and memory defragmentation with a low performance overhead. The experiment with the RV8 benchmark [5] demonstrates that Ashman introduces a 3.49% overhead in the worst case.
- We open-source our code of Ashman and experimental results on `github.com/Compass-All/Ashman` to facilitate further research.

The rest of the paper is structured as follows: Section 2 provides the background and related work of memory management in RISC-V and state-of-the-art TEEs. Section 3 introduces our design of the proposed memory management approach. Implementation details are listed in Section 4, and Section 5 shows the evaluation results of the approach. Section 6 discusses other probabilities in this approach and future work. The conclusion lies in Section 7.

2 BACKGROUND & RELATED WORK

2.1 RISC-V Security Mechanisms

2.1.1 RISC-V Privileges. The RISC-V standard [3] defines three software privilege levels: *User Mode* (U-mode), *Supervisor Mode* (S-mode), and *Machine Mode* (M-mode). Each processor can only run under a particular privilege level at a time. The privilege levels determine permissions for several operations, such as accessing specific memory, manipulating certain registers, etc. In general, user applications run in U-mode, the OS kernel or runtime lies in S-mode, and the firmware resides in M-mode together with the bootloader.

2.1.2 Physical Memory Protection (PMP). RISC-V introduces PMP to control access to the physical address space. Specifically, a PMP entry can be configured for a contiguous section of physical memory. The number of PMP entries is implementation-specific. For example, SiFive U74-MC [31] supports 8 entries, whereas the generic virtual platform on QEMU supports 16 entries [9].

2.2 Memory Management in Existing TEEs

Keystone [22] relies on Linux's memory allocation mechanisms (i.e., Buddy Allocator and the Contiguous Memory Allocator) to create and dynamically allocate the enclave memory regions. Each enclave application is discretely located in memory and requires a separate PMP entry to protect its memory area. Therefore, Keystone only supports a limited number of enclave applications. Besides, once an enclave is allocated, it cannot be migrated elsewhere by Linux. Hence, if there is insufficient memory adjacent to the enclave, the enclave application may fail to continue.

In the past few years, existing state-of-the-art TEEs have proposed the following strategies to resolve the memory management issues:

Additional address translation. vTZ [18] on Arm maintains an additional translation in hypervisor mode to support *intermediate physical addresses* (*ipa*) between *virtual addresses* (*va*) and *physical addresses* (*pa*). This way, it can configure the *ipa* space to utilize both contiguous memory protection in hardware and dynamic memory management in TEE-kernel. However, the hypervisor extension in RISC-V is not supported so far by real-world development boards. Therefore, the approach cannot be applied to RISC-V machines.

Customized hardware. On enclave initialization, CURE [13] depends on Contiguous Memory Allocator as well. The maximum size of the enclave memory is fixed, and dynamic memory allocation for enclaves is supported by either host OS (for *User-space* enclaves) or the enclave's own runtime (for *Kernel-space* enclaves). Memory accesses in CURE are controlled by modified arbiters on the system bus. Due to the hardware limitations on the arbiter, CURE can only support 13 enclaves concurrently with limited memory management.

TIMBER-V [24] utilizes tagged memory, which extends memory words with additional bits for metadata, to distinguish between the normal and secure memory. Additionally, an enclave can interleave normal memory regions to reduce memory fragmentation. By this way, TIMBER-V is able to provide dynamic memory management as well as unlimited number of concurrent enclaves. However, such features are based on additional customized hardware primitives, sacrificing the nativeness of enclave applications.

Penglai [16] also manages secure memory via the tagging mechanism. It relies on a modified allocator in Linux to manage host OS memory and enclave memory in unit pages. To ensure strict permission control, Penglai introduces self-designed registers and page table walkers to distinguish between host and enclave pages, bringing heavy modifications to hardware.

3 DESIGN

3.1 Overview

Figure 1 offers a brief overview of a generic TEE on RISC-V. The S-mode runtime, isolated from the host OS, provides essential supports for enclave applications to execute. It also maintains page

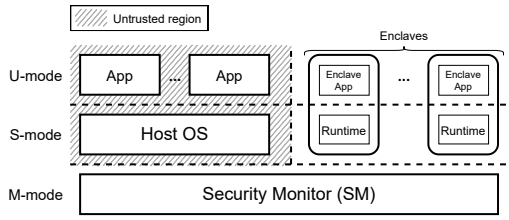


Figure 1: TEE Overview on RISC-V

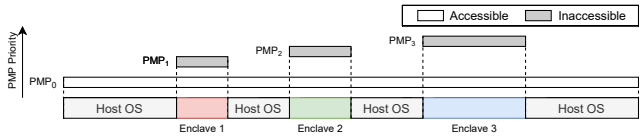


Figure 2: Existing Solution: Protecting Enclaves by Multiple PMP Entries in Host OS

tables for itself and the enclave application. The *Security Monitor* (SM) runs in M-mode to configure memory for enclaves. It also utilizes PMP to ensure memory isolation.

Our design principles are to provide a memory management model for enclaves similar to that of native user applications. Thus, we need to achieve the following goals beyond the essential requirements of security:

G1: Breaking the restriction on the number of concurrent enclaves. Generally, a PMP entry can only protect a contiguous memory region. As Figure 2 shows, the TEE protects each enclave with a unique PMP entry when the processor lies in the host OS. Therefore, the number of concurrent enclaves is limited by the number of PMP entries. One of our goals is to provide a memory management method to overcome the limit imposed by PMP.

G2: Making full use of physical resources for enclave memory support. Typically, a native user application may request memory resources at any time and can always obtain new memory regions whenever resources are sufficient. Thus, we need a dynamic memory allocation mechanism to satisfy the needs of enclave applications. Meanwhile, repeated allocations and deallocations may cause external memory fragmentation leading to difficulties in further allocation and unbearable memory waste. Therefore, an efficient countermeasure against fragmentation is our goal as well.

3.2 Memory Access Control

We should take measures to manage enclave memory securely while satisfying **G1**. That is, we need to ensure the isolation of enclaves without limiting the number of concurrent enclaves.

The isolation of enclaves consists of two parts: 1) a running enclave only has access to its own memory regions, and 2) the enclave memory should be invisible to the host OS. As Figure 3(a) demonstrates, when an enclave is executing, we only need to configure one PMP entry on its own memory region. In this way, the enclave can only see its own memory, and any access to unconfigured memory regions will be denied. When the processor is in the host OS, we need to ensure the sufficient number of PMP entries. We enforce

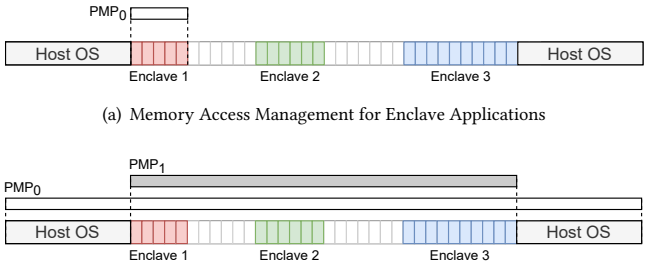


Figure 3: Ashman's Access Control in Different Scenarios

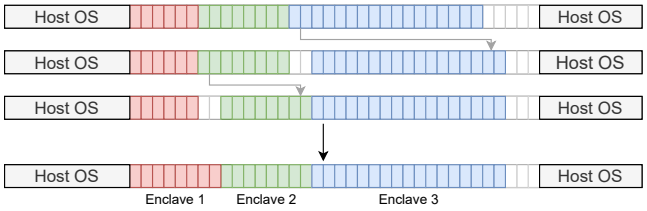


Figure 4: Multiple Migrations for Dynamic Memory Allocation

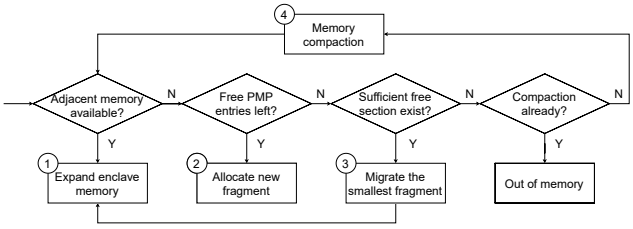


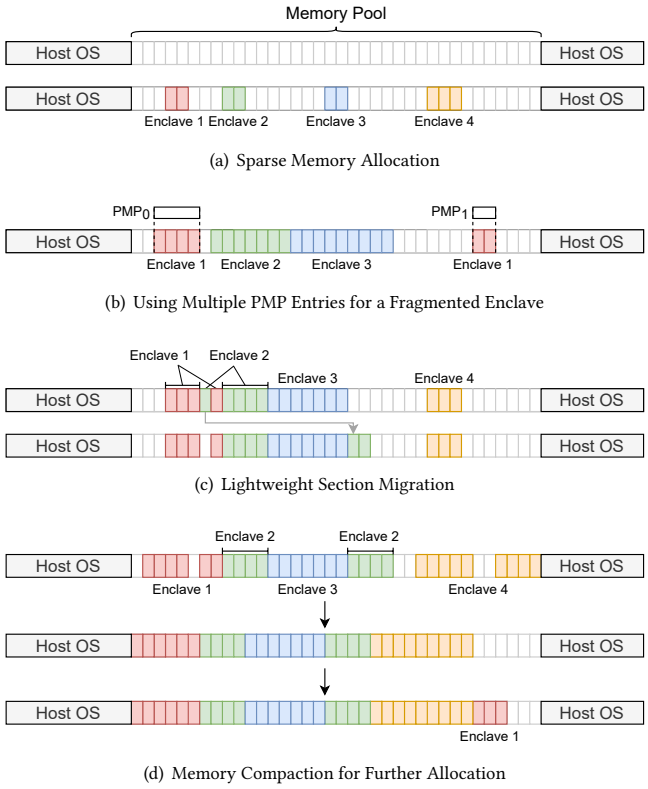
Figure 5: Flow Chart of Memory Allocation Algorithm

the host OS not to occupy memory regions between enclaves so that only one PMP entry is sufficient to protect all enclave memory. This approach naturally breaks restrictions on the number of concurrent enclaves.

3.3 Dynamic Memory Allocation

3.3.1 Naive Solution. To fulfill the requirements of **G2**, we need to make full use of PMP to utilize all memory regions for the host OS and enclaves. According to Figure 3, there exist unused memory regions between enclaves. To fully make use of these regions, we introduce a defragmentation approach for enclave memory.

For the most straightforward intuition, suppose we can only use one PMP entry to protect the physical memory of the enclave application. It means that we must ensure that all memory in an enclave is contiguous. Thus, all memory partitions are naturally used by enclave applications. However, as Figure 4 shows, even if Enclave 1 only needs two partitions, all other enclaves may be migrated to make room for the allocation. Note that each migration also requires rebuilding the page table of the migrated partitions; the overhead associated with frequent memory migrations is unacceptable.



(a) Sparse Memory Allocation

(b) Using Multiple PMP Entries for a Fragmented Enclave

(c) Lightweight Section Migration

(d) Memory Compaction for Further Allocation

Figure 6: Strategies of Memory Allocation

3.3.2 Complete Solution. To reduce the overhead, we investigate different cases of memory allocation carefully. The flow chart of the whole process is illustrated in Figure 5, and the entire process is depicted in Appendix A.

Case 1. Intuitively, if free adjacent memory is sufficient, Ashman expands the enclave memory contiguously. Furthermore, to make each enclave application's surrounding more likely to be free, Ashman allocates enclave memory regions sparsely in the reserved region, which is shown in Figure 6(a).

Case 2. If Ashman finds that there is not enough free space adjacent to an enclave application, Ashman puts the requested memory into any free area. In other words, Ashman allows fragmented enclave memory and utilizes multiple PMP entries when the enclave is running, as illustrated in Figure 6(b). We observe that in enclave contexts, Ashman only needs to protect the running one, so multiple PMP entries can be adopted to fulfill the protection for multiple memory fragments. Therefore, Ashman allows fragments as many as PMP entries for each enclave application.

Case 3. Further, when PMP entries run out, Ashman performs a lightweight migration strategy that moves the smallest fragment to a free memory region, as demonstrated in Figure 6(c). Note that Ashman only migrates the smallest fragment to minimize overhead. After the migration, the migrated fragment can expand the memory in the new area, as states in **Case 1**. To maintain the sparsity of the allocated regions, Ashman always chooses the largest free memory region for migration. In addition, the migration destination will be as close to the midpoint of the free memory as possible.

Case 4. Memory compaction is the rearrangement of memory for all enclave applications. It happens when Ashman does not find any available memory to perform **Case 3**. Ashman firstly identifies whether memory compaction has been performed previously in this allocation. A positive result indicates that there is no more space available at this point; otherwise, Ashman performs memory compaction.

Figure 6(d) depicts the process of memory compaction. In general, Ashman moves all allocated memory to one side; therefore, all possible free memory is merged to the other side. This way, all free memory in the memory pool can be used for memory allocation.

4 IMPLEMENTATION

In this section, we describe the implementation of Ashman in terms of two scenarios of memory allocation and migration. We integrate Ashman into our self-designed TEE Coffer [7] and deploy it on a real-world development board: Allwinner D1 Nezha board with T-HEAD C906 single core and 2GB RAM. Ashman is implemented on top of OpenSBI, which is the standard firmware for the RISC-V platform. The version we are using is OpenSBI v0.9 [4].

4.1 Memory Allocation

4.1.1 Memory Initialization. At first, we need to reserve a physical memory region for enclaves. To meet the isolation requirement, we modify Linux's memory allocator to exclude memory regions for enclaves. Specifically, at Linux boot time, we set the migrate type of most of the pages (i.e., all pages except those necessary for Linux kernel) managed by Linux to `MIGRATE_ISOLATE` [27], which enforces the Linux memory allocator not make use of these pages. The SM manages the reserved memory as the enclave memory pool.

To minimize the impact to host OS brought by the TEE, we allow Linux to utilize some of these pages temporarily. To be specific, we modify Linux's out-of-memory exception handler. When Linux runs out of memory, it will try to request a memory section from the SM. The SM checks whether the boundary section (i.e., the boundary between the enclaves and the host OS) is occupied. If so, the SM will perform memory migration, and the boundary section is allocated to Linux. Finally, The SM releases the corresponding PMP restriction so that Linux can access that memory region.

4.1.2 Memory Partition. The minimal memory management unit in Linux is one-page large (i.e., 4KB in size). Here, we introduce *memory partition* and increase the smallest memory management unit to 8MB in size. This way, Ashman assigns at least 8MB large memory for each allocation request. Although memory partition introduces internal fragmentation, it significantly reduces the number and overhead of free memory allocations and migrations.

4.1.3 Enclave Memory Paging. The TEE runtime and the enclave memory need different address spaces. The runtime needs to take charge of all available memory, so we firstly apply a simple mapping rule for the whole enclave memory by adding an offset to each physical address. Then, we perform an extra mapping on memory regions used by the enclave application. Besides, since the mapping rule for the enclave application is unordered, we record each

mapping to the inverse map. Therefore, the update of page tables during migration can be either located by the offset or the inverse map. Details of memory migration and inverse map are explained in Section 4.2.

4.1.4 Dynamic Memory Allocation. To support dynamic memory allocation, we maintain a memory pool inside the runtime for each enclave upon creation, which holds page information of unused memory in the allocated partition(s). When an enclave application requests memory, the runtime first checks whether the application memory pool can provide sufficient pages. If the pool runs out of memory, the SM allocates a new partition following the procedure described in Section 3.3 and informs the runtime of the corresponding information. The runtime then sets up the page table and refills the application memory pool for enclave applications.

4.2 Memory Migration

4.2.1 Migration Overview. The memory migration mechanism is the core part of our solution to enclave memory fragmentation. The migration process consists of building up the new partition and tearing down the old one. The SM first assigns the destination partition to the enclave, then copies the entire content over. Finally, the source partition will be zeroed and freed.

4.2.2 Updating Page Table. Enclave applications consistently access memory via virtual addresses. During the migration process, the physical address of the content has been changed, so we need to update the memory mapping for the migrated part to make sure that virtual addresses are correctly mapped to the new physical addresses.

As introduced above, we have two memory mappings inside one enclave for the runtime and enclave application. For runtime pages, the SM only needs to update the address offset because of the simple mapping strategy. As for enclave application pages, one intuitive approach is to traverse all page table entries and update the migrated physical address. This approach brings great overhead when the enclave is large, so we introduce the *inverse map* mechanism below to increase updating efficiency.

4.2.3 Inverse Map. By observations of our allocation algorithm, it can be implied that migrations usually take place in the smallest partition of an enclave. Thus, we can obtain the virtual address corresponding to physical memory in the migrated fragment so that the necessary page tables are updated.

We achieve this goal by introducing an *inverse map* to hold the physical-to-virtual correspondence. Before an enclave application starts, the runtime creates an inverse map with initial information and informs the SM. When the enclave is running, the inverse map will be updated along with the page table. If a partition needs migration, the SM looks up the inverse map for virtual addresses for page table updates so that the overhead is a constant.

Moreover, we reduce the size of the inverse map. In our design, the inverse map contains triples of the form (pa, va, n), where pa, va represent physical and virtual address and n marks the number of adjacent pages beginning with pa. The usage of n significantly reduces the size of the inverse map while ensuring the efficiency of indexing. Typically, for an enclave with 256MB memory, around 20 entries are sufficient in the inverse map.

Table 1: LoCs of Components in Ashman

Components	Lines of Code
Memory Pool Operations	40
Memory Allocation	178
Memory Migration	164
Memory Compaction	29
Total	411

5 EVALUATION

In this section, we first describe our experiment setup and then evaluate our design following three research questions:

- **RQ1: Does Ashman significantly increase the *Trusted Computing Base* (TCB)?**
- **RQ2: How many of the concurrent enclave applications can Ashman support?**
- **RQ3: Can Ashman make use of available memory as much as possible?**
- **RQ4: What efficiency can Ashman gain when performing memory migration and crunching?**

5.1 Experiment Setup

We prototype Ashman on Allwinner's D1 Nezha board (T-HEAD C906 single core, 2GB RAM), and QEMU (single-core, 4GB RAM) based on TEE architecture Coffer [7]. For D1 Nezha, the host OS is based on Linux 5.4; for QEMU, the host OS is based on Linux 5.13. Both require a lightweight modification (< 30 LoC) on Linux's *Out Of Memory* (OOM) handler.

We first evaluate the LoC of each component of Ashman to answer **RQ1**. Then we utilize QEMU of large memory size to study **RQ2** and **RQ3**, which are hardware independent. Last, we measure the performance overhead on the D1 Nezha board to answer **RQ4**.

5.2 TCB

Table 1 shows the *lines of code*s (LoCs) for each component of Ashman. Compared to the firmware Ashman based on, which consists of 16,605 LoCs, the added TCB of Ashman is small. Therefore, we answer the **RQ1** and Ashman is not supposed to increase the TCB of TEE desgin significantly.

5.3 Maximal Enclave Concurrency

In our prototype with QEMU, Ashman enables Coffer to successfully support up to 128 concurrent enclaves under a memory pool of 1GB. The upper limit of 128 here is because the Coffer implementation requires each enclave application to consume at least 8MB. Therefore, the maximal number of enclaves can be high on a device where the memory pool is large. Theoretically, given a device with enough memory, there is no limit on the number of concurrent enclave applications in Ashman.

Table 2 shows a comparison of different TEE architectures on the RISC-V platform. For other existing TEE platforms, there is either an upper limit of concurrent enclave number, or the architecture requires additional hardware other than standard PMP. Ashman

Table 2: Comparison of the Number of Concurrent Enclaves in Different TEEs on RISC-V

Name	# of Concurrent Enclaves	No Hardware Modification
Keystone [22]	# of PMPs	✓
CURE [13]	13	✗
Sanctum [15]	# of DRAM regions	✗
TIMBER-V [24]	Not limited	✗
Penglai [16]	Not limited	✗
Coffer with Ashman	Not limited	✓

supports a scalable number of enclaves without any hardware modification. Therefore, **RQ2** is answered.

5.4 Memory Utilization

Memory utilization characterizes how physical memory is actually used by enclave applications. The memory utilization rate is calculated as M_{used}/M_{pool}, where M_{used} is the memory used by an enclave and M_{pool} is the size of the memory pool. It should be noted that inner fragments are not taken into account in M_{used}. Inner fragments appear when an enclave allocates a memory partition but does not use it up.

To precisely evaluate memory utilization in Ashman, we design an experiment with concurrent enclaves keeping allocating memory from the pool. The concurrent enclaves are designed to have similar behaviors to real-world applications. They allocate memory of random sizes in turn. The size for each allocation ranges from 1MB to 128MB. We perform the experiment on Coffer with/without memory migration. Without memory migration, the allocating procedure has to stop when one of the enclaves fails to allocate memory from the pool. We calculated the memory utilization at this moment. With memory migration, the procedure carries on until all memory partitions are allocated by the enclave. The pool size in the experiment is chosen to be 256MB, 512MB, 768MB, and 1GB and the number of concurrent enclaves ranges from 16 to 128.

The experiment results are shown in Figure 7. The figure shows memory utilization rate improvement for different numbers of concurrent enclaves and different sizes of the memory pool. The dashed lines represent the memory utilization rate without Ashman, ranging from $14\% to 43\%$. The solid lines represent the memory utilization rate with Ashman, which ranges from 39% to 89%. Among different cases, Ashman improves the memory utilization rate by $149\% \sim 516\%$. Note that Ashman does not provide 100% memory utilization due to the minimum management unit of memory (memory partition). As the number of enclave applications increases, more memory regions are allocated to enclave applications while not used. When the number reaches 128, they consume 1GB of memory for creation. Subsequently, memory requests from either enclave application fail, but still much memory without in-use.

The experiment shows that Ashman significantly improves memory utilization. Therefore, we have answered **RQ3** that Ashman enables Coffer to make full use of physical resources.

5.5 Performance Overhead

To evaluate the performance overhead brought by our proposed enclave memory management, we focus on two aspects: (1) the efficiency of memory allocation and (2) the overhead of memory migration.

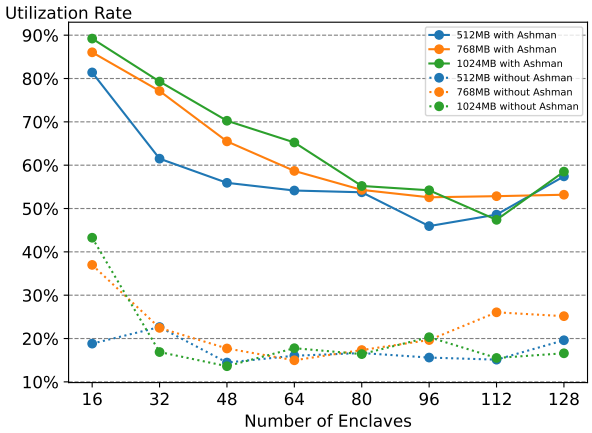


Figure 7: Memory Utilization Rate

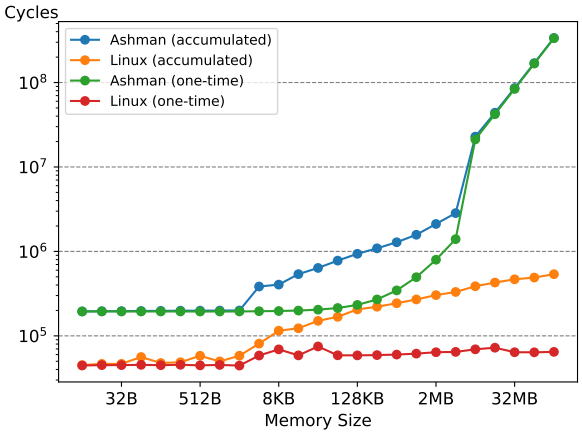


Figure 8: Memory Allocation Cycles

5.5.1 Memory Allocation Efficiency. We evaluate the efficiency by measuring: (a) the time consumption of an enclave allocating different sizes of memory, and (b) the memory allocation time as a percentage of total execution time for enclaves. The experiments are performed on a D1 Nezha development board.

For (a), we design two memory allocation tasks to run both in the enclave and the host OS and compare the time consumption measured in cycles. The first task is to simply allocate a given size of memory all at once, whereas the second is to allocate different sizes of memory randomly until the total amount reaches a given value. The experimental results are shown in Figure 8. The green/blue line stands for one-time/random-accumulated allocation in Ashman, respectively, and the orange/red line corresponds to one-time/random-accumulated allocation in host OS, respectively.

Figure 8 shows two turning points of time consumption in Ashman. The turning points are at 4KB and 8MB, corresponding to page size and memory partition size, respectively. They appear due

Table 3: Memory Allocation Time in RV8 Benchmarks

Name	qsort	aes	miniz	norx	primes
Size	193MB	98MB	27MB	98MB	6MB
Cycles	5.02×10^8	2.52×10^8	6.42×10^7	2.52×10^8	1.39×10^6
Percentage	3.49%	1.61%	0.15%	1.56%	0.01%

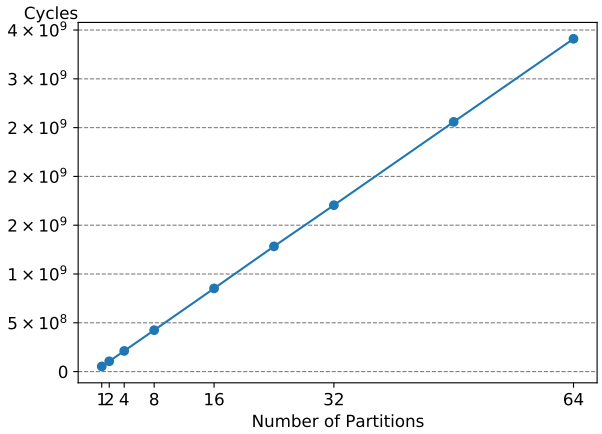


Figure 9: Time Consumption of Memory Migration

to the following reasons. When the requested memory size is below 4KB, the system call `brk` is not necessarily invoked. Otherwise, the enclave runtime handles `brk`, resulting in context switch overhead between U-mode and S-mode. When the requested size is larger than 8MB, the enclave runtime further requests the SM for one more memory partition, which additionally brings overhead caused by S- to M-mode context switches.

The allocation efficiency in the enclave is lower than that in the host OS for two main reasons. First, for a memory region larger than 8MB, a switch into M-mode is necessary for Ashman. Second, the enclave runtime utilizes a simpler memory management algorithm than the host OS to reduce the TCB.

As for (b), we measure the memory allocation time in RV8 benchmarks [5]. The experiment results are listed in Table 3. During the entire run, the memory allocation only takes 0.01% ∼ 3.49% of the total execution time. Therefore, Ashman does not add much performance overhead to the enclave.

5.5.2 Memory Migration Overhead. In the following, we evaluate the overhead caused by migrations. In Ashman, each enclave contains a primary partition that holds runtime settings such as page tables and extended partitions allocated for the enclave application. Intuitively, migrating a primary partition involves additional modification in runtime and thus takes a longer time.

We measure the time consumed to migrate both types of memory partitions by creating an enclave of different sizes and migrate the enclave entirely. The overheads, presented in cycles, are plotted in Figure 9, where the x-axis indicates the number of partitions to migrate. The difference between the primary partition and extended partitions slightly bends the slope down when there is more than one partition. On average, it takes around 5.7×10^7 cycles (~0.057s) to migrate a primary partition and 5.2×10^7 cycles (~0.052s) to migrate an extended partition. Thus, we can infer that the overhead is negligible compared to the whole execution time (RV8 benchmarks generally take 14s ∼ 28s to finish).

Therefore, we have answered **RQ4** that both memory allocation and memory migration are efficient in Ashman, bringing a small performance overhead to the tested TEE.

In summary, Ashman enables the support for executing a large number of enclaves simultaneously and a high physical resource utilization rate with a small performance overhead.

6 DISCUSSION AND FUTURE WORK

Shared Memory. Shared memory is a universal technique for native applications to communicate to others. Host OS provides some easy-to-use interfaces to map the same physical address to the virtual address of multiple processes [19]. So far, Ashman does not support shared memory due to memory isolation. Similar to Keystone, a designated "untrusted" area is pre-defined as a buffer for approximation of shared memory. We can achieve so by enforcing this area not migratable in the enclave memory pool.

Parallelism. Ashman can support parallelism by design, and we are on the way to implement it to a multi-core environment. The challenge raised by parallelism is that multiple enclave applications may require memory migrations simultaneously. Using locks with inter-core communication may solve this challenge.

Page Table. In our current prototype, we pre-allocate 1MB of memory for each enclave to store the page table (131,072 entries). This approach intends to simplify the process of mitigating memory that contains the page table. Therefore for each enclave application, the available memory region is actually limited by the size of the page table. In our current implementation, an enclave application can use 512MB (131,072 × 4KB) of memory at most. Leveraging RISC-V's *megapage* mechanism [3] can overcome the limitation and support up to 256GB of memory.

Memory Compaction. Currently, memory compaction simply migrates all memory partitions to one side. This way, it does not mitigate the fragmentation of enclaves. Therefore, if the memory is under high pressure, there may be only enough for new requests but not for the migration of existing fragments. There are many ways to alleviate this problem; for example, the entire memory pool can be extended. In addition, we can also introduce memory swapping during compaction to eliminate enclave fragmentation.

7 CONCLUSION

We propose a generic extension to the memory management of existing TEEs. Along with the careful design, we demonstrate the usability of our solution in high-workload scenarios. On top of our migration and compaction algorithms, TEEs provide a close-to-native memory management mechanism for enclave applications. With our algorithm, we hope it can be integrated into future TEE designs.

ACKNOWLEDGMENTS

We sincerely thank our anonymous reviewers for their insightful suggestions. Special thanks to Tai Yue for his kindly review and

helpful suggestions. We also appreciate Lei Zhou, Jinting Wu, and Jingquan Ge, who offer early discussion about the project. This work was supported by the National Natural Science Foundation of China under Grants 62102175 and 62002151, and the Science, Technology and Innovation Commission of Shenzhen Municipality under Grant SGDX20201103095408029.

REFERENCES

[1] 2008. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone.
[2] 2015. The Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.
[3] 2019. *The RISC-V Instruction Set Architecture (ISA) and Related Specifications.* https://riscv.org/technical/specifications/
[4] 2019. RISC-V Open Source Supervisor Binary Interface. https://github.com/riscv/opensbi
[5] 2019. RV-8 Bench. https://github.com/rv8-io/rv8-bench
[6] 2021. *AMD Secure Encrypted Virtualization (SEV).* https://developer.amd.com/sev/
[7] 2021. Coffer: A software-based TEE architecture on RISC-V. https://anonymous.4open.science/r/coffer/README.md
[8] 2021. *Penglai Enclave: Verifiable and Scalable RISC-V TEE System.* https://fosdem.org/2021/schedule/event/tee_penglai/
[9] 2021. QEMU. https://github.com/qemu/qemu
[10] Allwinner Technology. 2021. D1 Development board —— Nezha. https://d1.docs.aw-ol.com/en/d1_dev/
[11] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13).*
[12] ARM Ltd. 2009. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
[13] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves.
[14] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Proceedings 2019 Network and Distributed System Security Symposium.* San Diego, CA.
[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16).* USENIX Association.
[16] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).*
[17] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13).*
[18] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibing Guan, Haibo Chen, and Binyu Zang. 2017. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security 17).*
[19] Marty Kalin. 2019. Inter-process communication in Linux: Shared storage. https://opensource.com/article/19/4/interprocess-communication-linux-storage
[20] Keystone Enclave. 2020. seL4 in Keystone. https://github.com/keystone-enclave/keystone-seL4
[21] Lars Lühr. 2021. SGX-hardware list. https://github.com/ayeks/SGX-hardware
[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. 2019. Keystone: A Framework for Architecting TEEs. *CoRR* abs/1907.10119 (2019). arXiv:1907.10119 http://arxiv.org/abs/1907.10119
[23] Arm Limited. 2010. CoreLink TrustZone Address Space Controller TZC-380 Technical Reference Manual. https://developer.arm.com/documentation/ddi0431/c/introduction/about-the-tzasc
[24] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Proceedings 2019 Network and Distributed System Security Symposium.*
[25] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of 2016 Hardware and Architectural Support for Security and Privacy (HASP'16).* ACM Press.
[26] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13).*
[27] Nazarewicz Michal. 2012. A deep dive into CMA [LWN.net]. https://lwn.net/Articles/486301/
[28] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Larry Shi. 2018. A Comparison Study of Intel SGX and AMD Memory Encryption Technology. In *Proceedings of 2018 Hardware and Architectural Support for Security and Privacy (HASP'18).*
[29] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. 2020. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment.
[30] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Larry Shi. 2017. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In *Proceedings of 2017 Hardware and Architectural Support for Security and Privacy (HASP'17).*
[31] SiFive, Inc. 2021. *SiFive U74-MC Core Complex Manual.* https://starfivetech.com/uploads/u74mc_core_complex_manual_21G1.pdf
[32] Fengwei Zhang and Hongwei Zhang. 2016. SoK: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of Hardware and Architectural Support for Security and Privacy (HASP'16).*

A MEMORY ALLOCATION ALGORITHM

Algorithm 1: Memory Allocation Algorithm

Input : n: Number of requestsed partitions
R: Set of adjacent regions owned by the enclave
N: Maximum number of allowed PMP regions
Output: s: The first partition of requested memory

```
1  Function malloc(n)
       // 1. Expand region if possible
2      foreach r ∈ R do
3          s' ← find_largest_contiguous_nearby(r);
4          if sizeof(s') ≥ n then
5              s ← expand_region(r, s');
6              return s;
       // 2. Allocate new region if PMP sufficient
7      if |R| < N then
8          s ← find_contiguous_free(n);
9          if s ≠ NULL then
10             R ← R ∪ {s};
11             return s;
       // 3. Find a smallest region for migration
12     r' ← argmin_{r∈R} sizeof(r);
13     s ← find_contiguous_free(n  sizeof(r));
14     if s ≠ NULL then
15         migrate(s, r');
16         R ← R \ {r'} ∪ {s};
17         return s;
       // 4. Try performing compaction
18     if is_compacted() then
19         return out_of_memory();
20     else
21         memory_compaction();
22         return malloc(n);
```