UC Davis
UC Davis Electronic Theses and Dissertations

Title

Hardware/Software Co-Design for Secure High Performance Computing Systems

Permalink

https://escholarship.org/uc/item/83r536h8

Author

Akram, Ayaz

Publication Date

2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Hardware/Software Co-Design for Secure High Performance Computing Systems

By

Ayaz Akram

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Jason Lowe-Power, Chair

Venkatesh Akella

Sean Peisert

Committee in Charge

2023

Copyright © 2023 by

Ayaz Akram

All rights reserved.

Contents

List of Figures

List of Tables

Abstract

Hardware/Software Co-Design for Secure High Performance Computing Systems

High-performance computing (HPC) is increasingly becoming more data-centric, involving large data sets, rather than its historical focus on modeling and simulation. Sometimes, this data can be sensitive, provided by third parties to HPC centers or individual researchers, and raises security concerns regarding the confidentiality or integrity of the data. Our work aims to provide secure systems focused on HPC centers, without any significant performance reductions. Hardware-based trusted execution environments (TEEs) use hardware-backed techniques to provide some level of assurance for data and code confidentiality, and integrity. We first study the applicability of commercial hardware-based trusted execution environments (TEEs) to enable secure scientific computing. We rigorously analyze the performance impact of general purpose TEEs, AMD SEV, and Intel SGX, for diverse HPC benchmarks including traditional scientific computing, machine learning, graph analytics, and emerging scientific computing workloads. We also analyze the impact of the programming model required by these TEEs. The results show that commercial TEEs do not fit the HPC use case, either because their performance implications are intolerable, they require significant application changes (e.g., partitioning, linking applications against specific libraries), or their threat model does not include all system components that HPC applications might use. We provide a design point for enclaves that does not require an entire OS inside the enclave but can rely on a primarily untrusted OS for resource management. We implement a prototype data enclave, called *DESC*, with multithreading support on the RISC-V ISA that separates the management of the system from the protection of the sensitive data. We show how *DESC* allows an untrusted OS to maintain page tables, service system calls, and manage processes without compromising the enclave applications data confidentiality or integrity.

Cycle-level architectural simulation of trusted execution environments (TEEs) can enable extensive design space exploration of these secure architectures. Existing architectural simulators that support TEEs are either based on hardware-level implementations or abstract

analytic models. To this end, we enable a simulation environment using full-system architecture simulator, gem5, and a RISC-V based open source TEE, Keystone, and show how this simulation support opens new avenues for designing and studying these trusted architectures. Future HPC systems are expected to improve resource utilization by decoupling compute and memory extensively, leading to disaggregated architectures composed of different types of processing elements and remote memory pools. We also explore the expansion of our baseline TEE design (*DESC*) to provide scalable mechanisms that would allow a user to form a secure enclave spanning multiple processing elements.

Acknowledgments

I extend my heartfelt gratitude to my advisor, Prof. Jason Lowe-Power, for his invaluable guidance, unwavering support, and encouragement throughout my research journey. His mentorship has been an endless source of motivation during my PhD studies. I am deeply appreciative of the knowledge and wisdom I've gained from his expertise, and I will always hold profound gratitude towards him. I am equally thankful to my collaborators, Prof. Venkatesh Akella and Prof. Sean Peisert, for their contributions to this work.

I am indebted to the other members of the Qualifying exam committee, Prof. Houman Homayoun, and Prof. Sam King, for their insightful feedback and guidance, which significantly enriched this research.

My deepest appreciation goes to my parents for their unwavering support, sacrifices, and boundless love that have been instrumental in helping me reach my academic goals. I am also grateful to my siblings for their continuous encouragement and support.

I extend my thanks to all the members of the Davis Architecture Research Group (DArchR) at UC Davis. Engaging in discussions with this diverse and talented group has provided me with invaluable insights, both technical and non-technical, and has broadened my perspective across various fields. Their support and assistance have been invaluable to me.

Chapter 1

Introduction

1.1 Secure High Performance Computing

High-performance computing (HPC) is moving away from traditional simulation and modeling to large-scale computational problems involving large datasets. Sometimes this data can be sensitive, provided by third parties to HPC centers or individual researchers, and raises security concerns. This dissertation addresses the imperative need for secure systems tailored to the unique demands of HPC centers and their users while striving to minimize the performance impact.

Security guarantees are mainly encapsulated in the CIA triad (confidentiality, integrity, and availability). Confidentiality involves the protection of sensitive and private information from unauthorized disclosure. Integrity relates to the accuracy and reliability of data and systems. Data integrity ensures that information remains unaltered and trustworthy throughout its lifecycle. And availability ensures that compute and memory resources are always available to the application of interest.

In the world of scientific computing, ensuring security, including integrity and confidentiality, is paramount. Security measures for scientific computing aim to safeguard against cyber threats, preserving valuable computational research resources. Data integrity policies and mechanisms seek to guarantee the accuracy of findings by preventing unauthorized changes, while confidentiality policies and mechanisms seek to protect sensitive information, respecting ethical boundaries and legal requirements. These properties collectively seek to safeguard research investments, maintain reputations, facilitate collaboration, and ensure

adherence to regulations, contributing to the advancement of knowledge and innovation. By enabling secure HPC facilities, we can ensure regulatory compliance and ultimately propel the frontiers of scientific discovery.

Security solutions designed for the cloud computing environment may not apply to high-performance computing (HPC) due to differences in architectural requirements, performance priorities, and data handling workflow between the two types of platforms. On one hand, scientific computing revolves around research-oriented simulations and analyses, often necessitating specialized environments and high-performance computing capabilities. In contrast, cloud computing presents adaptable and diverse resources applicable to various industries, underpinned by features like virtualization and a wide array of services. The convergence of these domains is exemplified by cloud platforms accommodating scientific workloads, providing researchers with immediate access to potent computational resources. While shared security concerns like data integrity and unauthorized access are present in both realms, they manifest uniquely due to differing contexts. Scientific computing underscores research credibility and data safeguarding, whereas cloud computing prioritizes fortifying shared resources, data confidentiality, and adhering to regulations. Given these contrasting computational landscapes, solutions tailored for secure cloud computing might not seamlessly translate to high-performance computing (HPC) environments.

1.2 Trusted Execution Environments and High Performance Computing

Hardware-based trusted execution environments (TEEs) use hardware-backed techniques to provide a certain level of assurance for data and code confidentiality and integrity. In this thesis, we present a systematization of the existing trusted execution environments in industry and academia. We also highlight the common mechanisms these TEEs employ to provide different security guarantees and offer a detailed comparative analysis of different TEE proposals. TEEs are anticipated to be a promising solution for the security challenges in the high-performance computing (HPC) domain. However, we show why the existing TEEs are unsuitable for high-performance computing systems.

In this thesis, we rigorously analyze the performance impact of commercial general pur-

pose TEEs, AMD SEV [3], and Intel SGX [4], for diverse HPC benchmarks including traditional scientific computing, machine learning, graph analytics, and emerging scientific computing workloads. We also analyze the impact of the programming model required by these TEEs. The results show that the existing commercial TEEs do not fit the HPC use case, either because their performance impact is too high, they require significant application changes (e.g., partitioning, linking applications against specific libraries), or their threat model does not include all system components that HPC applications might use.

In this thesis, we present a novel data-centric confidential computing approach that leverages an operating system (OS) for resource management while circumventing the need to integrate the OS into the trusted computing base (TCB). Existing TEEs either require heavy application modifications (because they re-implement a limited set of system calls inside the enclave) or allow unmodified applications but include an entire OS as part of the TCB (e.g., VM-based enclaves). Both of these requirements pose challenges particularly in the context of high-performance computing (HPC) oriented systems.

Our contribution introduces an alternative enclave design paradigm that does not necessitate an entire OS within the enclave but instead relies on a mostly untrusted OS for resource management. To demonstrate this, we implement a prototype data enclave, named data enclave for scientific computing (*DESC*), with multithreading support on the RISC-V ISA that separates the management of the system from the protection of the sensitive data. We show how *DESC* allows an untrusted OS to maintain page tables, service system calls, and manage processes without compromising the enclave applications data confidentiality or integrity. We evaluate DESC using gem5 [5, 6] for performance and QEMU [7] for functional correctness (as executing complete applications in a simulated environment can be impractical). We show that our design correctly executes a set of scientific computing workloads (NAS Parallel Benchmarks, graph workloads, and other modern scientific computing and machine learning workloads) on 1–8 cores. Further, we show there is only minor overhead (less than 5% geometric mean for all benchmark suites) compared to running outside the enclave, even when modeling a 30-cycle memory encryption overhead (less than 20% geometric mean). Cycle-level simulation of TEEs using architectural simulators is vital for comprehensive a design space exploration of these secure architectures. Current TEE simulators stem

from hardware-level implementations or abstract models. To this end, we enable a simulation environment using the full-system architecture simulator, gem5, and the RISC-V based open-source TEE, Keystone [1]. This simulation framework provides fresh opportunities for the design and analysis of trusted architectures. We use the same simulation framework for evaluation of *DESC* referred in the previous paragraph.

Lastly, we delve into potential extensions of our proposed trusted execution environment to diverse heterogeneous environments, such as disaggregated memory systems and accelerators.

In conclusion, the growing utilization of large sensitive datasets within HPC centers underscores the paramount importance of secure data processing. The concept of confidential computing presents a potential avenue for mitigating security risks associated with sensitive data in HPC. Nevertheless, current TEEs, despite being pivotal to confidential computing do not align with the requirements of HPC due to performance issues, programming models, and thread models. Through the proposition of a data-centric TEE, *DESC*, and the vision of extending this paradigm to disaggregated memory systems and accelerator architectures, this dissertation paves the way for robust data protection in future HPC systems.

1.3 Contributions

Following are the main contributions of this work:

- **Striking the Balance Between Security and Performance in High-Performance Computing:** We emphasize the imperative of integrating robust security measures into high-performance computing systems while preserving optimal performance and usability.

- **Identifying Challenges and Advancements in TEE Applicability:** We delineate critical challenges and intricacies pertaining to the effective utilization of Trusted Execution Environments (TEEs) within the realm of high-performance computing.

- **Comprehensive Analysis of TEE-based Confidential Computing Architectures:** We rigorously evaluate the suitability of prevalent commercial TEE architectures for the unique demands of high-performance computing environments, shedding

light on performance, compatibility, and security considerations.

- **Data-Centric Enclave Design (*DESC*) for Scientific Computing Workloads:** This thesis presents an innovative enclave design paradigm, *DESC*, that eliminates the need for a complete OS within the enclave, and still allows an untrusted OS to manage resources for an unmodified enclave application. Tailored specifically for scientific computing workloads, the design choices underpinning *DESC* are both pragmatic and effective within HPC environments, yielding minimal performance overhead due to limited interaction between the OS and HPC applications. Moreover, *DESC* does not require modifications in the applications and allows them to leverage a multitude of operating system management optimizations – vital attributes for HPC workloads.

- **Simulation of Trusted Execution Environments in a Controlled Environment:** We present our approach to simulating trusted execution environments, contributing to the understanding and advancement of secure architectures.

- **Envisioning Future Enhancements and Extensions:** Our work sets the stage for future explorations and enhancements, providing a comprehensive list of potential extensions that can build upon the foundation of our proposed enclave design (*DESC*).

By addressing these key contributions, our research underscores the significance of secure data processing in high-performance computing and paves the way for innovative solutions that bridge the gap between security, performance, and usability.

1.4 Organization

The remainder of this document is structured as follows: Chapter 2 offers a contextual background on the problem of secure HPC and the historical development of confidential computing. Chapter 3 presents a systematic analysis of the existing confidential computing architectures, focusing on their limitations for HPC applications. Chapter 4 thoroughly examines the performance implications of current commercial TEEs on HPC applications and identifies the specific requirements for a TEE tailored to scientific computing needs. Chapter 5 outlines the details of our proposed data enclave design called *DESC*. In Chapter 6, we explore the support of *gem5*-based simulations for evaluating secure compute environments,

serving as the baseline for assessing the ideas presented in this document. Chapter 7 outlines the prospects for future research and development. Finally, our conclusions are summarized in Chapter 8.

Chapter 2

Motivation and Background

In this chapter, we delve deeper into the rationale behind the security requirements within HPC centers and explore the unfolding developments in confidential computing over recent years.

Table 2.1: HPC Use Cases

Domain	**Data Provider**	**Data types**	**Applications**
Health care	Hospital	Health records, medical images, gene sequences	Machine learning models
Transportation	Public transportation authority	Driving routes	Graph analysis
Energy	Utility company	Home and building energy usage	Real time demand or response

2.1 Security Issues in High Performance Computing Environments

Some computational scientific research requires the use of high-performance computing (HPC) centers, that can provide large-scale computing and storage resources to users (researchers). Some scientific computing problems are large-scale and involve large data sets as well. This data is often provided by a third-party (*data provider*) and involve sensitivity of some kind. Figure 2.1 provides a high level overview of how a data provider, an HPC platform provider,

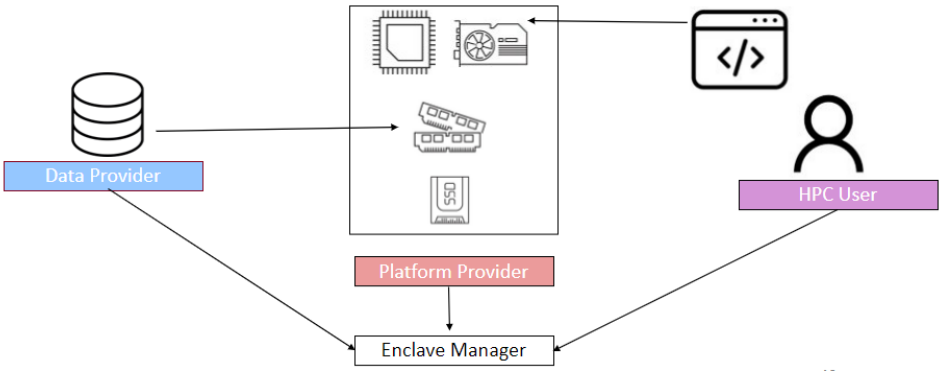


Figure 2.1: Interaction of multiple actors in an HPC center.

and an HPC user might interact with each other. Table 2.1 provides a few examples of scenarios where different data providers might provide some sensitive data to researchers to perform some type of analysis through their applications.

The use of sensitive data in HPC centers make it imperative to build HPC systems that can be secure and can be trusted by all the entities involved in a successful scientific workflow. These entities include, multiple users (who might be sharing all or a part of HPC system resources), HPC platform provider, and the data provider. HPC platform providers have tried to tackle this problem already [8, 9]. However, the current solutions have significant usability challenges. For example, processing sensitive health data requires dramatically different environments compared to those typically used in National Science Foundation (NSF) or Department of Energy (DOE) Office of Science high-performance computing facilities. Processing capabilities are limited to only a handful of racks and access requires virtual private networks (VPNs) and/or remote desktops. These onerous usability requirements are particularly cumbersome for the scientific community that is mostly used to working in very open, collaborative, and distributed environments, potentially with users from all over the globe.

2.1.1 HPC vs. Cloud Systems

The focus of this thesis is on high-performance computing systems, such as those that might be used in HPC centers (e.g., DOE National Labs).

HPC systems prioritize performance as their primary goal. This is in contrast to tra-

ditional virtualized cloud systems, which tend to emphasize manageability and flexibility over sheer performance. HPC systems are designed for executing complex computational tasks that require massive processing power, while virtualized cloud systems are more geared towards providing scalable and versatile computing environments for a wide range of applications. Multiple (sometimes heterogeneous) nodes, many cores per node, and integrated accelerators are some characteristics of the HPC systems. Moreover, applications on these specialized systems often bypass the OS for performance reasons. They do so to achieve peak processing power and minimal latency by establishing direct connections with hardware components, avoiding the overhead introduced by the operating system's abstraction layers. Unlike traditional HPC systems, cloud systems often employ multiple privilege layers. These layers provide enhanced control over access, resource allocation, and scalability, enabling greater adaptability and fine-grained permissions. With the advancements in cloud computing, the boundaries between scientific computing and cloud computing are increasingly getting blurred, however, the cloud computing is still not effective for large scale scientific problems [10, 11].

2.2 Confidential Computing

Confidential computing (which refers to the use of hardware-enforced (cryptographic) protection of data in **use** in contrast to the data at **rest** (storage) or in **transit** (I/O)) has recently emerged as a new paradigm of computing [12, 13]. Confidential computing creates trustworthy systems rather than point-wise solutions against particular attacks. There are two primary ways to enable confidential computing: privacy-preserving computation techniques (like homomorphic encryption[1] and multi-party computation[2]) and trusted execution environments (TEEs). A comparative analysis of these techniques suggests that hardware TEEs generally incur much lower performance costs than other methods like homomorphic encryption and multi-party computation [17]. TEEs scale well to larger data sizes [13] and generally provide several additional security properties like attestability and code confiden-

[1] *Homomorphic Encryption:* A form of encryption that allows computation on ciphertexts, generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext [14, 15].

[2] *Multi-party Computation:* A subfield of cryptography with the goal of creating methods for parties to jointly compute a function over their inputs while keeping those inputs private [14, 16].

Table 2.2: Primary security properties of TEEs

Property	**Definition**
Data Confidentiality	unauthorized view of data is not allowed.
Data Integrity	unauthorized entities are not allowed to alter the data.
Code Integrity	unauthorized entities cannot alter code in the TEE.
Code Confidentiality	unauthorized entities are not allowed to view the code inside the TEE.
Authenticated Launch	enforcement of authorization checks before process launch.
Programmability	if this is a TEE with arbitrary code or fixed function (code).
Attestability	if a TEE can provide evidence/measurement of its origin & current state.
Recoverability	if a TEE can be recovered from a compromised state.

tiality (as shown in Table 2.2). A TEE is defined as follows by the Confidential Computing Consortium [13]:

> "A Trusted Execution Environment (TEE) is commonly defined as an environment that provides a level of assurance of data integrity, data confidentiality, and code integrity. A hardware-based TEE uses hardware-backed techniques to provide increased security guarantees for the execution of code and protection of data within that environment."

Table 2.2 provides a set of fundamental properties inherent to a TEE. Figure 2.2 demonstrates how a trusted execution environment establishes a secure zone of trust for sensitive applications and their associated data. This process is elaborated upon in Section 3.4, that discusses the various mechanisms employed by current commercially-available TEEs to facilitate the creation of the secure enclave. In essence, a TEE constructs this realm of trust by capitalizing on the capabilities of hardware-based security features, secure boot protocols, attestation protocols, encryption, and other isolation mechanisms. These elements combine to isolate sensitive applications and data from external attackers and insider threats.

2.2.1 Beyond TEEs

Fully homomorphic encryption [15], secure multi-party computation [16], and functional encryption all represent methods for computing over encrypted data by leveraging software algorithms, rather than hardware properties. Similar protection properties would apply in these cases, but with two important caveats and one potential benefit: first, these techniques are computationally expense and are therefore significantly slower than hardware TEEs. This is true even though performance has improved from being on the order of 1 trillion times slower than computing in cleartext ten years ago to perhaps only ten to a hundred times slower than computing in cleartext, depending on the technique used and the operations needing to be computed under encryption. For example database searches have been shown to be relatively fast [18–21], but operations requiring both addition and multiplication are much slower. The second caveat is that programs typically need very significant modification to use this technique, often causing each application of the technique to require extensive adaptation of the underlying cryptographic approach. A potential benefit is that leveraging some of these approaches could allow the threat model to be expanded to include protecting against malicious users.

That said, TEEs could also be used to protect against malicious users by incorporating a guard on the output of computation, such as differential privacy [22]. Indeed, a "complete" architecture that we envision is one in which "sensitive" data cannot be computed upon unless inside the TEE, and similarly, that sensitive data cannot be output unless via the TEE, which also enables output to be forced to protected through differential privacy [22] or some other kind of "guard" or gating policy.

2.3 Confidential High-Performance Computing via TEEs

Current solutions to provide security in HPC centers usually require specialized computing facilities and access protocols which can be cumbersome for the users/researchers. In contrast, TEE based security solutions can be built in normal computing facilities (without any restrictions on users' access mechanisms). Figure 2.3 provides a high-level picture of how TEEs can help to create a zone of trust for sensitive data in HPC centers.

This thesis focuses on building trusted execution environments for HPC systems. There

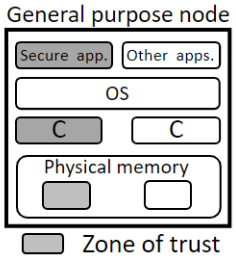


Figure 2.2: Trusted execution in traditional computing systems. 'C' stands for a core. Zone of trust referst to secure computational and memory resources used by a secure application that is enabled via the used of a TEE.

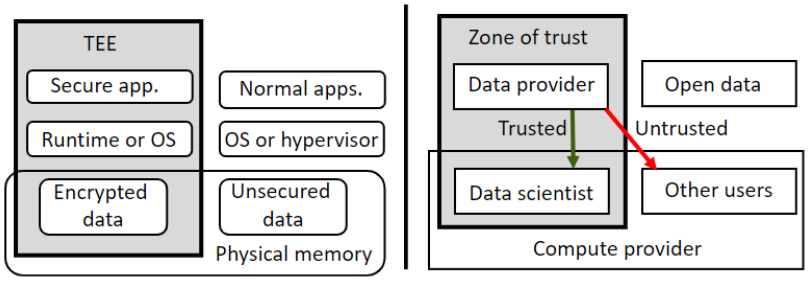


Figure 2.3: Creating a zone of trust for sensitive data in HPC centers. The figure on the left shows a general TEE and the figure on the right shows how that TEE can be used to enable a data scientist to compute on sensitive data provided by a trusted data provider and keep it secure from other entities in the system.

are a number of trusted execution environments that have been introduced by commercial processor vendors like Intel's SGX (Software Guard Extension) [4], ARM's TrustZone [23], AMD's SEV (Secure Encrypted Virtualization) [24], and research platforms like RISC-V's Keystone [1] and Sanctum [25]. A brief survey of these TEEs is provided in the Appendix. they support, the security properties they provide, and the mechanisms they use to pThese TEEs differ in terms of the programming modelrovide those properties.

We show that the current commerical TEEs do not work well for HPC use case because of multiple reasons: 1) their threat model is not a good fit for HPC, 2) programming model of current TEEs does not work well for HPC, and 3) their performance impact can be significant, specially for HPC scale workloads.

We propose a data-centric enclave design called *DESC* (discussed in Chapter 5) that enables secure scientific computing by protecting the data of scientific applications from other software (including the OS) running on a computing system. *DESC* allows the secure execution of unmodified applications while minimizing the TCB size.

Chapter 3

Limitations of Confidential Computing via TEEs for HPC Systems[1]

Trusted execution environments (TEEs) are primary enablers of confidential computing. This chapter presents a systematization of the existing trusted execution environments in industry and academia. We highlight the common mechanisms these TEEs employ to provide different security guarantees and offer a detailed comparative analysis of different TEE proposals. TEEs are anticipated to be a promising solution for addressing certain security challenges in the high-performance computing (HPC) domain. However, this chapter shows why existing TEEs are unsuitable for high-performance computing applications. Finally, we present our call for action to work to evolve the TEE technologies in conjunction with the evolving high-performance computing landscape.

3.1 Introduction

In this chapter, leveraging a survey of the existing literature, we identify the common mechanisms trusted execution environments (TEEs) use to isolate a sensitive application and its state from the rest of the system. We show how existing mechanisms do not fit well with the modern **high-performance computing** systems and what are the most promising directions to pursue to ensure that the high-performance computing systems can maintain

[1]**This work has been published in IEEE SEED 2022 [26]**

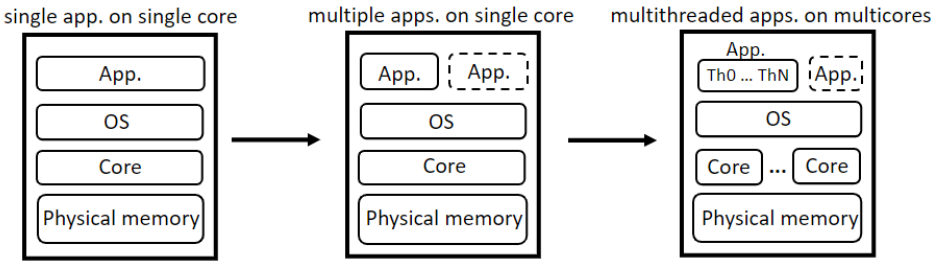


Figure 3.1: History of the computing landscape. This figure shows the evolution process of traditional high-performance computing systems. Computing systems have evolved from single processes on a single-core system to multi-threaded applications on heterogeneous multi-core systems.

isolation of sensitive data [8, 27]. In particular:

- We provide a categorization and systematization of existing trusted execution environments (TEEs).
- We group TEEs based on the key mechanisms/ideas they rely on to figure out the underlying principles that confidential computing is based on today.
- We undertake an examination of the historical progression of conventional computing systems and the implications of this evolution on their security. Using our observations, we point out many ways in which existing TEE technologies would not fit with modern high-performance computing systems in that: 1) they require large application modifications, 2) they have large TCBs, 3) they focus on core-level execution, and 4) they do not take side-channel attacks into consideration.
- We explore future research directions that can enable TEEs to be used for high-performance computing systems.
- We also use the insights discussed in this chapter to build an HPC focused TEE, *DESC*, discussed in details in Chapter 5.

3.2 Computing Landscape

3.2.1 History

Most protection and isolation mechanisms in computing systems (e.g, virtual memory, process isolation) were developed when the computing system model was very different from the

landscape of today [28, 29]. It is essential to look at the history of the computing landscape and its evolution over time. Initially, the computing system model was a single machine with a single core running one application (shown in the left-most part of Figure 3.1) [30]. The operating system would create an environment where the application perceives itself as the sole entity operating within the system [31]. This model had a vast TCB (required trusting all the components in the system). Over time, multiple applications started to share the hardware (still a single-core system, as shown in the middle part of Figure 3.1). The OS time multiplexed the applications on the same hardware. The OS started implementing virtual memory abstraction to isolate one application from the others. Eventually, the computing systems evolved such that the processor became a multi-core processor (shown in the rightmost part of Figure 3.1) [31]. Applications evolved and started to have more than one execution thread. In this model, the OS would manage multiple threads of execution on multiple cores. In summary, the systems became much more complex to manage; however, they still used the virtual memory based isolation primitives to ensure isolation among applications on these multi-core systems.

Traditionally, operating systems were responsible for managing most aspects of memory, IO, and computing. The virtual memory subsystem used for isolation also provided applications an abstract view of physical memory, allowing them to under-subscribe or over-subscribe physical memory. The virtual memory subsystem evolved, but the **coupling of isolation provision, and resource management** stayed intact [32, 33].

3.2.2 Current Computing Landscape

Next, we discuss the current (and future of) computing landscape. In Section 3.5, we will discuss how these advancements in the high-performance computing landscape become the reason for limited applicability of current TEE technologies for HPC. Figure 3.2 shows a system level view of modern high performance computing systems.

Accelerator Integration: Computing systems have started to integrate different types of accelerators with general-purpose CPUs. In modern computing systems, devices like GPUs, FPGAs, and other accelerators have become a part of the virtual memory subsystem and share virtual address space with applications running on the CPU. The memory allocation is still managed by the OS on the CPU, as the accelerators do not run an OS.

Heterogeneous Memory Systems: Heterogeneous memory systems have become much more prevalent today and rely on emerging memory technologies and more traditional DRAMs. For example, Intel's Sapphire Rapids [34] will include an HBM, a DDR5, and a (byte addressable) NVM (non-volatile memory). The rationale behind using different memory technologies is to allow for different memory types to be used for different applications or phases of a single application. This trend of heterogeneous memory systems makes it necessary to have the ability to migrate data from one device (physical address) to the other (physical address).

Remote Memory Systems: With an increasing adaption of systems where the memory might live remotely (be the non-uniform memory access (NUMA) systems or disaggregated systems [35]), memory management might not be entirely done by a (local) OS. Memory management might rely on some remote software/hardware. Network interfaces today have started to rely on RDMA (remote DMA), which bypasses OS and copies the data into a process's virtual address space directly.

Highly Multi-threaded Applications: Especially, in high performance computing systems, the applications are composed of multiple threads, which might execute on multiple cores. Traditionally, the threads executed on homogeneous CPU cores, and the OS managed which threads would execute and where would they execute. In the modern computing systems, the use of accelerators, scale out architectures, and disaggregation of memory resources lead to new models of computing. The host OS might not control all the threads of execution for an application.

Direct Memory Access by Devices: Historically, whenever the devices needed to access application's memory, they had to do that via OS as well. All the memory accesses from/to the device have historically being intercepted by the OS. This is not true anymore for high performance computing systems. DMA and RDMA allow direct access to memory by the devices.

3.3 HPC Focused Trusted Execution Environments

Considering the evolving computing landscape and the security concerns for high-performance computing environments, in this section, we point out important requirements for which we believe that the secure architectures focused on HPC should fulfill. Following are these

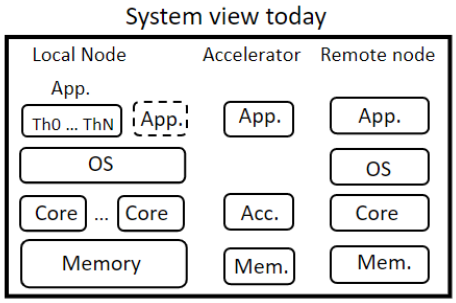


Figure 3.2: Modern high performance computing systems. Applications on these systems scale across local nodes, (integrated or remote) accelerators, and remote nodes.

requirements:

- (R1) *Requirement 1:* HPC-focused TEEs should have a minimum performance impact on HPC-style workloads (heavily multi-threaded and have large working sets).
- (R2) *Requirement 2:* TEEs should not require application modifications or linking against special libraries as HPC applications often rely on third party libraries. However, we note that the applicaiton modifications could benefit security although they might impose more usability requirements on users.
- (R3) *Requirement 3*: HPC-focused TEEs should exclude most of the OS from the TCB. Since, modern HPC applications often bypass the OS for performance benefits (by handing I/O in user-space libraries), reliance on OS security primitives should be minimal.
- (R4) *Requirement 4:* HPC-focused TEEs should be capable of expanding across compute nodes as HPC applications mostly scale across multiple nodes and rely on message passing run-times like MPI for communication across these nodes. Moreover, HPC centers (like data centers) are expected to rely on disaggregated architectures (e.g. pooling of memory resources), to increase the utilization of compute/memory resources and save the cost. An HPC focused TEE should consider disaggregated resources in its threat model as well.
- (R5) *Requirement 5:* HPC-focused TEEs should enable enclaves which can scale to processing elements other than the general purpose CPUs.

There exist multiple TEE technologies today. In the next section, we will analyze if these existing techniques fulfill the previously mentioned requirements of an HPC-focused TEE.

3.4 Systematization of TEEs

In this section, we systematize and classify the existing TEEs into different categories.

Generally, TEEs provide complete control over the trusted computing base (TCB) [12]. The data/code confidentiality and integrity properties of a TEE are usually enabled by isolating an enclave's memory (via the zone of trust shown in Figure 2.2) from the rest of the system while an enclave is in use. Before providing a classification of TEEs, we will first look at some of the common primitives TEEs use to isolate an enclave from the rest of the system. We also discuss the mechanisms/ways the software or other hardware components use these primitives.

3.4.1 Page Table Entry Metadata

Page table entry-level metadata refers to any physical page metadata that TEEs might maintain to identify an enclave page in the hardware. Multiple TEEs rely on this information to implement access control mechanisms. For example, SGX [4] maintains EPCM (enclave page cache map) entries which keep track of the enclaves that own the pages in EPC (enclave page cache), along-with information on the validity of the EPC page. Only SGX instructions can update the entries in the EPCM; therefore, the system software can track any unwanted change in the enclave's address map. Another example is AMD SEV [3], which uses bit 47 of the physical address in a page table entry to identify whether this page is secure. The hypervisor or the host OS manages this bit.

3.4.2 Encryption

Encrypting physical memory is a very common primitive used by TEEs (e.g., AEGIS [36], SGX [4], Graviton [37], HETEE [38], ARM RME [39]) to ensure confidentiality of data belonging to an enclave (and can be a strong mitigation against physical attacks). Usually, the TEEs rely on an encryption key that is generated and stored in an (isolated) trusted pro-

cessor (or some hardware component). For example, AMD SEV [3] relies on an ARM-based processor (AMD Platform Security Processor) for key management. As the cache blocks move from/to the processor chip to/from the DRAM, the key is used to encrypt/decrypt the cache blocks transparently.

3.4.3 Physical memory isolation via ISA extensions

RISC-V based TEEs (e.g., Keystone [1], CURE [40], Elasticlave [41], TIMBER-V [42]) rely on the PMP (physical memory protection) ISA extension. PMP controls U (user) and S (supervisor) modes' access to certain memory regions. The allowed access (r-w-x) permissions and the memory region can be configured using PMP address (pmpaddr) and configuration (pmpcfg) registers. There also exist proposals for providing physical memory protection to IO devices via IOPMP [43].

3.4.4 Use of tags/identifier in hardware

Some TEEs also use a tag or identifier to distinguish enclave data from other software in the system (for access control). For example, CURE [40] uses an enclave ID for bus arbitration to enable enclave to the peripheral binding. For this purpose, CURE [40] hardware relies on a filter engine on the system bus. Bastion [44] depends on a *module ID*, which is a new component in caches and TLB and acts as a tag for the currently executing process. ARM TrustZone [23] uses a single-bit identifier to distinguish between the secure and non-secure world (for device communication). SiFive's WorldGuard [45] can tag bus transactions to differentiate between software contexts that originated a request, allowing the target to determine if it trusts the requestor. HECTOR-V [46] also relies on identifiers embedded in interconnects, which helps create a safe IO path. AMD SEV [3] hardware tags all code and data with its VM ASID (inside the SoC), indicating the VM, which is the data owner.

3.4.5 Privileged Software/Hardware

Trusted execution environments mostly do not trust the host OS and try to bypass the host OS privileges. They usually do this through additional hardware/software components

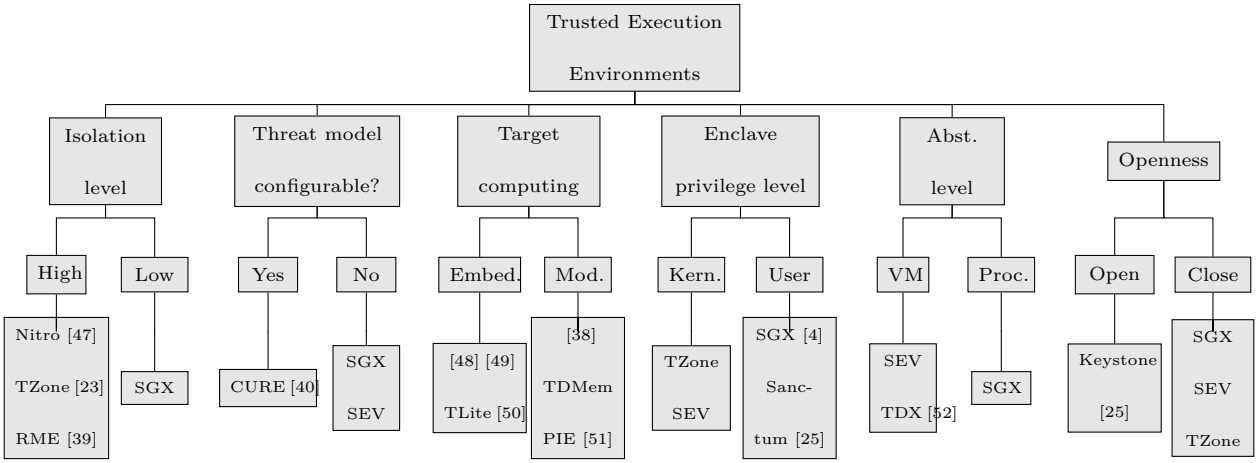


Figure 3.3: Classification of TEEs and some examples of each class. [**Note:** Emb. : Embedded, Mod.: Modern, Kern. : Kernel, Cont. : Container, Proc. : Process, VM : Virtual Machine]

to perform privileged operations focused purely on security. For example, CURE [40] uses a hardware-based security monitor to monitor the system bus's access. Keystone [1] uses an M-mode software-based security monitor to manage physical memory isolation primitive (i.e., PMP). Similarly, ARM RME (realm management extension) [39] relies on a monitor to enforce its security guarantees.

3.4.6 Classification of TEEs

We present a classification taxonomy of existing trusted execution environments to enable a better understanding of the vast design space that is covered by TEEs. Figure 3.3 shows this taxonomy with some examples of TEEs from each class. TEEs can be classified based on different factors. We use the following factors for this classification:

- *Isolation level:* This defines at what level the secure and non-secure components are isolated.
- *Threat model configurability:* This determines if the threat model of a TEE can be configured (either at the run time or the implementation time).
- *Enclave privilege level:* This is the privilege level at which the enclave operates.
- *Openness:* This determines whether a TEE is open-source or closed-source, with closed-source models often prevalent in industrial solutions and open-source models more

commonly found in academic contexts.

- *Abstraction level:* This is the level of abstraction at which the TEE provides an interface to the user.

- *Target computing:* This is the type of computing which the TEE is mainly designed for.

A more systematic and detailed comparison of different TEEs is provided in Table V (in Appendix section). Here, we provide some discussion and observations on different classes of TEEs that are shown in Figure 3.3 with important examples.

We observe that the current TEEs which provide the highest isolation level usually achieve it via physical isolation or partitioning at a very coarse granularity. For example, AWS Nitro enclaves are an example of highly isolated enclaves that provide (constrained) enclave virtual machines with no storage, network, or interactive access [47]. AWS Nitro enclaves have only a single point of connection to the outside world via a bi-directional virtual machine socket (*vsock*) between the parent instance and an enclave [47]. The major drawback of highly isolated enclaves is the difficulty to use them. For example, applications will have to rely on message passing, RPC or micro-services to interact with their secure compartment on the enclave virtual machine (in case of AWS Nitro). Other examples of highly isolated enclaves, ARM TrustZone [23] and Realms [39], partition the entire physical address space at a very coarse granularity (into secure and non-secure worlds).

There are also many examples of configurable TEEs (which can lead to variable TCB sizes). Configurability is a desirable property in the current heterogeneous world. Applications executing on a modern (heterogeneous) HPC system might not have the same sensitivity level (or require the same security guarantees, e.g., integrity is not essential if the application is not going to reuse previously written data). CURE [40] provides the ability to define enclave trust boundaries (at different granularity levels). AEGIS [36] provides the ability to have both a trusted and untrusted OS. ShEF [53], a trusted execution environment for FPGAs, provides the ability to customize encryption logic parallelism and authentication block size.

TEEs could opt for a specific security vs. cost tradeoff depending on the computing type

they are targeting. We observe that most of the earlier TEEs focused on general-purpose desktop/cloud or embedded computing (e.g., [3, 23, 48]). However, some recent examples of academic proposals target parts of modern computing systems. For example, HETEE [38] targets server rack-scale computing. Graviton [37] and HIX [54] tried to enable isolated execution on GPUs, ShEF [53] targets FPGAs and TDMem [55] focused on RDMA-based disaggregated systems.

Privilege-level based classification divides enclaves into kernel-space or user-space enclaves. Kernel-space enclaves can run trusted kernel-mode software inside the enclave, which means that these enclaves generally have a large TCB. For example, Keystone [1] requires having a kernel-space runtime (for user-space application's resource management) inside the enclave. SEV [3] allows kernel-space enclaves, where the guest OS is a part of the enclave. SGX [4], on the other hand, is a user-space enclave and has smaller TCB (compared to SEV). However, user-space enclaves have to pass the (trusted) user-space and (untrusted) kernel-space boundary to perform system-level services, which can also have security concerns.

3.5 Limitations of Existing TEEs

Next, we discuss some limitations of existing trusted execution environments and show how they hinder the adoption of secure execution environments for modern computing systems.

Confidential computing environments rely on hardware primitives to protect or isolate an enclave's memory from the rest of the system. These primitives can sometimes impose restrictions on the system's resource management, decreasing the usability and efficiency of the system. We now present the following observations on the kind of limitations confidential computing can impose on modern HPC environments.

3.5.1 Heavy Application Code Modifications

Currently, we lack the necessary primitives that allow fine separation of management and protection within a computing system. Consequently, in the context of confidential computing threat models, the entire operating system is generally considered untrusted.

This limitation significantly impacts the programming model of most TEEs, leading to reduced support for traditional C libraries. For instance, simpler libraries like *muslc* are favored over more complex alternatives such as *glibc*. Consequently, userspace applications,

particularly large ones found in high-performance computing systems, necessitate extensive modifications and experience functional limitations.

3.5.2 Large Trusted Compute Base (TCB)

Given the limited trust placed in the operating system within the confidential computing threat models, entrusting resource management to the OS can introduce vulnerabilities. For example, managing an enclave's address space allows a malicious OS to launch page fault-based attacks on enclaves leaking the access patterns of the sensitive application. These attacks are possible because OS can modify access permission of enclave's pages, which would lead to page faults, and thus OS can determine the enclave access pattern. Such attacks, called *controlled channel attacks* [56] are deterministic (and noise-free) and can have large leakage bandwidth compared to other noisy side channels. The proposed solutions to the controlled channel attacks require the enclave to control its page tables and enforce secure-paging policies within an enclave. Examples of such proposals include Autarky [57], Keystone [1], and CURE [40]. The drawback of these approaches is that they lead to a *larger TCB* and more complexity in the enclave.

The scheduling and synchronization of threads by an untrusted OS can lead to multiple security issues.[2] For example, an untrusted OS can influence a machine learning model leading to poisoning attacks by controlling the order in which the threads of the training algorithm are executed [58]. To solve this problem, some TEEs have implemented limited thread handling inside the enclave, which reduces the system's efficiency overall. For example, enclaves (like SGX [4]) enforce a static number of threads because they might only allow statically-defined entry points for executing threads. Many of the TEEs based on SGX have similar limitations. Enclaves like Keystone [1] do not support multi-threaded execution at all at the time of writing this thesis. In summary, today's enclaves generally do not have good support for multi-threaded execution unless they are willing to have a *large TCB*. Interestingly, virtual machine (VM) based enclaves include a guest OS in the TCB and allow multi-threaded applications to run transparently. Not only do the VM-based enclaves have a very large TCB, but multi-threaded execution in virtual machines can also have

[2]Scheduling based denial of service attacks are common, but generally not a part of the threat model of confidential computing systems.

significant performance implications. For example, when threads yield during synchronization operations, they can cause costly KVM exits [59, 60].

3.5.3 Focus on Core Level Execution

A significant limitation of most of today's TEEs is that they focus on a *core-level* view of memory permissions. This behavior limits the applicability of the TEEs to heterogeneous HPC systems.

The absence of an OS or other privileged software on accelerators implies that the memory management for accelerators would also be performed by the host OS.

The absence of an operating system (OS) or other privileged software on accelerators means that memory management responsibilities are delegated to the host OS. Furthermore, accelerators tend to be highly sensitive to address translation latencies, as highlighted in prior research [61], making access control through additional privileged components a complex undertaking. Consequently, when the memory-level view of access control is inaccessible, it becomes challenging for various computing elements to share a trusted memory space. This challenge also extends to disaggregated or remote memory systems, where the host OS and other privileged software or components on the host node do not possess complete control over remote resources. In such scenarios, relying on a core-level execution view to ensure security becomes increasingly difficult.

Another implication of *core-level* view of memory permissions is that they require synchronization of memory permissions across the cores, which are used to execute all the threads of an application. This synchronization, which is today done through inter-processor interrupts, can be costly [62]. Moreover, the synchronization becomes even more costly when the application scales to accelerators or remote compute nodes.

Devices also suffer because of the core-centric design approach of current TEEs. Today, most TEEs use untrusted (shared) memory buffers (e.g., bounce buffers in Linux) as temporary storage for the data moving between the devices and an enclave. The use of temporary buffers leads to extra copies of the data and has performance implications as well. This behavior also implies that the DMA functionality does not work securely with current TEEs.

3.5.4 No Consideration of Side Channels

TEEs do not consider system components that are not memory or cores. In other words, current TEEs generally do not focus on things that are not architecturally visible. This makes cache or system-bus-based side channel attacks possible on TEEs [63–65]. High bandwidth leakage channels can be possible, especially on modern high-performance computing systems with high-bandwidth links between physically isolated components.

3.5.5 Other Limitations

This subsection briefly discusses other, less critical, limitations of TEEs that do not fit in any of the above categories.

3.5.5.1 Memory Isolation Primitives and Fragmentation

Most of the hardware primitives used by TEEs today limit the maximum number of enclaves possible or cause fragmentation issues. For example, ARM TrustZone [23] uses an address space controller to create an OS hypervisor mapping. Keystone [1] relies on contiguous physical memory for an enclave (as PMP defined physical memory range has to be contiguous). Similarly, CURE requires the physical memory of an enclave to be contiguous. If multiple enclaves are executing simultaneously, the requirement of contiguous physical memory for each enclave can cause fragmentation and potentially overuse of resources.

3.5.5.2 Limitations on maximum number of enclaves

The memory isolation primitives used by the existing TEEs can also limit the maximum number of executing enclaves simultaneously. For example, PMP-based TEEs (like Keystone) cannot have more enclaves than the number of PMP entries (latest specifications [66] allow up to 64 entries). Similarly, AMD SEV has limitations on the number of maximum enclave VMs. The maximum number possible on the AMD EPYC system was 15 due to a fixed number of slots for encryption keys (one needed for each enclave VM) in the memory controller [67]. Sanctuary [68] also has a limitation on the maximum number of enclaves due to address space controller constraints [69]. CURE [40] can support 13 enclaves concurrently due to limitations of the hardware arbiter used. The limitation on a maximum number of

enclaves can be an important issue for multi-tenant computing systems.

3.5.5.3 Limitations on data movement

Cryptographic isolation primitives inhibit the transparent data movement in heterogeneous memory systems. For example, to ensure that two same plain text pages at different physical addresses have different cipher texts (as a protection mechanism against cipher-text block move attacks), AMD SEV uses a physical address-based tweak algorithm [70, 71] which uses a block's physical address and an encryption key (xor-encrypt-xor tweak [72]). Since the host-physical address is used to determine the cipher-text of a page, the hypervisor cannot move a page between the two physical addresses once it is allocated to the secure VM. The hypervisor has to lock the physical pages in memory which leads to pre-allocation of all the required physical memory and can cause under-utilization of resources and unintended effects on NUMA affine workloads [59]). The transparent movement of physical pages from one device to the other require the data to transit via the memory controller so that it can be decrypted and re-encrypted again using the new physical address, which can be costly.

3.5.5.4 Compute on Modern Computing Systems

The state of an enclave or secure process on context switch cannot be protected easily if the enclave is scaling across multiple computing elements, some of which may lack an operating system and be physically dispersed.

3.6 Potential Research Directions

In this section, we discuss the promising research directions enabling confidential computing on high-performance computing systems. These research directions can help in mitigating the critical limitations of today's TEEs: heavy application changes, large TCB, core-level isolation view, and inability to protect against side-channels.

Figure 3.4 shows the trust model we need for modern high-performance computing systems (like the one shown in Figure 3.2). The local (general purpose) node, accelerator node, and remote node share part of the trusted memory and should not need to trust any component other than the core(s) they are executing on.

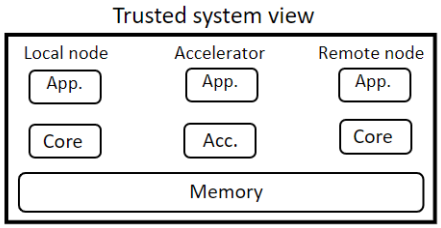


Figure 3.4: Required trusted system view. All compute elements and the memory employed by the secure application should exist within a unified trust boundary.

We argue that given the way computing is evolving, we do not necessarily treat security and performance as a trade-off, but we can achieve both together. We emphasize that the system view in Figure 3.4 also fits well with the optimizations which can extract more performance from a computing system. Therefore, synergistically building secure and performant systems is an appropriate approach. A similar observation by Orenbach et al. [73] suggests that enclaves have many similarities with accelerators: significant invocation overheads, space constrained private memory, and inability to directly invoke OS services such as network and I/O. The solutions to these problems for accelerators [74, 75] mostly involve bypassing the OS (for performance reasons), which is an attractive property for enclaves as well.

We will discuss some of the promising future directions, like new hardware primitives, horizontal privilege levels, and capability-based enclaves in more detail in Chapter 7. This section discusses a promising direction that we followed to build the proposed TEE in this thesis, i.e., data-centric enclaves.

3.6.1 Data Centric Enclaves

Current TEEs, when trying to isolate software from the rest of the system, for example via TEE-based containers, have usability constraints and eventually try to emulate existing system components (like POSIX or devices) inside the enclave systems and eventually have to deal with the same problems they started with [76]. Since today's threat models mainly consider untrusted software, the "unit of protection" should be individual data items [76]. Data-centric enclaves can solve this problem, which inherently rely on memory/data level isolation view rather than core-level isolation view. One example of such architecture is *Border Control* [77] which keeps the protection checks of IO-MMU consistent with the TLB

Table 3.1: Survey of Attacks on TEEs/Enclaves

Type of attacks	**Examples**
Side channel	[63–65]
Controlled channel	[56, 78]
Encryption Attacks	[70, 71]
IO Based Attacks	[72, 79]

Table 3.2: Example of Tools/Frameworks for TEEs

Type of tool	**Examples**
TEE Containers	Graphene [80], SCONE [81], SGX-LKL [82]
Simulation	FireSim [83], gem5 [84]
Profiling	TS-Perf [85], sgx-perf [86] , Tee-perf [87]

checks via a hardware structure. This way, *Border Control* can maintain the memory level view of permissions and protect against accelerator-based attacks.

We discuss more details of the data centric TEE, *DESC*, proposed in this thesis in Chapter 5.

3.7 Other Topics

In this section, we briefly discuss topics that are not in the main scope of this chapter.

3.7.1 Survey of Attacks on TEEs/Enclaves

There is a lot of research on bypassing the security guarantees of TEEs/enclaves. Table 3.1 shows some of the attacks that are possible on TEEs/enclaves.

3.7.1.1 Protection Against Side Channel Attacks

TEEs mostly do not protect against side channel attacks. However, there are a few exceptions [25, 40]. Komodo [88] obliviates computing to protect against side channels. Sanctum [25] protects against cache side channels by enforcing distinct cache sets per enclave. Keystone [1] also provides the ability to include side channels in the threat model.

3.7.1.2 Protection Against IO Attacks

Since IO devices are generally not a part of the CPU package and are not trusted, they can come from a malicious vendor. Such devices can break the confidentiality of the enclave's data when it leaves the CPU package. For example, Lee et al. [79] presented an off-chip attack on enclaves by snooping the memory bus. Some of the TEEs (e.g., CURE [40], HECTOR-V [46]) use enclave to peripheral binding to protect against IO-based attacks.

3.7.2 Tools for TEE Platforms

Table 3.2 provides a brief survey of different kinds of tools/infrastructure that can help the usage of TEEs or advance research on TEEs. TEE containers help in the execution of unmodified code on TEEs. These containers often provide an emulated view of specific system components and might have many of the limitations of the underlying TEE. Application profiling helps to understand their behavior better and potentially optimize their execution on given hardware. Standard profiling tools might not be able to interact with the enclave applications due to the specialized execution mode of enclaves. Though some specialized profiling tools exist for enclaves, as shown in Table 3.2, there is still room for improvement in this space. Simulation support for TEEs is essential. TEEs usually rely on a hardware-software co-design. However, most of the architectural simulators are not full-system and might not be able to support all components needed to simulate a TEE. On top of that, the details of the targeted TEE might not be openly available. However, there are options for the simulation of RISC-V-based TEEs (as shown in Table 3.2).

3.7.3 Formal Verification of TEEs

Formal verification provides means to evaluate if a security mechanism is correct and does what it claims. There are a few examples of TEEs which have been formally verified. Komodo [88] and Sanctum [25] are a couple of examples of TEEs with a formal proof of their correctness. RISC-V's PMP (which is used by many TEEs, e.g., Keystone [1]) has also been formally verified [89].

3.8 Conclusion

In this chapter, we provided a systematization study of existing trusted execution environments (TEEs) which are one of the main enablers of confidential computing. We discussed the primary mechanisms or primitives the existing TEEs use. We also provided a list of the limitations of the existing TEEs, which we believe are the main reasons why the current TEEs are not suitable for high-performance computing. The existing primitives to build TEEs require large application modifications, lead to large TCB, focus on core-level execution, and do not consider side channels a part of their threat model. These limitations make it very hard to run HPC applications under TEEs, cause significant slowdowns for HPC workloads, and do not ensure their security due to an insufficient threat model. We believe the existing TEE technologies are point solutions for different computing targets. And in the future, we need to either generalize the TEE technologies to be able to use them for any computing domain or come up with point solutions focused on high-performance computing. We also provided a list of the directions we believe can enable TEEs to be a good fit for high-performance computing systems.

Chapter 4

A Study on the Performance of Commercial TEEs for Scientific Computing [1]

CPU vendors have already introduced multiple TEEs which leads to an important question: *Are these commercial TEEs a good fit for HPC workloads in their current form?* Alongside their security features, analyzing these TEEs involves considering two crucial criteria: performance impact and usability. Therefore, this chapter discusses the performance impact of executing traditional scientific computing as well as modern HPC workloads in trusted execution environments in detail. We used x86 based commercial TEEs, Intel SGX [4] and AMD SEV [24], which focus on general purpose compute devices.

4.1 Threat Model

We assume that HPC system administrators are not trusted and that host operating systems and hypervisors are not trusted. However, the guest operating system of a virtual machine, which is owned by the user, is trusted. We assume very simple physical attacks are within scope, but that physical attacks that are more time consuming, such as opening a rack-mount HPC system and removing chips soldered on the board, are less important at this time because there are other means, such as video cameras pointed at the HPC systems, to monitor and mitigate such attacks. We assume HPC users themselves are trusted to not

[1] **This work has been published in IEEE IPDPS 2021 [59]. More details on this can be seen in the original paper [59] or the extended version [90].**

exfiltrate their own data, though we do not trust them to not attack others. Also, we focus on general-purpose computing hardware—FPGAs, GPUs, dedicated ASICs are not considered in this paper, mainly because no commercial TEEs yet exist for these hardware accelerators.

We assume that data providers trust the data users or that some other means (e.g., differential privacy [22]) will ensure the sensitive data is not improperly exfiltrated by the scientific application developers and users. Figure 2.2 shows how TEEs fit into this threat model.

4.2 Selected TEEs for This Study

Trusted execution environments in hardware, at minimum, provide some degree of hardware-enforced separation from other users and processes, and the ability of end users to verify through cryptographic attestation that execution is taking place within the TEE. Some TEEs, including Intel's Software Guard Extensions (SGX) and AMD's Secure Encrypted Virtualization (SEV), also support encrypted memory. Both SGX and SEV protect against malicious system administrators and host operating systems. TEEs have their roots in earlier cryptographic hardware functions, including Trusted Platform Modules. In this work, we analyze the performance of AMD SEV [24] and Intel SGX [4]. We exclude the other major commercially available option ARM TrustZone [23] from this study as existing TrustZone based TEEs mainly target embedded and mobile devices, not general purpose compute devices [91]. Table 4.1 shows a short feature comparison of both SGX and SEV.

4.2.1 Intel Software Guard Extensions (SGX)

Intel SGX divides the application into two code segments, untrusted and trusted (enclave) which cannot directly communicate and interact. Only the trusted part is allowed to access confidential data residing in encrypted form in a memory region called Enclave Page Cache (EPC). The need to split an application (manually) into trusted and untrusted parts can be a challenging task for HPC applications as they often rely on many third-party libraries. The size of the EPC is set to be 128MB, out of which almost 32MB is used to store the metadata needed to provide security guarantees [92]. In case of SGX, the MEE (memory encryption engine) which sits besides the memory controller on the CPU package is responsible for permission checks for EPC accesses, provision of data confidentiality by encrypting the data

when it leaves the CPU package to reside EPC and performs integrity tree operations on the data stored in the EPC.

Both parts of an SGX application communicate through an interface of in/out calls (*ecall/ocall*). *ecall* and *ocall* perform a secure context switch which includes: enabling/disabling of tracing mechanisms, permission checks for enclave memory, validation of enclave control structures and backing up/reloading of registers that represent untrusted execution context [93]. Similarly, enclave code cannot use normal system calls directly, rather the control needs to be transferred to the non-secure part of the application first using *ocall*. SGX requires application changes and/or recompilation. However, there are third-party solutions (e.g. SCONE [81]), which allow running unmodified workloads, but they have their own limitations (discussed in section 4.4.4). SGX also provides integrity guarantees through the use of integrity trees consisting of counters to keep track of version of EPC pages to protect against replay attacks.

4.2.2 AMD Secure Encrypted Memory (SEV)

In case of SEV, the protected memory can be equal to the size of the entire physical memory. AMD SEV provides transparent encryption of memory used by virtual machines (unique encryption key associated with each isolated guest). As a result, SEV has a larger trusted computing base (TCB), compared to SGX, which includes the guest OS, the hypervisor, and the CPU package. In contrast to SGX, which requires application modifications, SEV does not require changes in an application's code. However, the application needs to be run inside a VM managed by the hypervisor (QEMU). SEV lacks integrity support and does not provide protection against replay attacks. However, AMD had latern introduced SEV-ES [94] that adds encryption of guest register state to provide additional protection against VM state related attacks, and SEV-SNP [94] provides integrity checks. Our evaluation study presented in this chapter is based on only SEV.

4.3 Methodology

We picked traditional scientific computing workloads as well as modern applications which fit the criteria of HPC application domain. Table 4.2 provides a summary of the workloads evaluated in this work.

Table 4.1: Feature Comparison

Feature	**SGX**	**SEV**
Integrity Provision	Yes	No
TCB Size	Small	Large
Secure Memory Size	128 MB	Up to RAM size
Application Changes	Required	Not Required

4.3.1 Traditional HPC Benchmarks/Kernels (NPB)

We evaluate workloads traditionally used to benchmark HPC systems such as the NAS Parallel Benchmark suite (NPB) [95]. The NAS Parallel Benchmark suite, consisting of different kernels and pseudo applications, has been used to study HPC systems for a long time and is still being updated. These benchmarks can be used with multiple input data sizes, thus different class names. In this work, we used NPB Class C for both SEV and SGX and NPB Class D for SEV only.

4.3.2 Modern and Emerging HPC Workloads

Apart from the traditional scientific computing kernels/workloads, we also focus on workloads which characterize modern HPC usage. We selected a set of graph workloads (GAPBS) [96] with an input of a graph of road networks in the US. As a proxy for general machine learning training we used a decision tree workload (LightGBM) [99] (characterized by irregular memory accesses) which is trained using Microsoft's Learning to Rank (MSLR) data set. Finally, we used modern HPC workloads as well, including Kripke [97] (a particle transport simulation), LULESH [98] (a hydrodynamics simulation), Mobiliti [100] (a transportation benchmark), and BLAST [101] (a genomics workload). Kripke [97] is a highly scalable code which acts as a proxy for 3D Sn (functional discrete-ordinates) particle transport. The Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) [98] application solves a simple yet "full-featured" hydrodynamics simulation problem. Mobiliti [100] is a transportation system simulator (based on parallel discrete event simulation), designed to work on high performance computing systems. Basic Local Alignment Search Tool (BLAST) [101] is a well-known bioinformatics tool, which is used to search sequence similarity of a given

Table 4.2: Details of the workloads evaluated.

NAS Parallel Benchmarks NPB [95]		
Benchmark	**Description**	**Working-Set (C & D)**
bt	block tri diagonal solver	0.68 & 10.67 GB
cg	conjugate gradient	0.36 & 16.31 GB
ep	embarrassingly parallel	0.028 & 0.028 GB
is	integer sorting	1.03 & 33.1 GB
lu	lower-upper gauss-seidel solver	0.59 & 8.89 GB
mg	multi-grid method	3.3 & 26.46 GB
sp	scalar penta diagonal solver	0.78 & 11.62 GB
ua	unstructured adaptive mesh	0.47 & 7.30 GB
GAP Benchmark Suite [96] (road network)		
Benchmark	**Description**	**Working-Set**
bc	betweenness centrality	1.15 GB
bfs	breadth first search	0.97 GB
pr	page rank	0.97 GB
sssp	single-source shortest paths	1.39 GB
cc	connected components	0.96 GB
tc	triangle counting	0.57 GB
Other Modern HPC Workloads		
Benchmark	**Description**	**Working-Set**
Kripke [97]	Hydrodynamics Stencil Calculation	7.4 GB
LULESH [98]	Particle Transport Simulation	0.108 GB
LightGBM [99]	Microsoft Gradient Boosted Decision Tree Framework	5.4 GB
Mobiliti [100]	Transportation System Simulator	1.06 GB
BLASTN [101]	Basic Local Alignment Search Tool	26.20 GB

genome sequence compared to an existing database. We specifically use BLASTN in this work, which is a version of BLAST used to search a nucleotide sequence against a nucleotide database.

4.3.3 Hardware Platforms Used

Table 4.3 shows the configurations of the hardware platforms used for these experiments. For all of our evaluations, we evaluate *without* hyperthreading by limiting the number of threads to the number of cores on each platform.

We used three server class AMD machines. Figure 4.1 shows the detailed NUMA configuration of the AMD EPYC 7401P (Naples architecture, Figure 4.1a) and the AMD EPYC 7702 (Rome architecture, Figure 4.1b). The Naples-based system has 24 CPU cores with 6 cores on each of four dies in a single multi-chip module. Although this system is a single socket platform, it has four NUMA nodes. A multi-chip module package has characteristics similar to a multi-socket system in terms of latency and bandwidth between separate dies. With its four NUMA nodes the Naples-based system has high variation in memory latency depending on if the data is in the local NUMA node or one of the remote NUMA nodes.

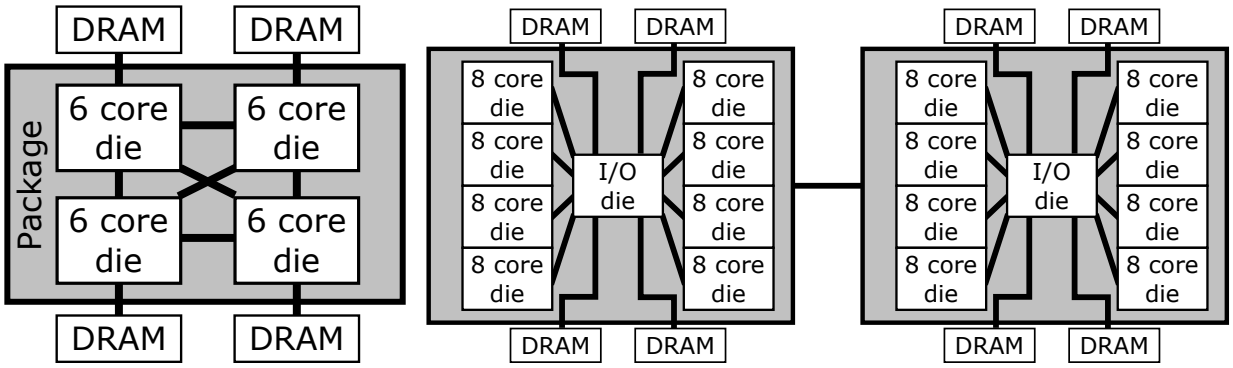
We also evaluated a recent Rome-based system since this design has a more uniform memory architecture. The Rome-based system has 64 cores with 8 cores on each of 8 dies in a multi-chip package, and it is a dual socket system for a total of 128 cores. The Rome system has more chips per package, but has a more uniform memory architecture since each die is equidistant from the I/O die with the memory controllers. In the Rome-based system we evaluated, there is only one NUMA node *per socket*. However, we used a dual socket system so our evaluations have two NUMA nodes. We also used an EPYC 7402P (Rome architecture, with one socket) system for validation of some results discussed in section 4.4.

The recently deployed supercomputers Frontier and El Capitan are based on AMD microarchitecture [102], though these are based on a recent microarchitectures (Frontier is Zen 3 based and El Capitan is Zen 4 based). The specific memory architecture of these devices support multiple sockets and have non-uniformity as the Rome-based system. Google's confidential cloud computing initiative also relies on AMD SEV for trusted execution support [103].

We use a desktop-class processor with 6 cores and a single NUMA node to perform Intel

Table 4.3: System Configurations. See Figure 4.1 for details on the two EPYC systems.

Feature	**AMD SEV 1**	**AMD SEV 2**	**AMD SEV 3**	**Intel SGX**
CPU	EPYC 7401P	EPYC 7702	EPYC 7402P	Core i7-8700
Sockets	1	2	1	1
Cores	24	128	24	6
NUMA	4 Nodes	2 Nodes	1 Node	1 Node
RAM	64GB	1TB	64GB	32GB



(a) AMD EPYC 7401P (Naples) (b) AMD EPYC 7702 (Rome)

Figure 4.1: Details of the non-uniform memory architecture for the two AMD systems evaluated.

SGX experiments, as there did not exist a server-class Intel processor with the support of SGX at the time of performing SGX experiments in this paper. Recently, Intel SGX is made available in one of the Intel Xeon parts (Xeon E3). However, the size of secure memory (doubled to be 256MB in total) is still significantly smaller than the working set of most of the workloads studied in this paper (only *ep* has a working set smaller than 256MB) and the conclusions drawn in this work (discussed in section 4.4) should still hold true.

4.3.4 Software Tools/Frameworks

To execute unmodified applications under SGX, we make use of SCONE [81] framework container. Programs are compiled statically and linked against a modified standard C library in SCONE. SCONE runtime also makes use of threads outside the enclave to perform asynchronous execution of system calls. We evaluated other SGX interfaces and picked SCONE as it provided the most complete support for unmodified applications. These other SGX pro-

gramming interfaces are discussed in section 4.4 (Finding 4.4). Rewriting HPC applications for SGX's programming model, by partitioning them into secure and un-secure components, is arduous but not impossible. However, in this work we focus on the use case of unmodified HPC applications. Also, the overhead of containerization like SCONE has been shown to be low. The original work [81] introducing SCONE showed that it has a 0.6–1.22× throughput compared to native execution for services like Apache, Redis, NGINX, and Memcached [81]. We also tested the performance of NAS parallel benchmarks in the "simulation mode" of SCONE. This mode uses all of the SCONE interfaces, but does not enable SGX. We found that the geometric mean of slowdown compared to native execution is 1.19×, which is insignificant compared to the slowdown of trusted execution (with SGX) in SCONE as shown in section 4.4 (Finding 4). Finally, we observed the performance of two memory intensive micro-benchmarks, partitioned into secure and un-secure parts directly using Intel SGX SDK, and found those numbers to be in line with our observations with SCONE as discussed in section 4.4 (Finding 4).

For SEV, we make use of the AMD provided scripts to set-up the SEV enabled host machine and the guest virtual machine (VM) managed by QEMU [7]. We also evaluated using Kata [104] which is a containerized interface to the hardware virtualization support in Linux. However, we found that Kata's support for SEV was too preliminary to get consistent results. Kata or other virtualized container interfaces may provide an even simpler programming interface to SEV in the future, but they will likely have the same performance characteristics as QEMU since they both use hardware support for virtualization. When running with QEMU, we assign all of the host cores to the guest and allocate enough memory on the guest to fit the entire resident memory of the application. The documentation and scripts required to set-up and run the experiments discussed in this work are available publicly.[2]

4.4 Understanding the Performance of TEEs

Next, we will present our findings on the performance impact of TEEs for scientific computing workloads and the reasons for the observed slowdowns. We make following main findings:

1. Finding 1: When the user configures the NUMA allocation policy correctly, SEV has

[2]https://github.com/lbnl-cybersecurity/tee-hpc

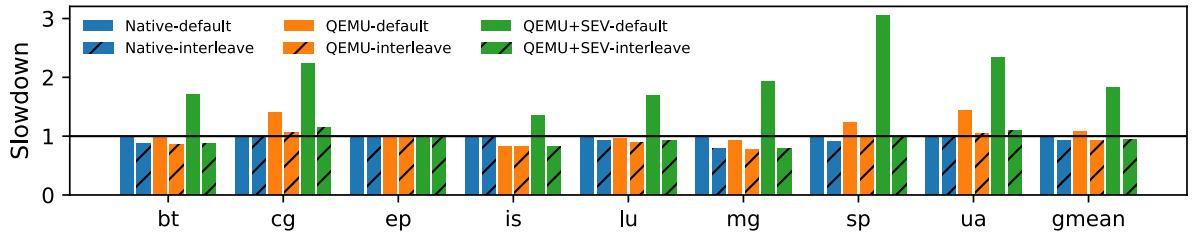


Figure 4.2: Performance impact of SEV for NPB C Class on AMD Naples (24 Threads). The SEV performance overhead is mainly because of default NUMA memory allocation, most of which goes away with interleaved NUMA allocation.

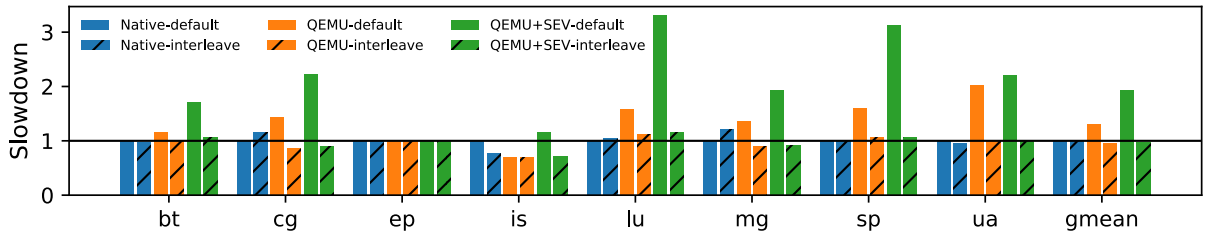


Figure 4.3: Performance impact of SEV for NPB D Class on AMD Naples (24 Threads).

small overhead for most workloads.

2. Finding 2: SEV relies on QEMU and hardware virtualization, which causes significant performance degradation for some irregular workloads, I/O intensive workloads, and workloads with high thread contention.

3. Finding 3: SEV initialization is slow and depends on the memory footprint of the application.

4. Finding 4: SGX has high performance overhead mostly due to its limited secure memory capacity and partially due to parallel scalability limitations and programming challenges.

The rest of this section provides details of these findings.

4.4.1 Finding 1: SEV can be used for secure scientific computing without significant performance degradation for most workloads if it is configured correctly.

SEV requires nested page tables [105] and is only available when running in a VM. Therefore, we compare three different cases: native (unsecure), QEMU (virtualized, but also no security gaurantees), and QEMU+SEV which provides security from the hypervisor and other users.[3]

Figures 4.2 and 4.3 show the performance of the NAS Parallel Benchmarks for the C and D class inputs relative to the "native" execution without any security guarantees. The solid bars on these figures show the performance of native execution, "QEMU" which is a KVM-based hypervisor running a virtual machine with the benchmark, and "QEMU+SEV" which has the SEV security extensions enabled (all relative to the performance of native execution). This shows that while the performance overheads of SEV (shown in green solid bars) are lower than SGX, using the default system configuration of SEV still results in significant performance degradation compared to the virtualized QEMU execution.

In this section, we will discuss how most of these slowdowns can be eliminated through careful NUMA data placement. We also present data from two different generations of AMD platforms to further investigate the overheads of SEV.

Finding 1.1: *Enabling SEV causes performance degradation beyond virtualization overheads.*

Although there is some overhead from virtualization for the NAS Parallel Benchmarks as shown in the orange bars of Figures 4.2 and 4.3, there is significantly more performance overhead when enabling SEV (green bars, up to 3× slowdown over the native execution).

Finding 1.2: *SEV overhead is because of NUMA placement.*

The reason QEMU+SEV suffers more performance overhead than QEMU is that when an SEV enabled virtual machine (VM) is launched, the memory pages allocated to the guest RAM are pinned by the hypervisor (QEMU) using *mlock* syscall. As a result, all data for the application is allocated on a single NUMA node and multi-threaded processes which expect performance improvements from running on large NUMA systems suffer from performance

[3]The initial implementation of SEV has many security vulnerabilities [70–72,106–108]. However, more recent implementations (e.g., Rome) fix many of the published vulnerabilities but still have similar performance characteristics to the systems we evaluate.

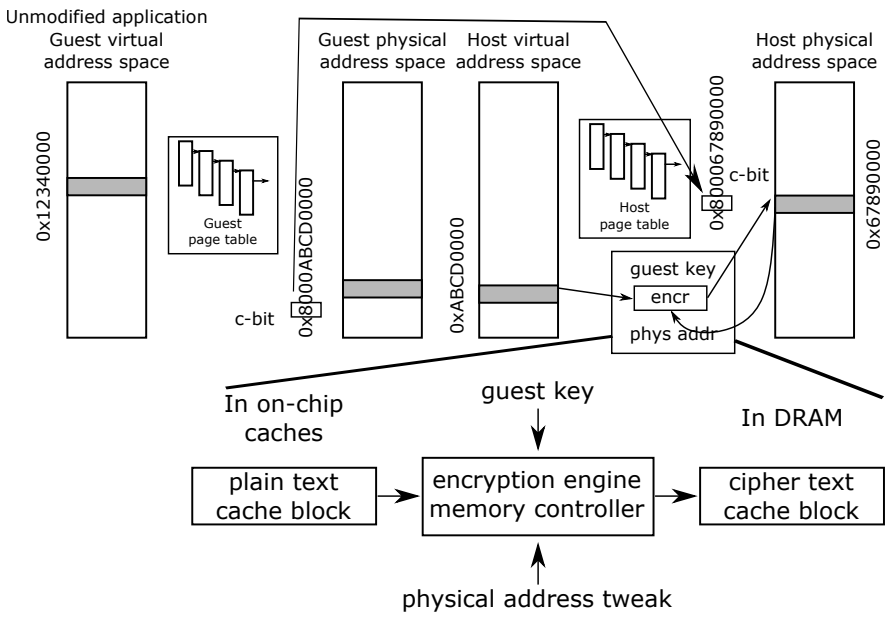
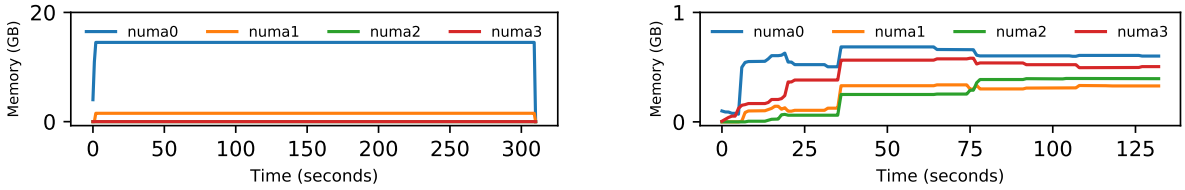


Figure 4.4: Details of SEV encryption implementation.



(a) SEV Default Allocation

(b) No SEV Default Allocation

Figure 4.5: Memory allocation over time using default policy.

degradation under SEV. QEMU without SEV does not have this restriction.

Why SEV requires locking pages to physical addresses?

Figure 4.4 shows details of how SEV is implemented. This figure shows both the interaction with the nested page table translation used for hardware virtualization acceleration and the memory encryption engine. First, this figure shows how the guest virtual address is translated through a nested page table since it must translate first into the guest physical address space then into the host physical address space. Importantly, the "c-bit" or encrypted bit is removed from the guest physical address by hardware and replaced after the host page table translates the address to the host physical address space. By removing and replacing the c-bit, the hypervisor is unaware of which pages are encrypted or not.

Second, SEV must guarantee that two identical plaintext pages present at different physical addresses in the memory will have different cipher texts to protect against cipher text block move attacks. To make this possible, SEV uses a physical-address based tweak algorithm [70, 71] as shown in Figure 4.4 with the physical address of the cache block influencing the cipher text via an xor-encrypt-xor tweak [109]. Since the host-physical address is used to determine the cipher-text of a page, the hypervisor cannot move a page between two physical addresses once it is allocated to the secure VM.

This limitation causes two performance issues when using SEV. First, all data pages for the guest are *pinned* in physical memory by the hypervisor [110]. In fact, because the default NUMA policy on Linux is "first-touch", all memory is allocated on a single NUMA node, which causes performance degradation for many of the scalable workloads evaluated in this work. Second, SEV-based guests can under-utilize the memory resources since they do not use on-demand paging.

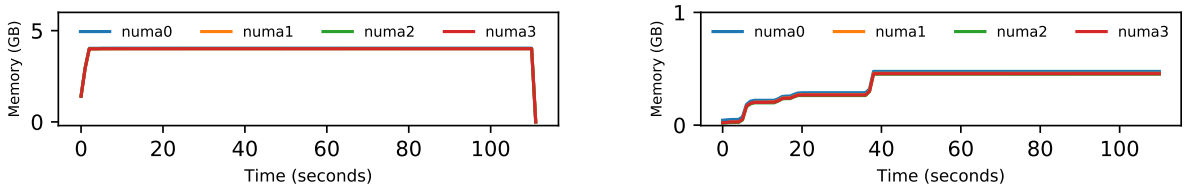
Figures 4.5a and 4.5b visualize the memory allocation process when using QEMU and QEMU+SEV. These figures show the memory allocation over time on different NUMA nodes on a system with four NUMA nodes when a VM with 16 GB memory is launched to run (for example) *sp* benchmark. Figure 4.5a shows that under SEV all data is allocated at the time of the VM launch *on a single NUMA node* as opposed to the non-SEV case (Figure 4.5b) which follows on-demand paging scheme and spreads the data across all four nodes.

For additional evidence, we conducted an initial study on a single-socket AMD Rome based system (AMD EPYC 7402P, 24 core system, similar to Figure 4.1b but with a single package and four core dies) using NPB D class workloads. This system has a uniform memory architecture, and that is why the slowdowns due to NUMA placement issues (observed previously) do not exist in this case as shown in Figure 4.10.

Thus, we conclude the SEV-specific overhead is due to the NUMA allocation policy.

Finding 1.3: *Explicit interleaving of data across NUMA nodes using* numactl *recovers most of the performance loss as shown in Figure 4.6.*

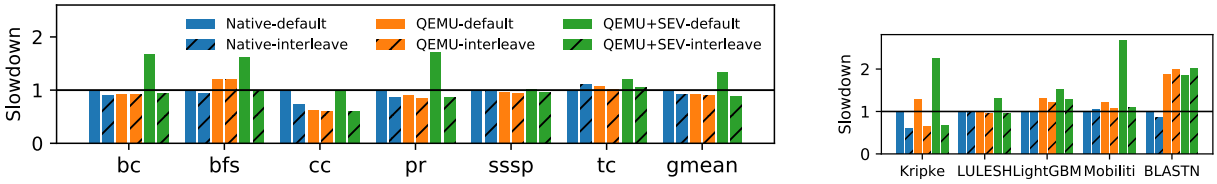
To mitigate the observed slowdown, we explicitly allocate memory pages across NUMA nodes rather than using the default NUMA memory allocation policy in the Linux kernel. We use `numactl` to allocate memory pages across NUMA nodes when the VM is launched



(a) SEV Interleaved Allocation

(b) No SEV Interleaved Allocation

Figure 4.6: Memory allocation over time using an interleave policy. Under SEV an equal amount of memory is allocated across all nodes.



(a) GAPBS (road network)

(b) Real world HPC workloads

Figure 4.7: Performance impact of SEV for GAPBS and other real world HPC workloads on AMD Naples (24 Threads). Interleaved NUMA allocation works for graph and other HPC workloads except BLASTN which shows high overhead mainly because of virtualized disk I/O operations.

under SEV. A visualization of the memory allocation using *interleaved* NUMA allocation policy is shown in Figure 4.6a. Under SEV, an equal amount of memory (4 GB on each node) is allocated across all nodes. We observe that the interleaved memory allocation across all NUMA nodes results in significant performance improvements for SEV. In fact, the performance differences between QEMU and QEMU+SEV shrink as shown in Figure 4.2 and 4.3 when enabling NUMA interleaving (hatched bars). This is in contrast to prior work which evaluated server-based applications and found that using a single NUMA node results in the best performance for virtualized workloads [111]. Importantly, we also observe that for native execution the interleaved allocation results in better performance compared to the default allocation for most of the cases (prominent examples are *Kripke, Mobiliti,* and *cc* from GAPBS).

In addition to the HPC kernels in the NAS Parallel Benchmarks, we also studied modern HPC workloads. Figure 4.7a shows the execution time for native, QEMU and QEMU+SEV cases for GAPBS workloads when executed using a road network graph. Similar to NPB,

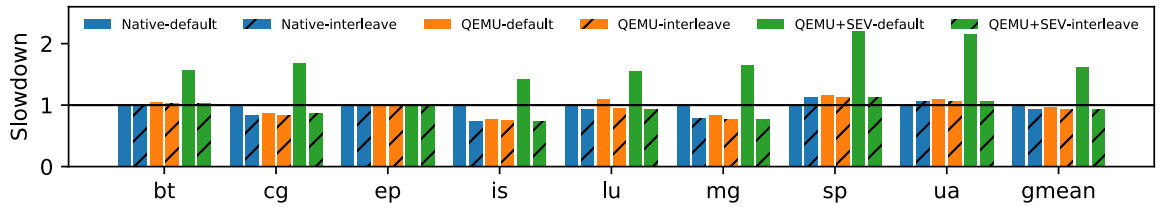
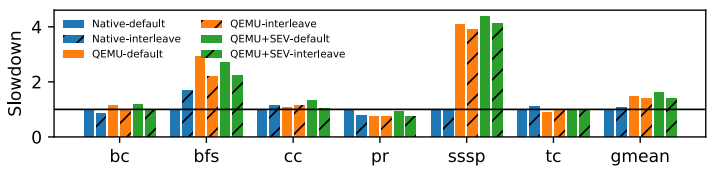
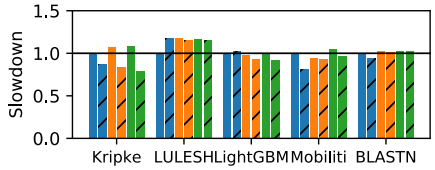


Figure 4.8: Performance impact of SEV for NPB D Class on AMD Rome (128 Threads)



(a) GAPBS (road network)



(b) Real world HPC workloads

Figure 4.9: Performance impact of SEV for GAPBS and other real world HPC workloads on AMD Rome (128 Threads). NUMA placement still matters on platforms with more uniform memory architecture. Two examples where main cause of overhead is virtualization are bfs and sssp.

NUMA interleaving reduces the difference between QEMU+SEV and SEV. Similar trends are found for other HPC workloads as shown in Figure 4.7b.

However, there are still some cases where QEMU and QEMU+SEV experience performance degradation compared to the native (unsecure) baseline. These differences can be attributed to *virtualization* overhead as discussed in Finding 2.

Finding 1.4: *NUMA placement still matters on new platforms with more uniform memory architecture (AMD EPYC 7702 (Rome architecture)) as shown in Figure 4.1.*

We also studied the performance of these benchmarks on another modern server class AMD machine EPYC 7702 (Rome architecture), which contains 2 NUMA nodes instead of four (see Figure 4.1). Figure 4.8, Figure 4.9a and Figure 4.9b show the relative performance of native, QEMU, and QEMU+SEV for the Rome system. Similar to the Naples system, there are significant overheads when using SEV unless the data is explicitly interleaved between NUMA nodes. Thus, even for systems that have more "uniformity" in their memory architecture, data placement is important for performance when using SEV.

Finding 1 summary: When enabling SEV, there are additional overheads beyond just

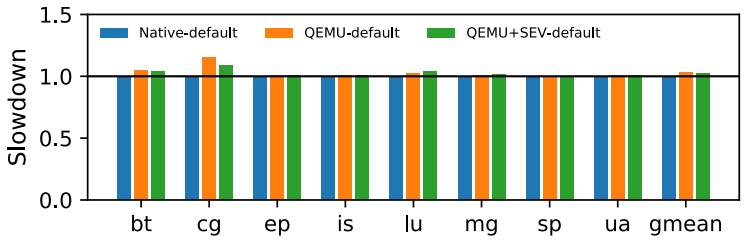


Figure 4.10: NPB D Class on AMD EPYC 7402P (24 Threads)

the virtualization platform overheads. These overheads are caused by the memory allocation restrictions of the SEV technology and persist even on the most recent architectures. However, we can overcome these SEV-specific overheads by explicitly interleaving data between NUMA nodes when the virtual machine is initialized.

4.4.2 Finding 2: The remaining SEV performance differences are due to virtualization overheads.

We find that in some cases there is performance degradation of the QEMU+SEV system compared to the baseline native execution even after applying our NUMA interleaving configuration change. These slowdowns come from the use of hardware virtualization and QEMU. For example, in Figure 4.9a, *sssp* with QEMU+SEV shows considerable slowdown compared to Native-default case irrespective of memory allocation policy (default or interleaved) on AMD Rome architecture. As visible in the Figure 4.9a, the performance of QEMU+SEV and QEMU match, indicating that the main cause of this slowdown is virtualization itself, not the SEV extension.

We observed that, when run with 128 threads (as in Figure 4.9a), *sssp* shows much higher number of kvm exits per second caused by the PAUSE instruction in comparison to the case when it is run with a smaller number of threads (e.g., 32). The PAUSE instruction is used to implement spinlocks and can cause KVM exits (i.e., a usermode to hypervisor switch) which has a higher latency than a normal context switch.

In fact, when executed with only 32 threads, the virtualization slowdown of *sssp* improves to 1.7× (in contrast to 4× in Figure 4.9a). Similarly, the QEMU overhead for *bfs* reduces to 1.6× with 32 execution threads in contrast to 2.6× with 128 execution threads (Ding et al. made similar findings [60]). Thus, when using QEMU or QEMU+SEV it is important to use

the appropriate number of execution threads for your workload and workloads with highly contended locks may result in significant performance degradation.

In Figure 4.7b, BLASTN also shows slowdown by virtualization on AMD Naples architecture. The nucleotide database which is used by BLASTN is approximately 245GB in size (much larger than the memory size of 64 GB on our AMD Naples system), which leads to many disk I/O operations and thus slowdown under virtualization. On the other hand, when the same workload is executed on AMD Rome system (which has 1 TB of memory), there is not any noticeable virtualization overhead as shown in Figure 4.9b since the workload can fit in the available system memory.

There is significant prior work quantifying the impact of virtualization on the performance of HPC workloads [10, 60, 112–116]. These prior works mostly focus on overheads from TLB misses and nested page table walks. Similarly, our results show the virtualization overheads grow as the working set of the applications grow and are worse for workloads with irregular access patterns (e.g., graph workloads). Prior work has shown you can reduce this overhead by using huge pages or through changes to the hardware (e.g., Virtualized Direct Segments [116]). Additionally, the work of Ding et al. [60] presents possible strategies to mitigate the virtualization slowdown caused by multithreaded application scaling.

4.4.3 Finding 3: SEV initialization is slow and depends on the memory footprint of the VM (1.1×–1.47× depending on the size of the VM memory (from 8 GB to 48 GB)).

We find that the time taken to initialize the workload is significant when using QEMU and increases when using QEMU+SEV. When using QEMU or QEMU+SEV, before running the workload the virtual machine guest operating system must complete the boot process. For QEMU this bootup time takes about one minute for our workloads.

However, when enabling QEMU+SEV, this boot time increases due to the hypervisor having to initialize the memory before handing it over to the guest OS. As shown in Figure 4.11, SEV can cause a slowdown (relative to QEMU-8 GB) of 1.1×–1.47× depending on the size of the VM memory (from 8 GB to 48 GB). In addition to the memory initialization, QEMU+SEV also needs extra time for key management when launching a guest with QEMU+SEV. However, we believe that the main source of SEV slowdown is the fact that

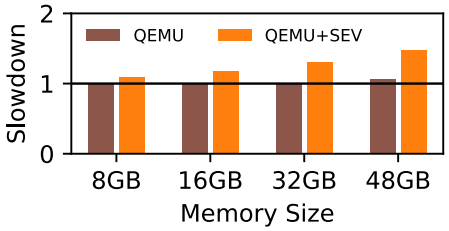
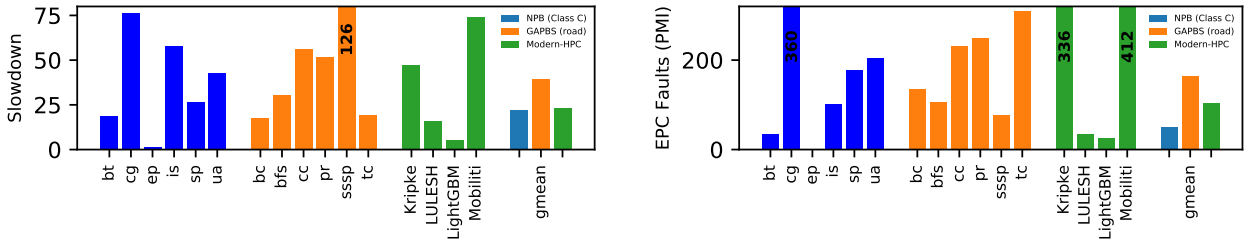


Figure 4.11: Performance of VM boot (relative to QEMU-8GB)



(a) Relative performance running under SGX compared to native execution.

(b) EPC Fault Rate (Per Million Instructions) when running under SGX

Figure 4.12: Performance Impact of SGX and its Relation to EPC (Enclave Page Cache) Faults. Slowdown and EPC faults show a strong correlation indicating that the workloads with higher secure to non-secure memory movement rates will exhibit higher slowdown.

the entire VM memory has to be allocated at once in case of QEMU+SEV in contrast to on-demand allocation in case of QEMU (as discussed in section 4.4.1), as evident by the increase in slowdown as the VM memory size is increased. This can specially become a bottleneck for the use cases where the user intend to launch their jobs in a new VM each time (e.g., when using Kata containers [104]).

4.4.4 Finding 4: SGX is inappropriate for unmodified scientific computing applications.

We find a number of reasons that SGX is not an appropriate technology for securing HPC workloads. A primary design goal of SGX is to enable a small trusted compute base, and SGX was not designed to support large scale workloads. We find that running HPC workloads under SGX causes a (1×–126×) slowdown (mostly due to its limited secure memory capacity as shown in Figure 4.12a), workloads exhibit poor thread scalability under SGX (as shown in Figure 4.13), and it is difficult to adapt HPC code to work under the SGX program-

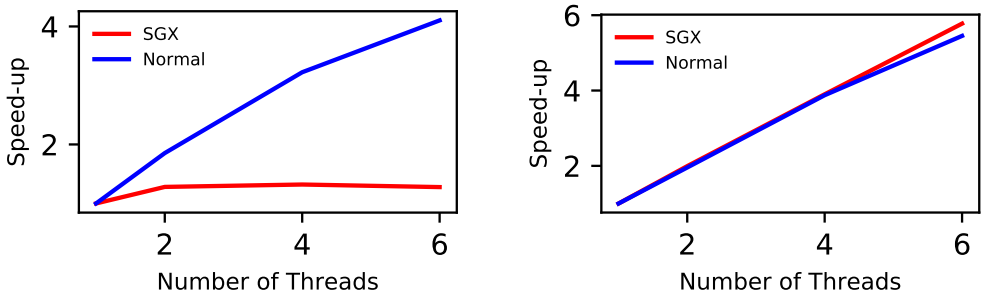
ming model. We observed that even with multiple third party solutions to run unmodified applications under SGX (e.g. SCONE [81], Graphene [80], Haven [117] and Asylo [118]) it is fundamentally difficult to use SGX to run HPC applications because these tools mostly use non-traditional C libraries, and have limited syscall support.

Finding 4.1: *Workloads with working sets larger than about 100 MB suffer large performance degradation under SGX.* Figure 4.12a shows the slowdown of HPC workloads under SGX compared to an un-secure baseline. For this experiment, we ran NPB with the "class C" inputs (blue in Figure 4.12a). We were limited to using the class C inputs, as most class D inputs were too large to run on the desktop systems that support SGX. However, we believe that running larger inputs under SGX would show at least as much performance overhead as the smaller inputs. We also show the relative performance of graph workloads and other modern HPC workloads in Figure 4.12a). We were not able to run BLASTN workload with SGX due its dependencies (discussed more in Finding 4.4).

Most of the performance degradation shown in Figure 4.12a can be explained by the overhead of moving data from un-secure memory into secure memory. SGX has a limited amount of secure memory, about 100 MB. Thus, any workload with a working set larger than 100 MB must use the secure memory as an *enclave page cache* (EPC). The EPC is managed by the SGX driver in software and has similar behavior to OS swapping and moving pages between normal and secure memory is a high latency event.

Figure 4.12b shows the number of EPC faults per million instructions for each of the workloads. This figure shows that most of the slowdown in Figure 4.12a can be explained by the EPC fault rate. The workloads with the highest rate of moving data between secure memory and normal memory (e.g., *cg* from NPB, Mobiliti, and Kripke) show very high slowdown. On the other hand, *ep* from NPB shows little performance overhead with SGX because it has a very small working set size (about 28 MB) which fits in the EPC and does not require data movement between secure and normal memory spaces.

Finding 4.2: *In some cases, SGX slowdown can be caused by system calls.* Applications under SGX exit the enclave to process a system call. This can become a problem for workloads with a large number of system calls. In the studied HPC workloads, the only case where we found system calls to be the dominant source of performance overhead is *sssp* benchmark



(a) cg (b) ep

Figure 4.13: Impact of Multiple Execution Threads. Workloads with high resident memory like *cg* do not scale well with the number of execution threads in contrast to low resident memory workloads like *ep*. Handling of EPC faults by the SGX kernel driver becomes the serializing factor in case of high resident memory workloads.

from GAPBS. As shown in Figure 4.12b, the slowdown for *sssp* does not correlate with EPC fault rate. *sssp* shows significantly higher number of enclave exits (and system calls) and is the main contributor to its performance overhead compared to the un-secure execution. Most of these system calls were *write* and *futex* calls, which are needed due to the benchmark printing progress to the terminal. The *futex* calls are used for synchronization (of multiple threads) before printing the status messages using *write* calls. The effect on slowdown because of *futex* system calls can be understood by the difference in the observed slowdown for six thread execution (126×) and single thread execution (20×), which does not need any synchronization.

Finding 4.3: *Workloads exhibit poor multithreaded scaling under SGX.* Another factor that aggravates the slowdowns under SGX is explained with the help of Figure 4.13, which shows the workload performance when increasing the number of threads. Figure 4.13a shows that *cg* only achieves a speedup of 1.4× with six threads when using SGX compared to about 4× speedup normally. We hypothesize that the handling of EPC faults by the SGX kernel driver becomes the serializing factor because all logical processors executing an enclave's code are required to exit the enclave whenever an EPC page is deallocated [4]. Similar behavior is exhibited by most of the other workloads with high resident memory size. On the other hand, workloads with working set sizes that fit in the EPC (e.g., *ep*) scale under SGX as they would under normal execution as shown in Figure 4.13b.

Finding 4.4: *SGX's programming model is a poor fit for HPC applications.* Intel distributes an official SDK [119] for SGX which requires users to re-write their application and divide it into two pieces, secure code and non-secure code. Due to the complex nature of HPC codes, dependencies on external libraries, and frequent use of legacy codes (including a non-trivial number of them written in Fortran), we investigated several alternative interfaces to SGX which reduce the burden on the programmer.

There are multiple third party solutions to run unmodified applications under SGX including SCONE [81], Graphene [80], Haven [117] and Asylo [118]. We did initial experiments with both Graphene and SCONE as they were the best supported third party solutions at the time we ran our experiments. SCONE provides containerized environment and is easier to set-up and has a better support of running diverse workloads without any modifications, so we used SCONE for our experiments. Although we only evaluated SCONE, all SGX programming interfaces have similar limitations due to SGX's design which limits the TCB. Graphene [80] is not as convenient to use as SCONE and Google Asylo's [118] recently added support to run unmodified applications still lags behind SCONE in terms of the number of supported use-cases. Open Enclave SDK [120] is another SDK to build enclave applications and does not support unmodified applications.

We found that even with SCONE, which promises to run unmodified applications with SGX, it is fundamentally difficult to use SGX to run HPC applications. In order to keep the library OS simple, SCONE makes use of the musl libc library, instead of more traditional C library glibc, along-with some containerized services using the Linux Kernel Library (LKL). The use of musl libc instead of glibc means many applications are not portable to SCONE (e.g., BLASTN failed to compile inside SCONE and many common frameworks such as TensorFlow require glibc instead of musl libc). Moreover, SCONE does not support some system calls like fork, exec and clone mainly due to its user space threading model and the architectural limitations of SGX [81] which further limits its applicability to scalable applications.

Finding 4 summary: The current implementation of Intel's SGX limits the secure memory size which severely affects the performance of any workload that has a working set that does not fit in this cache. Additionally, there is currently no stable support to run

unmodified workloads under SGX. The limited EPC capacity and application partitioning are a fundamental design constraints of SGX. Thus, we conclude that SGX is unsuitable for secure execution of HPC applications.

4.5 Beyond Single Node

Scientific computing workloads often scale across multiple machines (nodes). In this work, we only focus on a single node to isolate the performance impact of hardware TEEs. To understand the impact of communicating between TEEs on multiple nodes, we conducted a preliminary investigation of a multi-node system with support of SEV on CloudLab [121].

Current HPC systems mostly rely on high performance transport protocols like RDMA for communication among multiple nodes. However, RDMA does not provide any secure communication support although there is a recent research proposal for secure RDMA [122]. Therefore, we instead evaluated TCP for communication among machines using OSU MPI microbenchmarks [123]. For point-to-point bandwidth benchmarks, we observed a 2× reduction in bandwidth when comparing QEMU to QEMU+SEV, but the latency remains the same (approximately 1000 μs) for both QEMU and QEMU+SEV (ranging from 1 byte to 2MB packet sizes).

However, in this simple benchmark, the communication between nodes is insecure as there is no support for encrypted communication between multiple nodes in SEV automatically. A naive solution to make this communication secure is to use a VPN. We experimented with OpenVPN and found the slowdown of VPN based secure communication to be large. For example, for above microbenchmarks, the bandwidth number drops over 10× and latency increases by almost 20×.

Therefore, we conclude that there is a need to develop more performant architectures to enable TEEs across multiple nodes in a distributed memory. There are some existing software based solutions to enable encrypted communication across nodes, but they might not be sufficient for scientific computing scale workloads. For example, SCONE [81] provides a network shield which transparently protects the communication among multiple nodes (each with its SGX hardware). Asylo [118] (open-source framework for developing enclave applications) allows enclave based applications to be scaled across multiple machines using

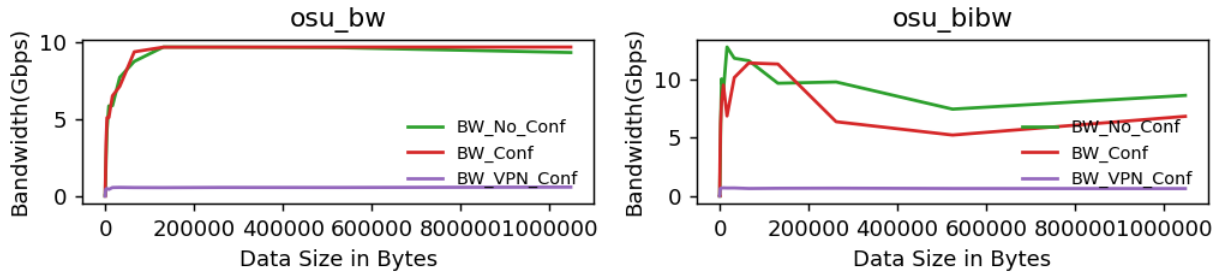


Figure 4.14: Bandwidth Test from OSU Microbenchmarks

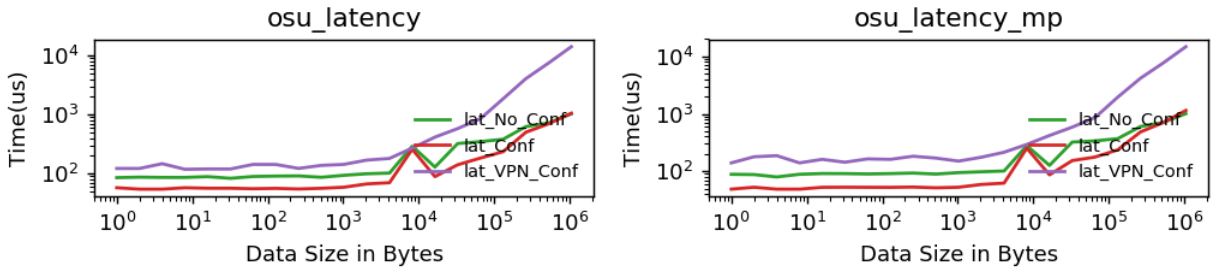


Figure 4.15: Latency Test from OSU Microbenchmarks

gRPCs (google remote procedure calls), while being agnostic to TEE implementation.

4.5.1 Trusted HPC in the Cloud

In this subsection, we will take a look into how to the performance implications of simple experimental set-up in Google Cloud for running HPC-like workloads securely. Recently, Google's confidential cloud computing initiative announced the availability of confidential virtual machines based on AMD's SEV (secure encrypted virtualization) for trusted execution support. When SEV is enabled, all data stored in the main memory for a particular virtual machine (VM) will be encrypted. This ensures the data cannot be read by other VMs, the hypervisor, or even individuals with physical access to the main memory hardware.

The HPC workloads are mostly distributed and scale across multiple machines/nodes (e.g. using OpenMPI). High bandwidth network interconnects are used for communication among these nodes at HPC centers. Unfortunately, SEV does not have support for secure multi-node computation yet. If you are executing workloads which scale across multiple VMs on different nodes, the data can't be sent encrypted across them with SEV since each VM has it's own key (generated at random by the hardware). Moreover, SEV makes use of the data's

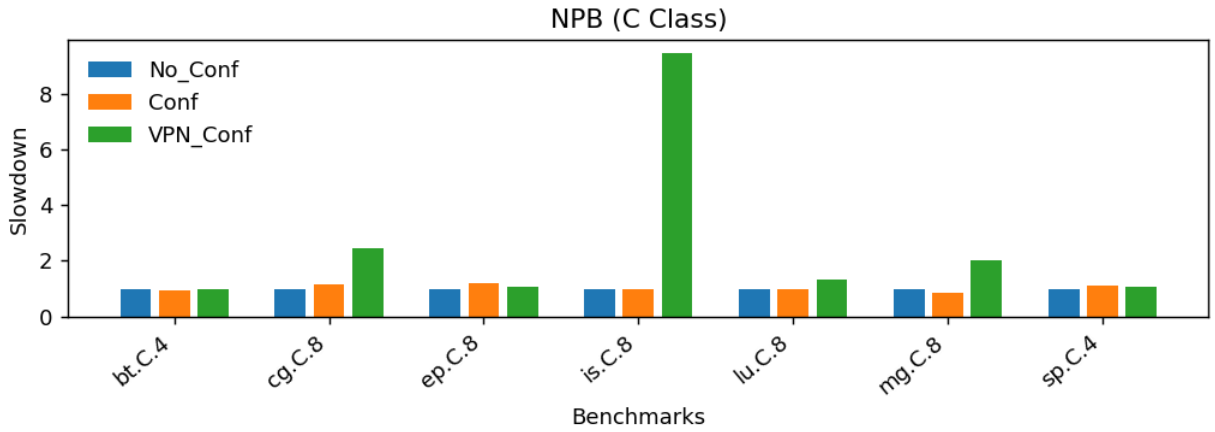


Figure 4.16: Slowdown for for NAS Parallel Benchmarks (C Class), 8 processes in total except bt and sp.

physical address in its encryption function in addition to the key, thus making the transfer of encrypted data even harder. Today, in Google Cloud infrastructure, if the boundary of one of Google Cloud's data-centers is not passed, there is some kind of authentication provided for the communication between different nodes, however the communication still happens in the clear. Therefore, in this blog post, we will make use of a VPN connection to enable secure communication. Although a VPN based secure communication is expected to be slow, currently it seems to be the only way of securing communication out of the box.

We used MPI workloads in our tests, which are very common in HPC domain. There are multiple open-source MPI implementations (OpenMPI, MPICH, MVAPIC) available and potentially can be used inside VMs. We use OpenMPI at this point and build and use it wih UCX which is a framework (collection of libraries and interfaces) to allow building various HPC protocols like RMA, fragmentation, MPI tag matching etc. and supports different transport protocols for communication like RDMA, TCP, shared memory etc. Steps taken to install OpenMPI, UCX and some MPI workloads are provided in the Appendix section.

Figure 4.14 and 4.15 show the bandwidth and latency numbers for selected micro-benchmarks from OSU MPI micro-benchmarks suite. Three different configurations shown in the figures are described below:

- **No_Conf:** No confidentiality support (SEV is disabled)
- **Conf:** Confidentiality support (SEV) is enabled, but communication channel is un-

encrypted

- **VPN_Conf:** Confidentiality support (SEV) is enabled, and communication channel is secure using a VPN connection

Figure 4.16 shows the slowdown (in comparison to un-secure execution) for NAS Parallel Benchmarks (C Class), which are often used to benchmark HPC systems, under three different configurations defined above. As can be seen in the figure, the slowdown with Conf configuration suggest no significant performance degradtion. The slowdown with VPN_Conf for most of the benchmarks is not as bad as in the above microbenchmarks. However, it should be noted that we were limited by the maximum number of processes which is one of the reasons the tests do not involve higher level of classes of NAS Parallel Benchmarks (e.g. D Class) which are more representative of the HPC workloads that scientists will be interested in.

To summarize the above results, while it is possible today to run HPC workloads securely in the cloud (like Google cloud), there are limitations:

- Limited number of vCPUs that can be deployed on secure machines. For example, only 8 total vCPUs (for now) in Google cloud, which severely limits the kind of HPC workloads which can be executed on these resources.
- While data is stored encrypted, communication is in the clear necessitating application-specific solutions or VPN.
- Using a VPN for secure communication is undesirable due to the performance overheads.

4.6 Observations on Security of SGX and SEV

Although the focus of this paper is the performance analysis of TEEs for secure HPC, we provide a brief security analysis of TEEs discussed in this paper. SGX provides integrity guarantees while SEV lacks such support. However, the weaker guarantees of SEV are considered to be good enough by Google's confidential cloud computing initiative, and these guarantees are becoming stronger. AMD has introduced SEV-ES [94] that adds encryption

of guest register state to provide additional protection against VM state related attacks, and SEV-SNP [94] provides integrity checks. These are encouraging developments from the security perspective, as they address some of the vulnerabilities and limitations of SEV. It should be noted that SEV-SNP [94] does not provide integrity guarantees using Merkle tree like data structures (as SGX does). Therefore, it is more scalable and can support larger secure memory sizes. Additionally, it seems Intel is also moving in the direction of full memory encryption and virtual machine-based trusted execution environments like AMD's SEV with total memory encryption (TME) and multi-key total memory encryption (MKTME) technologies [124, 125].

Finally, in this paper we have focused on just one aspect of the entire secure application workflow which may include other steps as well (like a secure connection to the computing resources) in addition to running it inside an SEV or SGX enclave. However, we believe that the execution of the workload itself in a secure enclave is the most important factor for performance analysis.

4.7 Scientific Computing Focused Trusted Execution Environment

This section discusses our observations which form the basis of an HPC-focused TEE proposed in this document. Multiple features distinguish HPC from general purpose computing environments and their significance for secure architectures. On one hand these features impose some restrictions on secure computing architectures, and on the other hand some of these features can be leveraged to simplify the TEE design. These observations have influenced the proposal of main requirements that an HPC-focused TEE should meet. These requirements are presented below:

- (R1) *Requirement 1: HPC-focused TEE should have minimum performance impact on HPC style workloads.* Scientific computing applications are heavily multi-threaded and have large working sets. This implies that the HPC centric TEE should be capable of supporting multiple execution threads and should have minimal performance overhead irrespective of the amount/size of the data that needs to be protected while in memory (or scale well with data size). As shown in the previous section, this requirement is not

fulfilled by the current TEEs. For example, SGX incurs high performance penalties for any workload with the working set larger than the enclave page cache size and SEV incurs performance overheads on irregular workloads because of its dependence on a virtual machine.

- (R2) *Requirement 2: HPC-focused TEE should not require application modifications or linking against special libraries.* HPC applications often rely on third party libraries, and it can be hard for such applications to be modified or re-written to port them to a different secure execution programming model. The porting effort can involve linking the applications against specific C libraries or sometimes big modifications in the applications. SGX requires application to be partitioned and SEV requires application to be run inside a virtual machine, both of these requirements do not fit well with the HPC compute model.

- (R3) *Requirement 3: HPC-focused TEE should try to not include OS in the TCB.* HPC applications rely on limited types of I/O. For example, the main type of I/O is network I/O which is mostly used to communicate with other nodes and mostly the disk accesses get reduced to network I/O as file systems are mostly maintained on remote nodes in an HPC center. Additionally, for performance reasons, the OS is mostly bypassed and I/O is handled in user-space libraries or run-times. For example, HPC centers rely on one-sided communication protocols like RDMA [126] on high speed network interconnects like InfiniBand (which supports 10s of GB/s of bandwidth). RDMA (which bypass OS mostly) provides performance benefits, but raises new security concerns because of its one-sided communication nature. At the same time the bypassing of OS provides an opportunity to exclude OS from the trusted computing base (or have less trust in the OS). SGX does fulfill this requirement to some extent, but has been shown to be successfully attacked via an untrusted OS. AMD SEV includes the guest OS in its TCB.

- (R4) *Requirement 4: HPC-focused TEEs should be capable of expanding across compute nodes.* HPC applications mostly scale across multiple compute nodes and rely on message passing run-times like MPI for communication across these nodes. This implies

that an HPC-focused TEE should expand its threat model to multiple nodes, and should be capable of building enclaves which will scale across nodes. Moreover, the support for some of the other TEE features like secure boot, authenticated launch of enclave, and attestation should be expanded across multiple nodes. None of the commercial TEEs support this.

- (R5) *Requirement 5: HPC-focused TEEs should enable enclaves which can scale to processing elements other than the general purpose CPUs.* HPC systems have also started to integrate accelerators (like GPUs and FPGAs) to offload certain applications or parts of applications to those processing elements. This necessitates the inclusion of these processing elements (and I/O in general) into trusted computing base as well. There are some academic TEEs for some specific processing elements, but they would not work for any type of accelerator.

Table V provides a taxonomy of multiple features that existing TEEs provide and the missing features that ideally an HPC-centric TEE would have. Some of the shown features are not HPC specific, but are presented for the purpose of completeness. The row showing the ideal HPC-centric TEE features also point out the particular requirements (mentioned above) that those features fulfill. Chapter 5 provides details of one of our proposed techniques to achieve these missing features.

Table V. Taxonomy of different TEE features. **HPC centric** (row in green shade) refers to what is best for HPC. Brown shaded columns are of special importance from HPC perspective.

TEE	Software Attacks[1]				Hardware Attacks[2]		Level[3]	TCB	I/O Handling	No Changes Needed		HPC Slowdown[4]
	From apps.	From OS/ hypervisor	From IO[8]	On IO[8]	From IO[8]	Physical Attacks				HW	SW	
SGX [4]	✓	✓	✓	✗	✗	✓	App.	App., CPU	outside enclave, in clear	✗	✗	large[5]
SEV [24]	✓	✓	✓	✗	✗	✓	VM	guest OS, App., CPU	using bounce buffers, in clear	✓	✓	minimal[6]
TrustZone [23]	✓	✓	✓	✗	✗	✗	system partition	trusted OS, CPU	I/O part of TCB	✗	✗	N/A
AWS-Nitro [47]	✓	✗		✗	✗	✗	VM	VM, hypervisor	VM socket	✓	✗	minimal
KeyStone [127]	✓	✓	✓	✗	✗	✓	App.	RT, SM, CPU	in clear	✓	✗	unclear[7]
HPC centric	✓	✓(R3)	✓(R5)	✓(R4)	✓(R5)	✓	App. (R1)	App.,CPU (R3)	secure	✓	✓(R2)	minimal (R1)
DESC[9]	✓	✓	✓	✓	✓	✓	App.	App.,CPU	secure	✓	✓	small

[1]Software attacks have software and [2]hardware attacks have hardware as the attack surface. [3]Level is the granularity/level at which protection is provided.

[4]No TEE supports multi-node trusted execution and use of software to create secure tunnel between TEEs on multiple nodes cause very high slowdown

[5]specially for multi-threaded and large memory Apps. [6]with careful memory allocation. [7]no support for multi-threaded enclave and has large slowdown for IO

Other Notes: These TEEs generally do not consider side channels. Threat of side channels depend on the data sensitivity and leakage rate.

Only SGX provides strong protection against integrity attacks. SEV-SNP provides some guarantees against integrity attacks. [8]I/O also includes GPUs, accelerators & FPGAs

[9]DESC is discussed in Chapter 5. [10]DM refers to disaggregated memory manager

Chapter 5

DESC – Data Enclaves for Scientific Computing

5.1 Introduction

Our goal is to enable secure scientific computing by protecting the data of scientific applications from other software (including the operating system) running on a computing system. We propose a new protection mechanism that allows the secure execution of unmodified applications while minimizing the trusted computing base (TCB) size. Our protection mechanism, *DESC (Data*[1] *Enclaves for Scientific Computing)*, is data-centric and protects an application's sensitive data at all times. Our approach contrasts with other protection approaches that try to create a new *execution* environment for the sensitive applications. *DESC* separates the *resource management* from *protection* by delegating these two functions to two different entities. The operating system (OS) manages the resources, and a higher privileged software (*Enclave Manager*) is responsible for protecting the sensitive application. Relying on the OS for resource management allows secure applications to utilize traditional kernel optimizations and management techniques.

DESC guarantees the protection of a sensitive application's memory at all times, even when relying on an untrusted OS for resource management. To implement *DESC*, we addressed three main challenges. First, *DESC* needs to ensure security during the execution mode switch between the OS and the sensitive application. Second, whenever the sensitive

[1]Data here refers to both data and instructions of a program in memory.

application explicitly shares data with the OS (such as during system call execution), *DESC* should ensure that only the shared data is exposed to the OS. Finally, *DESC* needs to account for OS-based resource management. In this thesis, we describe the mechanisms and techniques we have deployed to enable *DESC* in the context of these challenges.

This thesis focuses on the application of *DESC* to high performance computing (HPC) systems used for scientific research. We specifically focus on HPC in which the data and computation have elevated requirements for confidentiality or integrity — for example when the sensitive data provided by a third party is being computed upon, or when data is being used for a computation that automatically controls a closed-loop experimental workflow (such as a *self-driving lab* [128]), respectively.

HPC systems prioritize performance. Users expect their applications to run "bare metal" on the host OS, and not in a virtual machine that can result in significant performance overheads [59, 60, 129]. In contrast, cloud systems are virtualized and users expect applications to carry virtualization overheads. In addition, HPC centers allow a single tenant to occupy an entire node, in contrast to the multi-tenancy model that is common in cloud environments. Previous works [117, 130, 131] on enclave design primarily targeting cloud environments may not be entirely applicable to the HPC use-case. In contrast, the data enclave design presented in this thesis is intended for use in HPC and other similar environments where users expect bare-metal performance but also need to run computations on data with elevated security requirements.

Our design decisions for *DESC* are based on the specific requirements and characteristics of HPC applications. First, HPC applications often rely on third-party libraries and it can be challenging to modify or rewrite them to fit into a different secure execution programming model. Therefore, a data enclave approach that allows *unmodified* applications to run securely is well-suited for HPC.

HPC applications frequently bypass the operating system (OS) for I/O operations to enhance performance and minimize system noise. For example, HPC centers rely on one-sided communication protocols like RDMA [126] for performance benefits. DirectIO and userspace I/O system are other examples of this behavior. Figure 5.1 shows that for a variety of benchmarks that can be used as a proxy of modern high-performance computing

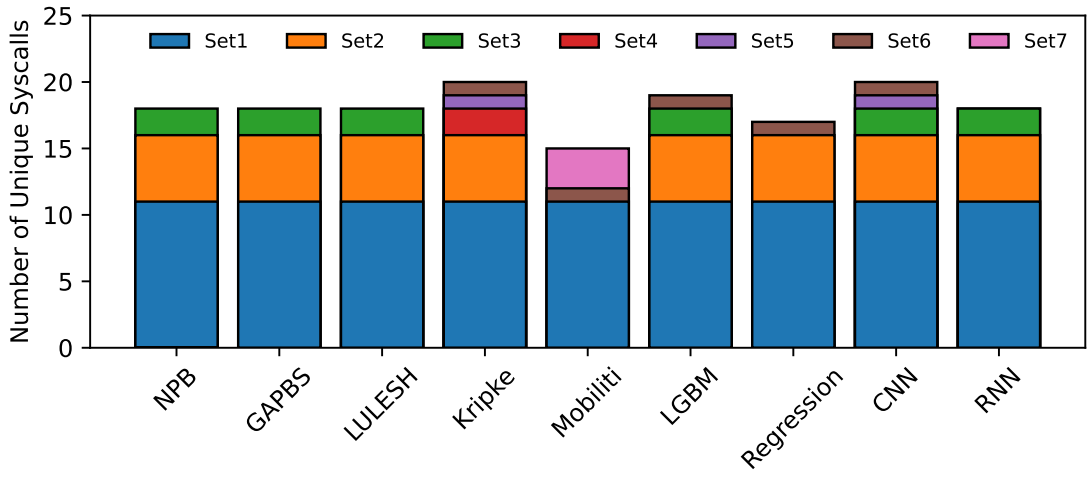


Figure 5.1: Unique system calls used by all the evaluated benchmarks. Each set refers to a collection of system calls that are common across the benchmarks. The total number of unique system calls used by the evaluated workloads is **6%** of the total available Linux system calls for RISC-V.

workloads, the number of unique system calls executed by these benchmarks is small, and these workloads mostly use the same system calls. Because of this limited interaction between the OS and (sensitive) HPC applications, the cost of supervising this interaction via the *Enclave Manager* (a trusted and higher privileged software) becomes tractable. Section 5.5 elaborates on the specific details of this supervision.

The main contributions of this work are:

- We implement a prototype data enclave on the RISC-V ISA that separates the management of the system from the protection of sensitive data.
- We show how the data enclave allows an untrusted OS to maintain page tables (with the extended PMP), supervise system calls (with the syscall interceptor), and manage processes and interrupts (through secure mode switching) without compromising the enclave applications data confidentiality or integrity.
- We show *DESC* has low overhead (less than 5% geometric mean) even with optional memory encryption (less than 20% geometric mean).
- We show *DESC* can correctly execute multithreaded applications in a secure environment showing low overhead compared to the untrusted execution.

5.2 Related Work on Confidential Computing

Confidential computing has similar security goals as *DESC*. It enables hardware-based protection of data in **use** in contrast to the data at **rest** (storage) or in **transit** (I/O)) [12], [13]. *Trusted execution environments (TEEs)* are the primary enablers of confidential computing. TEEs provide assurance of data integrity, confidentiality, and code integrity, using hardware-based techniques for increased security guarantees [13]. While TEEs can help create a zone of trust for sensitive data in HPC centers, the existing TEE technologies do not fully meet the goals and constraints of HPC.

Current TEEs have different approaches to protect against a malicious or buggy OS controlled by a system administrator with root privileges. Some TEEs use a supervisor runtime (e.g., Keystone [1], and Sanctum [25]) inside the enclave, while others rely on special containers with a library OS or special libc wrappers (e.g., Graphene-SGX [80], SGX-LKL [82], SCONE [132], Occlum [133], Chiron [134]), resulting in a large TCB or significant modifications to applications. We call these TEEs **runtime-based enclaves**. Since these TEEs try to emulate existing system components (like POSIX or devices) inside the contained systems, they might have to eventually deal with the same problems they started with vis-a-vis a large trusted compute base [76]. In some runtime-based TEEs, the runtime itself can be a complete operating system, such as the trusted OS in ARM Trustzone [23] and ARM Realms [39].This design choice results in a large TCB.

Some TEEs (e.g., AMD's SEV [3], SNP [135], AWS Nitro Enclaves [47], and H-SVM [136]) also require the use of virtual machines and make the guest OS part of the TCB, further increasing the TCB size. We call these TEEs **VM-based enclaves**. The first two rows of Table 5.1 provide the implications of the previously mentioned TEE technologies for a computing system. Users must either accept a large TCB, which increases the attack surface and compromises the system's security, or make significant modifications to their applications, which can be time-consuming and resource-intensive.

Multithreaded Execution – An Example of Limitations of Today's Confidential Computing Architectures

The existing TEE architectures incur usability challenges for the enclave applications because of their requirement of using special runtimes or virtual machines. Multithreaded

Table 5.1: Comparison of enclave types: *DESC* requires no application changes, has reduced TCB, and has smaller performance impact. Evaluation details of *DESC* are presented in Section 5.7.

Enclave types	**Application changes**	**Resource manager**	**Slowdown for HPC applications**	**TCB**
Runtime-based	High ✗	LibOS/runtime ✗	High ✗ [59, 137]	Large ✗ [138]
VM-based	Low ✓	Guest OS ✓	High ✗ [59]	Large ✗ [3]
DESC (ours)	Low ✓	Host OS ✓	Low ✓	Small ✓

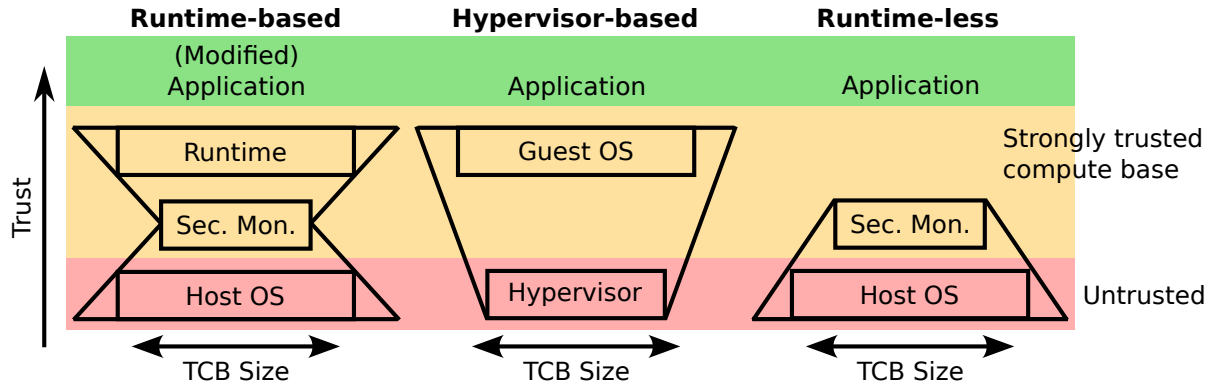


Figure 5.2: A comparison of TCB size and location of trust among different enclave styles (Runtime-based, Hypervisor-based, DESC). DESC achieves the lowest size of the strongly trusted compute base.

execution is an example of these limitations.

Today, some TEEs have implemented limited thread handling inside the enclave, which might reduce the system's efficiency overall. For example, enclaves (like Intel's SGX [4] and its variants) might enforce a static number of threads because they might only allow statically-defined entry points for executing threads. Enclaves like Keystone [1] do not support multithreaded execution at all at the time of writing this thesis [139]. VM-based enclaves include a guest OS in the TCB and allow multithreaded applications to run transparently. Not only do the VM-based enclaves have a very large TCB, but multithreaded execution in virtual machines can also have significant performance implications. For example, when

threads yield during synchronization operations, they can cause costly KVM exits [59,60]. In summary, today's enclaves generally do not have good support for multithreaded execution unless they are willing to have a large TCB.

Our Approach Towards Confidential Computing

In light of the observations illustrated earlier, we take a different approach from existing TEEs. We do not rely on any runtime or library OS for workloads and do not require virtual machines. Instead, we adopt a "trust but verify" approach, allowing the OS to manage the system and sensitive applications like normal applications while ensuring that a higher privileged software called the *Enclave Manager* and hardware primitives prevent the OS from compromising the security of the sensitive application.

Figure 5.2 shows how *DESC* leads to a smaller TCB compared to the traditional enclave styles. Runtime-based and hypervisor-based enclaves require a large runtime or a guest OS inside the TCB. In comparison, *DESC* does not require any application resource management software inside the TCB. Based on a conservative calculation, we observe that *DESC* has a TCB size that is approximately **50%** less lines of code compared to the TCB size of Keystone (with Eyrie runtime) [1]. Smaller TCB can open doors for formal verification [140] as done for other TEEs such as Komodo [88].

5.3 Threat Model

Since we are focused on HPC system platforms, our threat model includes three main entities:

- *Platform Provider*: The HPC system administrator provides compute platform and other resources like storage, memory, and network. The platform provider has root access to each node in the HPC system.

- *Data Provider*: The data provider is an entity that owns sensitive data. The data provider provides its data (partially or fully) to a user (or a set of users) of HPC resources. For example, the data provider could provide access to medical data to a vetted research scientist for some analysis.

- *Enclave User*: The enclave user runs an application (enclave) on the provided HPC platform which may access the sensitive data.

The data provider and the enclave user do not assume trust in the platform provider. This may be due to regulations required of the data that is sensitive (e.g., medical data) [8]. We assume that the data provider trusts the enclave user with all or a subset of the data. For OS-related threats, we adopt a similar threat model to other trusted execution environments (TEEs) like SGX. In this model, the OS is untrusted and may be malicious. For multithreaded applications, our threat model assumes that all threads have the same access permissions to the enclave memory (i.e., the enclave memory is shared among all threads).

Out of scope threats The following threats and attacks are out of scope of this work.

- *Side channel attacks*: We assume that the user has sole use of the compute node, as is common in HPC environments. Thus, we do not consider side-channels [78, 141, 142] within the node in our threat model.
- *Physical attacks*: We assume the physical security of the devices is secured by another means (e.g., cameras). Thus, cold boot attacks [143, 144] and physical tampering [145] are out of the scope of our threat model.
- *Denial of service*: Our threat model does not include the DOS attacks by the untrusted OS, for example not scheduling the trusted application to execute, or using other means to stop the progress of a trusted application. Such attacks are easily detectable and at worst, halt the forward progress of the program.
- *Data Files*: We assume that files are encrypted and protected with integrity measures such as hashing (as provided by the data provider) while they are in transit or at rest in storage, and that users will verify the contents' integrity before use.

5.4 *DESC* Based Computing Systems

5.4.1 Background on today's computing systems

Figure 5.3 depicts a RISC-V computing system with hardware and software components shown. The OS has access to all physical memory, but `OpenSBI` (library to implement firmware/bootloader) can limit the OS's access to certain regions. However, given that we do not trust the OS, without relying on a higher privileged software/hardware, we cannot ensure that an application can protect its state from the OS.

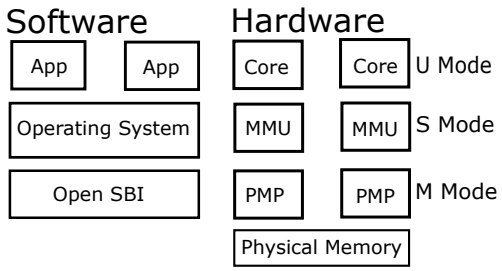


Figure 5.3: Overview of RISC-V based computing system.

5.4.2 RISC-V Isolation Mechanisms

RISC-V Privileged Modes: RISC-V provides three privilege levels to maintain execution mode isolation. The least privileged mode of execution is user-mode (U-mode), where normal user applications operate. The next level is supervisor-mode (S-mode), at which the OS operates. The most privileged mode is machine-mode (M-mode) at which software like the bootloader operates (as shown in Figure 5.3). Importantly, M-mode software cannot be modified by supervisor-mode software (e.g., by the root user or the OS) through the mechanism describe next.

RISC-V PMP (Physical Memory Protection): RISC-V's PMP feature controls access of user and supervisor mode to physical memory regions. The allowed access (*r-w-x*) permissions and the memory region can be configured using a set of PMP address (*pmpaddr*) and configuration registers (*pmpcfg*). These registers together constitute a PMP entry and can only be modified by the M-mode software (`OpenSBI` in Figure 5.3). Each entry defines a contiguous physical segment of memory with the same permissions. PMP entries define an *allow list* and every U/S-mode access needs to fall in some PMP range, otherwise an access fault is raised.

Virtual Memory Management: RISC-V provides different schemes for virtual memory management namely Sv39 (3-level page tables) and Sv48 (4-level page tables) for 64-bit systems. Virtual memory is managed by the S-mode software (e.g., the OS), not the M-mode software. When executing in M-mode, RISC-V always assumes the identity virtual-physical memory translation.

5.4.3 Security Guarantee of *DESC*

DESC provides the following security guarantee:

Enclave data (residing in memory or architectural registers) should be protected at all times (made inaccessible to other entities including the host OS).

DESC relies on RISC-V's hardware-based memory protection mechanism (PMP) to enforce this security invariant. We can utilize Figure 5.3 to explain the core concept of *DESC*, keeping in view that the OS is untrusted. The OS operates on all cores and manages the allocation of cores and memory sharing. The protection and isolation of memory are ensured by the *Enclave Manager* (extension of `OpenSBI`), a software running in M-mode, which operates on all cores. During a context switch, the OS manages the memory management unit (MMU), while the *Enclave Manager* updates the PMP entries to ensure protection on each core. Specifically, when an enclave application is active on a core, *Enclave Manager* configures the PMP registers to permit access solely to the memory addresses belonging to that application's data. Conversely, when the application is not running on a core, *DESC* configures the PMP registers to restrict access to those addresses. This prevents any other process or device from reading or modifying the application's data while it is inactive or suspended.

However, there are three specific cases that demand particular attention since we cannot trust the OS, but enclave applications still depend on it for resource management.

[C1] *Execution Mode Switch:* Enclave protection mechanisms must ensure that the enclave application state is not exposed when the core undergoes a mode switch from the application (U-mode) to the kernel (S-mode) or vice versa. The execution mode switch can occur due to either synchronous or asynchronous exceptions.

[C2] *Data Sharing:* When an application chooses to share data with the OS, such as during a system call, only the data explicitly marked as shared (e.g., as specified by the Linux kernel standard for system calls) should be exposed.

[C3] *OS-based Resource Management:* Since we rely on the untrusted OS for resource management, we must ensure that this resource management does not jeopardize the integrity or confidentiality of the application's data. For instance, we must ensure that the OS cannot modify the page mapping for the enclave application in a manner that

can compromise the confidentiality or integrity of the enclave application's sensitive data.

Prior research works [146–149] have already demonstrated the possible threats to an application's security because of an untrusted OS especially in the absence of any special care for the above three cases. In the following section, we outline the components of *DESC* and how they aid in ensuring security in these three special cases while implementing the aforementioned security invariant.

5.4.4 Design Principles for *DESC*

To fulfill the security guarantees mandated by *DESC*, we have established the following key design principles:

[**P1**] Preserve the security of enclave data, whether in memory or registers, at all times.

[**P2**] Ensure the protection of enclave state during execution mode switches.

[**P3**] Implement a data sharing model with the OS that exclusively exposes data explicitly marked as shared.

[**P4**] Proactively prevent unauthorized alterations to page mappings by the host OS.

In conjunction with these security principles, *DESC* adheres to the following design principles:

[**P5**] Enable the secure execution of **unmodified** applications.

[**P6**] Leverage the host OS for resource management to optimize efficiency.

These principles serve as the foundation for the design and operation of DESC, guaranteeing the system's security, integrity, and usability.

5.5 Design of Data Enclaves for Scientific Computing (*DESC*)

DESC separates **protection** from **resource management** by delegating these two functions to two different entities: protection is implemented by the enclave manager software and

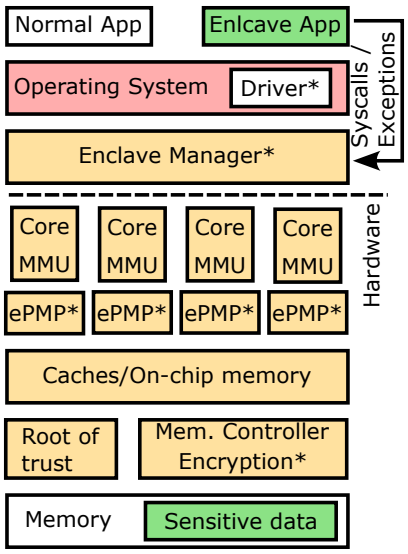


Figure 5.4: High-level overview of Data Enclave for Scientific Computing. Red is untrusted, orange is strongly trusted, green is sensitive. Stars (*) show the parts of the system we have added or extended. We discuss the driver in Section 5.5.1.2, the *Enclave Manager* in Section 5.5.1.1, the *ePMP* in Section 5.5.4.1, and encryption engine in Section 5.5.1.3.

hardware extensions, and resource management (e.g., tasks, memory, and I/O management) is handled by the OS such that the protection guarantees are not compromised. By leveraging an OS for resource management, we can take advantage of various built-in features, including page tables, preemption, and more, at no extra cost.

5.5.1 High-Level Overview

Figure 5.4 provides a high-level overview of our design of a data enclave. Green colored entities indicate that they are sensitive, red entities are not-trusted, and orange entities are trusted. Starred components (*) indicate *extended* software/hardware components required to implement *DESC* in a RISC-V based computing system.

DESC is made up of three main components:

- The Enclave Manager, an M-mode software, serves as the central component of *DESC*, responsible for managing and interacting with other components (e.g., the OS driver, memory protection hardware), tracking the state of enclave applications, and ensuring the system's security guarantees.

- Enclave driver which sets up the enclave and extends the OS so that it can support executing *DESC* applications.

- Hardware-based memory protection to enforce data confidentiality and integrity guarantees on ranges of physical addresses. We extend RISC-V's PMP registers (referred as *ePMP* in this thesis) for this enforcement.

Below, we provide a high-level overview of the above components, and then present details on how these components ensure enclave application data protection.

5.5.1.1 Enclave Manager

DESC takes inspiration from RISC-V based open-source TEEs, Keystone [1] and Sanctum [25], and uses a thin layer of trusted software (similar to reference monitor in kernel design) which runs at the highest privilege level (M-mode in RISC-V). We call this software *Enclave Manager*. The *Enclave Manager* cannot be accessed by a root-user (S-mode software) and is attested to ensure its integrity (e.g., during secure boot).

The *Enclave Manager* is responsible for three main tasks: 1) execution mode switch interception, 2) syscall interception, and 3) *ePMP* management. *ePMP* management allows the *Enclave Manager* to decide when a core should be allowed or disallowed access to an enclave application's memory.

1) Execution mode switch interception: In *DESC* all context mode switches are intercepted by the *Enclave Manager*. This interception allows the *Enclave Manager* to ensure that the enclave execution state would not get exposed to the OS. Similarly, the context switch to the enclave takes place via the *Enclave Manager* ensuring that correct execution state is restored. The secure execution mode switch triggers on both synchronous or asynchronous execution context switches.

2) Syscall interception: The *Syscall Interceptor* is a shim layer which intercepts system calls of sensitive applications and determines what data (or part of memory) an application intends to share with the OS. The *Enclave Manager* in turn exposes those parts of memory to the OS via reconfiguration of ePMP entries.

3) ePMP management: Enclave Manager is also responsible for configuring the *ePMP* registers to allow or disallow access to a protected memory region depending on if the executing context on the core belongs to an enclave application or some other application or the OS.

5.5.1.2 Enclave Manager Driver

To enable the proper execution of an enclave application, *DESC* requires minimal modifications in the underlying operating system (OS). These changes enable the *Enclave Manager* to safeguard the security properties of *DESC*, particularly in specific scenarios denoted as **C1** – **C3** previously. However, it is important to note that the OS itself remains outside the TCB. In the event of a compromise or bypass of our OS modifications, the potential consequence is termination of the enclave application, without compromising the overall system's security.

We made the following modifications in the OS through the use of a driver in our data enclave implementation:

- An OS driver to enable interaction with the *Enclave Manager*. This driver is responsible for requesting the *Enclave Manager* to create, run, or destroy an enclave via an SBI (supervisor binary interface) call.
- The idea of a secure process, to distinguish an enclave task from other tasks during task management.
- OS memory allocator modifications to allow physical memory allocation from a secure memory region for an enclave.
- Enabling control flow transfer to the *Enclave Manager* on returning from a system call execution so that the *Enclave Manager* can perform any sanity checks if needed. Moreover, the *Enclave Manager* is responsible for reconfiguring the memory protection unit to allow an enclave to execute.

Programming/Usage Model: We do not require any modifications in the sensitive applications. However, we use a runner application to launch an enclave. This runner application is responsible for interacting with the OS driver using an ABI (application binary interface) call which in turn interacts with the *Enclave Manager* using an SBI (supervisor binary interface) call to create and initialize the metadata to run an enclave. Similarly, this runner application will interact with the *Enclave Manager Driver* to destroy the enclave metadata once the enclave application has finished execution.

In our programming model, we assume that the entirety of the enclave application and the data it manipulates are sensitive. However, we can expand *DESC* to support the annotation of memory allocations as either sensitive or non-sensitive since we monitor all memory allocation system calls. By only protecting a subset of the application's memory, we may be able to reduce the overhead of *DESC*, although we have not evaluated this idea in this thesis.

5.5.1.3 Implementing Memory Protection via *ePMP*

Extended PMP (*ePMP*) is a per-core memory protection unit responsible for physical memory access controls. The extension to RISC-V's PMP registers ensures that the OS cannot break the address mapping integrity while managing application's page tables. The *Enclave Manager* is responsible for configuring the *ePMP* entries on every execution mode switch. The *ePMP* hardware primitive maintains access control at the granularity of arbitrary memory ranges. Range-based memory protection mechanisms (e.g., Mondrian [150] and its variants like PMP) fit well with HPC-style applications which generally require the same access permissions for large chunks of data. Moreover, scientific applications have fewer but larger memory allocations compared to the other types of workloads as observed by Ji et al. [151]. Therefore, tracking access permissions for each memory allocation separately becomes manageable for HPC applications. Though our implementation of *DESC* builds on RISC-V's PMP, *DESC* will work with any protection mechanism which provides hardware-enforced range-based access control.

Challenge of Using ePMP – Contiguous Physical Memory Regions: The memory region protected by a single *ePMP* entry must be physically contiguous. Instead of using the default Linux memory allocator where the corresponding physical memory may not be contiguous, we rely on the contiguous memory allocator (CMA) [152] to reserve a contiguous chunk of physical memory. This CMA region acts as a memory pool to handle all physical page requests for the enclave. We can use any implementation that gives contiguous memory allocations in both virtual and physical space (e.g., those found in prior works [129, 153, 154]), and chose to use the mainline CMA implementation for ease of implementation.

Figure 5.5 provides an overview of how physical memory is allocated for enclave applications given our modifications in the Linux kernel. Enclave applications rely on demand allocation like normal applications, where physical pages are allocated only when virtual

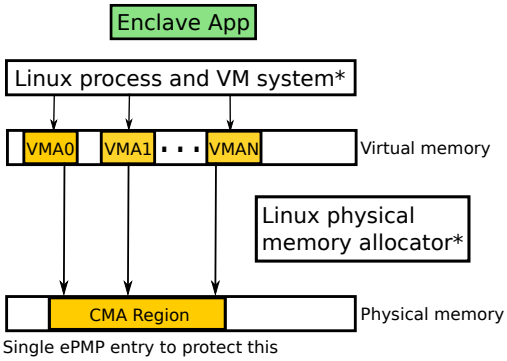


Figure 5.5: Modified Linux physical memory allocation. Starred entities are modified parts of the Linux kernel and interact with other kernel and data enclave components.

pages are accessed. We modify the Linux process and virtual memory system to track the status of enclave applications' VMAs, which are contiguous chunks of virtual memory that the kernel uses for different memory mappings (e.g., the code segment, stack, etc.) and tracking access permissions. The rationale for tracking memory allocations at the VMA level is discussed in Section 5.5.4. Each VMA is mapped to a contiguous physical region within the larger CMA region, and protected by *ePMP* entries.

Memory Encryption to Prevent Physical Attacks: Figure 5.4 shows an optional memory encryption engine in the memory controller as encrypting memory is sometimes required for compliance reasons. Our threat model does not strictly include physical attacks (e.g., cold boot attacks). However, our system uses an optional *Memory Encryption Engine (MEE)* in the memory controller to thwart some basic level of physical attacks. We assume a case where a per-enclave key will be generated in association with the *Enclave Manager*. Then the *MEE* will encrypt (or decrypt) every memory access leaving (or entering) the CPU package. We assume direct mode encryption, which can expose the encryption latency to read accesses and affect overall performance [155, 156]. In our evaluation, we assume a fixed latency of 30ns for each encryption operation. Section 5.7 of this thesis presents the slowdown of evaluated workloads with memory encryption.

Next, we discuss the three special cases that were listed in Section 5.4.

5.5.2 Case C1: Execution Mode Switch

DESC achieves secure execution mode switch by intercepting the control flow on execution mode switches. The *Enclave Manager* intercepts all the execution mode switches (from

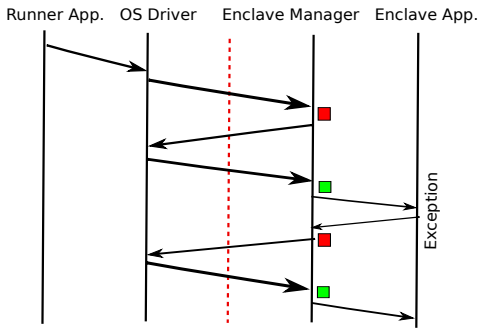


Figure 5.6: Secure control flow during context switch with *DESC* for a single thread of execution. Time moves forward vertically downwards. ■ and ■ show the points where the *Enclave Manager* intercepts the execution mode switch and disallow and allow access to the enclave memory on a particular core. Components on the right side of the red boundary are trusted, while untrusted components are on its left side.

a user mode to supervisor mode and vice versa) whenever an enclave application starts executing on a core. To enable this interception, we configure the RISC-V interrupt delegation registers [157] to not delegate interrupt handling to the operating system when the enclave application starts executing on a particular core.

Figure 5.6 provides a detailed view of how *DESC* ensures that the enclave application memory and its state are kept secure during context switches. As indicated by the trust boundary marked by a vertical red line in Figure 5.6, every time the execution control flow moves across the trust boundary, the *Enclave Manager* intercepts.

On interception at the time of exceptions (user to supervisor control flow), the *Enclave Manager* turns off the access to enclave memory region by configuring *ePMP* registers that belong to the enclave application memory. At this point, *Enclave Manager* also stores the enclave application register state and replaces it with a dummy state (unless application wants to share any of the registers, more on this in Section 5.5.3). The exception is eventually handled by the OS once the *Enclave Manager* passes the control to the OS.

When returning control to the enclave application from the operating system, the OS makes an SBI call to request that the *Enclave Manager* restore the application's context. If the exception was a system call, the SBI call also passes any return values. The *Enclave Manager* then swaps the dummy register state with the actual stored state of the enclave application, excluding any registers that were originally shared. Finally, the *Enclave Manager* makes the enclave application memory accessible to the core executing the application. The

Enclave Manager also guarantees that the execution state to which we are returning belongs to the enclave application. For this purpose, *Enclave Manager* consults the `EPC` (exception program counter) register of the stored execution state. If a malicious OS attempts to insert an incorrect control flow instead of returning to the enclave application, it would not be able to access the enclave application's memory. This is because the *Enclave Manager* would not have made it available on the core by reconfiguring the *ePMP*.

5.5.3 Case C2: Data Sharing

Applications interact with the OS via a system call interface. This interaction often includes an application sharing one or more parts of its state or memory with the OS. The data sharing is well-defined in time and space by the Linux system call interface. We rely on ***Syscall Interceptor*** and other parts of the *Enclave Manager* to ensure that only the intended data is shared with the OS and the rest of the enclave application state is kept secure.

Syscall Interceptor, intercepts on system calls and triggers relevant portions of the *Enclave Manager* to ensure safety of enclave application's data. The *Syscall Interceptor* itself is a simple look-up table like structure that makes the *Enclave Manager* aware of the system call being executed.

Unlike runtime-based enclaves (e.g., Keystone [1], Sanctum [25], and Graphene [80]), the *Enclave Manager* does not emulate system calls. Instead, it keeps its focus on ensuring *DESC's* security guarantee during system call execution via the mechanisms discussed in the previous sub section. Focusing on the security semantics of system calls leads to a smaller TCB compared to the alternative of emulation (reimplementing the entire system call functionality inside the TCB). To enable the sharing of data required for correct execution of the enclave application, *DESC* creates separate *ePMP* regions for the shared data (if it is in memory) or does not hide the state of the shared registers, while ensuring that only the intended data is shared.

Related work, such as Overshadow [158] and TrustShadow [159], typically uses argument marshaling to protect against potential security vulnerabilities. This involves copying the arguments from the application's address space to the OS visible address space. In contrast, the approach taken by *DESC* eliminates the need for extra copies.

Data Sharing via Syscall Arguments

To enable similar protection mechanisms for similar system calls, *DESC* groups them into classes based on how they specify their arguments. System call arguments directly or indirectly specify the shared data between the application and the OS.

1) Direct Arguments The first type of system call argument is direct arguments. This is the simplest case, where the arguments listed in the system call do not have any side effects, and the *Enclave Manager* only exposes these arguments, which are typically stored in the registers.

Examples of system calls that fall under this category include `nice` and `getpid`. In these cases, the arguments passed to the kernel only include basic information necessary for the system call to execute, such as the process ID for `getpid`.

2) Indirect Arguments In some cases, system call arguments are pointers to other structures or memory regions within the application's address space. For example, during `read` system call execution, the application passes a pointer to a buffer and the buffer size to the OS to write the data read from a file to the application's buffer. *Syscall Interceptor* intercepts the syscall and ensures that the OS will access only the region that the OS is allowed to access. For this purpose, *Enclave Manager* creates a separate permission region for the OS using a new *ePMP* entry, and once the call is completed, those permissions are revoked.

5.5.4 Case C3: OS-based Resource Management

The operating system (OS) manages hardware resources such as memory and execution cores via abstractions, such as virtual memory and processes. Virtual memory provides an abstraction for the physical memory of a computer, while a process is a unit of work that the OS schedules for execution on a CPU core. However, since the OS controls the hardware and the abstractions of the hardware provided to the user, a malicious OS could potentially exploit this control to access or leak an application's sensitive data. To prevent such attacks, *DESC* relies on *Enclave Manager* (especially *Syscall Interceptor*), OS modifications, and *ePMP*. Since, applications interact with the OS resource management tasks through system calls, *Enclave Manager* tracks the updates to resource management data structures in the OS. If the OS management deviates from the expected behavior of the application, *Enclave*

Manager detects such discrepancies and prevent the leak of sensitive data contained within the enclave.

5.5.4.1 Memory Management

DESC ensures confidentiality and integrity guarantee for enclave application's data in the context of OS-based memory management with the help of following mechanisms:

1) Tracking Application's Memory Allocations:

DESC tracks application's address space changes as a result of system calls like `mmap` and `brk`. This tracking is made possible by the OS changes that communicate any updates to application's VMAs to the *Enclave Manager* via an SBI call. As a result the *ePMP* entries responsible for protecting the physical memory of enclave application are updated.

2) Address Mapping Integrity via ePMP:

Since we allow a sensitive application to share the data directly via its address space, we must ensure that the address space (virtual to physical) mapping is not maliciously modified (e.g., page mapping vulnerabilities in Xen hypervisor [160]). Thus, we extend PMP registers to track both the physical and virtual address of each VMA region. These *ePMP* registers are inaccessible to the OS and can only be modified by the *Enclave Manager* ensuring their integrity.

In DESC, with *ePMP* registers, on each memory access the address translation produced by the TLB or the page table walker is *checked* by the *ePMP* register to ensure both the physical address is in the allow list and the virtual address check passes. Thus, it is not possible for the OS to modify the address translation without being detected. Figure 5.7, 5.8, and 5.9 provide more details of the *ePMP* entries and checks needed to validate a memory access.

Compared to previous works such as InkTag [161], our approach leads to simpler design. Instead of calculating a hash of the entire page table of a sensitive process every time a change is made to a page table entry, we effectively have an additional virtual to physical address translation stored in the *ePMP* entries that is not accessible to the OS.

5.5.4.2 Process Management

Once an enclave application process is launched, the OS driver labels it as a *secure process*. This labeling ensures that the OS can distinguish enclave process from other processes in

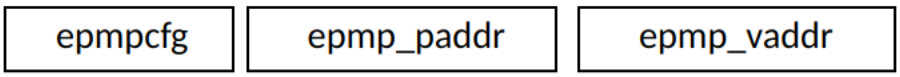


Figure 5.7: Single *ePMP* entry. *ePMP* stores virtual address of a memory range as well in addition to the physical address.

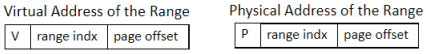


Figure 5.8: Virtual and physical addresses of a VMA range. Since, a VMA region is given a contiguous chunk of physical memory via the modified Linux memory allocator, VR and PR bits of the addresses should match.

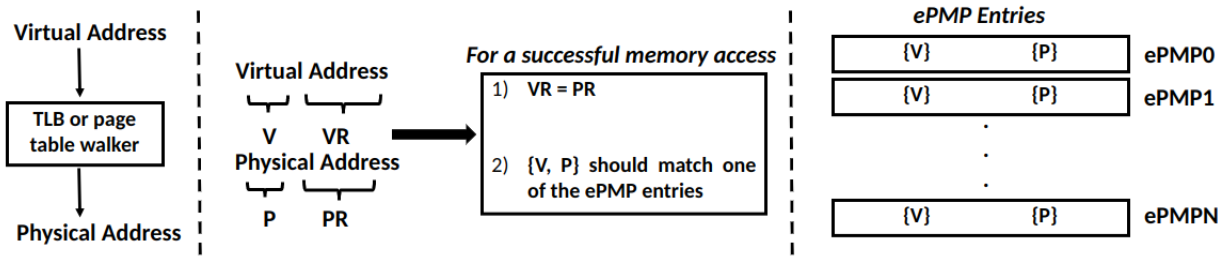


Figure 5.9: *ePMP* based memory access checks to ensure memory protection and address mapping integrity.

the system, and can keep *Enclave Manager* updated about the state of the secure process which is necessary for its correct execution/forward progress. An important aspect of process management for enclave applications is how *DESC* handles situations where an application creates a new thread using the `clone` system call. Following is a discussion on multithreaded execution of enclave applications.

Secure Multithreaded Execution: HPC or scientific computing applications are normally composed of multiple threads, which execute on multiple cores. One of the main benefits of relying on the OS resource management techniques, is that we can enable complex applications with much smaller effort compared to other TEEs.

Secure multithreaded execution is a consequence of our design choices, but can have a large impact as most of the existing TEEs do not have a good support of secure multithreaded execution. Today, TEEs have limited support for multithreaded execution which might reduce the system's efficiency overall.

We leave the thread management job to the OS and ensure that multiple threads of an

enclave will be executing on multiple cores. *Enclave Manager* manages the memory access permissions on all cores where the enclave threads are executing. *DESC* ensures when new threads are created using the `clone` syscall, they follow the same protocol for enclave memory and state protection as single threads would do (as explained in the previous subsections). Detailed workflow of new thread creation is discussed in Section 6.3.

The *Enclave manager* ensures atomic access to all enclave data structures when a multi-threaded enclave is executing and also takes care of synchronization of memory permissions across cores whenever *ePMP* configuration changes.

Control of synchronization primitives inside the OS leads to different types of attacks [58, 162]. *DESC* ensure that even when the scheduling and synchronization decisions are kept with the OS we can keep enclave application secure. `futex` system call is used by threads for synchronization access to critical sections. Using the data sharing methodology discussed earlier, on a `futex` call, *Enclave Manager* creates a new *ePMP* entry to make `futex` word readable to the OS. This mitigates any malcious tampering with the `futex` word.

5.5.4.3 Other Types of Resource Management

Following is a discussion on different types of resource management related system calls which may not pose a threat to our security guarantee, or are not included in our threat model.

- File Manipulation: File manipulation system calls, such as `open`, `close`, `seek`, and `read` have access to the data moving to and from the files used by the application. We assume the application will encrypt the data before moving it to the files. As future work, this can also be delegated to the *Enclave Manager*. However, through our data sharing mechanism, we ensure that only the relevant part of the enclave application memory is exposed during the `read` and `write` system calls.

- Communication among processes: These system calls (e.g., `pipe`) also usually rely on file interface to create communication channels among processes. We rely on similar assumptions for these system calls as above.

- Information from the OS: These system calls generally do not have input arguments and rely on OS to provide information from system resources (e.g., `time`). The only possible threat with these system calls requires checking the return value. Unless the

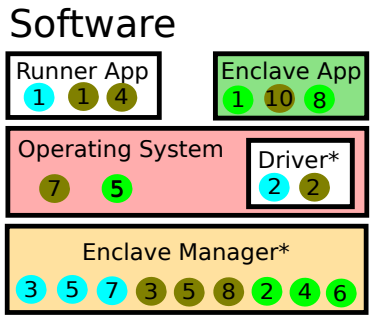


Figure 5.10: Software components involved during an enclave creation and execution. (x) shows the flow during enclave creation, (x) shows the flow during enclave execution, and (x) shows the flow during new enclave thread creation.

application's security properties depend on the returned value, a wrong return value will not expose application's sensitive data.

- Accessing I/O: I/O access via the file-system interface also assumes that the application protection mechanisms will be in place for any shared data with the I/O devices. For device specific communication, `ioctl` syscall is used to communicate to devices (e.g., tty). These calls pass a device file descriptor, device-specific request, and other device specific arguments (that can be pointers to user-space memory). Device specific arguments make it challenging to enable secure `ioctl`. Our approach towards protecting enclave data during `ioctl` calls is conservative. We create an allow list of the `ioctl` requests that the application trusts and only those requests are permitted to proceed.

5.5.5 Out of Scope Components of Enclave

The *Enclave Manager* is responsible for maintaining the security of the enclave's memory and state, both during enclave execution and during transitions between the enclave and the operating system. In addition to secure interaction between the enclave and the OS, the *Enclave Manager* performs basic operations such as initialization and authenticated launch of the enclave. Established strategies can be utilized for these operations, our focus is primarily on designing the data enclave, which is independent of these decisions.

5.6 *DESC* Workflow

Figure 5.10 to 5.12 depict the workflow of data enclaves in various scenarios: 1) enclave creation, 2) running enclaves, and 3) creation of new execution threads by a running enclave.

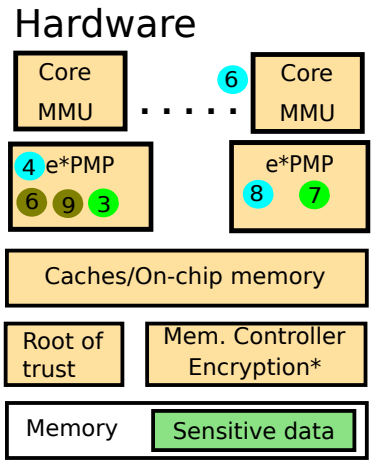


Figure 5.11: Hardware components involved during an enclave creation and execution. (x) shows the flow during enclave creation, (x) shows the flow during enclave execution, and (x) shows the flow during new enclave thread creation.

Figure 5.10 represents the software components of data enclaves, while Figure 5.11 illustrates the computing hardware components. Lastly, Figure 5.12 provides a memory protection view from each core's perspective. We discuss the above-mentioned three cases one by one:

5.6.1 Enclave Creation

Runner application requests the data driver to create the enclave (1), and the driver creates the contiguous memory region, which will be served as enclave memory (2). The driver then forwards the request of enclave creation (which includes the enclave memory information) to the *Enclave Manager* (3). The *Enclave Manager* initializes the metadata for the enclave and then configures an *ePMP* entry for the enclave's memory region so that the enclave memory can be made inaccessible to the core on which the *Enclave Manager* is currently executing (4) (in Figure 5.11). The *Enclave Manager* then sends IPIs (inter processor interrupts) to other cores to synchronize them with the same memory permissions (5). Once the other cores are interrupted (6), the *Enclave Manager* will take control of those cores (7) and configure the *ePMP* entries on those cores (8) to have a synchronized memory view across all cores. At this point, the enclave is created, and the enclave memory (EnM) will be inaccessible from all cores. The memory view of cores during the enclave creation process is shown in Figure 5.12, where the EnM is inaccessible on all cores after creation of the enclave as all cores are executing OS or any other normal application (NA).

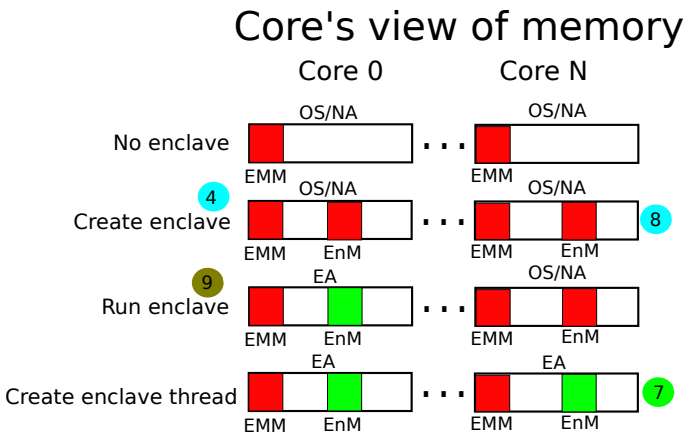


Figure 5.12: Memory accessibility view of processor cores. ■ indicates accessible memory region and ■ indicates an inaccessible memory region. NA: normal application, EMM: *Enclave Manager* memory, EnM: enclave memory, EA: enclave application. For simplicity, multiple VMAs of enclave application memory are shown as a single region i.e., EnM.

5.6.2 Enclave Running

Running the enclave application requires the runner application to first request the *Enclave Manager* (via the OS driver) (1) (2). The *Enclave Manager*, apart from updating any enclave metadata to indicate the current status of the enclave, configures the RISC-V interrupt delegation registers so that the *Enclave Manager* can intercept the interrupts on the core where enclave is going to execute (3). After this configuration, the *Enclave Manager* sends the control back to the runner application (4). Since data enclaves delegate the management responsibilities to the OS, we need to start the enclave application as a normal Linux process. Therefore, the runner application uses the exec call to execute the enclave application (4). The *Enclave Manager* (5) will intercept this exec syscall. The *Enclave Manager* will configure the *ePMP* entries (6) to ensure that the enclave memory cannot be accessed once the control is transferred to the OS (7). The OS will execute the exec syscall and initialize the process. Once the OS schedules the secure process (enclave application), the control will be first intercepted by the *Enclave Manager* (8). The *Enclave Manager* ensures that the *ePMP* is reconfigured to allow the core access to enclave memory (9). Finally, the *Enclave Manager* transfers the control to the enclave application (10).

Third row of Figure 5.12 shows what part of memory is accessible to all cores during the enclave execution. EnM is accessible only on the core where the enclave is executing (core 0 in the example) and is inaccessible on all other cores which might be executing OS or other

Table 5.2: Main feature of the configuration tested on gem5

Feature	FU740-like
Core Pipeline	5 stage in-order
Dcache size	32KB
Dcache assoc.	8
L2 cache	2MB
L2 cache assoc.	16
DTLB entries	128

applications.

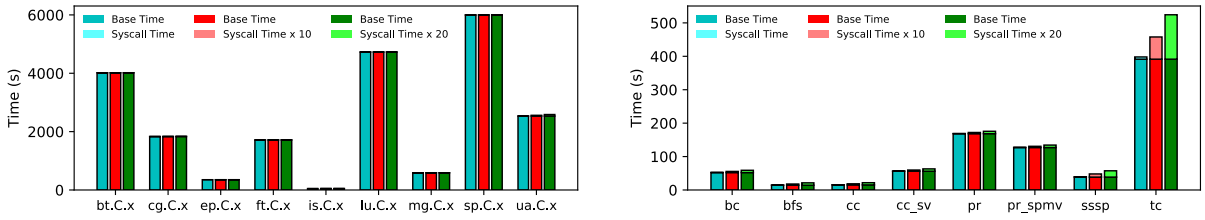
5.6.3 Creating New Enclave Thread

Once a running enclave wants to create a new thread, the enclave application will execute a `clone` syscall (1). The *Enclave Manager* will intercept this system call (2). Enclave manager will first configure *ePMP* to disable access to enclave memory (3) and will then give control to the OS (4) (5). The OS creates the new thread and will (eventually) schedule it to execute (on Core N in the example shown in Figure 5.11). The control to the new thread of enclave will take place via enclave manager (6), which reconfigures the *ePMP* entries (on Core N in the example) (7) to enable enclave memory access on the core and the control eventually goes to the newly created enclave thread. Independently, the main thread (the thread which created this new thread) will get scheduled by the OS on a different core.

Figure 5.12 shows what parts of memory different cores can access once the new enclave thread is created (the last row in the figure). The example assumes that the main thread of the enclave will keep on executing on Core 0 and the new thread will execute on Core N.

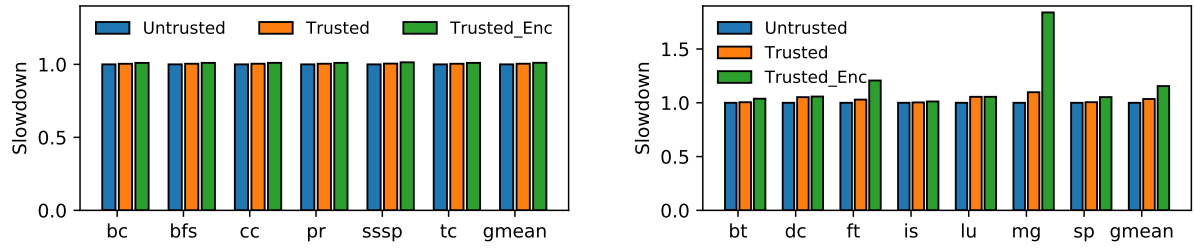
5.7 Results and Evaluation

We used NAS Parallel Benchmark suite (NPB) [95] to evaluate *DESC* that has been traditionally used to benchmark HPC systems. The NPB benchmark suite contains kernels and pseudo applications which can be used with different input data sizes. In addition to the conventional scientific computing kernels/workloads, we also use workloads that represent contemporary usage in high-performance computing (HPC). We picked a set of graph workloads, the GAP benchmark suite (GAPBS [96]), with a synthetic graph as an input.



(a) NPB (b) GAPBS

Figure 5.13: Impact of change in the system call execution time on the overall execution time. The motivation behind using syscall inspection for HPC-style workloads is that the time spent in system calls (even if scaled by a large factor) is a small fraction of the overall execution time.



(a) GAPBS (b) NPB

Figure 5.14: Comparison of slowdown (does not include enclave creation penalty) for GAPBS and NPB benchmark suite. Trusted refers to the trusted execution of the benchmarks (using *DESC*) and Trusted_Enc refers to the trusted execution with memory encryption (for data leaving the CPU package) on as well.

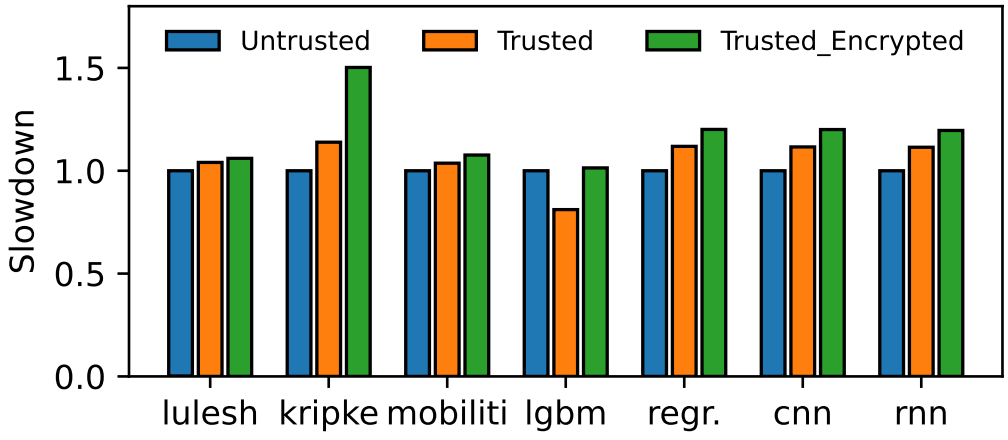
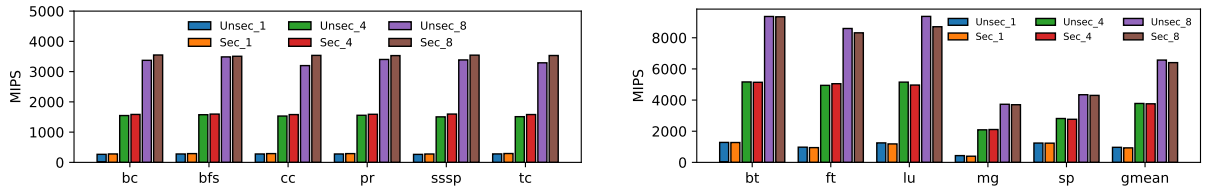


Figure 5.15: Slowdown for modern HPC and ML workloads. Regression (linear regression), CNN (convolution neural net.), and RNN (recurrent neural net.) are based on Torch.



(a) GAPBS (b) NPB

Figure 5.16: Million (usermode) instructions executed per second of simulation time. This is the sum of instructions across all cores. Unsec_[cores] refers to unsecure execution and Sec_[cores] refers to trusted execution with *DESC*, where [cores] is the number of threads of the benchmark and processing cores.

We used modern HPC workloads, including Kripke [97] (a particle transport simulation), LULESH [98] (a hydrodynamics simulation), and Mobiliti [100] (a transportation benchmark). Kripke [97] is a highly scalable code which acts as a proxy for 3D Sn (functional discrete-ordinates) particle transport. Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) [98] application solves a simple yet "full-featured" hydrodynamics simulation problem. Mobiliti [100] is a transportation system simulator (based on parallel discrete event simulation), designed to work on high performance computing systems.

As a proxy for general machine learning training we used a decision tree workload (LightGBM) [99] (characterized by irregular memory accesses) which is trained using Microsoft's Learning to Rank (MSLR) data set. In addition, we use Torch based machine learning workloads training, including linear regression, convolution neural networks, and recurrent neural networks.

We implemented our Linux kernel changes in Linux kernel v5.7. These changes include the kernel module to interface with the M-mode *Enclave Manager*, an updated memory allocator, and a notion of a secure process in the Linux kernel. We use the security monitor of Keystone as a baseline to implement our *Enclave Manager*. This software is compiled as part of the `OpenSBI` bootloader.

Evaluation Methodology: For functional evaluation, we successfully ran the selected workloads till completion in QEMU [7]. For performance evaluation, we simulated the selected workloads for a fixed 1 second of simulation time using an HiFive UnMatched Board (RISC-V FU740) like configuration on gem5 [5,6,163] simulator. The details of the RISC-V FU740 con-

figuration are shown in Table 6.1. We used gem5's `MinorCPU` to model the in-order pipeline of FU740. We vary the number of simulated cores from 1 to 8 in gem5. We use user-mode instructions executed in the fixed simulation time as an indicator of the workload progress to compare performance across different tested configurations.

The primary motivation behind syscall inspection is that the HPC applications do not spend much time on syscall execution. Our design of *Syscall Interceptor* is based on a study of the system calls which get executed in HPC applications. For this purpose, we collected traces of system calls and the execution time they take, using `strace` in Linux on a RISC-V machine.Figure 5.13 shows the system call execution time in comparison to the execution time for the rest of the program for NAS Parallel Benchmarks (NPB) and GAPBS (Figure 5.13a and 5.13b respectively). In the base case, the system call time is almost negligible and is not visible in the plotted bars.

Figure 5.13a and 5.13b also show the stacked bars with system call execution time scaled to much larger values. In case of NPB, even if the syscall time is to be increased by 20×, the overall syscall execution time stays insignificant in comparison to the rest of the execution time of the program (start to become visible when scaled by a factor of 200×, not shown in Figure 5.13a). For GAPBS, the behavior is similar except *tc*, which has a higher number of system calls primarily because of very aggressive use of print statements.

Figure 5.14 shows the slowdown of trusted execution for GAPBS and NPB workloads compared to the untrusted baseline. This figure includes three different configurations, 1) normal/untrusted execution, 2) execution with data enclave, and 3) execution with data enclave with encryption of data leaving CPU package. Figure 5.14 shows that with *DESC*, benchmarks do not show any significant slowdown. With encryption of memory in addition to execution under *DESC*, the geometric mean of slowdown is still small. The performance shown in Figure 5.14 does not include the enclave creation time. However, the fixed penalty of enclave creation is not significant and can be easily amortized for long-running HPC workloads. Figure 5.15 show slowdown of trusted/secure execution of other modern HPC workloads and training of some machine learning models. As shown in the figure, the maximum slowdown is less than 20% for these workloads. lgbm shows performance improvement when run only in Trusted mode (with *DESC*) due it's high affinity for contiguously allocated

physical memory.

One of our goals is to ensure that secure execution of multithreaded workloads scale similarly as untrusted execution. Therefore, we conduct performance comparisons between trusted and untrusted execution while running workloads with varying numbers of threads (on multiple cores). Figure 5.16 shows the performance of secure/trusted execution for multi-threaded applications. We present the throughput (cumulative user mode million instructions executed per second) of GAPBS and NPB benchmarks with different core counts (1, 4, and 8) for unsecure and secure execution. Figure 5.16 shows that the workloads' performance improves with increasing the core count (and thread count) similarly for both unsecure and secure execution of the benchmarks.

5.8 Conclusion

In this chapter, we introduced *DESC*, a novel design for trusted execution environments (TEEs). Unlike existing TEEs that require extensive application modifications or include an entire operating system (OS) in the trusted computing base (TCB), *DESC* leverages an untrusted OS for resource management while ensuring data confidentiality and integrity. We implement *DESC* on the RISC-V ISA.Our evaluation using gem5 and QEMU shows that *DESC* performs well across various scientific computing workloads, with minimal overhead compared to running outside the enclave.

Chapter 6

Simulation and Architectural Evaluation of TEEs[1]

Software simulators form the first level of "agile hardware design stack" [164], and are useful to iterate on high-level architectural tradeoffs and hardware/software co-design before focusing on the hardware implementation of a model (e.g, RTL). Therefore, it is important to build simulation models to be able to evaluate the hardware/software co-design ideas proposed in this document. To this end, we established a baseline simulation model to enable future architectural research on hardware/software co-design for secure compute environments. We focused on RISC-V based TEEs, Keystone [1], and a widely-used architectural simulator gem5 [5]. The RISC-V ecosystem has support to perform functional or RTL level simulation of RISC-V TEEs using QEMU [7] or FireSim [83]. However, the ecosystem lacks a tool or simulator to perform high-level architectural and micro-architectural studies of RISC-V TEEs at a cycle-level for the objective of early design space exploration, and researchers have to rely on analytical modeling or RTL implementations for their studies involving Keystone [165]. Keystone is proposed as a "customizable" TEE for RISC-V, which makes it an appropriate choice to use to provide simulation support to enable a baseline on top of which new designs can be implemented. This work is part of upstream gem5 [5].

[1] **This work has been presented at CARRV 2021 [163]**

6.1 Keystone[2] in gem5

In this work, we extended the privileged ISA support to add RISC-V PMP (physical memory protection) hardware in gem5 which enables running Keystone's Security Monitor (SM) on gem5. Figure 6.1 provides an overview of the PMP implementation in gem5. There are three components which interact with each other: the ISA subsystem, the MMU unit, and the PMP unit. Any read of the PMP registers returns the registers' value, and writing to a PMP register (eventually) triggers a call to update the PMP rules which are maintained in a PMP table (set of PMP entries). When a memory access is made and the MMU (TLB or page table walker) has generated the physical address corresponding to a program (virtual) address, a call is made to PMP unit to detect if a PMP check should be made or not (depending on the current mode of execution) [157]. If a check is desired, the PMP table is consulted to find out if there is an entry match/mismatch for both address and the permissions. If no match is found a fault is raised, otherwise control returns back to MMU with a successful check.

Keystone's SM is shipped as a part of both BBL and OpenSBI bootloaders. We have tested both bootloaders with the SM on gem5. We further set-up all Keystone components for simulation on gem5 and performed different tests to check the validity of runs. The components include:

- Bootloader (OpenSBI)
- SM (compiled as a part of OpenSBI)
- Linux kernel (compiled with OpenSBI)
- Keystone driver
- Benchmarks/tests with a test runner application
- Buildroot based disk image

Detailed instructions on how to build these components and use them with gem5 are provided in `https://github.com/darchr/Keystone-experiments`. This repository also con-

[2]**More details on Keystone are available in the Appendix section 8**

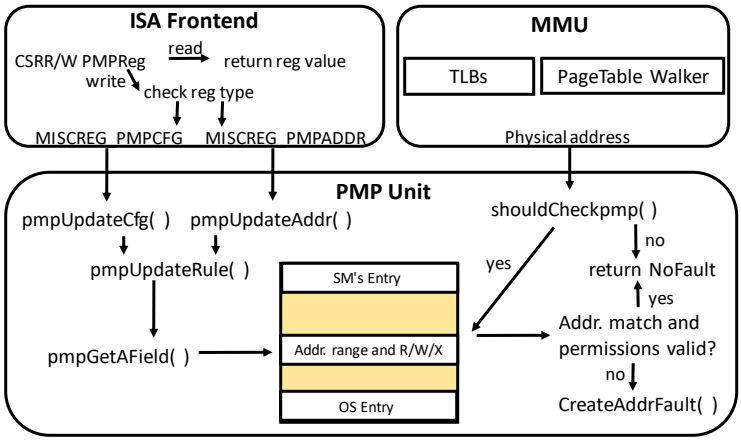


Figure 6.1: PMP implementation in gem5

tains scripts to launch gem5 based Keystone experiments using gem5art [166] (a tool to run gem5 tests in a structured and reproducible way).

6.1.1 Validation

In this section, we validate and evaluate Keystone's implementation in gem5. We relied on the following actions for the functional validation of this implementation:

- We performed physical memory access checks using Linux Busybox utility to test functionality of PMP, which passed successfully.
- We successfully ran primary Keystone tests, which, in addition to performing some basic functionality tests, check if an enclave access control is violated or not.
- Finally, we successfully tested the workloads used by Lee et al. [1] and found similar performance results.

In addition to the functional validation, we also validated the performance of gem5's Keystone implementation. To investigate this, we performed some experiments and collected performance numbers for Keystone benchmarks on gem5 and compared them with the performance numbers published in the Keystone paper [1].

Figure 6.2 shows a comparison of the slowdown experienced from enabling trusted execution on two different gem5 CPU models, and the slowdown numbers taken from the work of Lee et al. [1] for rv8 benchmark suite [2]. This figure shows that the Keystone simulations on

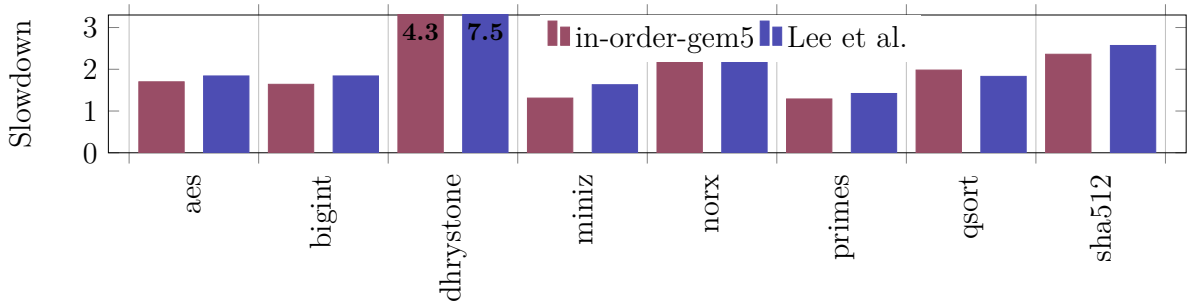


Figure 6.2: Comparison of slowdowns (incurred by trusted execution using Keystone) between gem5 and Lee et al. [1]. This slowdown includes enclave creation and management time as well.

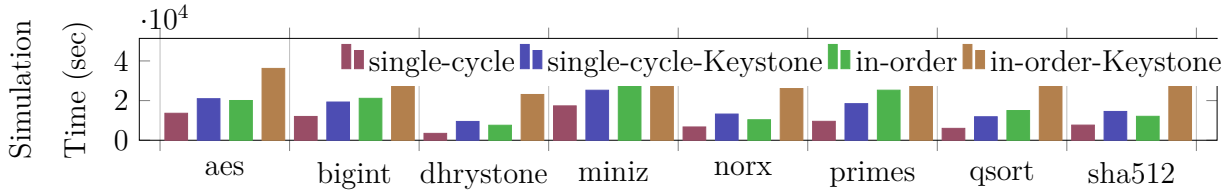


Figure 6.3: Time taken by gem5 to simulate rv8 [2] benchmarks on a single cycle (TimingSimpleCPU) and an in order (MinorCPU) CPU models of gem5 with and without Keystone.

gem5 exhibit similar performance numbers and trends as in the work of Lee et al. [1]. The slowdown numbers shown in Figure 6.2 include benchmark execution as well as the enclave creation, destruction and management time. *dhrystone* which has the smallest execution time in normal (untrusted) execution shows the biggest overhead for trusted execution, because the cost of enclave creation, and management becomes more dominant due to its small execution time. Similar is the case for *sha*512 and *norx*, which have slightly higher execution time compared to *dhrystone*, but still relatively less in comparison to other workloads.

Figure 6.3 shows the performance of gem5 itself (i.e., the time taken by gem5 to perform a simulation). Simulating an in order CPU (called MinorCPU in gem5) takes more time in comparison to a single cycle CPU (called TimingSimpleCPU in gem5). It should be noted that the difference in simulation time of a trusted and untrusted execution is because of the difference in amount of instructions/work that is simulated.

Next, we present a possible use-case[3] of the simulation support of Keystone in gem5.

[3]More use-cases can be seen in the original paper [163].

Table 6.1: Main feature of the configurations tested on gem5

Feature	**default**	**fu540-like**	**large**
Dcache size	32KB	32KB	512KB
Dcache assoc.	8	8	8
L2 cache	N/A	2MB	16MB
L2 cache assoc.	N/A	16	32
DTLB entries	64	128	2048

6.2 Case Study: Microarchitecture Impact on Performance of Secure Execution

This use-case discusses how changing the microarchitecture can impact performance of the trusted execution and how would it relate to the performance of the untrusted execution on the same platform. A secure computer architect can be interested in this kind of analysis while working on a new system. Cycle-level simulation is a quick way to perform this kind of design space exploration.

As an experiment, we picked single cycle (TimingSimpleCPU) and in order (MinorCPU) CPUs of gem5 and configured their memory and cache subsystems in three different ways (thus leading to six total configurations). The three memory and cache subsystems refer to def (default gem5 configuration), fu540 (fu540 like configuration), and large (a configuration with large structures and low latencies). Table 6.1 provides some details of these configurations.

We executed rv8 benchmarks in untrusted and trusted manner for all the six configurations, thus leading to 12 runs for a single benchmark. Figure 6.4 shows the execution time for all of these runs for each benchmark. We can observe that the overall execution time goes down as we move towards more aggressive configurations, however the ratio of trusted to untrusted execution time for each configuration stays similar. In other words, even on aggressive configuration, trusted execution incurs similar performance penalty (relative to untrusted execution) as it does on a simple configuration.

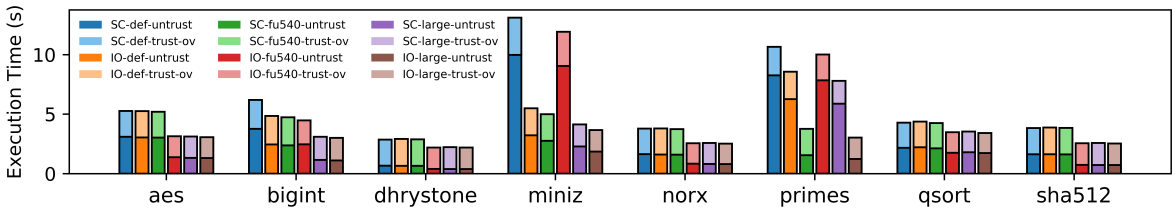


Figure 6.4: Microarchitecture impact on performance of secure compute environments. In the legend entries SC: single cycle, IO: in-order, def: default configuration from Table 6.1, fu540: fu540-like configuration from Table 6.1, and large: large configuration from Table 6.1. 'trust-ov' stands for overhead of trusted execution.

Chapter 7

Future Work

In this chapter, we discuss future research directions that can be followed to improve the state of the art of TEEs for HPC. These directions include ideas for existing commercial TEEs and also for our proposed secure architecture *DESC*.

7.1 Improving Existing TEEs

Based on our study of existing trusted execution environments presented in Chapter 4, we present a few research directions/aveneues that can be followed to make the existing commercial technologies more suitable for HPC.

7.1.1 Software Frameworks

First we discuss some research directions for software frameworks:

Intelligent Job Scheduling: Security requirements of HPC workloads can be diverse. At the same time, the HPC platforms can be heterogeneous possibly composed of nodes from multiple vendors e.g., Intel and AMD, thus making both SGX and SEV available in the same environment. Therefore, we propose the idea of an intelligent job scheduler which can allocate applications to an appropriate node depending on their sensitivity and the expected slowdown from the secure environment. The sensitivity of the workload can be fed to the scheduler from the user and the expected slowdown can be calculated using pre-trained models. As a proof of concept, we train a second-degree regression model to predict the slowdown of workloads under SGX, using only the features from normal execution of programs.

The slowdown under SEV can be assumed to be negligible if interleaved NUMA allocation is used as we showed in our performance analysis. For SGX, we can achieve a mean square error of 2.81×10^{-11} (an R2 score of 0.99) for a second-degree regression model. The features (normalized to per million instructions) used in the model include: resident_memory, native_time, syscalls, loads, and mem_accesses in the order of the highest to the lowest observed correlation (with the slowdown). For SGX, the highest correlation with slowdown is shown by resident_memory feature which is directly related to EPC faults which appears to be the primary reason for SGX slowdowns (section 4.4.4).

Automatic Application Partioning: Partitioning of applications into secure and non-secure parts is a difficult task, especially in HPC settings as HPC workloads often rely on various third-party libraries. Thus, there is a need for support of automated porting of applications to secure environments and their partitioning into secure and non-secure parts. For instance, Lind et al. [167] have already proposed Glamdring which is a framework for automatic application partitioning into secure and non-secure parts. Automatic application partitioning not only makes it easier to use secure environments, it can also help in mitigating the performance slowdowns by keeping the secure memory footprint to the minimum.

Dynamic Migration of Application Segments: Another research direction to explore is building of tools to shift sensitive functions or parts of the application transparently to enclaves at runtime. Similarly, the outsourcing of unsecure parts of the application to (untrusted) accelerators or general cores to improve performance can be done transparently using tools (if developed).

KVM Exit Clustering for Mitigating Slowdown: For virtual machine based TEEs, like AMD SEV, by scanning and decoding upcoming instructions the hypervisor can identify the ones that will cause a KVM exit. Then a cluster of exiting instructions can be formed which can be executed all at once, ultimately reducing the overall exit rate and the cycles spend for saving/restoring the VM's state leading to improved performance.

7.1.2 Research Avenues for Computer Architecture

Following are some research possibilities to explore for hardware enhancements to enable more optimized trusted execution in HPC settings:

Secure Memory Size: The limited secure memory size (as in SGX in the machines we tested in Chapter 4) is an issue that can have a severe impact on the performance of trusted execution of HPC workloads. One of the biggest hurdles in increasing the size of the EPC (secure memory) in SGX is the cost associated with the metadata that is needed to provide security guarantees of confidentiality and integrity. SGX maintains an integrity tree, composed of large counters, to keep track of the version of EPC pages to protect against replay attacks. We believe that there is an opportunity to optimize the design of integrity trees along with the other metadata to enable the bigger size of secure memory. For example, recent research by Taassori et al. [92] introduced a variable arity integrity tree (VAULT) that results in a compact design with low depth. Saileshwar et al. [168] proposed a compact integrity tree that uses morphable counters rather than fixed-size counters to accommodate more counters in the same memory area.

Currently in Intel SGX, whenever there is an EPC page fault, it is handled by a fault handler in the SGX kernel driver. This handling is a very expensive process (also requires the logical control flow to exit the enclave), which leaves room for some kind of hardware-based acceleration of this fault handling. An example of research efforts to reduce this cost is the work of Orenbach et al. [169], Eleos, which uses software-based address translation and management inside the enclave to eliminate the need of flowing out of the enclave in case of a page fault. Similarly, there lies an opportunity to explore the dynamic remapping of pages rather than actually copying them from one memory type to another.

Intelligent Page Prefetching and Eviction: By learning the memory access pattern of applications, the sensitive pages can be prefetched in the secure memory from non-secure memory regions even before they are actually accessed, thus reducing the number of access faults. Currently, before control is transferred to the kernel driver to handle EPC fault, lower 12 bits of CR2 register (holding the faulting virtual address) are cleared because of security

concerns. Thus the driver is not able to use those bits to help in predicting memory access patterns. Moreover, Gjerdrum et al. [170] have shown the page fault handler to be over-eager and unable to utilize EPC exhaustively. There lies an opportunity to enable kernel driver to use the application's memory access patterns to prefetch pages anticipated to be used or perform smart page eviction. Similarly, the pre-fetching and page eviction can be handled entirely in the hardware making it part of the TCB.

7.2 Exploration of New Ways to Build TEEs

Based on our analysis and study of TEEs presented in Chapter 3, we enlist following directions to pursue to explore new ways of building TEEs that will be suitable for HPC as well.

7.2.1 New Hardware Primitives

Most of today's commonly used operating systems (Linux, Windows, macOS) use a monolithic kernel, which provides a fixed abstraction to the user-mode applications and depends on a generic or a homogeneous view of applications while managing their resources. However, many other kernel designs have been proposed in the past. For example, exokernel [32] was based on the idea of application-level resource management. The applications again are pursuing this direction by trying to relinquish the monolithic nature of the kernel. Today, the applications are mainly doing this for performance reasons.

For example, in modern computing systems, HPC applications often bypass the OS and handle I/O in user-space libraries or run-times via a specialized HPC networking stack (e.g., RDMA [126] via InfiniBand [171]). The noteworthy point is that these libraries tend to make certain assumptions about HPC. For example, modern MPI libraries assume that they can fully utilize CPU cores to spin on network hardware resources to check for progress. Less dependence/reliance on a (primarily untrusted) OS fits well with the confidential computing model. Moving the resource management code to user space also means a larger software TCB for the enclave.

A larger TCB also typically means more exploitable bugs. On the other hand, hardware is more trustworthy due to two primary properties: immutability and privilege [172]. Therefore, we emphasize focusing on increasing the hardware TCB components. We believe that there is an opportunity for computer architects to think of new hardware primitives as the ones

we have today do not work well, as discussed in Section 3.5.

7.2.2 Horizontal Privilege Levels

As discussed before, the vertically integrated model of privilege levels does not work for evolving high-performance computer systems. We emphasize horizontal privilege levels, where there can be a variable number of vertical layers in each horizontal privilege level (e.g., there might not be an OS in the privilege hierarchy of an accelerator). Depending on the threat model, one horizontal layer can have more privilege than the other. An example of a similar system is ARM TrustZone [23] which divides the system into a secure and a normal world. However, it does that only for a CPU system and cannot create a secure world for accelerator or other remote computing elements/memory nodes.

7.2.3 Capability Based Enclaves

The idea of capabilities (first proposed a few decades ago [173]) provides ways to enable compartments at different abstraction layers of computing systems. Capability based machines have also existed for a long time now (e.g., M-Machine [174], Rice research computer [175], CHERI [176]). Capability based architectures inherently do not have to follow a vertically integrated privilege hierarchy, rather these architectures rely on capabilities of different components in the system to perform access control checks. Therefore, using capabilities to implement enclaves seems a promising idea for (heterogeneous) high-performance computing systems.

7.3 Future Work on *DESC*

In this section, we present the possible future extensions to the proposed TEE in this thesis.

7.3.1 D-DESC – Disaggregated Data Enclaves for Scientific Computing

Today's HPC systems over-provision resources to make sure that certain applications will have their requirements satisfied. HPC nodes might use less than 25% of their overall memory at times [177, 178]. Future HPC systems are expected to decouple compute and memory extensively, thus leading to disaggregated architectures, where multiple processing elements (of different kinds) can potentially access a large pool of memory resources [179]. These dis-

aggregated architectures are expected to improve the resource utilization, but can increase security concerns (due to lack of system level access controls) especially if different processing elements are used by multiple users at the same time. In addition to access control concerns, another unresolved problem is of expanding enclaves across these disaggregated architectures. To this end, we propose to expand our baseline TEE design (DESC) to provide scalable mechanisms which can allow a user to form a secure enclave spanning across multiple processing elements.

Though the disaggregation of resources could be across all dimensions, in this work we focus on disaggregated memory systems due to our focus on data enclaves for scientific computing. An example disaggregated memory system is shown in Figure 7.1. Multiple nodes are connected to a pool of memory resources through a coherent interconnect fabric (e.g. CXL [35], Gen-Z [180]). These systems are also referred as Fabric Attached Memory (FAM) systems. These systems often employ a memory broker/manager node to keep track of memory resources needed by different nodes. There are two fundamental ways of managing FAM systems: 1) entire memory pool is exposed to the OS on all nodes (this requires modifications in the OS) or 2) each node provided with a view of flat memory space that belongs to it and a translation module/unit performs translation of node address to FAM address (transparently to the node). The second mechanism of translation seems promising from the perspective of security as it provides a way to perform access control checks (during memory translation unit), similar to what happens in traditional MMU systems. However, the translation based management can be costly as two level address translation (similar to two-dimensional page table walks in virtualized systems) might need as many as 24 memory accesses for a single translation [181].

Threat Model:

A single node threat model remains the same as it was discussed in Chapter 5. Multiple nodes do not trust each other. However, the *Enclave Manager* running on each node can be considered as trusted by the enclave user and the data provider. Moreover, we assume that the *Enclave Manager* will be performing a secure launch after ensuring the integrity of *Enclave Manager* on other nodes if the enclave is to be scaled across those nodes. Protocols like CXL provide link-level protection mechanisms which we can rely on to consider interconnect fabric

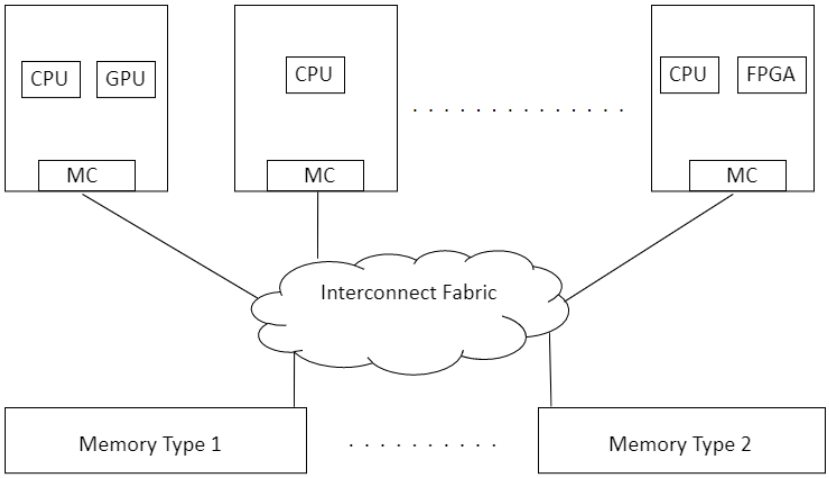


Figure 7.1: An example of a disaggregated memory system (MC: memory controller).

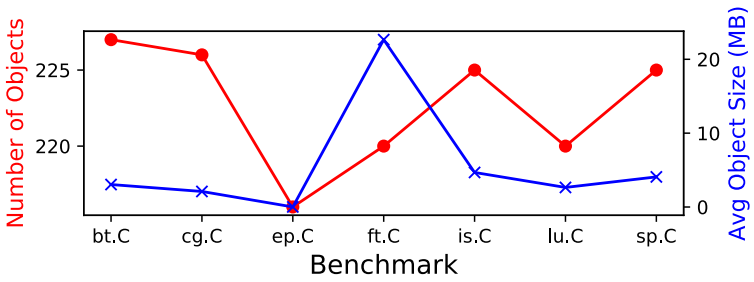
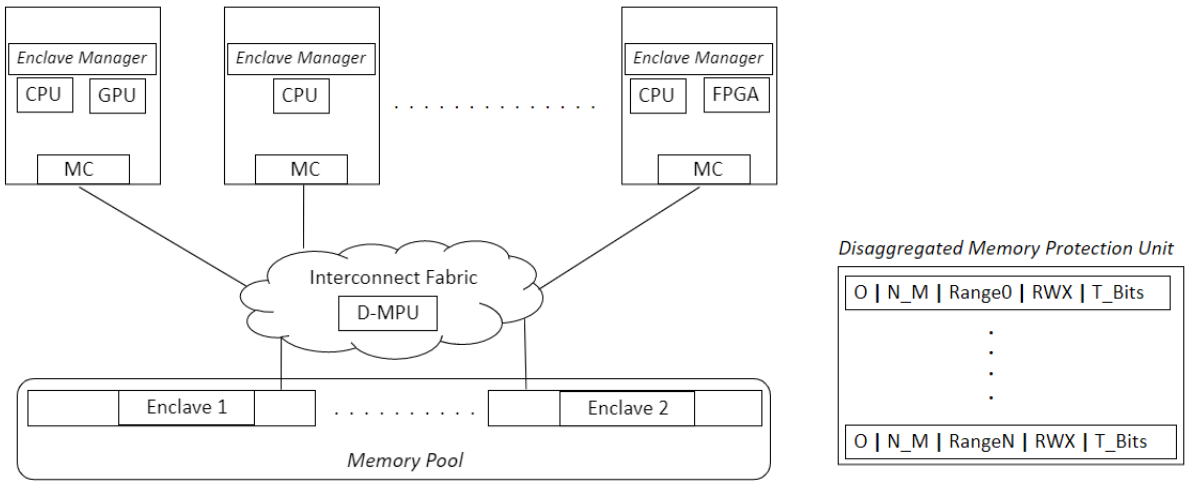


Figure 7.2: Memory allocations of NPB.

within trust boundary.

Overview of D-DESC (Disaggregated data enclaves for scientific computing):

We note that HPC workloads are characterized by a few and large memory allocations. For example, Figure 7.2 shows that the HPC applications like NPB, have a limited number of large allocations. The maximum number of allocations/objects created by *ft.C* is around 225. And the average object size is multiple MBs for all the benchmarks. Similar behavior has been observed by Ji et al. as well [151]. This observation motivates the use of range based checks and translation mechanism (our proposal is independent of the actual translation mechanism used). A high level overview of D-DESC is shown in Figure 7.3. A disaggregated memory protection unit (D-MPU) is added to the interconnect fabric, which is managed by the *Enclave Managers* on all nodes. Figure 7.3b provides details of D-MPU. Each entry in D-MPU is supposed to provide protection to a single object (memory alloca-



(a) D-DESC (b) Memory Protection

Figure 7.3: High level overview of Disaggregated Data Enclaves for Scientific Computing (D-DESC)

tion) of any enclave. 'O' refers to the owner of this object (or in other words the node where the enclave start to execute). Only M-mode software of 'O' node is allowed to change the range entry in D-MPU. An object might be accessible by multiple nodes, when an enclave has scaled across multiple nodes. To take care of that, each entry in D-MPU provides a node map 'N_M'. 'RWX' permissions define what kind of permissions a particular object has (read/write/execute). It should be pointed out that this D-MPU will be in addition to the node level memory protection checks performed by MPU inside each node. Since, the 'O' node of any memory range will be the only entity allowed to update a memory range or the relevant entry in D-MPU, we can avoid expensive synchronization across nodes during context switches. Application ID or enclave ID can also be made a part of the D-MPU entry. This allows to associate a particular memory object with an enclave irrespective of the nodes that enclave is executing on.

In summary, *D-DESC* will be relying on two main principles: 1) tracking protection metadata per object of any enclave, and 2) tracking the object creation and the nodes which might be able to access an object through *Sycall Inspector* (which is a part of *Enclave Manager* on every node). Using these principles, we can implement the security properties discussed above.

Chapter 8

Conclusion

In summary, this thesis has explored the evolving landscape of high-performance computing (HPC) in the context of increasingly data-centric demands and security concerns. The traditional focus of HPC on modeling and simulation has given way to a new paradigm where large and sensitive datasets are integral to scientific computing.

Our investigation began by evaluating the applicability of commercial hardware-based trusted execution environments (TEEs) in the context of secure scientific computing within HPC centers. The comprehensive performance analysis, encompassing diverse HPC benchmarks, revealed that while these TEEs offer some assurances for data and code confidentiality and integrity, they are not an appropriate fit for the unique requirements of HPC. This is primarily due to either unacceptable performance overheads, substantial application modifications, or incomplete threat models.

To address these limitations, we introduced a novel enclave design, *DESC*, which permits secure data processing while relying on a primarily untrusted operating system for resource management. This innovative approach maintains data confidentiality and integrity without necessitating extensive application modifications.

Looking ahead, the future of HPC systems is anticipated to feature disaggregated architectures, decoupling compute and memory resources extensively. In response, we also investigated the extension of our baseline TEE design (*DESC*) to accommodate secure enclaves spanning multiple processing elements.

In conclusion, this thesis contributed to the field of secure scientific computing in HPC

environments. By addressing the challenges of data confidentiality and integrity while minimizing performance overhead and application modifications, our work paves the way for more robust and efficient solutions.

Appendix: Summary of Comparative Analysis of TEEs

Table V. Taxonomy of different TEE features. Each column shows the status of a particular TEE property and the mechanism to achieve that property is shown in parentheses. Blank entries indicate that the information on that property was not available.

TEE	Data Conf.	Data Integrity	Code Integrity	Code Conf.	Auth. Launch	Attest.	Custom.	Isolation	Software Attacks[1] Protected	Hardware Attacks[2] Protected	TCB	Level[3]	Changes (HW,SW)	IO Handl.[4]	Use Cases
Autarky [57]	✓	✓	✓	✗	✗	✓	✗	✗	✗	physical attacks	CPU, OS Driver, LibOS	process	(✓, ✓)	clear	desktop & cloud
AWS Nitro [47]	✓(P, VM)	✓(P, VM)	✓(P, VM)	✓(P, VM)	✓(E)	✓(E)	✗	✓(VM)	other VMs	✗	VM, OS, Nitro HV	VM	(✗, ✓)	vsock	cloud VMs
AEGIS [36]	✓(PTR, E)	✓(IT)		✓(PTR)	✓(H)	✓(H)	✓	✓	processes, OS	physical attacks	CPU, OS	processes	(✓, ✓)	clear	desktop & cloud
Bastion [44]	✓(MP)		✓	✓	✓	✓(H)	✗	✓(IA, MP)	processes, OS	physical attacks	CPU, HV, gOS	VM	(✓, ✓)		cloud
CURE [40]	✓(IA)	✓(IA)	✓(IA)	✓(IA)		✓	✓	system bus arbiter (IA)	processes, OS	IO attacks, physical attacks	config.	process (user & kernel space)	(✓, ✓)	✓(enclave to peripheral binding)	variable
Elasticlave [41]	✓(MP)	✓		✓		✓	✓	✓(MP)	processes, OS	physical attacks	CPU, SM, RT	process	(✗, ✓)	secure	variable
ERTOS [182]	✓(MP)	✓(MP)	✓(MP)	✓(MP)			✗	✓(MP)			ERTOS module of FreeRTOS	tasks	(✗, ✓)		embedded systems
Graviton [37]	✓(E, I)	✓(MAC)	✓(MAC)	✓(E)	✓	✓	✗	✓(P, IA)	OS, HV, processes		GPU, on-package memory	GPU kernels	(✓, ✓)		GPU computing
HECTORV [46]	✓(MP)	✓			✓	✗	✗	✓(SP, IA, MP)	processes	other peripherals on SoC	HW	process	(✓, ✗)	peripheral binding	het. systems

HETEE [183]	✓(E)	✓(E)	✓(E)	✓(E)	✓	✓	✗	✓	processes, OS		hardware security controller, PCIE fabric	multi-node (computing elements)	(✗, ✗)		rack-scale computing
HIX [54]	✓(E)	✓	✓	✓		✓	✗	✓	OS, processes		GPU	CPU-GPU applications	PCIE root complex & MMU, GPU driver	secure	GPU (heterogeneous computing)
Iso-X [184]	✓(P)			✗		✓	✗	✓			HW only	process			
IceClave [185]	✓(E)	✓(IT)		✓(E)			✗	✓	processes	physical attacks	embedded processor (in SSD controller)	offloaded applications			in-storage computing (flash based SSDs)
Komodo [88]			✓	✓		✓	✗	✓	OS, processes	physical attacks					
KeyStone [1]	✓(MP)	✓(MP)	✓(H)	✓(MP)	✓(E)	✓	✓	✓(MP)	OS, processes		SM, CPU, runtime	process (U+S mode)	(✗, ✓)	✗	variable
ARM Realms [39]	✓(P)			✗		✗		✓(cache access checks)				process			embedded, mobile
Sanctum [25]	✓				✓	✓		✓	processes, OS	✗		process			
Sancus [48]	✓			✗	✓	✓					HW only				embedded
SecureBlue++ [186]	✓			✓		✗		✓			HW only				
ShEF [53]	✓(E, MAC)	✓(E, MAC)	✓(E, MAC)	✓(E, MAC)	✓(E)	✓(E)	✓	✓	CPU OS, processes		FPGA, shell logic	FPGA bit-stream	(✗, ✓)		cloud FPGAs
SGX [4]	✓	✓	✓	✗	✗	✓	✗	✗	✗	Small desktop Apps.	CPU, SGX driver	process	(✓, ✓)	outside enclave, in clear	desktop

SEV [3]	✓(E)	✗	✗	✓	✓	✓(hashing VM)	✗	VM level (through page table's C bit)	processes, OS	physical attacks	gOS, CPU	VM	(✗, ✗)	clear	cloud
SEV-ES [187]	✓(E)	✗	✗	✓	✓	✓		VM level	processes, OS	physical attacks	gOS, CPU	VM	(✗, ✗)		cloud
SEV-SNP [135]	✓	✓(nested paging)	✓(nested paging)	✓	✓	✓	✗	VM level	processes, OS	physical attacks	gOS, CPU	VM	(✗, ✗)		cloud
Penglai [188]	✓(MP)	✓(MT)	✓	✓			✗		OS, processes		CPU	process			
TDX [52]	✓	✓	✓	✓				VM level	HV, OS, processes			VM			cloud VMs
TDMem [55]	✓(MP, E)	✓(MAC)			hashing of FPGA bitstream		✓(FPGA based)	✓(MP, address translation tables)	memory access attacks from the donor and donee		kernel (donee only), FPGA boards	RDMA disaggregated systems	needs FPGA, ✓(kernel)		cloud disaggregated systems
TrustZone [23]	✓(P)	✗	✗	✓(P)		✗	✗	✓(cache access checks)	non-secure world	✗	CPU, secure OS	process	(✓, ✓)	clear	embedded, mobile
TIMBERV [42]	✓(MP)	✓(MT)	✓	✓		✓(E)	✗	tagged entry points, regions MPU	processes, OS	✗	CPU	process	(✓, ✓)	clear	embedded systems

Note: Conf. : Confidentiality, Auth. : Authenticated, E: Encryption, P: Partitioning, VM: virtual machine, HV : hypervisor, IT: integrity tree, H : hashing, MP : memory protection checks, IA : id assignment, gOS : guest OS, MAC : message authentication code, SP : secure processor, Custom. : Customizability, RoT : root of trust, MT : Memory Tagging

[1]Software attacks have software and [2]hardware attacks have hardware as the attack surface. [3]Level is the granularity/level at which protection is provided.

Other Notes: These TEEs generally do not consider side channels. Threat of side channels depend on the data sensitivity and leakage rate.

SGX provides strong protection against integrity attacks. SEV-SNP provides some gaurantees against inegrity attacks. [4]I/O includes GPUs, accelerators and FPGAs as well.

Appendix: A Survey of Trusted Execution Environments

This appendix briefly describes some details of some of the representative examples of TEEs from different classes.

Industrial TEEs

Intel's Trusted Execution Environments

Intel SGX [4] is one of the earliest industrial TEE solutions. SGX creates usermode enclaves and assumes an untrusted operating system. The programming model of SGX requires a user application to be divided into two segments, untrusted and trusted (enclave) which cannot directly communicate and interact. Only the trusted part is allowed to access confidential data residing in encrypted form in a memory region called Enclave Page Cache (EPC). The size of the EPC was intially limited to 128MB, which increased to 256MB in later Intel machines with SGX support. In case of SGX, the MEE (memory encryption engine) which sits besides the memory controller on the CPU package is responsible for permission checks for EPC accesses, provision of data confidentiality by encrypting the data when it leaves the CPU package to reside EPC and performs integrity tree operations on the data stored in the EPC. Both parts of an SGX application communicate through an interface of in/out calls (ecall/ocall). ecall and ocall perform a secure context switch which includes: enabling/disabling of tracing mechanisms, permission checks for enclave memory, validation of enclave control structures and backing up/reloading of registers that represent untrusted execution context. Similarly, enclave code cannot use normal system calls directly, rather the control needs to be transferred to the non-secure part of the application first using ocall. SGX requires application changes and/or recompilation. SGX also provides integrity guarantees through the use of integrity trees consisting of counters to keep track of a particular version of the page.

Intel has recently moved towards the encryption and virtual machine-based trusted execution environments like AMD's SEV with total memory encryption (TME) and multi-key total memory encryption (MKTME) technologies [124, 125].

AMD's Trusted Execution Environments

AMD SEV [3] provides transparent encryption of memory used by virtual machines (unique encryption key associated with each isolated guest). As a result, SEV has a larger trusted computing base (TCB), compared to SGX, which includes the guest OS, the hypervisor, and the CPU package. In contrast to SGX, which requires application modifications, SEV does not require changes in an application's code. However, the application needs to be run inside a VM managed by the hypervisor (QEMU). SEV lacks integrity support and does not provide protection against replay attacks.

Later, AMD introduced SEV-ES [187] that adds encryption of guest register state to provide additional protection against VM state related attacks, and SEV-SNP [135] which provides integrity checks. It should be noted that SEV-SNP [135] does not provide integrity guarantees using Merkle tree like data structures (as SGX does). Therefore, it is more scalable and can support larger secure memory sizes.

ARM's Trusted Execution Environments

ARM TrustZone is the first well-known TEE introduced by ARM. TrustZone enables secure execution environments by partitioning a computer system between two worlds (secure world, and normal world). The secure world needs to include a trusted operating system as well. ARM TrustZone finds most of the applications in embedded systems.

Recently, ARM has introduced ARM Confidential compute architecture [189]. Realm management extension (RME) is the hardware extension to enable ARM's confidential computing [190], and builds on ARM TrustZone technology. RME relies on high-privileged software which is responsible for allocating and managing the resources that a 'realm' uses. However, this higher-privileged software cannot access the contents of the realm or affect its execution flow. RME enabled systems include memory encryption and can potentially include integrity as well.

Academic TEEs

Keystone

Keystone [1] is a customizable TEE (can target variable threat models), which tries to decouple isolation mechanism from decision of resource management. The design of Keystone

is inspired from the use of reference monitors in kernel design. It relies on RISC-V's primitives like PMP (physical memory protection) and enables enclaves with both user-level and supervisor-level code. The security functions are managed by a highest privileged mode software (called *Security Monitor*).

Sanctum

Sanctum [25] is another example of academic TEE solutions, which only allows enclaves to run at user-level in contrast to Keystone [1]. Sanctum does not provide protection against phsyical attacks, but protects against side channels by enforcing distinct cache sets per enclave. Sanctum also manages its own page tables, like Keystone.

TEEs for Heterogeneous Systems

Graviton

Graviton [37] is one of the first works to enable isolation of GPU kernel from host OS/hypervisor/and other software. It does so by making hardware changes in the peripheral components of GPUs (specifically GPU command processor and PCIe control engine). Graviton performs memory checks in GPU channels to ensure the memory proection and isolation of multiple regions. Importantly, Graviton's threat model assumes the on-package memory to be a part of the TCB. Graviton requires modifications in the CUDA runtime and GPU driver, and makes it the responsibility of the runtime to authenticate the GPU.

HETEE

HETEE [38] relies on a centralized controller to manage trusted execution on all computing units including GPUs and accelerators and is mainly focused on server rack scale protection. Ir primarily relies on PCIe Express fabric (softwaare defined and flexible topology) to dynamically allocate computing resources for secure and non-senitive computing tasks across multiple servers.

References

[1] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[2] "rv8-bench," https://github.com/michaeljclark/rv8-bench, 2021, [Online; accessed 5-May-2021]. [Online]. Available: https://github.com/michaeljclark/rv8-bench

[3] "Secure Encrypted Virtualization (SEV)," https://github.com/AMDESE/AMDSEV.

[4] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, 2016, https://eprint.iacr.org/2016/086.

[5] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[7] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*. Anaheim, CA, Anaheim, CA, 10-15 April 2005, pp. 41–46.

[8] S. Peisert, "Security in high-performance computing environments," *Communications of the ACM*, vol. 60, no. 9, pp. 72–80, 2017.

[9] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

[10] A. Gupta *et al.*, "The Who, What, Why, and How of High Performance Computing in the Cloud," in *IEEE CloudCom*, 2013.

[11] E. Roloff, M. Diener, A. Carissimi, and P. O. Navaux, "High performance computing in the cloud: Deployment, performance and cost efficiency," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 371–378.

[12] M. Russinovich, M. Costa, C. Fournet, D. Chisnall, A. Delignat-Lavaud, S. Clebsch, K. Vaswani, and V. Bhatia, "Toward confidential cloud computing: Extending hardware-enforced cryptographic protection to data while in use," *Queue*, vol. 19, no. 1, pp. 49–76, 2021.

[13] C. C. Consortium, "A Technical Analysis of Confidential Computing v1.1," https://confidentialcomputing.io/white-papers-reports/, 2021.

[14] ——, "Confidential Computing Consortium Scope," 2021, retrieved August 5, 2021 from https://github.com/confidential-computing/governance/blob/master/scoping.md.

[15] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford University, 2009.

[16] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 1986, pp. 162–167.

[17] C. C. Consortium, "Confidential Computing: Hardware-Based Trusted Execution for Applications and Data," https://confidentialcomputing.io/white-papers-reports/, 2021.

[18] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Processing Queries on an Encrypted Database," *Communications of the ACM*, vol. 55, no. 9, pp. 103–111, 2012.

[19] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A Strongly Encrypted Database System," *IACR Cryptology ePrint Archive*, vol. 2016, p. 591, 2016.

[20] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "SoK: Cryptographically Protected Database Search," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, San Jose, CA, 2017.

[21] A. J. Titus, A. Flower, P. Hagerty, P. Gamble, C. Lewis, T. Stavish, K. P. OConnell, G. Shipley, and S. M. Rogers, "SIG-DB: leveraging homomorphic encryption to Securely Interrogate privately held Genomic DataBases," *arXiv preprint arXiv:1803.09565*, 2018.

[22] C. Dwork, "Differential Privacy," in *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, part II (ICALP)*, ser. Lecture Notes in Computer Science, vol. 4052, July 2006, pp. 1–12.

[23] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security," *Information Quarterly*, pp. 18–24, 2004.

[24] D. Kaplan, J. Powell, and T. Woller, "AMD MEMORY ENCRYPTION," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[25] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[26] A. Akram, V. Akella, S. Peisert, and J. Lowe-Power, "Sok: Limitations of confidential computing via tees for high-performance compute systems," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 121–132.

[27] S. Peisert, "Trustworthy scientific computing," *Communications of the ACM*, vol. 64, no. 5, pp. 18–21, 2021.

[28] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Computing Surveys (CSUR)*, vol. 8, no. 4, pp. 409–445, 1976.

[29] P. J. Denning, "Virtual memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

[30] R. M. Davis, "Evolution of computers and computing," *Science*, vol. 195, no. 4283, pp. 1096–1102, 1977.

[31] P. B. Hansen, "The evolution of operating systems," in *Classic operating systems: from batch processing to distributed systems*. Springer, 2001, pp. 1–34.

[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr, "Exokernel: An operating system architecture for application-level resource management," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.

[33] S. Biggs, D. Lee, and G. Heiser, "The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, 2018, pp. 1–7.

[34] N. Nassif, A. O. Munch, C. L. Molnar, G. Pasdast, S. V. Lyer, Z. Yang, O. Mendoza, M. Huddart, S. Venkataraman, S. Kandula *et al.*, "Sapphire rapids: The next-generation intel xeon scalable processor," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 44–46.

[35] D. D. Sharma, "Compute express link," *CXL Consortium White Paper.[Online]. Available: https://docs. wixstatic. com/ugd/0c1418 d9878707bbb7427786b70c3c91d5fbd1. pdf*, 2019.

[36] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2003, pp. 357–368.

[37] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 681–696.

[38] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp.

991–1006. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00054

[39] "Realm Management Extension," https://developer.arm.com/documentation/den0126/latest.

[40] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with {CUstomizable} and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1073–1090.

[41] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[42] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v." in *NDSS*, 2019.

[43] P. S.-C. Ku, "IOPMP Updates Protection Of IOPMP Andes Technology," 2021, rISC-V Summit.

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

[45] B. Wheeler, "Sifive secures risc-v," *Microprocessor report*, 2019.

[46] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 187–199.

[47] "AWS Nitro Enclaves," https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[48] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 479–498.

[49] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: Tiny trust anchor for tiny devices," in *DAC*, 2015, pp. 1–6.

[50] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[51] M. Schneider, A. Dhar, I. Puddu, K. Kostiainen, and S. Capkun, "Pie: A platform-wide tee," 2021.

[52] *Intel Trust Domain Extensions (Intel TDX).* [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[53] M. Zhao, M. Gao, and C. Kozyrakis, "Shef: shielded enclaves for cloud fpgas," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1070–1085.

[54] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[55] T. Heo, S. Kang, S. Lee, S. Hwang, and J. Huh, "Hardware-assisted trusted memory disaggregation for secure far memory," *arXiv preprint arXiv:2108.11507*, 2021.

[56] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 317–328.

[57] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[58] J. R. Sanchez Vicarte, B. Schreiber, R. Paccagnella, and C. W. Fletcher, "Game of threads: Enabling asynchronous poisoning attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 35–52.

[59] A. Akram, A. Giannakou, V. Akella, J. Lowe-Power, and S. Peisert, "Performance analysis of scientific computing workloads on general purpose tees," in *Proceedings of the 35th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2021.

[60] X. Ding, P. B. Gibbons, and M. A. Kozuch, "A Hidden Cost of Virtualization when Scaling Multicore Applications," in *5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[61] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 743–758, 2014.

[62] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares, "Mind the gap: Reconnecting architecture and os research." in *HotOS*, 2011.

[63] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1852–1867.

[64] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[65] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[66] "RISC-V Instruction Set Manual," 2022, https://github.com/riscv/riscv-isa-manual.

[67] *AMD SEV*. [Online]. Available: https://docs.openstack.org/nova/latest/admin/sev.html

[68] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[69] H. Li, W. Huang, M. Ren, H. Lu, Z. Ning, H. Cui, and F. Zhang, "A novel memory management for risc-v enclaves," 2021.

[70] L. Wilke *et al.*, "SEVurity: No Security Without Integrity - Breaking Integrity-Free Memory Encryption with Minimal Assumptions," in *IEEE S & P*, may 2020, pp. 1431–1444.

[71] Z.-H. Du *et al.*, "Secure Encrypted Virtualization is Unsecure," *arXiv preprint arXiv:1712.05090*, 2017.

[72] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1257–1272.

[73] M. Orenbach and M. Silberstein, "Enclaves as accelerators: learning lessons from gpu computing for designing efficient runtimes for enclaves."

[74] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: Integrating a file system with gpus," in *ASPLOS*, 2013.

[75] S. Shahar, S. Bergman, and M. Silberstein, "Activepointers: a case for software address translation on gpus," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 596–608, 2016.

[76] B. Laurie, "How To Ruin A Perfectly Good Container," https://medium.com/@benlaurie_18378/how-to-ruin-a-perfectly-good-container-d33250fca595, 2029.

[77] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 470–481.

[78] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[79] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An {Off-Chip} attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[80] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC)*. usenix.org, 2017.

[81] S. Arnautov *et al.*, "SCONE: Secure Linux Containers with Intel SGX," in *USENIX OSDI*, 2016, pp. 689–703.

[82] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.

[83] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.

[84] A. Akram, V. Akella, S. Peisert, and J. Lowe-Power, "Enabling design space exploration for risc-v secure compute environments," 2021.

[85] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "Ts-perf: General performance measurement of trusted execution environment and rich execution environment on intel sgx, arm trustzone, and risc-v keystone," *IEEE Access*, vol. 9, pp. 133 520–133 530, 2021.

[86] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 201–213.

[87] M. Bailleu, D. Dragoti, P. Bhatotia, and C. Fetzer, "Tee-perf: A profiler for trusted execution environments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 414–421.

[88] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 287–305.

[89] K. Cheang, C. Rasmussen, D. Lee, D. W. Kohlbrenner, K. Asanovic, and S. A. Seshia, "Verifying risc-v physical memory protection," in *SECRISC-V*, 2020.

[90] A. Akram, A. Giannakou, V. Akella, J. Lowe-Power, and S. Peisert, "Performance analysis of scientific computing workloads on trusted execution environments," *arXiv preprint arXiv:2010.13216*, 2020.

[91] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Computing Surveys*, vol. 51, no. 6, Jan. 2019.

[92] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, Mar. 2018, pp. 665–678.

[93] O. Weisse *et al.*, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *ACM/IEEE ISCA*, 2017.

[94] https://developer.amd.com/sev/.

[95] D. H. Bailey *et al.*, "THE NAS PARALLEL BENCHMARKS," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, 1991.

[96] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, 2015.

[97] A. J. Kunen *et al.*, "KRIPKE - A Massively Parallel Transport Mini-App," LLNL, Livermore, CA, Tech. Rep., 2015.

[98] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[99] G. Ke *et al.*, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *NIPS*, 2017, pp. 3146–3154.

[100] C. Chan *et al.*, "Mobiliti: Scalable Transportation Simulation Using High-Performance Parallel Computing," in *ITSC*, 2018, pp. 634–641.

[101] S. F. Altschul *et al.*, "Basic Local Alignment Search Tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[102] AMD, "Powering the Exascale Era," 2020, accessed April 20, 2020 from https://www.amd.com/en/products/exascale-era.

[103] "Expanding Google Cloud's Confidential Computing portfolio," https://cloud.google.com/blog/products/identity-security/expanding-google-clouds-confidential-computing-portfolio.

[104] "Kata Containers," Available: https://katacontainers.io/.

[105] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 26–35.

[106] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose, "The SEVer-ESt Of Them All: Inference Attacks Against Secure Virtual Enclaves," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 73–85.

[107] M. Morbitzer *et al.*, "Severed: Subverting AMD's Virtual Machine Encryption," in *EuroSec*, 2018, pp. 1–6.

[108] F. Hetzelt and R. Buhren, "Security Analysis of Encrypted Virtual Machines," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2017, pp. 129–142.

[109] P. Rogaway, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," in *ASIACRYPT*, 2004.

[110] D. Kaplan, "AMD x86 Memory Encryption Technologies," in *Linux Security Summit*, 2017.

[111] U. Kamio and Y. Kinoshita, "KVM/QEMU tuning of NUMA and Memory," in *Open Infrastructure Summit*, 2017.

[112] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," in *2010 IEEE second international conference on cloud computing technology and science*, 2010, pp. 159–168.

[113] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of Virtualization Technologies for High Performance Computing Environments," in *IEEE 4th International Conference on Cloud Computing*, 2011, pp. 9–16.

[114] S.-H. Ha, D. Venzano, P. Brown, and P. Michiardi, "On the Impact of Virtualization on the I/O Performance of Analytic Workloads," in *2nd IEEE International Conference on Cloud Computing Technologies and Applications (CloudTech)*, 2016, pp. 31–38.

[115] A. Kudryavtsev, V. Koshelev, and A. Avetisyan, "Modern HPC Cluster Virtualization Using KVM and Palacios," in *2012 19th IEEE International Conference on High Performance Computing*, 2012, pp. 1–9.

[116] J. Gandhi *et al.*, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *47th IEEE/ACM MICRO*, 2014.

[117] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[118] "Asylo," Available: https://asylo.dev/.

[119] "GET STARTED WITH THE SDK," Available: https://software.intel.com/en-us/sgx/sdk.

[120] "Open Enclave SDK," Available: https://openenclave.io/sdk/, accessed: 2019-9-3.

[121] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[122] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler, "sRDMA–Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 691–704.

[123] "Osu micro-benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/.

[124] "Intel Promises Full Memory Encryption in Upcoming CPU," https://arstechnica.com/gadgets/2020/02/intel-promises-full-memory-encryption-in-upcoming-cpus/.

[125] "Intel® Architecture Memory Encryption Technologies Specification," Intel Corporation, Tech. Rep. Rev. 1.2, April 2019.

[126] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[127] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[128] H. G. Martin, T. Radivojevic, J. Zucker, K. Bouchard, J. Sustarich, S. Peisert, D. Arnold, N. Hillson, G. Babnigg, J. M. Marti, C. J. Mungall, G. T. Beckham, L. Waldburger, J. Carothers, S. Sundaram, D. Agarwal, B. A. Simmons, T. Backman, D. Banerjee, D. Tanjore, L. Ramakrishnan, and A. Singh, "Perspectives for Self-Driving Labs in Synthetic Biology," *Current Opinion in Biotechnology*, vol. 79, p. 102881, 2023.

[129] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 237–248, 2013.

[130] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from hypervisor and host operating system exploits," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[131] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 121–135.

[132] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, Nov. 2016, pp. 689–703.

[133] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[134] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[135] "AMD SEV-SNP," https://developer.amd.com/sev/.

[136] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 272–283.

[137] S. Mofrad *et al.*, "A Comparison Study of Intel SGX and AMD Memory Encryption Technology," in *HASP*. ACM, Jun. 2018, p. 9.

[138] W. Liu, H. Chen, X. Wang, Z. Li, D. Zhang, W. Wang, and H. Tang, "Understanding tee containers, easy to use? hard to trust," *arXiv preprint arXiv:2109.01923*, 2021.

[139] S. Kaminsky, "Secure multi-threading in keystone enclaves," 2021.

[140] P. Gaddamadugu, "Formally verifying trusted execution environments with uclid5," Ph.D. dissertation, MA thesis. EECS Department, University of California, Berkeley, 2021.

[141] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[142] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[143] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[144] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 390–397.

[145] S. Skorobogatov, "Tamper resistance and physical attacks," *Summer School on Cryptographic Hardware, Side-Channel and Fault Attacks*, 2006.

[146] D. R. Ports and T. Garfinkel, "Towards application security on untrusted operating systems." in *HotSec*, 2008.

[147] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[148] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[149] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *NDSS*, 2021.

[150] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 304–316.

[151] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S. S. Vazhkudai, W. Xue, and D. Sanchez, "Understanding object-level memory access patterns across the spectrum," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[152] M. Nazarewicz, "Contiguous memory allocator," in *Proc. LinuxCon Eur.*, 2012.

[153] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 637–650.

[154] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 515–528.

[155] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 339–350.

[156] S. Yuan, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Analyzing secure memory architecture for gpus," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 59–69.

[157] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume 2: Privileged architecture version 1.7," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[158] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 2–13, 2008.

[159] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 488–501.

[160] S. Luan, "Exploit two xen hypervisor vulnerabilities," *Alibaba Cloud*, 2016.

[161] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2013, pp. 265–278.

[162] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer, 2016, pp. 440–457.

[163] A. Akram, V. Akella, S. Peisert, and J. Lowe-Power, "Enabling design space exploration for risc-v secure compute environments," in *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*, 2021.

[164] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[165] J. Tullos, S. Graham, and P. Patel, "Applied analytical model for latency evaluation of risc-v security monitor," in *16th International Conference on Cyber Warfare and Security*. Academic Conferences Limited, 2021, p. 354.

[166] B. R. Bruce, A. Akram, H. Nguyen, K. Roarty, M. Samani, M. Friborz, T. Reddy, M. D. Sinclair, and J. Lowe-Power, "Enabling reproducible and agile full-system simulation," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 183–193.

[167] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic Application Partitioning for Intel SGX," in *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, Jul. 2017, pp. 285–298.

[168] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable Counters: Enabling Compact Integrity Trees for Low-Overhead Secure Memories," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[169] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *Proceedings of the 12th ACM European Conference on Computer Systems*, Belgrade, Serbia, Apr. 2017, pp. 238–253.

[170] A. T. Gjerdrum *et al.*, "Performance of Trusted Computing in Cloud Infrastructures with Intel SGX," in *CLOSER*, 2017.

[171] I. T. Association *et al.*, "Infiniband architecture specification release 1.2," *http://www.infinibandta. org*, 2000.

[172] L. Zhao and D. Lie, "Is hardware more secure than software?" *IEEE Security & Privacy*, vol. 18, no. 5, pp. 8–17, 2020.

[173] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[174] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 319–327, 1994.

[175] E. A. Feustel, "The rice research computer: a tagged architecture," in *Proceedings of the May 16-18, 1972, spring joint computer conference*, 1971, pp. 369–377.

[176] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 457–468.

[177] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 183–190.

[178] J. Shalf, G. Michelogiannakis, B. Austin, T. Groves, M. Ghobadi, L. Dennison, T. Gray, Y. Shen, M. Y. Teh, M. Glick *et al.*, "Photonic memory disaggregation in datacenters," in *Photonics in Switching and Computing*. Optical Society of America, 2020, pp. PsW1F–5.

[179] A. Awad, S. Hammond, and C. Hughes, "Hw/sw codesign for disaggregated memory architectures: Opportunities and challenges," in *ASCR Workshop on Reimagining Codesign*, 2021.

[180] G. Casey, S. Team *et al.*, "Gen-z an overview and use case s," in *Open Fabric Allience, 13th annual workshop*, 2017.

[181] V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Deact: Architecture-aware virtual memory support for fabric attached memory systems," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 453–466.

[182] A. Thomas, S. Kaminsky, D. Lee, D. Song, and K. Asanovic, "Ertos: Enclaves in real-time operating systems."

[183] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, L. Zhao, F. Yuan, P. Li, Z. Wang, B. Zhao *et al.*, "Enabling privacy-preserving, compute-and data-intensive computing using heterogeneous trusted execution environment," *arXiv preprint arXiv:1904.04782*.

[184] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *47th IEEE MICRO*, 2014.

[185] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M. J. Kang, H. J. Lim, B. Jacob, and J. Huang, "Iceclave: A trusted execution environment for in-storage computing," in *54th Annual IEEE/ACM MICRO*, 2021, pp. 199–211.

[186] R. Boivie and P. Williams, "Secureblue++: Cpu support for secure execution," *IBM, IBM Research Division, RC25287 (WAT1205-070)*, pp. 1–9, 2012.

[187] "Protecting Register State with AMD SEV-ES," https://developer.amd.com/sev/.

[188] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the penglai enclave," in *15th USENIX OSDI*, 2021, pp. 275–294.

[189] "Arm Confidential Compute Architecture," https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture, 2021.

[190] "Realm Management Extension," https://developer.arm.com/documentation/den0126/latest.