Enclave Application Cache for RISC-V Keystone

Takumu Umezawa
Waseda University
Tokyo, Japan
umetaku@akane.waseda.jp

Akihiro Saiki
Waseda University
Tokyo, Japan
s.akihiro_lx@akane.waseda.jp

Keiji Kimura
Waseda University
Tokyo, Japan
keiji@waseda.jp

2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) | 979-8-3315-9546-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/EUROSPW67616.2025.00055

***Abstract*—The development of the IoT society and the spread of cloud computing have also increased the number of insecure devices and vulnerable host OSs. We must protect the applications that deal with sensitive information, even from untrusted OSs. Trusted execution environment (TEE) technologies have attracted much attention because they enable an isolated, secure program execution environment from OSs. However, the booting phase of application execution in a TEE requires an expensive hash calculation in the measurement process for its executable binary image to guarantee its integrity. To mitigate this overhead, this paper proposes an enclave application cache mechanism for RISC-V Keystone, a representative RISC-V TEE. It bypasses the measurement process by storing the previously executed binary image in a secure memory space. The performance evaluation shows the proposed cache can achieve a 6–40× speedup in a TEE application's execution time.**

1. Introduction

Various large and small computer systems surround us throughout our society, from edge devices to cloud computing nodes. This situation has also increased the risks of introducing thoughtlessly mismanaged computers and vulnerable OSs. Attackers try to steal and break our sensitive information by using these vulnerable systems as footholds. The importance of security technologies to protect computer systems is increasing.

Trusted Execution Environment (TEE) is one of the most representative security technologies. It executes applications in a secure execution environment isolated from other software, including OSs. While various CPUs have provided their own TEE technologies [1], [4], [6], [8], [9], [12], [13], RISC-V keystone has been proposed as an open-source TEE framework [6]. This enclave-type TEE provides a securely isolated execution environment for each enclave application (eapp), same as Intel SGX and ARM OP-TEE [2], [9].

Before executing an eapp, the Keystone system performs a measurement process, including calculating the hash value of the extracted application memory image, to ensure its identity. While the hash calculation can attest to the eapp memory image's integrity and safety, it poses a long latency before the application starts [3], [14]. This could become a serious problem when an eapp is used in a server application or a serverless service [7]. When the server invokes an eapp at each query arrival time to keep the program organization simple and ensure independence from other queries, it causes an expensive hash calculation for every query. Besides, a Keystone eapp's memory image tends to become large since Keystone requires statically linked binary, and an enclave also contains its runtime, resulting in a long hash calculation time.

To overcome this problem, several works have been proposed for RISC-V platforms: One introduced a hash calculation hardware accelerator [14]. Another proposed a separate eapp binary loading mechanism that allows a user to create a special enclave and load applications into it in advance (Shadow Enclave) with measurement to quickly launch its instance at the invocation time [3].

This paper proposes an enclave application cache mechanism for RISC-V Keystone. It keeps eapp memory images and their hash values in a secure memory area once they are executed. When one of the same applications is invoked again, the system builds a new enclave by copying the previously kept memory image to the enclave without its measurements. Unlike the Shadow Enclave, which requires the user's explicit operation, the system automatically processes it. The main contributions of this paper can be summarized as follows: (1) We propose an enclave application cache for RISC-V, which reduces the eapp startup time without the user's intervention. (2) We analyze the security of the proposed mechanism and confirm that it keeps the same level of security as the original Keystone. (3) We evaluate the performance of the proposed cache mechanism and reveal that it can accelerate the eapp execution time 6–40× compared to the original Keystone implementation.

The remaining part of this paper is organized as follows: Section 2 and Section 3 review the related works and Keystone architecture, respectively. Section 4 evaluates the overhead of the eapp startup. Section 5 explains the proposed enclave cache architecture. Section 6 assesses the security risk of the proposed enclave cache, and then Section 7 conducts its experimental evaluation.

2. Related Works

Hoang et al. proposed a Keystone-compatible RISC-V system with hardware accelerators for hash computation and digital signature algorithms [14]. Most security algorithms run on software, but hardware implementation can improve the computational speed. The Enclave cache method is software-based and can be used with a hardware accelerator.

Penglai Enclave is a TEE implementation for RISC-V [3]. It has a feature to create another type of enclave called

DOI 10.1109/EuroSPW67616.2025.00055

"Shadow Enclave". Shadow Enclave is a non-executable enclave. Its memory map validation and hash calculation for attestation are processed before execution. When it is created, the user is returned an Enclave ID. By specifying this ID, the user can create and launch an enclave instance based on the contents in the Shadow Enclave. Since the Shadow Enclave's memory map has already been measured, creating an enclave instance does not require it, allowing for a faster enclave launch. Shadow Enclave requires the user's explicit operations on its creation and the eapp execution using it. Cerberus introduces a similar approach as Penglai's Shadow Enclave [5].

3. RISC-V Keystone

This section reviews RISC-V Keystone, particularly regarding RISC-V's privilege levels, memory protection function (PMP), and Enclave.

3.1. RISC-V Privilege Levels

RISC-V has three privilege levels, each with a corresponding execution mode: machine-mode (M-mode), supervisor-mode (S-mode), and user-mode (U-mode) [15]. M-mode has the highest privilege level and controls all physical resources and interrupts. S-mode has the second-highest privilege level and is used for the OS kernel, including device drivers and kernel modules. U-mode has the lowest privilege level and is used for general user processes.

3.2. Keystone Architecture

In Keystone, an enclave is a basic unit of an isolated program execution environment. It comprises an enclave application (eapp) and the runtime (RT). Each of them is executed in U-mode and S-mode, respectively. Untrusted applications and OS in the Untrusted Region are also executed in U-mode and S-mode, respectively. However, as explained in the next subsection, the memory areas for the Untrusted Region and enclaves are separated. These regions use shared memory to exchange information. M-mode executes a Security Monitor (SM) that manages the creation, execution, and destruction of enclaves. It also keeps metadata, including information such as each enclave's address, size, and the hash value used for the attestation.

3.3. Physical Memory Protection (PMP)

Keystone uses the Physical Memory Protection (PMP) defined by the ISA specification for RISC-V [15] to create an isolated memory region for an enclave. PMP controls the access rights, such as reading, writing, and executing, of the specific physical memory regions for U-mode and S-mode via PMP entries attached to each RISC-V core in a system.

PMP entries are statically prioritized. The corresponding PMP entries check every memory access issued by a RISC-V core. The PMP entry with the highest priority, which matches any accessed bytes, determines whether access is allowed or denied. The access rights of the memory region defined at the higher-priority entries override those at the lower-priority entries.

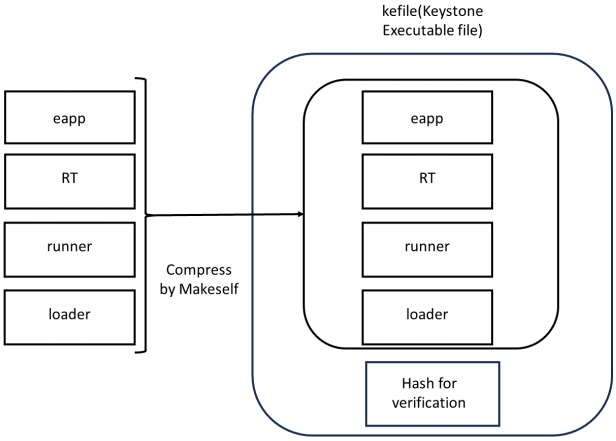

Figure 1. Components of Keystone Executable file.

Keystone configures the highest priority PMP entry for the SM's memory region at the system boot time. In contrast, it configures the lowest priority PMP entry to cover all memory regions with full access permissions (OS PMP entry). Thus, the OS can have access permission to all regions not protected by other PMP entries. At the enclave startup time, the OS allocates the appropriate contiguous memory region for the created enclave. Then, the OS passes the allocated memory region to the SM to configure a PMP entry to protect it from the OS [6].

When a CPU core switches its context to an enclave, the SM configures the PMP entry so that the relevant enclave memory region is fully accessible. Simultaneously, the SM disables all access permissions in the OS PMP entry to prevent the enclave from accessing other memory regions. Then, when a CPU core switches to something other than the enclave, the SM disables the access permissions for the enclave region and resumes the OS PMP entry's permission, allowing the OS's ordinary memory access. Thus, the Keystone can isolate an enclave memory region from other enclaves and the OS.

3.4. Components of Keystone Enclave Application File

An eapp executable file is a self-extracting file created by Makeself [11]. As depicted in Fig. 1, the self-extracting file consists of a TAR archive and a shell script. Further, the TAR archive contains an eapp, an RT, a loader, and a runner: The loader loads the RT into the enclave memory region and creates page table entries (PTE). The RT loads and executes the eapp and provides a small execution environment, such as a proxy of syscalls for the host OS. The runner, executed on the host OS, controls the startup and execution of the eapp in cooperation with the Keystone device driver and the SM.

3.5. Enclave Startup Process

This section explains the startup process of an eapp in Keystone.

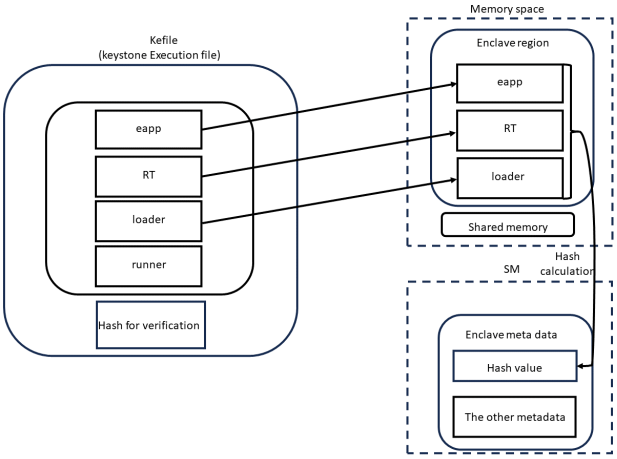

Figure 2. Deployment of Enclave in Main Memory at Its Startup.

First, the runner file is executed, and the control is transferred to the Keystone device driver. The Keystone device driver calculates the required memory size based on eapp, RT size, etc., and then allocates the necessary memory from the Linux kernel based on this information. The driver also obtains a shared memory region for exchanging information between the enclave and the OS. Next, the OS copies the eapp, runtime, and loader file images to the allocated enclave memory region. Finally, the driver passes this memory information to the SM, and the CPU control is switched to the SM.

Based on the information sent from the driver, the SM creates metadata containing information such as the address and size of the enclave's memory region. Next, the SM configures PMP entries for enclave and shared memory regions as described in Section 3.3. It also measures the eapp and the runtime by calculating a hash value for the attestation. The SM stores it in the metadata. As a result of the process above, the eapp, RT, and the loader are finally deployed in the isolated memory region, as depicted in Fig. 2.

After the enclave creation, the SM can switch its control to the eapp. At the first context switch, the loader extracts the runtime memory map from its file image and copies it to the enclave memory, and then the runtime extracts the eapp memory map from its file image. Finally, the eapp starts the execution.

4. Preliminarly Evaluation: Enclave Startup Overhead

Here, we evaluate the enclave startup overhead and the hash calculation overhead inside it to determine the expected performance improvement by the proposed enclave cache.

4.1. Evaluation Environment

We use SiFive's Hifive Unmatched RISC-V multicore platform [10] throughout the evaluation in this paper. It has SiFive Freedom U740 SoC, composed of four SiFive U74 cores and one SiFive S7 core, as shown in Table 1. Each U74 core has a 32KiB L1D cache and a 32KiB L1I cache, while the S7 core has a 16KiB L1I cache. The L2 cache is 2 MiB shared by all cores. Note that all programs evaluated in this paper run on a U74 core. The operating frequency is 1.2 GHz [10].

TABLE 1. SPECIFICATION OF HIFIVE UNMATCHED [10]

Core	U74	S7
Number of Cores	4	1
L1I Cache	32KiB	16KiB
L1d Cache	32KiB	N/A
L2 Cache	2MiB	
Main Memory	DDR4 16GiB	
Clock Frequency	1.2GHz	
Storage	Samsung SSD 970 EVO Plus 250GB (MZ-V7S250BW)	

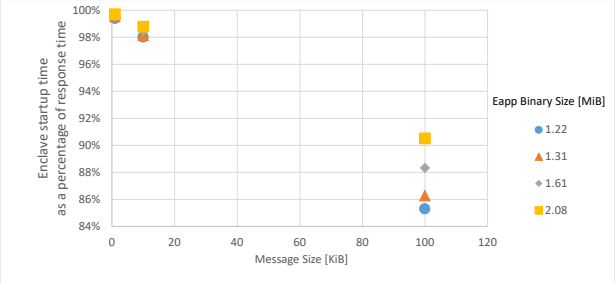

Figure 3. Evaluation Result of Enclave Startup Overhead.

4.2. Evaluation Program

We made a digital signature server program for the evaluation. The core module of the calculation is implemented as an eapp containing a secret key. When the program receives a query with a message, it executes the eapp. The eapp then calculates a digital signature for the message using ED25519, SHA3-512, and the secret key. Finally, the eapp replies to the client with the calculated signature.

We use the ED25519 and SHA3-512 implementations included in the Keystone source tree. The program's binary size can be changed to assess the impact of the size on the hash calculation in the enclave measurement process. We conducted the evaluations 10 times and used the average value.

4.3. Evaluatoin Result

Fig. 3 shows the eapp startup overhead evaluation result. The horizontal and vertical axes represent the input message size and the ratio of the eapp startup time to the eapp execution time, respectively. Note that the majority of the startup overhead is hash calculation time.

The result shows that even the cases of 100KiB message sizes spent over 85% of execution time. This result shows we can reduce the eapp execution time by reducing the hash calculation.

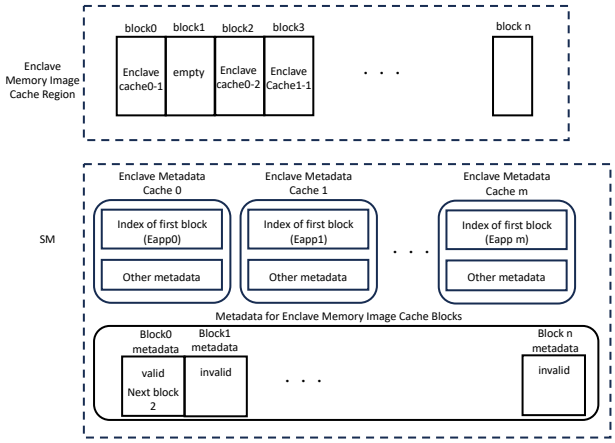

Figure 4. Proposed Enclave Cache Structure.

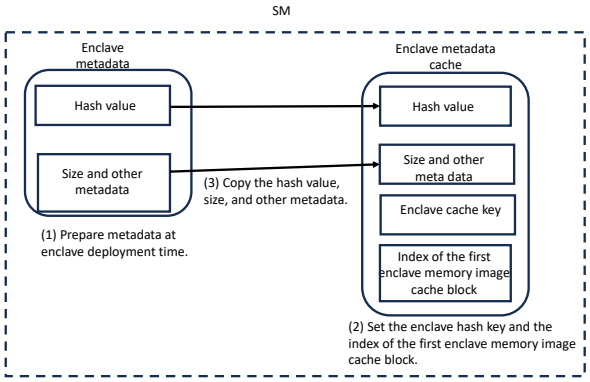

Figure 5. Setting metadata cache using the created enclave's metadata at the cache miss.

5. Proposed Enclave Cache Design

5.1. Enclave Cache Structure

The proposed enclave cache holds multiple eapp memory images in a PMP-secure space. It consists of the memory image cache and the metadata cache, as depicted in Fig. 4. The memory image cache is located in the dedicated memory region separately allocated from the SM. The metadata of the memory image cache and the metadata cache are located in the SM.

The memory image cache is divided by 4KiB blocks, each with an associated metadata. Each block holds a 4KiB block of an eapp memory image, and its associated metadata holds a valid bit and an index of the next block. The number of blocks is fixed in the current implementation.

The metadata cache consists of multiple entries: the number of entries defines the maximum number of eapps that can be held in the cache. Each entry holds the Keystone enclave metadata, such as the eapp hash value, size, etc. It also contains the index of the first memory image cache block of that eapp and the key to check whether the invoking eapp is in the cache.

We introduce an eapp's digital signature, which contains information about the eapp, as the enclave cache key. This is provided along with the eapp's execution file and its public key in the current implementation. The signature contains the provider's ID, the application's ID, RT's ID, the loader's ID, and their version number. Thus, each eapp image can have its unique signature. When an eapp is updated, the SM can avoid executing an older one in the cache and appropriately create an enclave for the newer one.

Note that an entire eapp must be held in the cache. If a part of the eapp must be loaded from the file system, the measurement must be processed again since the binary file may be falsified. Thus, if the eapp's memory image size exceeds the cache size, the cache cannot hold that eapp.

5.2. Caching Eapp

When an eapp is invoked, its digital signature and public key are sent to the SM. It validates the public key with the root public key. If the public key is valid, then the SM checks whether the eapp is in the cache by checking the metadata cache using the digital signature and the public key. If the eapp exists in the cache (cache hit), a new enclave is created from the memory image held in the cache, as described in the later section (Section 5.3). Otherwise, the SM processes the cache miss procedure.

At the cache miss, the SM creates an enclave, same as the normal enclave startup: extracting the file images of the eapp, RT, and the loader from the Keystone executable into the memory, memory protection by PMP, and hash calculation as described in Section 3.5.

Then, the size of the created enclave is checked to determine whether it is smaller than the cache size. If it is larger than the cache, the enclave is not cached. If the enclave size is smaller than the cache size and larger than the cache's free space, the SM invalidates one of the cached enclaves to allocate sufficient free space. If the total number of cached enclaves exceeds its limitation, the SM also chooses one of them to be invalidated so that the SM allocates the entry in the metadata cache for the newly cached one. The SM uses LRU to choose the invalidating enclave.

After that, the SM sets the enclave metadata, its digital signature as the enclave cache key, and its first block in the enclave cache in the metadata cache as depicted in Fig. 5, and the newly created enclave's memory image is copied into the allocated blocks as depicted in Fig. 6. The SM also sets the valid bit and the next block index in each memory image cache block's metadata. In Fig. 6, blocks 0, 1, and 3 are allocated for the newly cached eapp. These blocks are validated by setting 1 for the valid bits in the associated metadata. Also, the SM sets "1" in the next block field of block 0, "3" for block 1, and so on.

5.3. Creation of Enclave from Enclave Cache

When an invoked eapp is in the cache (cache hit), the SM creates an enclave memory image from the cached information. Once the cache hit happens by checking the enclave cache key in the metadata cache, the SM copies the memory images of the eapp, RT, and the loader to the created enclave region from the enclave memory image cache blocks by traversing the next block indexes in the associated blocks' metadata. Next, the SM creates the enclave metadata and copies the hash value from the

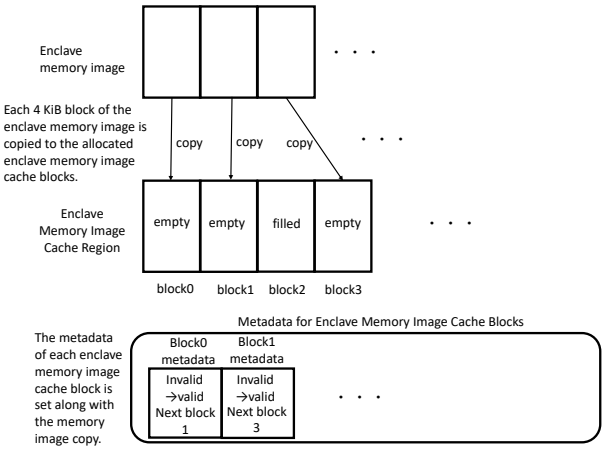

Figure 6. Data flow of enclave cache creation. The created enclave's memory image is divided into 4KiB blocks and copied into the allocated blocks of the memory image cache. The associated metadata for each memory image cache block is also configured.

enclave cache metadata to the created metadata. Then, the loader starts the RT and the eapp, like the normal execution.

6. Security Analysis

This section first describes the threat model. We follow the same threat model of RISC-V Keystone for our enclave cache. Then, we argue that the proposed cache can protect its contents from unauthorized external memory access, just as the original Keystone can.

6.1. Threat Model of RISC-V Keystone

RISC-V assumes four classes of attackers: a physical attacker, a software attacker, a side-channel attacker, and a denial-of-service attacker [6]. The most critical attacker class in our proposal is the software attacker, which can control the software behavior of host applications and the untrusted OS on the victim host. It can also modify the unprotected memory regions. Since the proposed enclave cache is in memory space, we must confirm that it is protected from unauthorized memory access. More on this in the next subsection.

The proposed cache can also offer the same security strength as the original Keystone for other attacker classes. Side-channel attacks with off-chip components are out of the scope of this paper, as is Keystone. If we need a higher security strength, we can combine existing protection mechanisms, such as introducing an isolated scratch pad memory for the enclave cache location to prevent side-channel attacks.

6.2. Protection Against Malicious Memory Access

RISC-V keystone assumes that software attackers cannot access memory regions protected by PMP [6]. Since a software attacker only has U-mode and S-mode privileges, protection by PMP prevents memory access and tampering. The SM calculates the hash value for the created enclave at cache miss time, and its memory image is stored in the cache. Since both are appropriately protected by PMP, the attacker cannot tamper with the cached application after the hash value calculation. Also, the metadata area of the enclave cache is managed by the SM and is protected by PMP, so the attacker cannot tamper with the metadata to alter hash values and addresses. Therefore, the same security strength as the original Keystone is maintained even when the enclave cache is introduced.

7. Experimental Evaluation

This section reports the performance evaluation result of the proposed enclave cache on RISC-V Keystone. The evaluation environment is the same as in Section 4, using Hifive Unmatched, and the average value of 10 measurements is taken. We first evaluate the primitive performance of the proposed enclave cache, then we use the digital signature server program used in Section 4 for an end-to-end performance evaluation.

7.1. Evaluation of Primitive Performance

We evaluate two items: The first evaluation is the speedup at the enclave cache hit compared to the one without the cache (normal startup). The second evaluation is the overhead of the enclave cache implementation, comparing the enclave cache miss to the normal startup. Since we assume the number of cached applications is small (at most ten), we do not evaluate the application look-up overhead. Note that we ensure the platform's hardware cache is cleared before the measurement to compare it with the case without an enclave cache.

Fig. 7 and Fig. 8 depict the evaluation results. For both graphs, the horizontal and the vertical axes show the eapp file size and the enclave startup time, respectively.

Fig. 7 depicts the comparison result of the enclave startup time between the normal startup (dotted line) and the enclave cache hit (solid line). It shows that the enclave cache hit achieves approximately 28.5–55.5$\times$ faster startup time than the normal.

Similarly, Fig. 8 depicts the comparison result of the enclave startup time between the normal startup (dotted line) and the enclave cache miss (solid line). It shows that the startup time with the enclave cache miss is approximately 1.03$\times$ larger than normal. This confirms that the overhead introduced by the proposed enclave cache is negligible.

When the size of the eapp file changed from 1.6 KiB to 2.0 KiB, the speedup on cache hits decreased from about 55.5$\times$ to 31.9$\times$. This is because the Keystone device driver allocates the memory for an enclave in pages of powers of 2, and the allocated memory size is changed between these two file sizes. The overhead caused by this memory allocation does not appear to be significantly different when an enclave is normally started since it is relatively smaller than the hash value calculation. However, for the proposed enclave cache, the impact of the memory allocation on the entire system becomes large due to bypassing the hash calculation, resulting in a relatively shorter startup time.

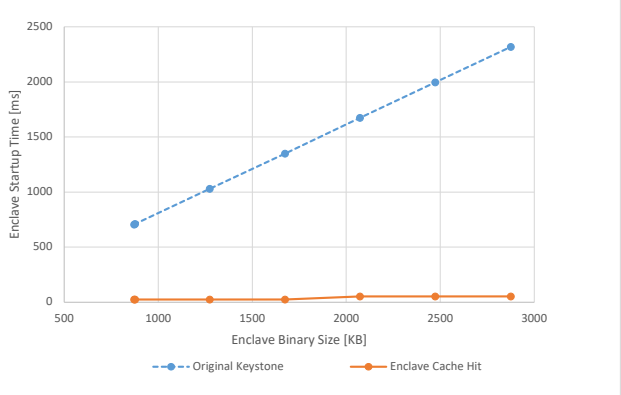

Figure 7. Enclave start-up time comparison between normal start-up and cache hit.

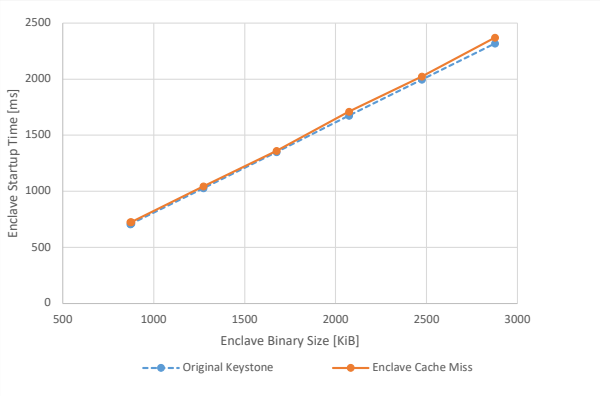

Figure 8. Comparison of Enclave start-up time between normal start-up and cache miss.

7.2. Evaluation on Digital Signature Program

We evaluate the proposed enclave cache with the digital signature server program used in Section 4. Fig. 9 depicts the evaluation result. The horizontal axis shows the input message size, and the vertical axis shows the speedup of the response time at the cache-hit time compared to that on the original Keystone. We varied the server's binary size.

The evaluation shows that the proposed cache obtains a 6–40× speedup. Though the longer message requires more calculation and startup time becomes relatively short, the 100 KiB message size case can still obtain a speedup of about 6–8×. The proposed cache can contribute to reducing the response time of the server application.

7.3. Discussion: Comparison with Shadow Enclave

We also implement another type of enclave cache, which creates a special enclave holding the memory image of a cached enclave: at the cache-hit time, the SM creates an enclave instance from the special enclave holding the cached enclave's memory image. This implementation mimics the eapp startup overhead for Penglai's Shadow Enclave [3] in the Keystone environment. Note that Penglai introduces page-based memory protection

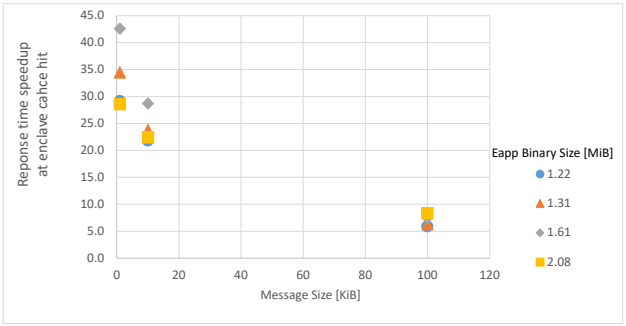

Figure 9. Speedup of server response time at enclave-hit against the original Keystone.

(Guarded Page Table) by a hardware extension. Unlike Penglai, we must use one PMP entry to protect a special enclave in the Keystone environment.

We compare the startup time of the proposed enclave cache with this special enclave-based cache implementation. The evaluation shows the latter one has approximately 10–15 [ms] shorter startup time than the proposed cache, depending on the binary file size. They are about 60–74% of the startup time for the proposed cache. This is because the proposed cache must gather the memory image from multiple memory image cache blocks to create the enclave instance, while the latter just copies an entire memory image from the special enclave. On the other hand, the latter implementation requires one PMP entry for each special enclave, while the proposed cache uses only one PMP entry for the memory image cache, which can hold multiple eapps' memory images. The proposed approach can realize an efficient enclave cache mechanism on an ordinary RISC-V platform by consuming only one additional PMP entry.

8. Conclusion

This paper proposes an enclave cache mechanism to accelerate the startup of an enclave on RISC-V Keystone. We first investigated the overhead of response time for a server application and revealed that the hash calculation for the enclave measurement process can account for more than 85% of the response time. The proposed enclave cache can reduce startup time by bypassing the hash value calculation in the measurement process if the system executes it once and the cache keeps its information. The evaluation result has shown the proposed enclave cache obtained 6–40× speedup compared to the original Keystone implementation. We have also shown that the overhead introduced by the cache is only 3% by comparing the startup time at the cache miss to the normal one.

Acknowledgement

A part of this paper is supported by JSPS KAKENHI Grant Number JP23K11040.

References

[1] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. https://www.amd.com/content/

dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, January 2020.

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

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

[4] Intel. White paper: Intel trust domain extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419, February 2023.

[5] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. Cerberus: A formal approach to secure and efficient enclave memory sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1871–1885, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] 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, 2021.

[8] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.

[9] Linaro [n. d.]. Open portable trusted execution environment ([n. d.]). https://www.op-tee.org/.

[10] SiFive [n. d.]. HiFive Unmatched datasheet ([n. d.]). https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf.

[11] Stéphane Peter [n. d.]. makeself - make self-extractable archives on unix ([n. d.]). https://makeself.io/.

[12] Arttu Paju, Muhammad Owais Javed, Juha Nurmi, Juha Savimäki, Brian McGillion, and Billy Bob Brumley. Sok: A systematic review of tee usage for developing trusted applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. CoVE: Towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, page 315–321, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Hoang Trong-Thuc, Duran Ckristian, Nguyen-Hoang Duc-Thinh, Le Duc-Hung, Tsukamoto Akira, Suzaki Kuniyasu, and Pham Cong-Kha. Quick boot of trusted execution environment with hardware accelerators. *IEEE Access*, 8:74015–74023, 2020.

[15] Andrew Waterman, Krste Asanović, and SiFive Inc. The RISC-V instruction set manual, volume II: Privileged architecture. https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.