Dynamic Linking in Trusted Execution Environments in RISC-V

Catherine Lu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-142
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-142.html

May 18, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Dynamic Linking in Trusted Execution Environments in RISC-V

by Catherine Lu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanovic
Research Advisor

(Date)

* * * * * * *

Professor John Wawrzynek
Second Reader

(Date)

Dynamic Linking for Trusted Execution Environments in RISC-V

Catherine Lu

Abstract

Dynamic linking is an important feature for many applications, making it an integral function to provide in Trusted Execution Environments (TEEs) to improve the ease of use and encourage adoption of TEEs for security and privacy. However, the implementation of dynamic linking in enclaves poses many challenges, for example, the proper verification of code loaded after an enclave has begun execution and the efficient initialization of enclaves. Previous TEEs have included dynamic linking functionality but either fail to accommodate library sharing or require that libraries be manually loaded in advance. Our work draws upon previous studies to provide two methods of dynamic loading that allow for customization to users' needs. First, we provide trusted loading after enclave execution as the default loading method that can automatically load in shared libraries by coordinating with an untrusted OS. Second, we provide library enclaves as an optimization for commonly used libraries to be pre-loaded and verified prior to enclave execution. These library enclaves allow for sharing between multiple enclave applications. While previous work has been done on proprietary software, we implement our design in Keystone, an open-source RISC-V framework for constructing customizable TEEs, making our work accessible for future research and study. Finally, we expand on previous dynamic library implementations by accounting for mitigations against side-channels attacks. We propose a cache tagging solution to defend against the Flush+Reload side-channel attack and analyze its performance tradeoffs through simulation.

1 Introduction

Data privacy and the security of sensitive code has become an increasingly large concern, especially with the growth of cloud computing. Typically, encryption is used to ensure the confidentiality and integrity of data. However, when data is actively in use, operating on encrypted data can be very slow. Instead, Trusted Execution Environments (TEEs) have seen rising utilization to preserve the privacy of sensitive code and data. TEEs, also known as enclaves, are isolated environments that are used to protect the confidentiality and integrity of programs and data within the region from an untrusted host. They typically provide a security property called shielded execution [13] which hides execution from the host and prevents host tampering. Under the confidentiality guarantees of shielded execution, only inputs and outputs of an enclave are revealed, but any intermediate state is hidden. With the integrity guarantees of shielded execution, no part of the host system may affect behavior of programs executing within an enclave.

To facilitate the adoption of TEEs, it is necessary for TEEs to support unmodified legacy applications. Additionally, many applications rely on dynamically-linked libraries (DLLs) for core functionalities. Dynamic libraries are efficient for compilation because the library and application are compiled separately, preventing the other component from needing to be re-compiled when a change is made to only one or the other. Furthermore, when multiple applications reference the same shared library, only one copy must be kept in memory. Load time is also reduced if shared library code is already present. To enable these optimizations, it is necessary to support dynamic linking in TEEs.

Although shared libraries are commonly used outside of TEEs, allowing dynamic linking within enclaves poses several challenges. Enclave memory is typically isolated from the rest of the processor in order to preserve the confidentiality and integrity of programs running within it. The enclave must be verified to ensure that its memory has not been tampered with. These requirements make it difficult to share code or link a dynamic library at runtime. Additionally, the dynamic linking implementation must not significantly increase enclave start up time.

Some previous studies have proposed solutions to the dynamic linking problem for TEEs. Graphene-SGX [13], now renamed to Gramine, implements shared libraries with a trusted loader that verifies each of the loaded libraries. This is performed by maintaining a manifest file for each enclave which

contains the list of loadable binaries and their hashes. During the loading process, the trusted loader will load the library OS, the standard libraries, and then load any other libraries specified by the user by comparing the SHA-256 hash of the library with the hash within the manifest file. If the hash does not match, the loader will not open the library.

While Gramine provides a system for shielding dynamic libraries, which are used in a large majority of applications, it comes with some drawbacks. Because Gramine requires that all needed libraries be included in the enclave and manifest file prior to measurement, it may be necessary to over-approximate the needed libraries in order to cover possible libraries that will be accessed. Additionally, since integrity is verified through a manifest file, when libraries are updated, it is also necessary to update the hash of the library within the manifest file.

Plug-in enclaves provide an alternative solution to dynamic linking. With the plug-in enclave (PIE) [9] model, a new shared enclave region primitive is created. The shared enclave is an immutable memory region that can be mapped to other host enclaves. Dynamic libraries can be added by running a shared enclave containing the desired library. The library will be attested on enclave initialization. If a host enclave requires the dynamic library, the shared enclave can then be mapped into the host enclave's virtual memory.

This paper presents another implementation of dynamic libraries in enclaves using Keystone [8], an open-source framework for trusted execution environments in RISC-V. We draw from both Gramine and PIE dynamic loading. We provide a trusted loading verified by a manifest file as the default loading scheme. Additionally, we allow users to construct library enclaves, read-only enclaves that can be mapped into various host enclaves, as an optimization to the default loading method. The use of library enclaves in conjunction with the trusted loading scheme allows commonly used libraries to be pre-loaded and verified, reducing enclave initialization times.

Introducing dynamic libraries in Keystone allows our TEEs to support a wider range of applications. However, the use of shared memory also gives rise to new vulnerabilities, namely cache side-channel attacks. Side-channel attacks target the security of a system by exploiting indirect effects of the system or hardware instead of directly attacking the code. Cache side-channels typically utilize minute differences in cache access times to infer information about the instructions or data being accessed.

Side-channel attacks are a growing threat and as a result there have been many prior publications on timing- and cache-related side-channels including Spectre [7], Meltdown [10], Prime+Probe [15], and Flush+Reload [14]. We will focus on the potential Flush+Reload side-channel attack introduced by adding shared libraries in Keystone. The Flush+Reload attack takes advantage of a cache side-channel that occurs when different processes run shared code. This attack utilizes the last-level cache (LLC), an inclusive cache that is also shared by multiple cores on the same processor. An inclusive cache is one that contains copies of data at all higher cache levels. Thus, an attacker can take advantage of these two properties by flushing target memory lines from the LLC. To execute an attack on a victim process, the adversary can evict the desired memory line, wait for a chosen period to allow the victim time to access the line, then measure the time it takes to access the same line again. If the victim had accessed the line, the time to load should be short since the memory line should be in the cache again; otherwise, the data must be retrieved from memory so the load time should be much longer. This vulnerability exists when using library enclaves, since two enclaves that share the same library enclave will be directly sharing those physical pages. We propose a cache-line tagging scheme to defend against the Flush+Reload attack.

Our contributions in this paper include:

- An implementation of dynamically linked libraries in an open-source RISC-V based TEE framework
- The design of a cache-line tagging system to defend against the Flush+Reload attack
- A cache simulation analysis of the performance impacts of cache tagging

The rest of the paper is organized as follows. Section 2 describes the background on Keystone and the threat model for our TEE framework. Section 3 details the design of our dynamic linking implementation in Keystone and analyzes the security tradeoffs of DLLs in enclaves. Section 4 introduces our proposed cache tagging design for defending against the Flush+Reload attack in TEEs and presents the results of our cache simulation and analysis. Finally, we conclude our paper in Section 5.

2 Keystone Background

Keystone is an open-source framework that allows for customizable TEEs in RISC-V, enabling more developers to make use of TEEs for data security and privacy. At its core, Keystone relies on restricting application and operating system access to memory using physical memory protection (PMP) registers. An overview of the Keystone system is shown in 1. The following sections will provide further detail of the relevant components.

2.1 RISC-V Privilege Modes

RISC-V has three privileged modes that Keystone uses to enforce access and execution permissions. These three modes, with increasing permissions, are User (U-mode), Supervisor (S-mode), and Machine (M-mode). U-mode supports user processes such as untrusted applications and enclave applications (EAPPs). S-mode, which is above U-mode, supports the

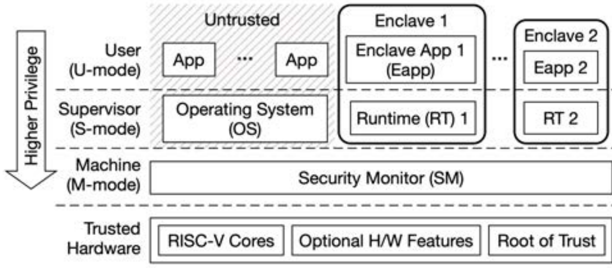

Figure 1: Keystone System Diagam [8]

Figure 2: Enclave Lifecycle [8]. On **creation**, memory is allocated for the enclave and binaries are loaded. Then, the SM sets PMP entries for the enclave region and removes access. Finally, the SM verifies the loaded binaries. On **execution**, the SM provides the enclave r-w-x permissions on entering the enclave, and removes them when stopping or exiting. On **destruction**, enclave memory is cleared and PMP entries are deallocated.

host operating system as well as the runtime kernels running within each enclave. Finally, M-mode is the highest privilege level with no limitations and operates on physical memory. In Keystone, the Security Monitor (SM) is the only component that runs in M-mode and comprises Keystone's TCB.

2.2 RISC-V PMP entries

Keystone's main access control comes from RISC-V's PMP standard. PMP registers are control and status registers (CSRs) that are used to control S- and U-Mode accesses to a region of memory. Each PMP entry may be assigned to a region of physical memory and can be set to allow or deny r-w-x permissions. PMP entries are statically prioritized, with the 0th entry having highest priorty and the $(N-1)$th entry having the lowest.

2.3 Keystone Security Monitor

Keystone's security monitor (SM) is the M-mode software responsible for ensuring the security and isolation of Keystone enclaves. The SM ensures proper memory isolation by configuring PMP entries throughout the enclave lifecycle. The full enclave lifecycle is shown in Figure 2. When an enclave is created, it sets a PMP entry for the enclave's physical memory region and another PMP entry for a shared region between the enclave and the host OS. These entries are initially set to no access so that the host OS and other applications may not access enclave memory. When executing an enclave, the SM flips the PMP entries so that the enclave has access to it's own memory and no other regions.

In addition to memory isolation, the SM also handles measurement and verification of enclaves prior to execution. The SM must hash and measure each enclave to ensure enclave integrity before it is allowed to run.

Furthermore, the SM is also responsible for handling traps and interrupts. These may be resolved directly by the SM or forwarded to the host OS. Additionally, communication to the OS through system calls are also passed through the SM.

2.4 Keystone Runtime

The Eyrie Runtime (Eyrie-RT) is a lightweight kernel running within each enclave that handles memory management and environment calls. It is responsible for enclave initiated communication to the SM and untrusted OS including network calls, access to the untrusted file system, I/O, and other functionality. The RT executes in S-Mode within the enclave and is trusted by the enclave application.

2.5 Threat Model

To discuss the Keystone threat model, we must first elaborate on Keystone's trusted components. Keystone creates a chain of trust beginning from the hardware root-of-trust, which is used in the verification of the SM. The SM must then verify all enclave components, including the RT and EAPP. Thus, the RT and EAPP trust the SM. The RT loads the EAPP, so the EAPP also trusts the RT. All components must trust the SM. These chains of trust are also dependent on a correct hardware implementation.

The Keystone threat model is similar to that of many other TEEs and considers the host OS, other enclaves, and other programs running on the system as untrusted. We assume that the attacker may have access to physical hardware and can measure outputs coming off the chip. Attackers can also take control of system software, such as the OS, and may also create their own enclaves. We do not consider denial-of-service attacks in the scope of our attacker model, since the host OS can deny service at any time by refusing to schedule enclave threads.

This work expands on the initial scope of the Keystone threat model by taking into consideration side-channel attacks. We account for an attacker that may be able to observe and measure the timing of cache accesses.

3 Enabling Dynamic Linking in Keystone

In the Gramine dynamic linking model, shared libraries are linked after enclave execution by loading DLL object files directly into enclave physical memory from the host operating system. Plug-in enclaves take a different approach by pre-loading dynamic libraries into a separate enclave to be mapped to existing host enclaves. In our implementation of dynamic linking in Keystone enclaves, we provide both options to ensure correct functionality of dynamically linked programs, while taking advantage of the efficient memory usage and fast enclave initialization provided by shared libraries. In this section, we detail the design of our dynamic library implementation in Keystone and provide a security analysis of our implementation.

3.1 Design

We propose two dynamic linking methods for Keystone: the first introduces a new Keystone primitive, library enclaves, and the second relies on trusted loading with a user-provided manifest file.

Library enclaves allow DLLs to be shared securely across multiple enclaves and applications. These can be used for commonly used shared libraries, so that the physical memory region can be shared between enclaves. Additionally, library files into library enclaves before the host enclave is created, allowing for faster host enclave start-up times since the library can be pre-verified by the SM.

When a library enclave does not exist for a required shared library, enclaves will default to the trusted loading method that is verified with a manifest file. This is advantageous for DLLs that are not as frequently used or shared by other applications. With this method, libraries are loaded after the enclave is executed and the EAPP is parsed for needed libraries. It ensures that an EAPP can still run, even if a shared library has not yet been pre-loaded, as long as the user has provided the expected hash within the manifest file. Additionally, it allows us to conserve PMP entries, since libraries are directly added to the host enclave and do not require a separate memory region. Our trusted loading design also simplifies the measurement and validation process compared to statically-linked enclave applications in Keystone. The following sections will describe the design and evaluation of these two linking methods.

3.1.1 Library Enclaves

We created a new Keystone primitive, the library enclave, to facilitate efficient linking of shared libraries. The library enclave is a read-only enclave that stores a dynamic library ELF file. It does not run independently but can be mapped into a host enclave's memory for execution. Because library enclaves are immutable and DLLs are typically accessible code, they can be securely mapped to multiple host enclaves, allowing the physical DLL object files to be shared by different enclaves.

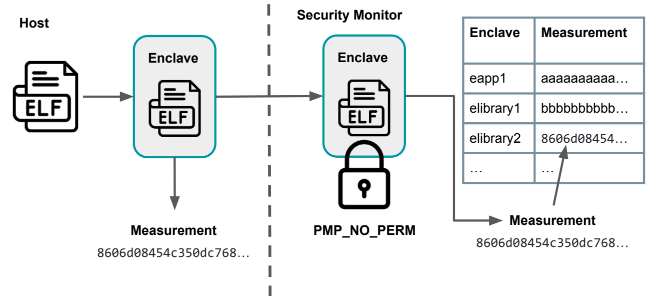

Figure 3: Library Enclave Creation

In our implementation, the security monitor is responsible for linking library enclaves to hosts and controlling library enclave permissions. On library enclave creation, a PMP entry is allocated for the library enclave memory region, and it is set so that r-w-x permissions are removed from library enclave memory. When a library enclave is linked to a host enclave, the SM creates a reference from host to library enclave. On host enclave execution, the SM loops through library enclaves referenced by the host and modifies each library enclave's PMP entry so that the permissions are changed to read-only, prior to context switching to the host enclave.

Since library enclaves are not executed directly, we simplify its design from host enclaves by removing execution structures such as the runtime kernel and the untrusted buffer. This allows library enclaves to be easily verified by the SM by computing a hash over the library enclave physical memory. The library enclave only stores the DLL, so this hash can be directly compared to the expected hash of the DLL object files. This is a simplification from the prior host enclave verification process, where the SM needed to walk the enclave page tables to iteratively build a measurement of the entire enclave memory. This is no longer necessary because library enclaves store only the library ELF and are mapped and loaded into a host enclave at runtime. Thus, they can be verified before virtual memory is setup.

3.1.2 Trusted Loading by the Runtime Kernel

For our default loading scheme, we implemented a trusted loader within the Runtime kernel. This loader is used to load any necessary libraries that have not yet been added as library enclaves. The loader first parses the enclave application ELF file to identify required libraries. Then, the loader communicates with the SM to select relevant library enclaves

and determine unloaded libraries. To load these remaining DLLs, the loader communicates with the host operating system through the enclave's untrusted buffer to copy the library ELFs into enclave memory.

Due to the untrusted nature of this channel, it is necessary to verify the loaded libraries before linking them for execution. This verification is performed using a list of hashes provided within the manifest file on enclave creation. The loader hashes the library ELFs and compares these measurements to the expected hashes within the manifest file. If any hashes are incorrect or are not present, the loader initiates the enclave destruction subroutine.

Once the DLLs are validated, they can be safely loaded and mapped with the enclave's page tables in preparation for execution.

3.1.3 Memory Modifications for Trusted Loading

In order to support dynamic linking of enclaves after enclave execution, we made significant changes to the memory structure for host enclaves. In Keystone's previous implementation, enclave memory is entirely initialized by the host operating system. During the enclave creation process, the host OS parses and maps the runtime and enclave application ELF files into a physical memory region designated for the enclave. It then loads each ELF object file, creating a preliminary page table for the enclave before the enclave begins execution.

In our design, instead of loading the runtime and enclave application binaries prior to enclave execution, we initialize the enclave by simply copying the ELF files into enclave physical memory. All loading functionality is performed by a Trusted Bootloader (TB) that is responsible for parsing RT and EAPP ELF files and setting up enclave page tables so that the RT kernel may have access to all of the enclave's DRAM. To facilitate the enclave boot process, we initialize the enclave's physical memory with the TB ELF file, the RT ELF file, and the EAPP ELF file.

Because our design does not have virtual memory on initialization, it significantly reduces the complexity of the initial enclave memory system. This clarifies the verification process for the SM. With Keystone's previous enclave memory layout, the SM measured enclave memory by walking the enclave's page tables and computed a hash iteratively by measuring each page. In our design, the SM can hash each segment of the EPM separately and compare the computed measurement with the expected hash of each component's ELF file.

3.1.4 Putting it all together: Lifecycle of a Dynamically Linked Enclave

A dynamically-linked Keystone enclave goes through several different stages. **Prior to enclave creation**, the programmer can create library enclaves to store DLL files. This preloading allows these DLLs to be verified prior to actual en-

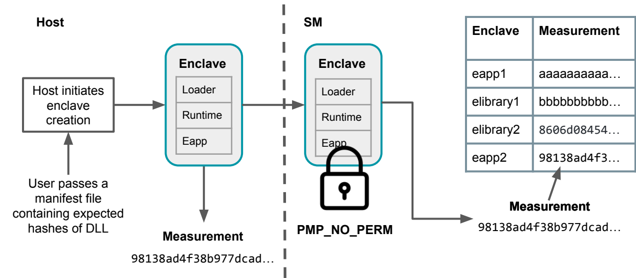

Figure 4: Host Enclave Creation

clave creation, speeding up the enclave initialization process. On **enclave creation**, the EPM is initialized with the Trusted Bootloader, Runtime kernel, and enclave application. The SM measures each of these components to verify their integrity. During the **first execution**, the TB boots the RT, which then parses the EAPP for its required DLLs. The RT loader communicates with the SM to request existing library enclaves to be linked with the current host enclave. The SM creates references from host to the relevant library enclaves and updates PMP entries to allow the host enclave read-only access. Any remaining DLLs will be resolved by the RT loader by communicating with the host OS through the enclave's untrusted buffer. After these DLLs are verified by the SM, the loader will parse and map each DLL the enclave's virtual memory. When the enclave is **destroyed**, the SM removes references from host to library enclaves and cleans up enclave memory. In the event that a library enclave is destroyed, the SM will immediately destroy any enclaves that are dependent on the library enclave.

3.2 Implementation

In our implementation of dynamic linking in Keystone, we added 213 lines of code to the SM for our library enclave creation and 1522 lines to the RT kernel for trusted loading. The Trusted Bootloader that we implemented was 3645 lines. Our implementation has resulted in only minor increases to the TCB and trusted code. Some key details from the Trusted Bootloader are discussed in the following section.

3.2.1 Trusted Bootloader

Our Trusted Bootloader (TB) is a lightweight program that boots the runtime kernel. In our design, the bootloader must be the first program that is executed upon transferring control to the enclave.

The context switch from security monitor into the enclave occurs using the `mret` instruction, which on the first execution, switches the RISC-V privilege mode from M-mode to S-mode to boot the Runtime. To ensure that the bootloader is the first program that runs when entering into enclave memory, we copy the loader program into the first physical page of

enclave DRAM and load this address into the `mepc` register, so that the *pc* is pointing to the beginning of DRAM when execution begins. Because we want this instruction to be the first instruction of the TB, we extract only the text section of the TL ELF to be copied to enclave memory.

Before the enclave begins execution, its memory contains only unloaded ELF files. Thus, the TB initially operates only on physical addresses. After setting up the enclave's page tables, it must then switch from physical to virtual addresses. We perform this change by setting `stvec`, the register containing the address of the trap handler, to the start address of the RT kernel. Then, we change the `satp` CSR from 0, which indicates execution with physical addresses, to the physical address of the root page table. After switching to virtual addressing, the processor initially still tries to execute the next physical address, which will result in a trap since the address is likely not to correspond to a valid instruction in virtual memory. Since `stvec` is set to the starting address of the RT, the trap will cause the `pc` to be set to the RT start, triggering execution of the RT with virtual addressing.

3.3 Security Analysis

The addition of dynamically linked libraries reduces the strict memory isolation of prior enclave designs. Thus, it is necessary to discuss the security trade-offs of the dynamic library implementation.

For the library enclave implementation, once the DLL is loaded into a library enclave, the SM measures and verifies that the hash of library enclave memory matches the expected measurement of the DLL object files. This verifies that the DLL has not been modified, which ensures the integrity of the library enclave. All following accesses to the library enclave will be read-only accesses facilitated by the trusted SM. Therefore, we can be confident that the integrity of library enclave contents are preserved.

For the DLLs that are loaded later by the RT loader, the loader partners with the SM to measure each library file before execution. The measurements are compared to the expected hashes provided within the manifest file. The SM requires that the measurements match to allow the enclave to continue execution. Then, the RT loader can load the dynamic libraries for execution. In this scheme, the SM verifies the integrity of the library, but we must also trust the RT loader for correct loading and linking of the DLLs.

4 Mitigating Cache Side-Channels with Shared Libraries

Our dynamic linking design enables efficient code sharing and fast enclave start-up times. However, sharing libraries across enclaves also creates a security risk due to side-channel attacks on shared memory. One such well studied attack is the Flush+Reload cache side-channel attack [14]. In this section, we give an overview of how the Flush+Reload attack could be implemented on our linked enclaves and propose a mitigation using cache tagging. In addition, we simulate caches with and without our proposed modifications to analyse the trade off between enhanced security and performance.

4.1 The Flush+Reload Side-Channel Attack

Even the most cryptographically secure systems may leak indirect information from computation. This leakage of information results in vulnerabilities that may be exploited through side-channel attacks. Side-channel attacks can exploit information such as power consumption, timing channels, or electromagnetic emissions to gain an understanding of the system or running processes.

In computer architecture there have been a number of well studied side-channel attacks on enclave-based systems. For example, Nilsson et al. [12] break down well known attacks on Intel-SGX into control-channel attacks, cache attacks, branch-prediction attacks, speculative-execution attacks, rogue data-cache loads, microarchitectural data sampling, and software-based fault-injection attacks just to mention a few.

In this work we will focus on cache side-channel attacks. Cache side-channels take advantage of the cache structure and the indirect information created by caching memory from the DRAM. Typically, this includes timing information from cache hits or misses. These are accessed from outside the protected memory region since caches are shared between processes and enclaves. Some major variations of cache side-channel attacks include Prime+Probe [15], Evict+Reload [5], and Flush+Reload. In the Prime+Probe attack, the attacker primes the cache with their data and can determine if a victim has accessed certain physical addresses if they discover that their data has been evicted. The Evict+Reload and Flush+Reload attacks rely on having some shared physical memory between the victim and the attacker. In these attacks, the attacker evicts a cache line and then measures the timing of their next access to determine if the victim accessed this cache line. Since our DLL implementation introduces shared memory, we analyze the Flush+Reload attack on our system in the following sections.

4.1.1 The Attack

The Flush+Reload attack [14] targets the last-level cache (LLC) of a processor. This attack targets the LLC because it is generally shared across all processors in modern hardware implementations. We show an example of the Intel Ivy cache structure in Figure 5.

Since pages within the LLC are shared, an attack process can evict memory from the cache and then monitor the cache for timing information about what is accessed. Flush+Reload is executed in three phases. First, the attacker flushes the memory line that is of interest. Second, the attacker waits

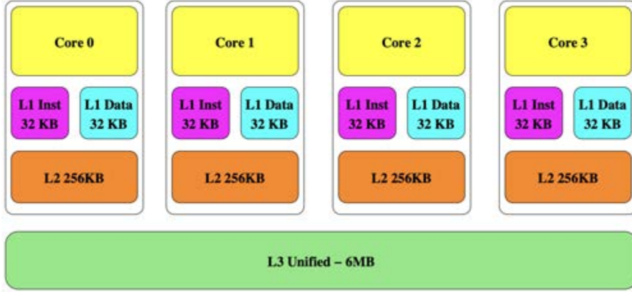

Figure 5: Intel Ivy Cache Structure [14]

to give the victim process time to access data in the cache. When the victim accesses the flushed cache line, data will be loaded into the cache. In the final stage, the attacker reloads the memory line. Using timing analysis the attacker can tell if data was loaded by the victim. The data will be quickly loaded if the victim accessed the data, and the data will take longer if there is a cache miss because data will needed to be loaded from the DRAM.

Yarom et al. [14] highlight the risk of this attack by demonstrating that they were able to successfully extract private encryption keys from Intel x86 processors.

4.1.2 Flush+Reload Mitigations

Yarom et al. [14] provide some suggestions for mitigating the Flush+Reload attack, including limiting the use of the `clflush` command in x86, limiting sharing of pages between processes, and reducing clock resolution or adding noise so that an attacker cannot accurately measure accesses.

There is also related work on evaluating and defending against the Flush+Reload attack in the literature that we draw inspiration from. CATalyst [11] uses Intel Cache Allocation Technology to partition the LLC into secure and non-secure partitions. The secure partition stores cache-pinned secure pages that can be assigned to VMs on launch. Thus, security-sensitive code is always a cache hit since they are essentially pre-loaded to the secure partition. Dynamically Allocated Way Guard (DAWG) [6] is another mitigation against cache timing side-channels. DAWG securely partitions set associative caches so that hits and line replacements are restricted to the partition that issued a memory access. This prevents processes in other partitions from learning metadata about other partitions memory accesses. SiFive's WorldGuard security feature [4] also provides another implementation of a generic system for separating processes into protection domains, or worlds. These protection domains can likewise be used to partition cache accesses based on a world ID.

Similar to many of these approaches we restrict accesses to the shared cache. However, our approach is unique in that it enables enclaves to still make efficient use of the entire cache space without allowing a malicious program to gain detailed timing information.

4.1.3 Flush+Reload on Keystone

Now that we have explored the structure of the Flush+Reload attack, we elaborate on the affects of Flush+Reload on the Keystone framework. Generally, Keystone is not vulnerable to Flush+Reload attacks because there is no shared physical memory between enclaves. However, in our dynamic library implementation, we allow library enclaves to be shared between enclaves, introducing a cache side-channel vulnerability. We will analyze the risk of a successful Flush+Reload exploit on our design and provide a solution for mitigation.

Attack Scheme. The Flush+Reload attack depends on being able to flush specific lines or all of the cache. RISC-V's recently added cache management extensions, Zicbom [2], provide instructions for flushing, invalidating, and cleaning a cache line. When this extension is enabled, these instructions would allow an attacker to execute Flush+Reload. To attack a victim enclave in Keystone, an attacker can create their own enclave with an EAPP that uses a dynamic library also shared by the victim. The attacker can then periodically flush cache lines shared by their enclave and the victim enclave. By measuring the access time for these memory lines, the attacker can learn information about the victim enclave process.

4.2 Cache Tagging

To defend against this information leakage, we propose a cache-tagging mechanism to prevent an attacker from gaining information about data that is loaded into the cache by separate enclaves. The attacker is able to gain information about victim enclave memory accesses because the attacker is able to (1) flush cache lines corresponding to victim enclaves and (2) hit on cache lines loaded by victim enclaves. We prevent (2) by creating security domains for different enclaves and tagging each cache line with a corresponding domain ID. On a memory access, this domain tag is checked along with the address tag to allow a cache hit only if the memory access comes from an enclave corresponding to the same ID. Otherwise, the cache access results in a cache miss, and a new cache line is loaded in with the enclave's domain ID and the requested data.

By default, each running enclave is assigned a separate domain ID. Since the number of enclaves is limited by the number of PMP entries, we allocate $logN$ bits in each cache line for the domain ID where N is the number of PMP entries on the system. To improve performance, multiple enclaves or PMP regions may be assigned to the same domain ID when mutually trusted, for example, when created by the same user. A diagram showing how our domain ID is added to the cache tag is shown in 6.

The key insight of our method is that every process can still execute **evictions** (1), from any location in the shared cache,

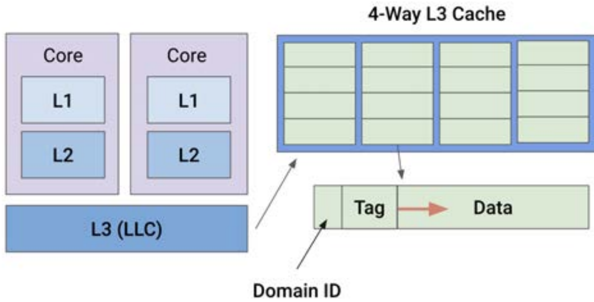

Figure 6: Additional Domain ID is Added to the Cache Tag

but cannot gain significant timing information about cache lines loaded by another enclave.

4.3 Cache Simulation

We use Spike, the RISC-V ISA simulator [1], to create a simulation of our cache-tagging mechanism. We implemented cache tagging on the Spike source code. Finally, we ran our code on two commonly used benchmarks and compare our results to the untagged Spike source code. We find that while performance decreases, the average memory access time (AMAT) only increases by about 4.26% on average.

While SiFive's WorldGuard security feature [4] provides an implementation of a generic system for separating processes into protection domains that can similarly be used to partition caches, we did not have access to the necessary hardware, so we chose to evaluate the performance trade offs of the cache partitioning defense for Keystone through simulation.

4.3.1 Assumptions

We make a few key assumptions when simulating our cache-tagging proposal. First, we discovered that Spike does not have the full support for running the entire Keystone framework. Thus, although Keystone does not require that enclaves are run in separate processes, we model each enclave with a separate simulated hardware thread for the sake of simulation and ease of tagging.

Second, we trust that Spike provides an accurate representation of cache access performance. Spike uses separate instruction and data caches as its L1 cache and uses a unified L2 cache. We assume that the L1 caches are not shared, but the L2, which is the LLC in this case, is shared between processors. Thus, we perform our analysis on the L2 cache, which would be susceptible to Flush+Reload.

Finally, we also assume, along the lines of other work on side-channel mitigations, that denial-of-service is out-of-scope. Since we allow the attacker to flush the cache, just not measure what data is reloaded, the attacker could prevent other enclaves from benefiting from the shared cache. This could significantly reduce performance. While a real concern, this work does not address this risk. Additionally, we do not provide a comprehensive analysis of side-channels within the system. Rather we attempt to prevent risks and evaluate the performance cost of the side-channel attack **exposed** by our DLL implementation.

Cache Sizes	Block Sizes (bytes)	Cache Ways
256KB	64	4
512KB	128	-
1MB	-	-
2MB	-	-
4MB	-	-
8MB	-	-

Table 1: Cache Parameters

4.3.2 Implementation

In Spike, we modify the cache `access` function. When an enclave accesses an address in memory, we check the thread ID of the running thread against the domain ID of the cache line, in addition to the address tag. If the IDs match, then the access is allowed and is treated as a cache hit. In the case that the IDs do not match, this access is treated as a cache miss even if the same block of physical memory is present. On eviction, we do not check domain IDs and evict according to Spike's normal LRU policy.

4.3.3 Experiment Setup

To compare the performance cost of cache tagging, we ran our modified code on RISC-V benchmarks [3] for matrix multiply and vector add. We execute each benchmark with multiple threads, where each simulated hardware thread represents an enclave process.

Initially, when running tests, we observed that the size of the cache negatively correlates with the number of misses. To account for this behavior we test with cache sizes both larger and smaller than the datasets we used. For the vector addition benchmark, we use an 80 KiB dataset, and for matrix multiplication we use a 1 KiB dataset. The dataset sizes differ because it took a prohibitively long time to generate a larger dataset for the matrix multiply benchmark.

Similarly, for our L1 cache design, we observed that a large L1 cache resulted in fewer hits in the L2 cache. Though this is desirable behavior for normal execution, we wanted to compare L2 performance which necessitates L2 usage. Therefore, we restrained our L1 cache to a 4-way cache with only 16 sets and 64-byte cache blocks for both the instruction and data cache. This is much smaller than typical modern day processors, but for our experimental setup it is reasonable because we are focused on the shared LLC only. We also carefully selected LLC cache parameters to use based on

modern processor designs. The parameters we tested with are shown in Table 1.

We ran two types of experiments using Spike. The first used an area-neutral model to examine the effect of adding additional domain ID bytes to the cache line on the cache miss rate. Recall that only an additional $logN$ bits, where N is the number of PMP entries in the system, are needed in each cache line to represent the domain ID. Since RISC-V supports 16 PMP entries in the most conservative case we need 4 bits per cache line for the domain ID. Our baseline Spike simulation used 8 byte tags with 128 byte cache lines. Therefore, to achieve the same cache data capacity we would need a 0.42% area overhead. Unfortunately Spike and typical processors only accept blocks and sets in powers of two. Therefore, in our experiments, we were limited to simulating either half the number of sets or half the block size, representing a 50% area overhead. This provides an extremely conservative estimate of performance decrease due to the overhead cache space used by the domain ID tags.

The second experiment examines the impact of the cache-tagging implementation on miss rates when multiple enclaves are running at once. We expect the miss rates to increase when multiple enclaves are sharing the LLC because the cache lines loaded by one enclave may not be utilized by an other enclave, rather a second enclave must also load the data from DRAM. This creates the effect of aliasing within the cache, decreasing cache efficiency. To test the performance impacts of this phenomenon, we run up to five threads at once using our modified Spike code which implements domain ID tagging.

4.3.4 Results

In our simulations, we found that although miss rates did increase due to our security feature for both the matrix multiplication and vector addition benchmarks, these changes were very small and only had minor impacts on performance. We found that reducing sets to manage the cache capacity had less of an effect on miss rate compared to reducing the block size. Additionally, adding more enclaves in parallel increased the overall miss rates.

Cache Capacity Overhead. In our first experiment we modeled the area overhead by reducing either the number of sets or the block size while keeping the cache size consistent. We found that in our experiments modeling area overhead by reducing sets, L2 miss rates were similar to the baseline, with relatively larger increases on smaller cache sizes for the vector addition benchmark. The baseline parameters used a block size of 128 bytes and sets ranging from 512 to 16384.

For vector addition, we measured an increase of at most 10% when sets were reduced. Results were much less encouraging for reduced block size which miss rates reaching well above 60%. Vector addition results are shown in Figure 7. For matrix multiplication, reducing the sets increased the miss

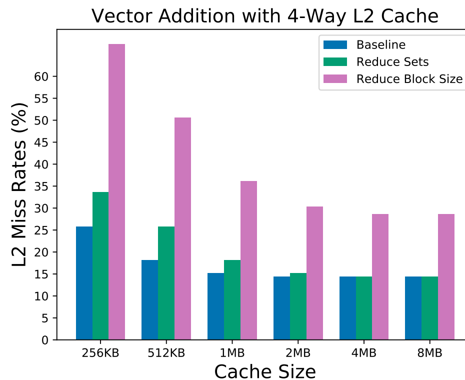

Figure 7: Vector addition benchmark running on Spike with the baseline parameters, sets reduced by a factor of two and block size reduced by a factor of two.

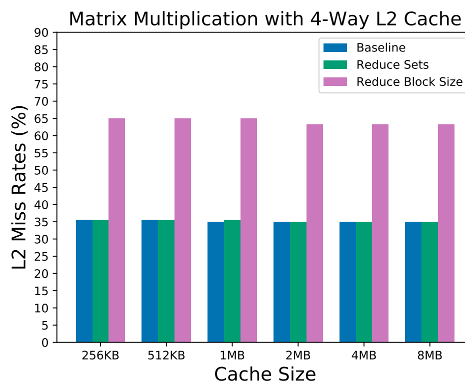

Figure 8: Matrix multiplication benchmark running on Spike with the baseline parameters, sets reduced by a factor of two and block size reduced by a factor of two.

rate by only a few percentages, while again reducing the block size increased the miss rate to approximately 65%. Matrix multiplication results are shown in Figure 8. These results indicate that for our parameters and computational loads it makes sense to compensate for the added space due to domain IDs using additional sets.

Multiple Enclaves. In our second experiment we examined the impact of our cache tagging implementation on L2 miss rate. This was modeled by running up to five threads at a time.

For vector addition and matrix multiplication, Figure 9 and Figure 10 respectively show the difference in miss rate from the baseline unmodified code. For vector addition, when only one enclave is running, the increase is marginal. However the delta is around 25% when five enclaves are running. For matrix multiplication, there is an increase of about 10% for two enclaves. Again the delta gets to around 25% when three to five enclaves are running.

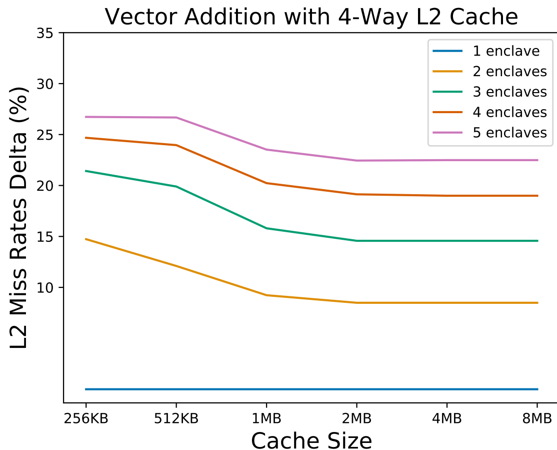

Figure 9: Vector addition running on Spike with one to five enclaves

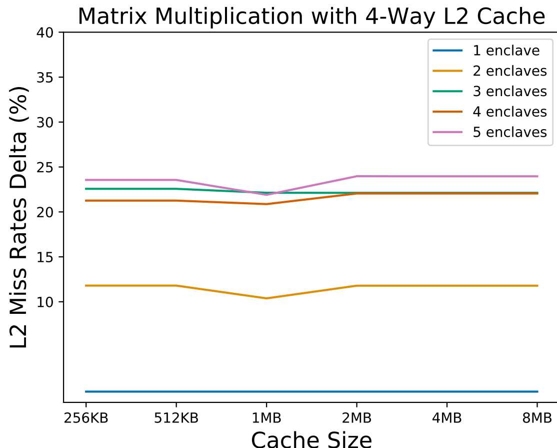

Figure 10: Matrix multiplication running on Spike with one to five enclaves

For this experiment, we also computed the AMAT while running the experiment with different numbers of enclaves. Since our experiments are performed on a simulator, we did not have physical cache hit times from hardware. Our computations assume that L1 hit time is 4 cycles, L2 hit time is 10 cycles, and DRAM accesses take 100 cycles. The percentage increases in AMAT for each benchmark are shown in Table 2.

We found that our tagged implementation had an AMAT of 4.48 cycles on average, which was only a minimal increase compared to the baseline with an AMAT of 4.3 cycles. In the worst case, we found that for the the vector addition benchmark with five enclaves, the tagged simulation had an 11.82% increase resulting in an AMAT of 4.85 cycles. These results show that using this cache tagging security feature will have little impact on the performance of enclaves in Keystone.

Number of Enclaves	% Increase in AMAT for `vvadd`	% Increase in AMAT for `matmul`
1	0.00%	0.00%
2	5.17%	0.96%
3	8.52%	1.92%
4	10.36%	2.10%
5	11.82%	1.75%

Table 2: Percent Increase in AMAT for Vector Addition and Matrix Multiplication Benchmarks

4.4 Discussion

We provide a simulation and evaluation of a viable Flush+Reload attack mitigation for shared dynamic library linking. This mitigation further illustrates the viability of dynamically linked libraries on a RISC-V based enclave system. As with most systems, however, there are trade offs and limitations to our approach.

We see that there are indeed some performance drawbacks to preventing Flush+Reload attacks on our system. When five enclaves are running together on the same system the miss rate increases substantially. This matches our intuition because when there are more enclaves there is more cache space that will miss because it was accessed by another enclave. We argue that these increases are relatively minor and would be acceptable in cases where security is of the utmost importance. For example, for medical or financial data enclave users may value data security over performance. Additionally, the added performance benefits of DLL may outweigh the performance costs of cache tagging for applications that make use of many shared libraries.

There are a few limitations to our approach. Due to time constraints, we only ran one dataset size for each benchmark. It would be interesting to examine how the cache miss rate changes based on the dataset size. While we simulated the effect of adding tags on an area neutral model, adding tags to the cache in hardware may have slightly different effects on performance. We leave the hardware implementation and evaluation to future work.

Unfortunately, we only had the time to study the Flush+Reload attack, but there are still potential side-channel attacks that could be executed against Keystone enclaves. One particular cache side-channel attack that we considered was the Prime+Probe attack. The Prime+Probe attack differs from Flush+Reload in that the attacker primes the cache by filling it with their own data and then measures the timing of accesses to their own addresses to figure out if the victim has accesses physical addresses that map to the same way. While the Prime+Probe attack can still be executed with our cache tagging scheme if an attacker fills an entire cache way with their data, this attack does not rely on shared memory. Thus, Keystone may be vulnerable to this attack, even without the addition of our dynamic library implementation, so we leave

the analysis and mitigation of this side-channel vulnerability to future work on Keystone.

5 Conclusion

We present an implementation of dynamic libraries in an open-source RISC-V framework for customizable TEEs. With our design, we enable efficient memory usage by allowing sharing of dynamic libraries and expedite enclave initialization times by removing library loading from the enclave startup sequence. We analyze a potential cache-side channel attack that results from our library design and provide an efficient and secure cache-tagging solution. This new dynamic library functionality with aid adoption and ease of use for Keystone and encourage further TEE research.

Acknowledgements

I would like to thank Dayeol Lee for the direction he has given me in the preliminary design discussions for the dynamic library implementation and cache tagging schemes. Additionally, I would like to thank Tess Despres for her work on implementing the Spike simulation and data collection for the cache tagging experiments as well as her contribution to the first drafts of this paper, in particular to the section for Mitigating Cache Side-Channels with Shared Libraries. I would also like to thank Anay Wadhera for his help on the initial memory management implementation for the Trusted Bootloader. Thank you also to Professor John Wawrzynek for his encouragements to delve deeper into the the architectural components of this work. Finally, I would like to thank Professor Krste Asanovic for his mentorship, guidance, and feedback throughout this year that have shaped this project. This research was partially funded by ADEPT industrial sponsors and affiliates and Amazon Web Services.

References

[1] RISC-V ISA simulator. https://github.com/riscv-software-src/riscv-isa-sim, 2021.

[2] Cache management operations for RISC-V. https://github.com/riscv/riscv-CMOs, 2022.

[3] RISC-V tests. https://github.com/riscv-software-src/riscv-tests, 2022.

[4] Shield SOC security - SiFive. https://www.sifive.com/technology/shield-soc-security, 2022.

[5] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[6] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[7] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[8] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Xiaodong Song. Keystone: an open framework for architecting trusted execution environments. *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[9] 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.

[10] 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.

[11] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[12] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *ArXiv*, abs/2006.13598, 2020.

[13] 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.

[14] Yuval Yarom and Katrina E. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[15] Younis A Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. A new prime and probe cache side-channel attack for cloud computing. In *2015 IEEE International*

Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, pages 1718–1724. IEEE, 2015.