THE UNIVERSITY *of* EDINBURGH

Secure Storage Systems for Untrusted Cloud Environments

Maurice BAILLEU

Doctor of Philosophy

Adviser: Prof. Pramod BHATOTIA

Assistant advisor: Prof. Vijay NAGARAJAN

Committee members: Prof. Peter PIETZUCH, Prof. Antonio BARBALACE

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2023

Abstract

The cloud has become established for applications that need to be scalable and highly available. However, moving data to data centers owned and operated by a third party, i.e., the cloud provider, raises security concerns because a cloud provider could easily access and manipulate the data or program flow, preventing the cloud from being used for certain applications, like medical or financial.

Hardware vendors are addressing these concerns by developing Trusted Execution Environments (TEEs) that make the CPU state and parts of memory inaccessible from the host software. While TEEs protect the current execution state, they do not provide security guarantees for data which does not fit nor reside in the protected memory area, like network and persistent storage.

In this work, we aim to address TEEs' limitations in three different ways, first we provide the trust of TEEs to persistent storage, second we extend the trust to multiple nodes in a network, and third we propose a compiler-based solution for accessing heterogeneous memory regions. More specifically,

- SPEICHER extends the trust provided by TEEs to persistent storage. SPEICHER implements a key-value interface. Its design is based on LSM data structures, but extends them to provide confidentiality, integrity, and freshness for the stored data. Thus, SPEICHER can prove to the client that the data has not been tampered with by an attacker.

- AVOCADO is a distributed in-memory key-value store (KVS) that extends the trust that TEEs provide across the network to multiple nodes, allowing KVSs to scale beyond the boundaries of a single node. On each node, AVOCADO carefully divides data between trusted memory and untrusted host memory, to maximize the amount of data that can be stored on each node. AVOCADO leverages the fact that we can model network attacks as crash-faults to trust other nodes with a hardened ABD replication protocol.

- TOAST is based on the observation that modern high-performance systems often use several different heterogeneous memory regions that are not easily distinguishable by the programmer. The number of regions is increased by the fact that TEEs divide memory into trusted and untrusted regions. TOAST is a compiler-based approach to unify access to different heterogeneous memory regions and provides programmability and portability. TOAST uses a load/store interface to abstract most library interfaces for different memory regions.

Lay summary

This thesis focuses on the security and privacy of data stored in the cloud. As more services transition to cloud infrastructure for improved reliability, scalability, and maintainability, there are concerns that the cloud provider or an attacker could access and manipulate the data, especially for sensitive services like finance and healthcare. This has resulted in hesitation to adopt cloud services for such applications.

To address these concerns, the thesis explores a technology called Trusted Execution Environments (TEEs). These environments provide a secure memory area and processor state that protect applications from unauthorized access. However, this protection does not extend to storage. The thesis aims to extend trust to the storage.

Three projects are introduced in the thesis to tackle different aspects of secure storage subsystems:

- SPEICHER: This project enhances the security of data stored in individual computer systems. It achieves this by improving the underlying data structures used for storage and encrypting any data that leaves the secure memory area. This ensures the confidentiality and integrity of the stored data. SPEICHER employs specific techniques like flat Merkle trees and an asynchronous trusted counter to prevent potential attacks.

- AVOCADO: This project focuses on creating a trusted distributed key-value store. It utilizes TEEs to prevent situations where different nodes in a distributed system claim different values. This ensures consistent and trusted replication of data, even in the presence of failures. AVOCADO also offers a service that facilitates easy addition and removal of nodes from the network, ensuring scalability and availability.

- TOAST: This project improves programmability and portability in modern systems that employ different types of memory. With the introduction of TEEs, the number of memory regions doubles. TOAST provides a unified interface for accessing these different memory types. It also includes a proxy and protection library that simplify the adoption of new technologies and prevent data leaks.

By utilizing these three projects, it is possible to build a more powerful and specialized trusted storage system that ensures the security and privacy of data in the cloud.

Publications

This thesis is based on the following conference papers.

1. SPEICHER : *Securing LSM-based Key-Value Stores using Shielded Execution* by Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani, In the proceedings of *Usenix FAST 2018* [1]

2. AVOCADO : *A Secure In-Memory Distributed Storage System* by Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia, In the proceedings of *USENIX ATC 2021* [2]

3. TOAST : *Heterogeneous Memory Management* by Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Deepak Garg, and Pramod Bhatotia, under review in *IEEE/ACM CGO 2024*

Other conference papers during my PhD.

4. *TEE-Perf : A Profiler for Trusted Execution Enviroments* by Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzer, In the proceedings of *IEEE/IFIP DSN 2019* [3]

5. TREATY : *Secure Distributed Transactions* by Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia, In the proceedings of *IEEE/IFIP DSN 2022* [4]

Table of Contents

1 Introduction **3**

1.1 Motivation . . . 3
1.2 Problem statement . . . 4
1.3 Approach and contribution . . . 5
1.4 Thesis outline . . . 7

2 Background **9**

2.1 Storage Systems . . . 9
2.1.1 Log Structure Merge Tree . . . 11
2.1.2 Distributed KVS . . . 11
2.1.3 Persistent Memory . . . 13
2.2 Trusted Execution and Memory Safety . . . 13
2.2.1 Trusted Execution Environments . . . 13
2.2.2 Remote attestation . . . 15
2.2.3 MPK . . . 16
2.2.4 Capabilities . . . 17
2.3 Direct I/O Stack . . . 18
2.3.1 DPDK . . . 18
2.3.2 SPDK . . . 19
2.3.3 eRPC . . . 19

3 SPEICHER: A Secure LSM-based KV Store **21**

3.1 Motivation . . . 22
3.1.1 Threat Model . . . 24
3.2 Design . . . 25
3.2.1 Design Challenges . . . 25
3.2.2 System Components . . . 27
3.2.3 Algorithms . . . 33
3.2.4 Optimizations . . . 36

3.3 Implementation . . . 37
3.4 Evaluation . . . 39
3.4.1 Experimental Setup . . . 40
3.4.2 Performance of the Direct I/O Library . . . 40
3.4.3 Impact of the EPC paging on MemTable . . . 42
3.4.4 Throughput and Latency Measurements . . . 42
3.4.5 Performance of the Trusted Counter . . . 45
3.4.6 I/O Amplification . . . 46
3.5 Related Work . . . 47
3.6 Summary . . . 49

4 AVOCADO: A Secure In-Memory Distributed Storage System 51
4.1 Motivation . . . 51
4.2 System Model . . . 54
4.3 Design . . . 55
4.3.1 Network Stack . . . 56
4.3.2 Replication Protocol . . . 58
4.3.3 Configuration and Attestation Service . . . 63
4.3.4 Single-node KVS . . . 64
4.4 Implementation . . . 67
4.4.1 System Components . . . 67
4.4.2 Optimizations . . . 68
4.5 Evaluation . . . 68
4.5.1 Network Stack . . . 69
4.5.2 Replication Protocol . . . 71
4.5.3 Configuration and Attestation Service . . . 74
4.5.4 Single-node KVS . . . 74
4.5.5 Distributed KVS . . . 76
4.5.6 Scalability . . . 78
4.5.7 Number of Keys . . . 79
4.6 Related Work . . . 80
4.7 Summary . . . 81

5 TOAST: Heterogeneous Memory Management 83
5.1 Introduction . . . 83
5.2 Motivation: The 4P Challenges . . . 86
5.3 Overview . . . 88
5.3.1 The TOAST Approach . . . 89

5.3.2 System Model . . . 92
5.3.3 Example Revisited . . . 92
5.4 Design . . . 93
5.4.1 TOAST Compiler . . . 93
5.4.2 TOAST Runtime Library . . . 94
5.4.3 Protection Library . . . 95
5.5 Implementation . . . 96
5.5.1 TOAST Compiler . . . 96
5.5.2 TOAST Runtime Library . . . 96
5.5.3 TOAST Protection Library . . . 97
5.5.4 Allocation Wrapper . . . 98
5.6 TOAST Case-studies . . . 100
5.7 Evaluation . . . 101
5.7.1 Experimental Setup . . . 101
5.7.2 Programmability . . . 102
5.7.3 Performance . . . 102
5.7.4 Portability . . . 105
5.7.5 Protection . . . 106
5.8 Related Work . . . 109
5.9 Summary . . . 111

6 Conclusion and Future Work **113**
6.1 Conclusion . . . 113
6.2 Future Work . . . 114
6.3 Code availability . . . 116

A SPEICHER Algorithms **117**

List of Figures

2.1 Desgin of a LSM based KVS . . . 10
2.2 Overview of a distributed KVS . . . 12
2.3 Overview of TEEs . . . 14
2.4 Overview of MPK . . . 16
2.5 Overview of direct I/O stack . . . 17

3.1 SPEICHER overview . . . 28
3.2 SPEICHER MemTable format based on skip list design . . . 30
3.3 SPEICHER SSTable file format . . . 31
3.4 SPEICHER append-only log file format . . . 32
3.5 Performance of direct I/O library for shielded execution vs native SPDK. 41
3.6 Impact on the random accessing time of EPC paging on the MemTable. . 43
3.7 SPEICHER's performance for different R/W workloads . . . 44
3.8 SPEICHER's performance for different value sizes . . . 45
3.9 SPEICHER's performance for different number of threads . . . 46

4.1 AVOCADO's system overview. . . . 56
4.2 AVOCADO's network stack. . . . 59
4.3 AVOCADO's message format. . . . 59
4.4 Example of Put request in AVOCADO protocol. . . . 62
4.5 AVOCADO's single-node KVS. . . . 65
4.6 Performance comparison of the AVOCADO network stack . . . 70
4.7 AVOCADO's performance for different workloads . . . 72
4.8 AVOCADO's performance for different value sizes . . . 72
4.9 AVOCADO's performance for different number of threads . . . 73
4.10 AVOCADO single-node KVS's performance for different value sizes . . . 75
4.11 AVOCADO single-node KVS's performance for different workloads . . . 76
4.12 Performance overhead of TEE and encryption in AVOCADO . . . 77
4.13 AVOCADO scalability experiment . . . 78

4.14 AVOCADO's performance for different number of distinct keys 79

5.1 Virtual address space layout in TOAST 88
5.2 Overview of TOAST 90
5.3 Overview of TOAST's protection mechanism 95
5.4 TOASTPtr's performance for a replication protocol for different workloads 103
5.5 TOASTPtr's performance for a replication protocol for various value sizes 104
5.6 TOASTPtr's performance for persistent KVS for different workloads . . . 105
5.7 TOAST's protection mechanism overhead on memcpy 106
5.8 TOAST's performance for shared log 107
5.9 TOAST's protection mechanism overhead for KVS 108

Signed declaration

I declare that the thesis has been composed by myself and that the work has not been submitted for any other degree or professional qualification. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and those of the other authors to this work have been explicitly indicated below. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others. The work presented in § 3 was previously published in USENIX FAST 2018 as "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution" by Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia (PhD supervisor), Christof Fetzer, Michio Honda, and Kapil Vaswani. § 4 was submitted to USENIX ATC 2021 with the title "AVOCADO: A Secure In-Memory Distributed Storage System" by the authors Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia (PhD supervisor). Lastly, § 5 is under review in IEEE/ACM CGO 2024 with the name "TOAST: Heterogeneous Memory Management" by Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Deepak Garg, and Pramod Bhatotia (PhD supervisor).

Maurice Bailleu

Chapter 1

Introduction

1.1 Motivation

Cloud computing is becoming increasingly popular for its benefits of scalability, availability, and maintainability. A driving factor for the adoption of cloud infrastructure is the rapid increase of data being collected, stored, and processed by modern services. However, the use of cloud infrastructure also raises privacy and security concerns, as the cloud provider has full access to the hardware and software stack, including the operating system (OS) and hypervisor. This privileged control allows the cloud provider to intercept all data transfers and inspect and control memory, processor states, network traffic and stored data of the services hosted in the cloud. A potential attacker could, therefore, manipulate data stored in the cloud, posing security and privacy risks, which prevents entire groups of services to migrate to the cloud infrastructure, e.g., financial industry and health care.

To address these concerns, hardware vendors have proposed and implemented Trusted Execution Environments (TEEs), such as Intel SGX [5], Intel TDX [6], ARM TrustZone [7], ARM Realms [8], AMD SEV-SNP [9], and RISC-V Keystone [10]. TEEs provide a secure memory region that is protected against access and manipulation by other software, including privileged software like OSs and hypervisors. To prevent CPU data leakage through the register and cache state, TEEs also disallow the operating system from inspecting the CPU state of processes running inside a TEE. Combined with a remote attestation service, this allows for the creation of fully trusted applications.

These TEEs are a promising solution for adopting security- and privacy-critical applications to the cloud, as they allow service providers to protect their data and application state from the cloud provider. As a result, major cloud providers have begun adopting TEEs and offering them to their customers, e.g., Google Cloud [11],

Microsoft Azure [12], and Alibaba Cloud [13]. This has led to increased interest in providing a full system stack for security and privacy-sensitive applications, allowing these applications to take full advantage of the cloud and reducing their dependency on their own data centers.

However, TEEs are designed for local transient processes, making them insufficient to build an entire system stack. Modern services often require to store and retrieve data, that might be stored on a persistent storage device, like SSD or persistent memory (PMEM), or might be distributed over multiple nodes. This gives a potential attacker a vast attack surface, as TEEs do not give any guarantees for data residing outside of the secured memory region. Furthermore, TEEs do not provide any guarantees for the correctness of the code. By leaking a private or encryption key, a simple programming mistake can unrecoverably compromise the entire system. TEEs, further, exacerbate the problem by dividing the system into a trusted and untrusted partition, which can be indiscriminately accessed by the same process.

This raises the challenging question of building a trusted storage system for untrusted cloud environments using TEEs. This thesis identifies three main challenges in addressing this question: *(i)* How to extend the trust provided by TEEs to persistent storage, since TEEs only protect the processor state and memory. *(ii)* How to provide trust in a distributed system, given that TEEs are bound to a single node. *(iii)* How to provide programmability, portability and high performance for heterogeneous memory systems in building a trusted storage system.

1.2 Problem statement

Persistent storage is desirable in cloud storage environments compared to in-memory systems, due to its lower cost per GB of data, its durability, i.e., being able to recover the data after system shutdown, and vertical scaling capabilities, as a single machine can incorporate larger amounts of persistent storage than volatile memory. To leverage the advantages of persistent storage in a trusted storage system based on TEEs, which provides confidentiality, i.e., data can only be accessed by authorized entities, integrity, i.e., unauthorized changes to the data can be detected, and freshness, i.e., stale data can be detected, we have to overcome two challenges: firstly, extending the confidentiality and integrity of TEEs to persistent data structures, secondly, to provide freshness guarantees, building a trusted counter service, which fulfills the performance requirements of modern storage systems.

Distributed storage systems provide scalability and availability, by laying out their data over different nodes. Designing a trusted storage system in distributed

systems poses additional challenges as TEEs are limited to a single node and cannot protect data in transit between the different components. Additionally, conventional Byzantine Fault Tolerance (BFT) protocols are costly and required to ensure secure communication between untrusted nodes. In order to overcome these limitations and design a trusted distributed storage system that provides confidentiality, integrity and fault tolerance, we propose a scaleable attestation mechanism to establish trust between servers and clients. Our solution also enables mutual trust between nodes, which allows for secure data transmission and a more efficient protocol for Byzantine settings that provides confidentiality and fault tolerance with less overhead than conventional BFT protocols.

Modern storage systems use multiple types of memory for improved performance, security, and reliability. These memory types can be found throughout the system, including in network and storage devices. However, accessing these heterogeneous memory types can be difficult due to device-specific implementations and the need for additional actions such as reading and writing to specific memory addresses and cache flushes. This can lead to challenges in portability for programmers who must understand the details of each device. Additionally, TEEs complicate the problem by further dividing the address space into trusted and untrusted regions, making it difficult for programmers to distinguish between memory types and leading to potential errors such as sensitive information leakage or mistakenly persisting temporary data. To address these challenges, we propose a simplified, generic programming model that uses the established load/store interface and includes an error handling mechanism and a protection library to isolate different memory regions.

1.3 Approach and contribution

This thesis aims to establish a trusted storage solution in the untrusted cloud. We introduce three projects that focus on different aspects of storage systems and security and can be combined to secure a storage system which the cloud provider cannot access or manipulate. By addressing the question of how to ensure security for data and storage operations in untrusted cloud environments with hardware-based TEEs, this thesis explores three projects: SPEICHER [1], AVOCADO [2], and TOAST. Each of these projects addresses different challenges of secure storage for the cloud. These are persistence and freshness for SPEICHER; scalability and availability for AVOCADO; programmability and portability for TOAST.

SPEICHER is a log-structured merge tree (LSM) [14] based key-value store (KVS) that establishes trust for persistent storage on a single node. It offers confidentiality, integrity, authenticity, and freshness by taking advantage of the properties of LSM data structures. For instance, the fact that LSM KVS structures store data in multiple levels allows for the implementation of a fast and secure MemTable that partitions data between trusted TEE memory and untrusted host memory. Additionally, the write-only-once nature of LSM KVS enables SPEICHER to guarantee the freshness of persistent data by building a Merkle tree [15] over parts of the KVS, with the root node stored persistently along with a trusted counter value.

SPEICHER also provides a fast asynchronous counter service to overcome the performance limitations of hardware-based trusted counters, which can have latencies of up to 200 ms. Additionally, it implements a user space direct I/O storage library based on SPDK [16], allowing for efficient access to data stored on the persistent storage (e.g., SSD) without the need for expensive world switches between the TEE and the untrusted host.

In our evaluation, SPEICHER showed a reasonable overhead of 15 to 35 $\times$ compared to an unsecured RocksDB [17] version in the YCSB benchmark [18].

AVOCADO extends the trust provided by TEEs to multiple nodes, offering scalability, availability, and performance without providing persistence. It uses the TEE to harden a non-Byzantine fault (BFT) replication protocol, specifically multi-writer-ABD, for use in an untrusted infrastructure. This allows AVOCADO to exclude equivocation, reducing the number of nodes necessary for fault tolerance in an untrusted system from BFT's $3f + 1$ to $2f + 1$.

AVOCADO also provides confidentiality, integrity, authenticity, and freshness for the stored data. It introduces a secure user space network stack based on eRPC [19] and DPDK [20], allowing AVOCADO to bypass the kernel and avoid expensive switches between the trusted environment and untrusted host system. Similarly to SPEICHER, AVOCADO splits in-memory data into trusted and untrusted memory, greatly reducing the amount of required trusted memory.

In our evaluation, AVOCADO outperformed BFT by 4 to 65 $\times$ while using fewer replicas and providing confidentiality in the YCSB benchmark.

TOAST is a compiler-based heterogeneous memory management system that prevents users from unintentionally leaking information. Modern systems often incorporate heterogeneous memory devices, such as NICs, SSDs, or persistent memory, in order to improve performance. These devices expose an interface over

the memory bus by allocating a memory area, allowing direct control from user space processes and reducing syscall overheads. However, this poses programmability and portability issues, as the programmer has to associate a pointer with a specific memory area and use specific libraries to access it. Additionally, the programmer is unable to distinguish between pointers referring to different memory areas, which poses protection challenges.

TEEs further complicate matters by doubling the number of heterogeneous memory regions by dividing the entire system into trusted and untrusted parts. Any information leakage can compromise the security guarantees provided by the system. TOAST addresses all three challenges by providing a unified load-store interface for different types of heterogeneous memory, unified callback-based error handling, and a runtime protection library that prevents information leakages.

The TOAST compiler transforms pointers referencing heterogeneous memory regions, instruments load and store operations as function calls to the library necessary for accessing the memory region, and checks the temporal and spatial validity of the pointer to prevent information leakage.

1.4 Thesis outline

In the following chapters, we will introduce the background (Chapter § 2), and provide an in-depth look at each of the three projects: SPEICHER (Chapter § 3), AVOCADO (Chapter § 4), and TOAST (Chapter § 5). The thesis will conclude in Chapter § 6.

Chapter 2

Background

This chapter serves as an introduction to the foundational concepts upon which the subsequent chapters' three projects are built. The primary goal of the thesis is to develop tools for constructing a trusted storage system. Therefore, this chapter commences with an overview of storage technologies (Section § 2.1) that are employed in the various projects:

SPEICHER utilizes an LSM-based key-value store (KVS) (Subsection § 2.1.1). AVOCADO focuses on a trusted distributed KVS (Subsection § 2.1.2). TOAST examines the management of heterogeneous memory, specifically persistent memory for storage (Subsection § 2.1.3).

All three projects make use of trusted execution environments (TEEs). Consequently, this chapter provides an introduction to TEEs, with a specific emphasis on Intel SGX, as it is the TEE category employed in the three projects (Subsection §2.2.1).

The concept of remote attestation (Subsection § 2.2.2) is discussed, as it guarantees the integrity of the software running on the cloud provider's platform.

Furthermore, we introduce the memory safety mechanisms employed by TOAST (Subsection §2.2.3 and § 2.2.4).

Finally, the chapter concludes with a description of direct I/O technologies (Section § 2.3) that are utilized in all three projects.

2.1 Storage Systems

Due to the increased demand in data storage and processing, multiple storage designs have emerged. This thesis aims to build an hardware assisted secure storage system. As already mentioned in the introduction (see § 1.2) and also discussed multiple times in this thesis (e.g., see § 3, § 3.1, § 3.2.1, § 3.6, § 4, § 4.1, § 4.3, § 4.7, and § 6.1), TEEs

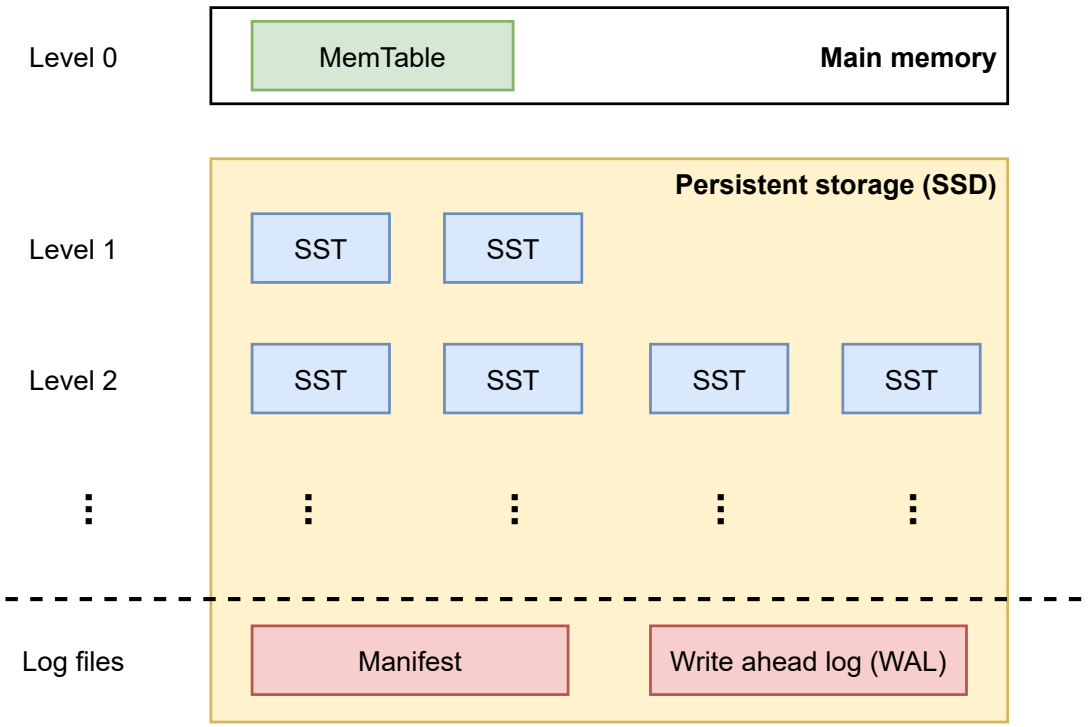

Figure 2.1: Example configuration of a log-structured merge tree (LSM) based KVS. Puts are added to the lowest level, i.e., level 0, which is hold in memory for faster accesses. When a level reaches its maximum size, a compaction event is triggered which merges the level with the next higher level. In this example all levels except level 0 are stored on the persisent storage. All `puts` also get appended to the write ahead log (WAL), to be able to recover them, in case of a system reboot. The manifest hold the current state of the LSM-KVS.

trust guarantees do not extend to the persistent storage or over the network. Modern storage systems, however, rely on these technologies to provide scalability (vertical and horizontal) and availability.

This section provides an overview of storage technologies used in the three projects that comprise the work of this thesis. Log-structure merge tree (see § 2.1.1) for persistence is found in SPEICHER (see § 3). AVOCADO (see § 4) is a replicated key value store, based on common distributed key value store technologies (see § 2.1.2). TOAST (see § 5) provides a heterogeneous memory abstraction to improve programmability. We implemented TOAST abstraction on top of persistent memory (see § 2.1.3) and direct I/O libraries, like DPDK, SPDK, and eRPC (see § 2.3).

2.1.1 Log Structure Merge Tree

Log structured merge tree (LSM) based key value stores (KVSs), such as LevelDB [21] and RocksDB [17] have gained popularity due to their superior write throughput. LSM-based KVSs organize the data using three constructs: *MemTables*, *static sorted tables* (SSTables or simply SSTs), and log files. Figure 2.1 shows an overview of a LSM-based KVS. In the following description of the function of an LSM-based KVS we focus on RocksDB, as SPEICHER is based on it. However, other LSM-based KVSs perform equivalent operations.

RocksDB inserts `put` requests to a memory-resident MemTable that is organized as a skip list [22]. For crash recovery, these `puts` are also sequentially logged to the *write-ahead-log* (WAL) file backed by persistent storage medium with checksums. When the MemTable fills up, it is moved to an SSTable file backed by an SSD or HDD in a batch to ensure sequential device access (this thus can cause scanning the skip list).

The SSTable files are grouped into levels with increasing size (typically $10\times$). The process of moving data to the next level is called *compaction*, which ensures the SSTables to be sorted by keys, including the ones being merged from the previous level. Since SSTables are immutable, compaction always creates new SSTables on the persistent storage medium. Any state changes in the entire storage system, such as creation and deletion of SSTable and WAL files, are recorded to the *Manifest*, which is a transactional and persistent log.

On a `get` request, RocksDB first searches the MemTable for the key, then searches the SSTables from the lowest level in turn; at each level, it binary-searches the corresponding SSTable. Using this primitive, it is trivial to process range and iterator queries, where the latter only differs in the client interface. RocksDB maintains an index table with a Bloom filter attached to each SSTable in order to avoid searching unnecessary SSTables.

While restarting, RocksDB establishes the latest state in a `restore` operation. To this end, the Manifest and the WAL are read and replayed.

2.1.2 Distributed KVS

Distributed key-value stores (KVSs) are designed to scale horizontally across multiple machines, allowing them to handle a large amount of data and a high volume of requests. In distributed KVSs, partitioning and replication are often used together to ensure scalable, available, and transparent data distribution. Partitioning involves dividing the data space into smaller chunks, called partitions, each of which contains a portion of the overall data set. Replication involves creating multiple

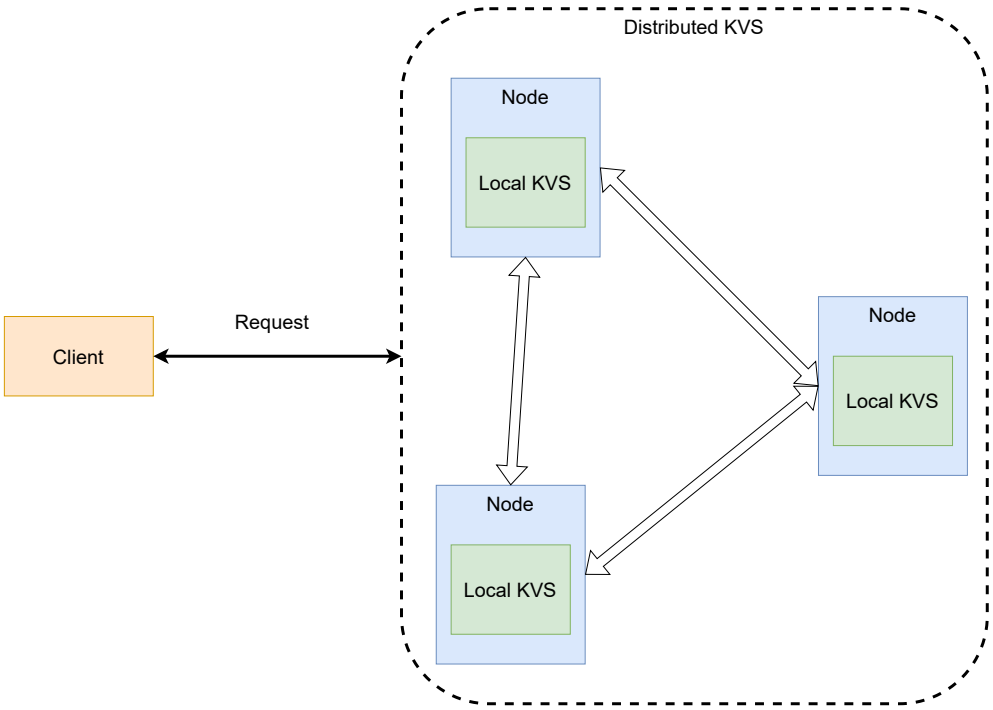

Figure 2.2: Distributed KVS act as a single system towards the client. A distributed KVS provides scalability and availability by partitioning and replicating the data over the nodes in the system.

copies, or replicas, of each partition and distributing them among the members of the cluster. Each member is typically responsible for a single replica of a partition.

Replication techniques in distributed databases can vary in how data is accessed and updated, which impacts consistency and availability. One approach is leader-based replication, in which a specific replica, known as the leader, primary, master, or similar, is designated as the main source of data access and updates. Changes made to the primary replica are then propagated to the other replicas. This technique is powerful in preventing conflicts and deadlocks among the replicas, but can become a bottleneck as all access and updates must go through the primary replica. An alternative approach is to eliminate the leader concept and treat all replicas as equal (leaderless), allowing data to be accessed and updated using any replica. This can increase the availability of the system, but it complicates a consistent view of a system between replicas [23].

These systems provide a foundational building block for modern online services [24–31]. RAMCloud [32] pioneered this design space by providing logging and recovering mechanism for distributed in-memory key-value stores. Prominent systems such as FaRM [33] and Pilaf [34] provide a shared address space between nodes, and take advantage user space networking, especially RDMA [35]. In contrast

to our second project AVOCADO, these system did not consider security for the untrusted infrastructure.

2.1.3 Persistent Memory

Persistent memory (PMEM) [36] is byte-addressable persistent storage which is directly connected to the memory bus of the system together with DRAM (NVRAM), with comparable latency and throughput. This is in contrast to more conventional block storage devices, which have to read and write and entire fixed size block, increasing latency and decreasing throughput for small accesses. Software frameworks like PMDK [36] implement routines and data structures to keep the persistence guarantees of PMEM in cases of crashes or unexpected failures.

2.2 Trusted Execution and Memory Safety

This thesis explores methods and tools for constructing a secure storage system in untrusted environments. SPEICHER and AVOCADO leverage trusted execution environments (TEEs) (see § 2.2.1) and their remote attestation capabilities (see § 2.2.2) to create a trusted key-value store. TOAST acknowledges that TEEs result in an increased number of heterogeneous memory regions, as TEEs divide memory into trusted and untrusted segments. This exacerbates the programmability and portability issues of heterogeneous memory. To avoid sensitive information leaks, TOAST implements two protection libraries: a hardware-based Intel MPK (see § 2.2.3) and a software capability-based one (see § 2.2.4), which provide protection against accidental memory leaks.

2.2.1 Trusted Execution Environments

Trusted execution environments (TEEs) [5–10, 37] are tamper-resistant processing environments that guarantee the authenticity, the integrity and the confidentiality of their executing code, data and runtime states, e.g. CPU registers and memory. Figure 2.3 shows a general overview of TEEs. Their content remains resistant against all software attacks even from privileged code, e.g., OS and hypervisor, as well as any physical attacks, such as memory probes, performed on the main memory of the system.

TEEs can be classified into three distinct categories, each offering unique characteristics and functionalities.

The first category comprises TEEs that strictly enforce isolation between the trusted and untrusted environments or "worlds." In this setup, the software stack

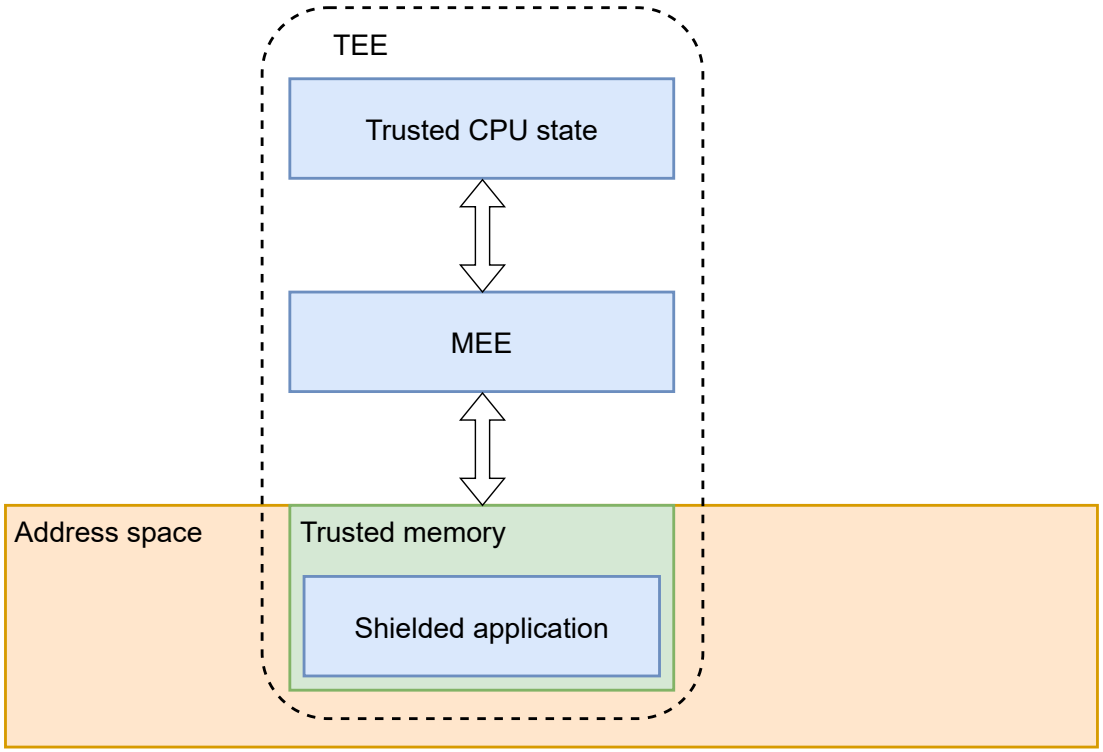

Figure 2.3: Trusted execution environments (TEEs) secure the CPU state and guarantee that the trusted memory is access through the memory encryption engine (MEE) only. The TEE keeps the de-/encryption key private to a specific execution.

cannot be shared between the two worlds. While this approach ensures a robust separation, it presents challenges when it comes to sharing data between them. Examples of TEEs in this category include ARM TrustZone [7] and RISC V Keystone [10]. Although the techniques discussed in this thesis can be adapted for use with this type of TEE, their primary focus is not on providing cloud services since they have limitations on the number of trusted worlds they support.

The second category of TEEs builds the trusted environment within an application, resulting in a minimal trusted computing base (TCB). The TEE can be invoked from within the application, and while it can access the application's memory, it does not have access to the memory of other processes. To handle memory management, these TEEs employ a hardware-based memory management systems since applications typically do not implement page fault handlers. Intel SGX [5] serves as a prime example of this type of TEE. A more comprehensive description of SGX is provided later in this section, as all three projects presented in this thesis are based on SGX.

The final category is represented by TEEs such as Intel TDX [6], AMD SEV-SNP [9], and ARM Realms [8]. These TEEs focus on providing trusted virtual

machines (VMs). They enable a larger TCB within the trusted environment and offer software-based memory management. This approach facilitates easier development, as system designers can create a typical software environment within the trusted environment. Over the past few years, there has been a growing trend towards this category of TEEs, reflecting their increasing popularity and adoption.

SGX SGX, a TEE implementation by Intel, offers the abstraction of an isolated memory called *enclave*. Enclave pages reside in the enclave page cache (EPC) — a specific memory region (up to 128 MiB for v1 and 256 MiB for v2) which is protected by an on-chip memory encryption engine (MEE). To support applications with larger memory footprint (up to 4 GiB in v1 and with support of the OS up to 256 GiB in v2) SGX implements a *paging* mechanism. However, the EPC paging mechanism incurs high overheads [1, 38].

This isolation prohibits SGX applications from executing outside-of-the-enclave code directly, e.g., system calls. Thus, enclave threads need to *exit* the trusted environment and further copy all associated data out of the enclave since kernel code cannot access it. Afterwards, threads have to *enter* the enclave again. We refer to this as *world switch*.

With the adoption of TEEs in the cloud, shielded execution frameworks are being developed and adopted to provide strong security properties for unmodified/legacy applications running in the untrusted environment. Prominent examples for the SGX-based shielded execution framework include Haven [39], SCONE [40], Graphene [41], Panoply [42], Eleos [38], Asylo [43], Google's Confidential VMs [11], Open Enclave SDK [44] and CCF [45]. This thesis leverages the advancements in shielded execution frameworks; in particular, we use SCONE [40] to build distributed storage systems.

2.2.2 Remote attestation

Remote attestation is a powerful mechanism used to verify that a system stack on a remote machine is in the expected state. It ensures that the correct version of the software stack is running on the intended hardware. This mechanism relies on a trusted entity (i.e. root of trust), such as a Trusted Execution Environment (TEE) or Trusted Platform Module (TPM), which performs measurements by calculating secure hashes over the loaded software components.

Initially, the root of trust, whether it be a TEE or TPM, measures the integrity of crucial software elements like the bootloader, operating system kernel, hypervisor, or software within a TEE. Subsequently, the loaded software can measure additional components, such as programs, and inform the root of trust about their measurements.

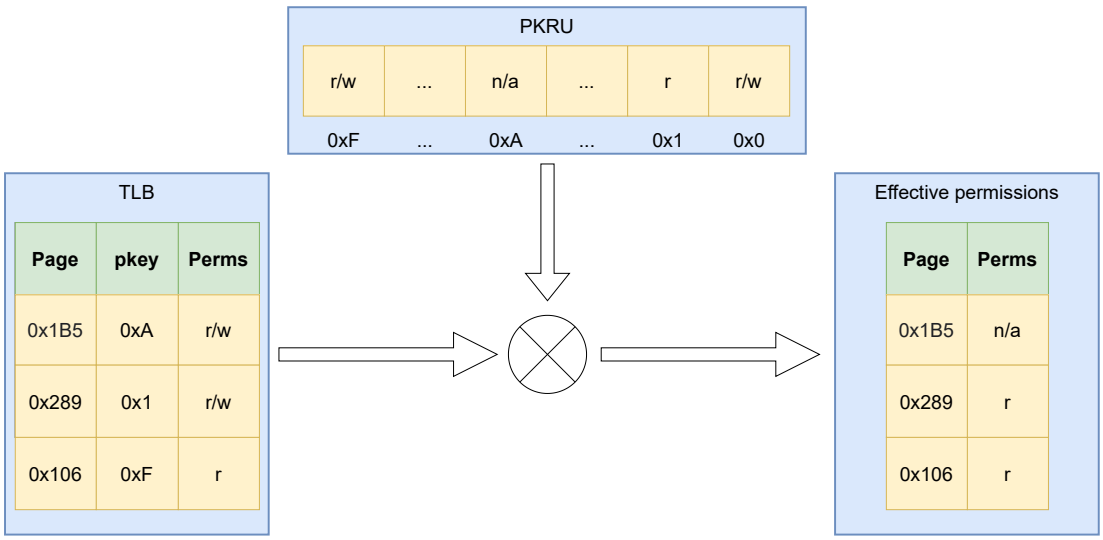

Figure 2.4: MPK uses 4 previously reserved bits to store a tag per entry in the TLB. It then combines the access rights for the page together with the access rights stored in the user-controlled thread-local PKRU register for the tag to generate the effective access permissions of the page. The effective permissions is the intersection of the page permission and tag permission.

The root of trust combines its own calculated hash with the new measurements received, resulting in a new measurement value. At this stage, a process can request the root of trust to provide and sign the new hash value. With this signed value in hand, a remote process can easily validate the system stack's configuration. It accomplishes this by verifying the signature of the root of trust and comparing the measurement value (hash value) against a pre-established known value.

By employing remote attestation, organizations can gain confidence in the integrity and security of remote system stacks by verifying their compliance with expected configurations.

2.2.3 MPK

MPK [46] is an x86 ISA extension that allows for user space page-level access control. It leverages 4 previously unused bits of every page-table entry to place a tag. The allocation and release of a protection key as well as the page tagging operation require elevated privileges and, therefore, are performed via system calls. However, a process can change the granted permissions for the pages tagged with a specific key directly in user space by updating a special register (PKRU). Note that, the PKRU value is thread-local; each thread can manage the permissions of its keys autonomously. Figure 2.4 shows how the page permissions interact with the tag permissions to evaluate the

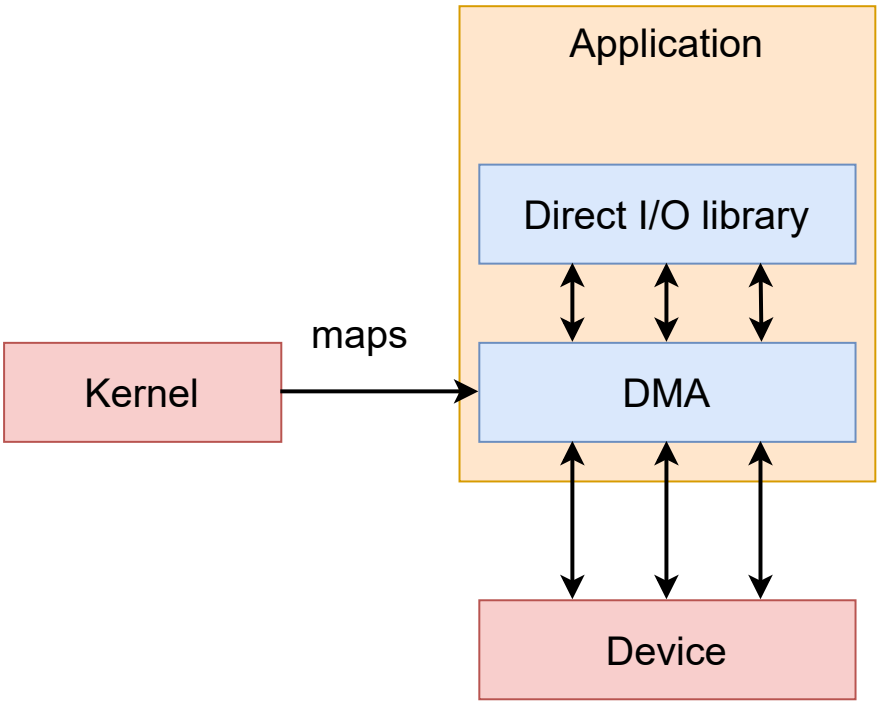

Figure 2.5: The direct I/O library communicates over the direct access memory (DMA) mapping directly with the device. This eliminates system calls.

effective permissions.

2.2.4 Capabilities

Capabilities are an efficient token-based method to perform resource management and fine-grained access control, suitable for security-critical systems [47–56]. Precisely, a capability is a reference to an object or resource together with its access rights.

To allow for flexible resource access control, capabilities can typically be exchanged between processes and protection domains. Four core operations are commonly supported; *(i)* create, where a new capability is created for a resource either from scratch or by inheriting the rights of an existing capability referring to the same resource, *(ii)* modify, which alters the access rights to a resource of an existing capability and *(iii)* revoke, that deletes the existing capability and the ones derived from it, *(iv)* invoke, which access the resource referenced by the capability. When an application attempts to access a resource (e.g., memory, storage) managed by a capability, the system examines the current capability rights and permits the access or aborts the operation based on them.

2.3 Direct I/O Stack

This thesis comprises three projects that utilize direct I/O libraries for faster processing. SPEICHER uses SPDK (see § 2.3.2) to enhance persistent storage performance. AVOCADO employs a combination of DPDK (see § 2.3.1) and eRPC (see § 2.3.3) to establish a secure network layer. TOAST unifies heterogeneous memory access with direct I/O libraries. All three direct I/O libraries (SPDK, DPDK, and eRPC) are utilized in TOAST's case studies.

Direct I/O in user space bypasses parts or all of the OS kernel stack and has become a popular alternative to traditional OS kernel drivers due to its ease of development and deployment and its performance advantages. This is especially true in high-throughput/low-latency applications such as high-performance networking (over 100 Gbit) or NVMe, where context switches can add a significant overhead [57–61]. This has led to the adoption of direct I/O libraries in these settings to avoid kernel context switches.

Direct I/O libraries operate by establishing a direct communication channel between the device and the system through a mechanism known as direct memory access (DMA). DMA enables devices to access the system's main memory directly, bypassing the need for CPU involvement. This approach offers several benefits, such as reduced latency and decreased CPU utilization, particularly for high-performance devices.

Additionally, the operating system has the capability to map the physical memory range utilized by the device into the address space of an application. This mapping grants the user-space process control over the device. Consequently, the application gains the ability to manipulate and manage the device's operations.

To further optimize the performance of direct I/O libraries, one can employ a polling-based system and instruct the kernel to disable interrupts for the specific device. By eliminating the need for context switches, the number of interruptions caused by device communication is significantly reduced.

2.3.1 DPDK

Data Plane Development Kit (DPDK) [20] is a user space direct I/O library for different platforms, i.e., x86, ARM, and PowerPC, and different OSes, i.e., Linux, FreeBSD, and Windows. It abstracts network interface controllers (NIC) easing the development of network applications. DPDK expect the user to provide their own network stack, for session and transport layer.

2.3.2 SPDK

The Storage Performance Development Kit (SPDK) [16] is based on DPDK and provides a user space direct I/O abstraction for Non-Volatile Memory (NVM) Express (NVMe). NVMe is a standardized interface for accessing storage media such as solid state drives (SSDs) over PCI Express (PCIe). It was designed to fully leverage the capabilities of modern SSDs. SPDK offers low-level block access to the underlying storage device, as well as an object allocator called Blobstore [62] and a filesystem based on Blobstore called BlobFS [63].

2.3.3 eRPC

eRPC [19] is a state-of-the art general-purpose and asynchronous remote procedure call (RPC) library for high-speed networking for lossy Ethernet or lossless fabrics. eRPC uses a polling-based network I/O along with user space drivers, eliminating interrupts and system call overheads from the datapath. Particularly, application threads create exclusive `RPC` objects and one-to-one connections between them in order to `enqueue` and `receive` requests and responses. Each `RPC` reserves `hugepages`' memory for packet I/O, i.e., Rx and Tx queues.

Furthermore, eRPC provides a UDP stack, leveraging optimization techniques, (zero-copy reception, congestion control, etc.) while it remains generic; it supports a wide range of transport layers such as RDMA, DPDK, and RoCE.

Chapter 3

SPEICHER: A Secure LSM-based KV Store

We introduce SPEICHER, a secure storage system that not only provides strong confidentiality and integrity properties, but also ensures data freshness to protect against rollback/forking attacks. SPEICHER exports a Key-Value (KV) interface backed by Log-Structured Merge Tree (LSM) for supporting secure data storage and query operations. SPEICHER enforces these security properties on an untrusted host by leveraging shielded execution based on a hardware-assisted trusted execution environment (TEE)—specifically, Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure SGX enclave memory region to ensure that the security properties are also preserved in the stateful (or non-volatile) setting of an untrusted storage medium, including system crash, reboot, or migration.

More specifically, we have designed an authenticated and confidentiality-preserving LSM data structure. We have further hardened the LSM data structure to ensure data freshness by designing asynchronous trusted counters. Lastly, we designed a direct I/O library for shielded execution based on Intel SPDK to overcome the I/O bottlenecks in the SGX enclave. We have implemented SPEICHER as a fully-functional storage system by extending RocksDB, and evaluated its performance using the RocksDB benchmarks. Our experimental evaluation shows that SPEICHER incurs reasonable overheads for providing strong security guarantees, while keeping the trusted computing base (TCB) small.

3.1 Motivation

With the growth in cloud computing adoption, online data stored in data centers is growing at an ever increasing rate [64]. Modern online services ubiquitously use persistent key-value (KV) storage systems to store data with a high degree of reliability and performance [17, 21]. Therefore, persistent KV stores have become a fundamental part of the cloud infrastructure.

At the same time, the risks of security violations in storage systems have increased significantly for the third-party cloud computing infrastructure [65]. In an untrusted environment, an attacker can compromise the security properties of the stored data and query operations. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems [66–72].

However, securing a storage system is quite challenging because modern storage systems are quite complex [68, 73–75]. For instance, a persistent KV store based on the Log-Structured Merge Tree (LSM) data structure [14] is composed of multiple software layers to enable a data path to the storage persistence layer. Thereby, the enforcement of security policies needs to be carried out by various layers in the system stack, which could expose the data to security vulnerabilities. Furthermore, since the data is stored outside the control of the data owner, the third-party storage platform provides an additional attack vector. The clients currently have limited support to verify whether the third-party operator, even with good intentions, can handle the data with the stated security guarantees.

In this landscape, the advancements in trusted execution environments (TEEs), such as Intel SGX [5] or ARM TrustZone [7], provide an appealing approach to build secure systems. In fact, given the importance of security threats in the cloud, there is a recent surge in leveraging TEEs for shielded execution of applications in the untrusted infrastructure [38–42]. *Shielded execution* aims to provide strong security properties using a hardware-protected secure memory region or *enclave*.

While the shielded execution frameworks provide strong security guarantees against a powerful adversary, they are primarily designed for securing “stateless" (or volatile) in-memory computations and data. Unfortunately, these stateless techniques are not sufficient for building a secure storage system, where the data is persistently stored on an untrusted storage medium, such as an SSD or HDD. The challenge is *how to extend the trust beyond the “secure, but stateless/volatile” enclave memory region to the “untrusted and persistent” storage medium, while ensuring that the security properties are preserved in the “stateful settings”, i.e., even across the system reboot, migration, or crash.*

To answer this question, we built a secure storage system using shielded execution targeting all three important security properties for the data storage and

query processing: (a) *confidentiality* — unauthorized entities cannot read the data, (b) *integrity* — unauthorized changes to the data can be detected, and (c) *freshness* — stale state of data can be detected as such.

To achieve these security properties, more specifically, we need to address the following three architectural limitations of shielded execution in the context of building a secure storage system: Firstly, while modern TEEs like Intel TDX and AMD SEV-SNP have no limitation on the trusted memory size, Intel SGX's, on which this project is based on, secure enclave memory region is quite limited in size, and incurs high performance overheads for memory accesses. It implies that the storage engine cannot store the data inside the enclave memory; thus, the in-memory data needs to be stored in the untrusted host memory. Furthermore, the storage engine persists the data on an untrusted storage medium, such as SSDs. Since the TEE cannot give any security guarantees beyond the enclave memory, we need to design mechanisms for extending the trust to secure the data in the untrusted host memory and also on the persistent storage medium.

Secondly, the syscall-based I/O operations are quite expensive in the context of shielded execution since the thread executing the system call has to exit the enclave, and perform a secure context switch, including TLB flushing, security checks, etc. While existing shielded execution frameworks [38, 40] proposed an *asynchronous* system call interface [76], it is clearly not well-suited for building a storage system that requires frequent I/O calls. To mitigate the expensive enclave exits caused by I/O syscalls, we need to design a direct I/O library for shielded execution to completely eliminate the expensive context switch from the data path.

Lastly, we also aim to ensure data freshness to protect against *rollback* (replay old state) or *forking attacks* (create second instance). Therefore, we need a protection mechanism based on a trusted monotonic counter [77], for example, SGX trusted counters [78]. Unfortunately, the SGX trusted counters are extremely slow and they wear out within a couple of days of operation. To overcome the limitations of the SGX counters, we need to redesign the trusted monotonic counters to suit the requirements of modern storage systems.

To overcome these design challenges, we propose SPEICHER, a secure LSM-based KV storage system. More specifically, we make the following contributions.

- **I/O library for shielded execution:** We have designed a direct I/O library for shielded execution based on Intel SPDK. The I/O library performs the I/O operations without exiting the secure enclave; thus it avoids expensive system calls on the data path.

- **Asynchronous trusted monotonic counter:** We have designed trusted counters

to ensure data freshness. Our counters leverage the lag in the sync operations in modern KV stores to asynchronously update the counters. Thus, they overcome the limitations of the native SGX counters.

- **Secure LSM data structure:** We have designed a secure LSM data structure that resides outside of the enclave memory while ensuring the integrity, confidentiality and freshness of the data. Thus, our LSM data structure overcomes the memory and I/O limitations of Intel SGX.

- **Algorithms:** We present the design and implementation of all storage and query operations in persistent KV stores: get, put, range queries, iterators, compaction, and restore.

We have built a fully-functional prototype of SPEICHER based on RocksDB [17], and extensively evaluated it using the RocksDB benchmark suite. Our evaluation shows that SPEICHER incurs reasonable overheads, while providing strong security properties against powerful adversaries.

3.1.1 Threat Model

In addition to the standard SGX threat model [39], we also consider the security attacks that can be launched using an *untrusted* storage medium, e.g., persistent state stored on an SSD or HDD. More specifically, we aim to protect against a powerful adversary in the virtualized cloud computing infrastructure [39]. In this setting, the adversary can control the entire system software stack, including the OS or hypervisor, and is able to launch physical attacks, such as performing memory probes.

For the untrusted storage component, we also aim to protect against rollback attacks [77], where the adversary can arbitrarily shut down the system, and replay from a stale state. We also aim to protect against forking attacks [79], where the adversary can attempt to fork the storage system, i.e., the KV Store, e.g., by running multiple replicas of the storage system.

Even under the extreme threat model, our goal is to guarantee the data integrity, confidentiality, and freshness. Lastly, we also aim to provide crash consistency for the storage system [80].

However, we do not protect against side-channel attacks, such as exploiting cache timing and speculative execution [81], or memory access patterns [82, 83]. Mitigating side channel attacks in the TEEs is an active area of research [84]. Further, we do not consider the denial of service attacks since these attacks are trivial for a third-party operator controlling the underlying infrastructure [39]. Lastly, we assume that the

adversary cannot physically open the processor packaging to extract secrets or corrupt the CPU system state.

3.2 Design

SPEICHER is a secure persistent KV storage system designed to operate on an untrusted host. SPEICHER provides strong confidentiality, integrity, and freshness guarantees for the data storage and query operations: `get`, `put`, `range queries`, `iterators`, `compaction`, and `restore`. We implemented SPEICHER by extending RocksDB [17], but our architecture can be generalized to other LSM-based KV stores.

3.2.1 Design Challenges

As a strawman design, we could try to secure a storage system by running the storage engine inside the enclave memory. However, the design of a practical and secure system requires addressing the following four important architectural limitations of Intel SGX.

I: Limited EPC size The strawman design would be able to protect the in-memory state of the MemTable using the EPC memory. However, EPC is a limited and shared resource. Currently, the size of EPC is 128 MiB. Approximately 94 MiB are available to the user, the rest is reserved for the metadata. To allow creation of enclaves with sizes beyond that of EPC, SGX features a secure paging mechanism. The OS can evict EPC pages to an unprotected memory using SGX instructions. During eviction, the page is re-encrypted. Similarly, when an evicted page is brought back, it is decrypted and its integrity is checked. However, the EPC paging incurs high performance overheads (2–2000 $\times$) [40].

Therefore, we need to redesign the shielded storage engine, where we allocate the MemTable(s) outside the enclave in the untrusted host memory. Since the secure enclave region cannot give any guarantees for the data stored in the host memory, and the native MemTable is not designed for security - we designed a new MemTable data structure to guarantee the confidentiality, integrity and freshness properties.

Modern TEEs, like Intel TDX [6] and AMD SEV-SNP [9], do not have the same memory limitation as Intel SGX. This in turn removes the need of redesigning the in-memory MemTable, as it fits in the trusted memory area of the TEE. However, SPEICHER targets Intel SGX and therefore has to consider the limited EPC size.

II: Untrusted storage medium The storage engine does not exclusively store the data in the in-memory MemTable, but also on a persistent storage medium, such as on an SSD or HDD. In particular, the storage engine stores three types of files on a persistent storage medium: SSTable, WAL and the Manifest. However, Intel SGX is designed to protect only the volatile state residing in the enclave memory. Unfortunately, SGX does not provide any security guarantees for stateful computations, i.e., across system reboot or crash. Further, the trust from the TEE does not naturally extend to the untrusted persistent storage medium.

To achieve the end-to-end security properties, we further redesigned the LSM data structure, including the persistent storage state in the SSTable and log files, to extend the trust to the untrusted storage medium.

III: Expensive I/O syscall To access data stored on an SSD or HDD (in the SSTable, WAL or Manifest files), conventional systems leverage the system call interface. However, the system call execution in the SGX environment incurs high performance overheads. This is because the thread executing the system call has to exit the enclave, and the syscall arguments need to be copied in and out of the enclave memory. These enclave transitions are expensive because of security checks and TLB flushes.

To mitigate the context switch overhead, shielded execution frameworks, such as SCONE [40] or Eleos [38], provide an *asynchronous* system call interface [76], where a thread outside the enclave asynchronously executes the system calls without forcing the enclave threads to exit the enclave. While such an asynchronous interface is useful for many applications, it is not clearly suited for building a storage system that needs to support frequent I/O system calls.

To support frequent I/O calls within the enclave, we designed a new I/O mechanism based on a direct I/O library for shielded execution leveraging storage performance development kit (SPDK) [16].

IV: Trusted counter In addition to guaranteeing the integrity and confidentiality, we also aim to ensure the freshness of the stored data to protect against rollback attacks [77]. To achieve the freshness property, we need to protect the data stored in the untrusted host memory (MemTable), and those on the untrusted persistent storage medium (SSTable, WAL and Manifest files).

For the first part, i.e., to ensure the freshness of MemTable allocated in the untrusted host memory, we can leverage the EPC of SGX. In particular, the Memory Encryption Engine (MEE) in SGX already protects the EPC against rollback attack.

Therefore, we use the EPC to store a *freshness signature* of the MemTable, which we use at runtime to verify the freshness of data stored as part of the MemTable in the untrusted host memory.

However, the second part is quite tedious, i.e., to ensure the freshness of the data stored on untrusted persistent storage (SSTables and log files), because the rollback protected EPC memory is stateless, or it cannot be used to verify the freshness properties after the system reboots or crashes. Therefore, we need a rollback protection mechanism based on a trusted monotonic counter [77]. For example, we could use SGX trusted counters [78]. Unfortunately, the SGX trusted counters are extremely slow ($60 - 250$ ms) [85]. Furthermore, the counter memory allows only a limited number of write operations to NVRAM, and it easily becomes unusable due to wear out within a couple of days of operation. Therefore, the SGX counters are impractical to design a storage system.

To overcome the limitations of SGX counters, we designed an asynchronous trusted monotonic counter that drastically improves the throughput and mitigates wear-out by taking advantage of the crash consistency properties of modern storage systems.

3.2.2 System Components

We next detail the system components of SPEICHER. Figure 3.1 illustrates the high-level architecture and building blocks of SPEICHER. The system is composed of the controller, a direct-I/O library for shielded execution, a trusted monotonic counter, the storage engine (RocksDB engine), and a secure LSM data structure (MemTable, SSTable, and log files).

SPEICHER controller The controller provides the trusted execution environment based on Intel SGX [40]. Clients communicate over a mutually authenticated encrypted channel (TLS) to the controller. The TLS channel is terminated inside the controller. In particular, we built the controller based on the SCONE shielded execution framework [40], where we leverage SCONE's container support for secure deployment of the SPEICHER executable on an untrusted host.

The controller provides the remote attestation service to the clients [86, 87]. In particular, the SGX enclave generates a signed measured of its identity, whose authenticity can be verified by a third party. After successful attestation, the client provides its encryption keys to the controller. The controller uses the client certificate to perform the access control operation. The controller also provides runtime support for user-level multithreading and memory management inside the enclave. The

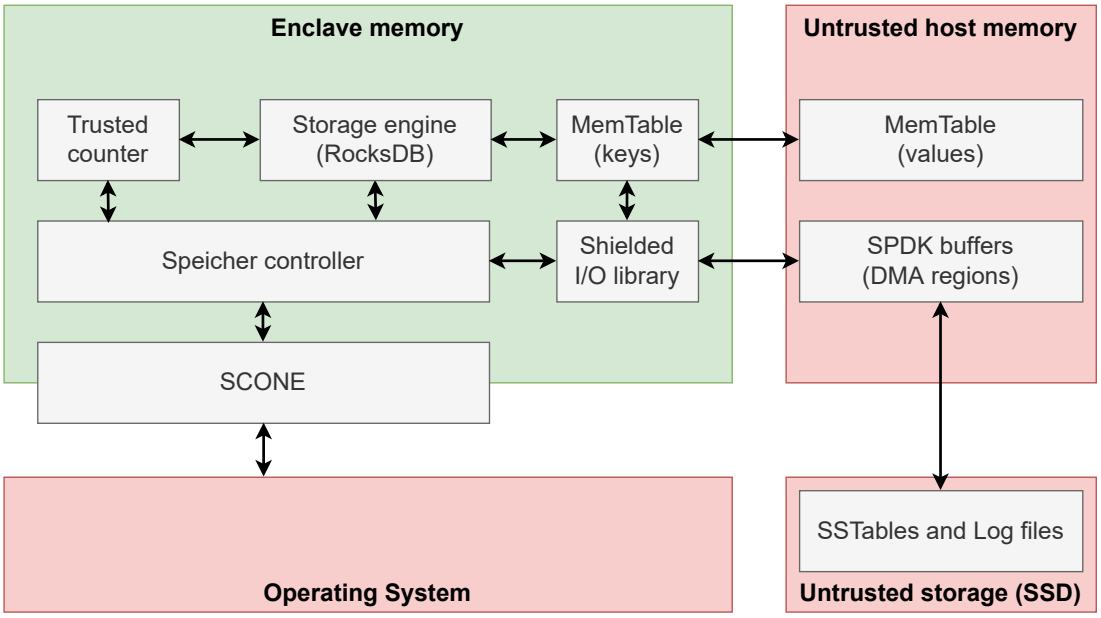

Figure 3.1: SPEICHER overview

controller leverages the asynchronous system calls interface (SCONE libc) on the control path for the system configuration. For the data path I/O, we built a direct I/O library, which we describe next.

Shielded direct I/O library The I/O library allows the storage engine to access the SSD or HDD from inside the SGX enclave, without issuing the expensive enclave exit operations. We achieve this by building a direct I/O library for shielded execution based on SPDK [16].

SPDK is a high-performance user-mode storage library, based on Data Plane Development Kit (DPDK) [20]. It eliminates the need to issue system calls to the kernel for read and write operations by having the NVMe driver in the user space. SPDK enables zero-copy I/O by mapping DMA buffers to the user address space. It relies on actively polling the device instead of interrupts.

These SPDK features align with the goal of SPEICHERof exit less I/O operations in the enclave, i.e., to allow the shielded storage engine to interact with the SSD directly. However, we need to adapt the design of SPDK to overcome the limitations of the enclave memory region. In particular, our shielded I/O library allocates huge pages and SPDK ring buffers outside the enclave for DMA. The host system maps the device in an allocated DMA region. Afterwards SPDK can initialize the device. To reduce the number of enclave exits, SPDK's device driver runs inside the enclave. This enables efficient delivery of requests from the storage engine to the driver, which explicitly copies the data between the host and the enclave memory.

Trusted counter In order to protect the system from rollback attacks, we need a trusted counter whose value is stored alongside with the LSM data structure. Intel SGX provides monotonic counters, but their update frequency is in a range of 10 updates per second, and we indeed measured approximately 250 ms to increment a counter once. This is far too slow for modern KV stores [88].

To overcome the limitations of SGX counters, we designed an Asynchronous Monotonic Counter (AMC) based on the observation that many contemporary KV stores do not persist their inserted data immediately. This allows AMC to defer the counter increment until the data is persisted without loosing any availability guarantees. As a result, AMC achieves 70 k updates per second in the current implementation.

AMC provides an asynchronous increment interface, because it takes a while since the counter value is incremented until it becomes *stable*, which means the counter value cannot be rolled back without being detected. At an increment, AMC returns three pieces of information: the current stable value, the incremented counter value, and the *expected time* for the value to be stable. Due to the expected time and the controller having to be re-authenticated after a shutdown, the client only has to keep the values until the stable time has elapsed, to prevent any data loss in case of a sudden shutdown.

AMC's flexible interface allows us to optimize update throughput and latency by increasing the time until a trusted counter is stable. This also allows users to adjust trade-off between the wear out of the SGX monotonic counter and the maximum number of unstable counter increments, which a client might have to account for. SPEICHER generates multiple counters by storing their state to a file, whose freshness is guaranteed through the use of a synchronous trusted monotonic counter. For instance, we can employ SGX monotonic counters [78], ROTE [85] or Ariadne [89] to support our asynchronous interface. Therefore, we can have a counter with deterministic increments for WAL and the Manifest, making it possible to argue about the freshness of each record in the files.

MemTable As detailed in §3.2.1, the EPC is limited in size and the EPC paging incurs very high overheads. Therefore, it is not judicious to store large MemTables or multiple MemTables within the EPC. Further, since SPEICHER uses the EPC memory region to secure the storage engine (RocksDB) and the shielded I/O library driver, it further shrinks the available space.

Due to this memory restriction, we need to store the MemTable in the host memory. Since the host memory is untrusted, we need to devise a mechanism to ensure the

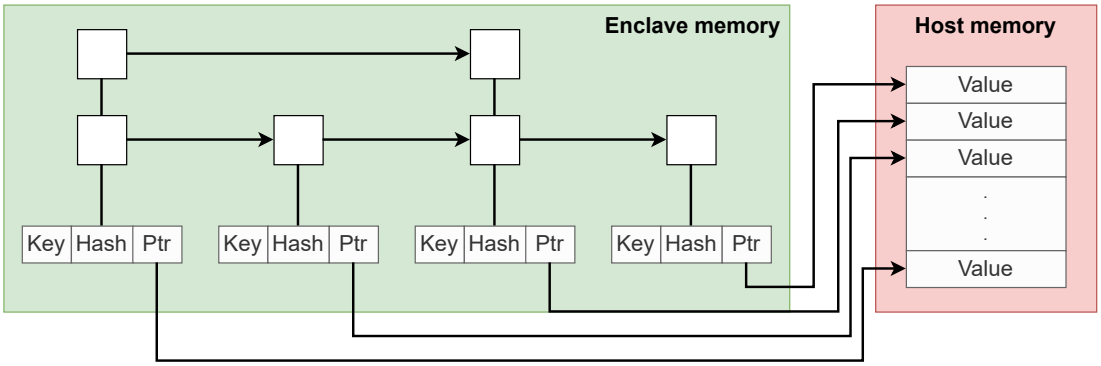

Figure 3.2: SPEICHER MemTable format based on skip list design

confidentiality, integrity, and freshness of the MemTable.

In our project, we tried three different designs for the MemTable. Firstly, we explored a native Merkle tree that generates hashes of the leafs and stores them in each node. Thus, we can verify the data integrity by checking the root node hash and each hash down to the leaf storing the KV, while allowing the MemTable to be stored outside the EPC memory. However, the native Merkle tree suffers from slow lookups as the key has to be decrypted on each traversal. Further, it requires multiple hash recalculations on each lookup and insertion.

Secondly, we tried a modified Merkle tree design based on a prefix array, where a fixed size prefix is used as an index into the array of Merkle trees. An array entry holds the root node of a Merkle tree, which holds the actual data. This should reduce the depth of the search tree compared to the native Merkle tree; thus, reducing the number of necessary hash calculations and decryptions of keys. However, while we were able to increase the lookup speed compared to the native Merkle tree, it still suffered from the same problem of having to decrypt a large number of keys in a lookup, and causing a large number of hash calculations.

Lastly, our third attempt of the MemTable design reuses the existing skip list data structure for the MemTable in RocksDB. Figure 3.2 shows SPEICHER's MemTable format. In particular, we partition the existing MemTable in two parts: key path and value path. In the key path, we store the keys as part of the skip list inside the enclave. Whereas, the encrypted values in the MemTable are stored in the untrusted host memory as part of the value path. This partitioning allows SPEICHER to provide confidentially by encrypting the value, while still enabling fast key lookups inside the enclave. To prevent attacks on the integrity or the freshness of the values, SPEICHER stores a cryptographic hash of the value in each skip list node together with the host memory location of the value.

While the first two designs removed almost the entire MemTable from the EPC,

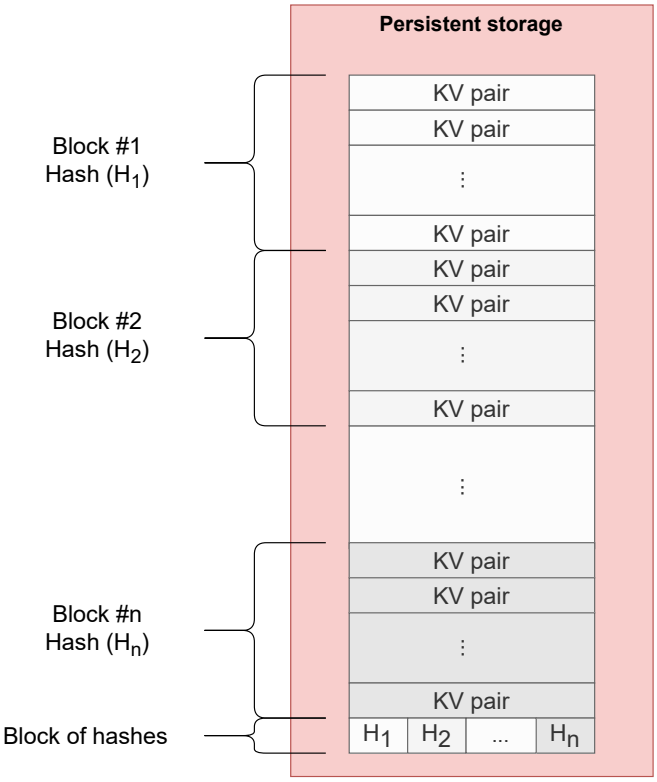

Figure 3.3: SPEICHER SSTable file format

the last design still maintains the keys and hash values inside the enclave memory. To determine the space requirements of our MemTable in comparison to the regular RocksDB's MemTable, we use the following formula:

$$S = n * (k + v) + \sum_{i=0}^{m} p^i * n * ptr$$

Where S represents the entire size of the skip list, n is the number of KV pairs, k is the key size, v is the value size or the size of the pointer plus hash value for our skip list, p is the probability for being added into a specific layer of the skip list, m is the maximum number of layers, and ptr is the size of a pointer in the system.

For instance, in case of the default setting for RocksDB, with a maximum size of 64 MiB, key size of 16 B, value size of 1024 B, pointer size of 8 B, p of 1/4, m of 12 and for SPEICHER's skip list a hash size of 16 B — SPEICHER's MemTable achieves a space reduction of approximately 95.2 %. Further, the reduction ratio increases with increased value size.

SSTables The SSTable files maintain the KV pairs persistently. These files store KV pairs in the ascending order of keys. This organization allows for a binary search within the SSTable, requiring only a few reads to find a KV-pair within the file. Since

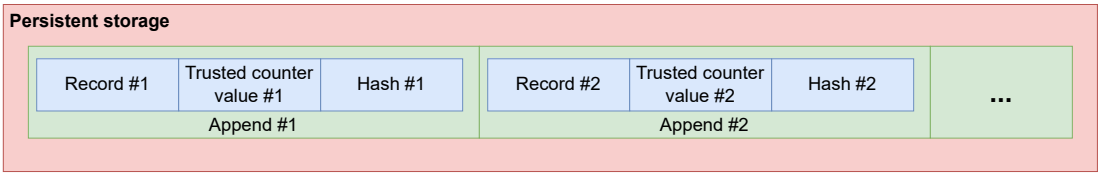

Figure 3.4: SPEICHER append-only log file format

SSTable files are optimized for block devices, such as SSDs, they group KV pairs together into blocks (the default block size is 32 KiB, as this is also the default in RocksDB).

SPEICHER adapts SSTable file format to ensure the security properties (see Figure 3.3 for SPEICHER's SSTable file format). The confidentiality is secured by encrypting each block of the SSTable file before it is written to the persistent storage medium. Additionally, SPEICHER calculates a cryptographic hash over each block. These hashes are then grouped together in a block of hashes and appended at the end of the SSTable file. When reading SPEICHER can check the integrity of each block by calculating the block's hash and comparing it to the corresponding hash stored in the footer. To protect the integrity of the footer an additional hash over the footer is calculated and stored in the Manifest. Since the Manifest is protected against rollback attacks using a trusted counter, the footer hash value stored in the Manifest is also protected from the rollback attacks. Thus, SPEICHER can use this hash to guarantee the freshness of the SSTable file's footer and transitively the freshness of each block in the SSTable file.

Log files RocksDB uses two different log files to keep track of the state of the KV store: (a) WAL for persisting inserted KV pairs until a top-level compaction; and (b) the Manifest to keep track of live files, i.e., the set of files of which the current state of the KV store consists. SPEICHER adapted these log files to ensure the desired security properties, as shown in Figure 3.4.

Regarding WAL, every `put` operation appends a record to the current WAL. This record consists of the encrypted KV pair, and an encrypted trusted counter value for the WAL at the moment of insertion, and a cryptographic hash over both. Since the records are only appended to the WAL, SPEICHER can use the trusted counter value and the hash value to verify the KV pair, and to replay the operations in a restore event.

The Manifest is similar to the WAL; it is a write-append log consisting of records storing changes of live files. We use the same scheme for the Manifest file as we do for the WAL.

3.2.3 Algorithms

We next present the algorithms for all storage operations in SPEICHER. The associated pseudocodes are detailed in the appendix.

I: Put Put is used to insert a new KV pair into the KV store, or to update an existing one. We need to perform two operations to insert the KV pair into the store (see Algorithm 1). First, we need to append the KV pair to the WAL for persistence. Second, we need to write the KV pair to the MemTable for fast lookups.

Inserting the KV pair into the WAL guarantees that the state of the KV store can be restored after an unexpected reboot. Therefore, the KV pair should be inserted into the WAL before it is inserted into the MemTable. To add a KV pair to the WAL, SPEICHER encrypts the pair together with the next WAL trusted counter value and a cryptographic hash over both the data and the counter. The encrypted block is then appended to the WAL (see the log file format in Figure 3.4). Thereafter, the trusted counter is incremented to the value stored in the appended block. In addition, the client is notified when the KV pair will be stable; thereafter, the state cannot be rolled back. In case of a system crash between generating the data block and increasing the trusted counter value, the data block would be invalid at reboot, because the trusted counter would point the block to a future time. This operation is safe as the client can detect a reboot when SPEICHER tries to authenticate itself. After the reboot the client can ask the KV store about what the last added key was, or can simply `put` the KV pair again in the store as another request with the same key supersedes any old value with the same key.

In the second step, SPEICHER writes the KV pair into the MemTable and thereby making the `put` visible to later `gets`. SPEICHER first encrypts the value of the KV pair and generates a hash over the encrypted data. The encrypted value is then copied to the untrusted host memory, while the hash with a pointer to the value is inserted into the skip list in the enclave, in accordance to SPEICHER's MemTable format (Figure 3.2). Since the KV pair is first inserted into the WAL, and only if this is successful, i.e., the WAL and trusted counter are updated, we can guarantee that only KV value pairs whose freshness is secured by the trusted counter are returned.

II: Get Get may involve searching multiple levels in the LSM data structure to find the latest value. Within each level, SPEICHER has to generate either the proof of existence, or the proof of non-existence of the key. This is necessary to detect insertion or deletion of the KV pairs by an attacker.

Algorithm 2 details the `get` operation in SPEICHER. In particular, SPEICHER begins

with searching the MemTable. SPEICHER searches the skip list for the node with the key. Either the key is in the MemTable, then the hash value is calculated over the value and compared to the hash stored in the skip list, or the key could not be found in the skip list. Since the skip list resides inside the protected memory region, SPEICHER does not need to make the non-existence proof for the MemTable because an attacker cannot access the skip list. If the KV store finds a key in the MemTable and the existence proof is correct, i.e., the calculated hash value is equal to the stored hash value, the value is returned to the client. If the proof is incorrect, the client is informed that the MemTable is corrupted. Since the MemTable can be reconstructed from the WAL, the client can then instruct the SPEICHER to recreate the KV store state in the case of an incorrect proof.

When the key is not found in the MemTable, the next level is searched. All levels below the MemTable are stored in SSTables. The SSTable files are organized in a way that no two SSTables in the same level have an overlapping key-range. Additionally, all the keys are sorted within an SSTable file. Due to this, any given key can only exist in one position in one SSTable file per level. This allows SPEICHER to construct a Merkle tree on top of the SSTable files of a level. With the ordering inside the SSTable, SPEICHER can correlate a block in the file with the key. This allows SPEICHER to calculate a hash over this block, which then can be checked against the stored hash in the footer. The hash of the footer can then be checked against the Merkle tree over the SSTable files in that level. It gives SPEICHER the proof of non-/existence for the lookup, and possibly the value belonging to the key. If the proof fails, the client is informed. In contrast to an incorrect proof in the MemTable, SPEICHER is not able to recover from this problem since the data is stored on the untrusted storage medium. If SPEICHER finds the KV pair and the proof is correct, it returns the value to the client. If the key does not exist, that is SPEICHER could not find it in any level and all level proofs are correct, an empty value is returned.

The freshness of data is guaranteed either by checking the value against the securely stored hash in the EPC for the case where the key has been found in the MemTable, or by checking the hash values of the SSTables against a Merkle tree. Additionally, as any key can only be stored in one position within a level, SPEICHER can also check against deletion of the key in a higher level, which is also necessary to guarantee freshness.

III: Range queries Range queries are used to access all KV pairs, with a key greater than or equal to a start key and lesser than an end key (see Algorithm 3). To find the start KV pair, we need to do the same operation as in get requests. Furthermore,

it requires to initialize an iterator in each level, pointing to the KV pair with a key greater or equal to the starting key. These iterators are necessary as higher levels have the more recent updates, due to keys being inserted into the highest level and being compacted over time to the lower levels, and lower being larger in size and therefore having more KV pairs. If the next KV pair is requested the next key of all iterators is checked and the iterators with the smallest next key are forwarded.

In case the next key is in multiple levels, the highest level KV pair is chosen. Therefore, SPEICHER has to do a non-/existence proof at all the levels, before it returns the chosen KV pair. If any of these proofs fails, the client is informed about the failed proof. Identical to the `get` operation, the client can then decide to either restore the KV store or to restore a backup.

Similar to the `get` operation, the hash value stored in the EPC and the Merkle tree over the SSTables are used to guarantee the freshness of the returned values.

IV: Iterators Iterators work identical to the range queries; they just have a different interface (see Algorithm 4).

V: Restore After a reboot, the KV store has to restore its last state (see Algorithm 5). This process is performed in two steps, first collecting all files belonging to the KV store, and then replaying all changes to the MemTable. In the first step the Manifest file is read. It contains all necessary information about the other files, such as live SSTable files, live WAL files, smallest key of each SSTable file. Each changing event about the live file is logged into the Manifest by appending a record describing the event. Therefore, at a restore all changes committed in the Manifest have to be replayed. This means that the SSTable files have to be put in the correct level. Each record in the Manifest is integrity-checked by a hash, and the freshness is guaranteed by the trusted counter for the Manifest. Since the counter value is incremented in a deterministic way, SPEICHER can use this value to check if all blocks are present in the Manifest. After the SSTable files in the levels are restored, and the freshness of all the SSTable files is checked against the Manifest by comparing the hash with the hash stored in the Manifest, the WAL is replayed.

Since each `put` operation is persisted in the WAL before it is written into the MemTable, replaying the `put` operations from the WAL allows SPEICHER to reconstruct the MemTable at the moment of the shutdown. Each `put` in the WAL has to be checked against the stored hash in the record, and the stored counter value. Additionally, since the counter value of the WAL is checked whether it equals to that of the Manifest counter, SPEICHER can check for the missing records. Records that

have a counter value being in the future, i.e. a counter value higher than the stored stable trusted counter value are ignored at restore. Further, due to the deterministic increase of the counter, SPEICHER can check against the missing records in the log files. If in any of these steps one of the checks fails, SPEICHER returns the information to the client, because SPEICHER is not able to recover from such a state.

VI: Compaction Compaction is triggered when a level holds data beyond a pre-defined threshold in size. In compaction (see Algorithm 6), a file from Level_n is merged with all SSTable files in Level_{n+1} covering the same key range. The new SSTables are added to Level_{n+1}, while all SSTables in the previous level are discarded. Before keys are added to the new SSTable file, the non-/existence proof is done on the files being merged. This is necessary to prevent the compaction process from skipping keys or writing old KV to the new SSTable files.

Since hash values are calculated over blocks of the SSTable files, a new block has to be constructed in the enclave memory, before it is written to the SSD. Also, all hash values of the blocks have to be stored in the protected memory until the footer is written and a hash over the footer is created. The file names of newly created SSTables and footer hashes are then written to the Manifest file, with the new trusted counter value. This is similar to the `put` operation. After the write operation to the Manifest completes and the trusted counter is incremented, the old SSTable files are removed from the KV store and the new files are added to Level_{n+1}. Since the hash values of the new SSTables are secured with a trusted counter value in the Manifest file, the SSTables cannot be rolled back after the compaction process.

3.2.4 Optimizations

Timer performance As described in §3.2.2, in order to prevent every request from blocking for the trusted counter increment, we leverage asynchronous counters written in files whose freshness is guaranteed by synchronous counters (or SGX counters). We use one counter for the WAL and another for the Manifest so that SPEICHER can operate on them independently. Although this method drastically improves throughput by allowing SPEICHER to process many requests without waiting for the counter to be stable, it also poses on the client the need for holding its write requests until the counter value is stable. This is why we designed and implemented the interface of AMC that reports the expected time for the counter to be stable. Because of this interface, the client does not need to frequently issue the requests to check the current stable counter value.

SPDK performance SPDK is designed to eliminate system calls from the data path, but in reality its data path issues two system calls on every I/O request: one for obtaining the process identifier and the other for obtaining the time. They are executed once in an I/O request that covers multiple blocks and their costs are normally amortized. However, since the context switch to and from the enclave is an order of magnitude more expensive, these costs are not amortized enough. We modified them to obtain the values from a cache within the enclave that are updated only at the vantage points. As a result, we achieved 25× improvements over the naive port of SPDK to the enclave.

3.3 Implementation

Direct I/O library Our direct I/O library for shielded execution extends Intel SPDK. Further, the memory management routines and the `uio` kernel module that maps the device memory to the user space are based on Intel DPDK [20]. Although the device DMA target is configured outside the enclave, the SSD device driver and library code, including BlobFS in which SPEICHER stores RocksDB files, entirely run within the enclave.

We use SPDK 18.01.1-pre and DPDK 18.02. In SPDK, 56 LoC are added, and 22 LoC are removed. In DPDK, 138 LoC are added and 72 LoC are removed. These changes were made to replace the routines that cannot be executed in the enclave.

Trusted counters AMCs are implemented using the Intel SGX SDK. A dedicated thread continually checks if any monotonic counter value has changed. If a counter value has been incremented, the thread writes the current value to the file. The storage engine can query the *stable value* of any of its counters, i.e., the last value that has been written to disk. Note that this value cannot be rolled back since it is protected by the synchronous SGX monotonic counter. Overall, our trusted counter consists of 922 LoC.

SPEICHER controller The SPEICHER controller is based on SCONE. We leverage the Docker integration in SCONE to seamlessly deploy SPEICHER binary on an untrusted host. Further, we implemented a custom memory allocator for the storage engine. The memory allocator manages the unprotected host memory, and exploits RocksDB's memory allocation pattern, which allows us to build a lightweight allocator with just 119 LoC. Further, the controller employs our direct I/O library on the data path, and the asynchronous `syscall` interface of SCONE on the control path

for system configuration. The controller also implements a TLS-based remote attestation for the clients [87]. Lastly, we integrated the trusted counter as a part of the controller, and exported the APIs to the storage engine.

Storage engine We implemented the storage engine by extending a version of RocksDB that leverages SPDK. In particular, we extended the RocksDB engine to run within the enclave, also integrated our direct I/O library. Since the RocksDB engine with SPDK does not support data encryption and decryption, we also ported encryption support from the regular RocksDB engine using the Botan Library [90] (1000 LoC). In addition to encrypting data files, we extended the encryption support to ensure the confidentiality of the WAL and Manifest files. We further modified the storage engine to replace the LSM data structure and log files with our secure MemTable, SSTables, and log files. Altogether, the changes in RocksDB account for 5029 new LoC and 319 changed LoC.

MemTables RocksDB as default uses a skip list for MemTable. However, it does not offer any authentication or freshness guarantees. Therefore, we replaced MemTable with an authenticated data structure coupled with mechanisms to ensure the freshness property. Our MemTable uses the `Inlineskiplist` of RocksDB and replaces the value part of the KV-pair with a node storing a pointer to and the size of the value as well as an HMAC. For the en-/decryption as well as for the HMAC we used OpenSSLs AES128 in GCM mode. This results in a 16 B wide HMAC. This implementation consists of 459 LoC. As discussed previously, we also implemented MemTable with a native Merkle tree (1186 LoC) and a Merkle tree with a prefix array (528 LoC). However, we did not use them eventually since their performance was quite low.

SSTables To preserve the integrity of the SSTable blocks, we changed the block layer in RocksDB to calculate the hash before it issues a write request to the underlying layer. The hash is then cached until the file is flushed (258 LoC). Thereafter, hashes of all blocks are appended to the file coupled with the information about the total number of blocks, and the hash of this footer. When a file is opened, our hash layer loads the footer into the protected memory and calculates the hash of the footer. It then compares the value against the hash stored in the Manifest file. Only if these checks are passed, it opens the corresponding SSTable file and normal operations proceed. At reading, the hash of the block is calculated and checked against the hashes stored in the protected memory area, before the block data is handed to the block layer of

RocksDB. We further enabled AES-128 encryption to ensure the confidentiality of the blocks (188 LoC). The hashes used in the SSTables are SHA-3 with 384 bit.

Log files Log files including the WAL and the Manifest use the same encryption layer as the SSTable files. However, the validation layer is different, and comes before the block layer since the operation requires knowledge of the record size. While writing, the validation layer adds the hash and the trusted counter value to the log files.

The validation layer uses the knowledge that log files are only read sequentially at startup for restoring purpose. Therefore, at the start up, the layer allows any action written in the log file as long as the hash is correct, and the stored counter increases as expected. At the end of the file, SPEICHER checks if the stored counter is equal to the trusted counter. The last record's freshness is guaranteed through the trusted counter. Integrity of all the records is guaranteed through the hash value protecting also the stored counter value. This value can then be checked against the expected counter value for that block. Since the counter lives longer than the log files, the start record value has to be secured too. In case of WAL, this is achieved by storing the start counter value of the WAL in the Manifest. The start record of the Manifest is implicitly secured, since the record must describe the state of the entire KV store.

3.4 Evaluation

Our evaluation answers the following questions.

- What is the performance (IOPS and throughput) of the direct I/O library for shielded execution? (§3.4.2)
- What is the impact of the EPC paging on the MemTable? (§3.4.3)
- What are the performance overheads of SPEICHER in terms of throughput and latency measurements? (§3.4.4)
- What is the performance of our asynchronous trusted counter? And what stability guarantees it has to provide to be compatible with modern KV stores? (§3.4.5)
- What is the I/O amplification overhead? (§3.4.6)

3.4.1 Experimental Setup

Testbed We used a machine with Intel Xeon E3-1270 v5 (3.60 GHz, 4 cores, 8 hyper-threads) with 64 GiB RAM running Linux kernel 4.9. Each core has private 32 KiB L1 and 256 KiB L2 caches, and all cores share a 8 MiB L3 cache. For the storage device our testbed uses a Intel DC P3700 SSD. The SSD has a capacity of 400 GB and is connected over PCIe 3.0 x4 with a sequential throughput of 2800 MB for reads and 2000 MB for writes, and 460 k IOPS random reads and 175 k IOPS for random writes. While this machine is outdated today, it was at the time of the experiments the most powerful and best representing single machine setup which supported SGX.

Methodology for measurements We compare the performance of SPEICHER with an unmodified version of RocksDB. The native version of RocksDB does not provide any security guarantees, i.e., it provides no support for confidentiality, integrity and freshness of the data and query operations.

Importantly, we stress-test the system by running a client on the same machine as the KV store. This is the worst-case scenario for SPEICHER since the client is not communicating over the network. Usually, the network slows down client's requests, and therefore, such an experimental setup is unable to stress-test the KV store. We avoid this scenario by running the client as part of the same process on the same host. This eliminates further the need for enclave enters and exits, which would add a high overhead, making a stress-test impossible.

Compiler and software versions We used the RocksDB version with SPDK support (git commit `3c30815`). We used SPDK version 18.01.1-pre (git commit `73fee9c`), which we compiled with DPDK version 18.02 (commit `92924b2`). The native version of SPDK/ DPDK and RocksDB was compiled with `gcc 6.3.0` and the default release flags. The SPEICHER version of SPDK/DPDK and RocksDB was compiled with the same release flags but `gcc version 7.3.0` of the SCONE project.

RocksDB benchmark suite We use the RocksDB benchmark suite for the evaluation. In particular, we used the `db_bench` benchmarking tool which is shipped with RocksDB [91] and Fex [92]. The benchmark consists of three workloads as shown in Table 3.1. Workload A is the default workload.

3.4.2 Performance of the Direct I/O Library

We first evaluate the performance of SPEICHER's I/O library for shielded execution. The I/O library is designed to have fast access to the persistent storage for accessing

Workload	Pattern	Read/Write ratio
A *(default)*	Read-write	90R—10W
B	Read-write	80R—20W
C	Read only	100R—0W

Table 3.1: RocksDB benchmark workloads.

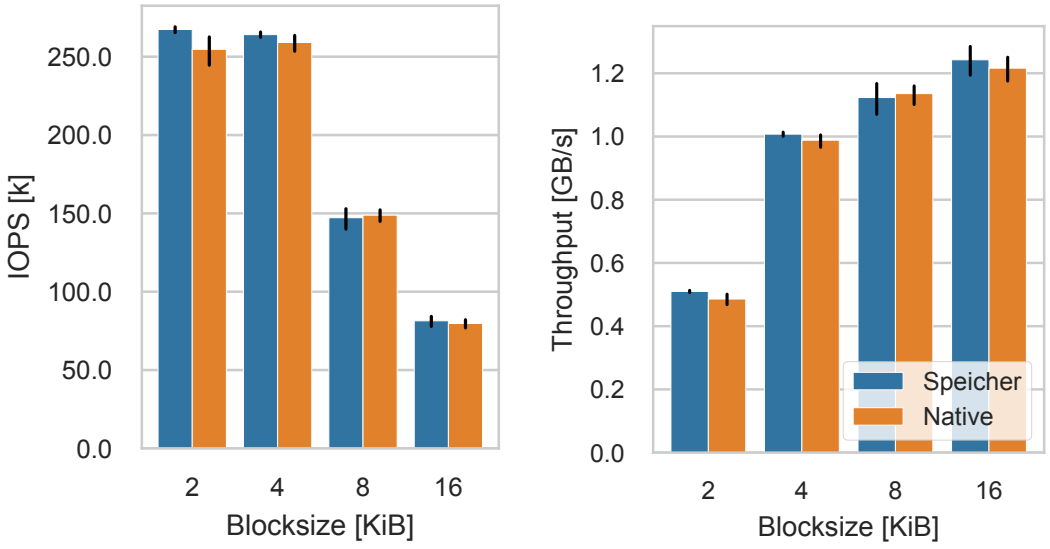

Figure 3.5: Performance of direct I/O library for shielded execution vs native SPDK.

the KV pair stored on the SSD (§3.2.2). We run the performance measurement 20 times for every configuration of block size for the native execution and SPEICHER. Figure 3.5 shows the mean throughput and IOPS with our I/O library and those with the native RocksDB-SPDK with a confidence interval of 95 %. We use Workload B (80 %R—20 %W). Since the communication between SPDK and the device is handled completely over DMA, our direct I/O library does not suffer from context switches. Additionally, due to storing the buffers outside of the enclave, we also do not require expensive EPC paging, which would drastically reduce the performance of the I/O library. Our performance evaluation of the direct I/O library shows that it does not suffer from any performance deprecation compared to the native SPDK setup. We were not able to perform precise latency measurements, due to the fact that SGXv1 does not allow to take a timestamp within the enclave, requiring an expensive world switch.

3.4.3 Impact of the EPC paging on MemTable

We next study the impact of EPC paging on MemTable(s). Note that a naive solution of storing a large or many MemTables in the EPC memory would incur high performance overheads due to the EPC paging. Therefore, we adopted a split MemTable approach, where we store only the keys along with metadata (hashes and pointers to value) inside the EPC, but the values are stored in the untrusted host memory (§3.2.3). To confirm the overheads of the EPC paging on accessing a large MemTable which are incurred in our rejected design, we measure the overheads of accessing random nodes in a MemTable completely resident in the enclave memory.

Figure 3.6 shows the performance overhead of accessing memory within the SGX enclave. The result shows that as soon as SGX has to page out MemTable memory from the EPC, which happens at 96 MiB, the performance drops dramatically. This is due to the en-/decryption and integrity checks employed by the MEE in Intel SGX. Therefore, it is important for our system design to keep the data values in the untrusted host memory to avoid the expensive EPC paging. Our approach of only keeping the key path of the MemTable inside the EPC requires a small EPC memory footprint. Therefore, our MemTable does not incur the EPC paging overhead.

3.4.4 Throughput and Latency Measurements

We next present the end-to-end performance of SPEICHER with different workloads, value sizes and thread counts. In our setup client and server are running inside the same enclave, as this is the worse case for SPEICHER. We measured the average throughput and latency for each of our benchmarks. Figures 3.7, 3.8, and 3.9 shows

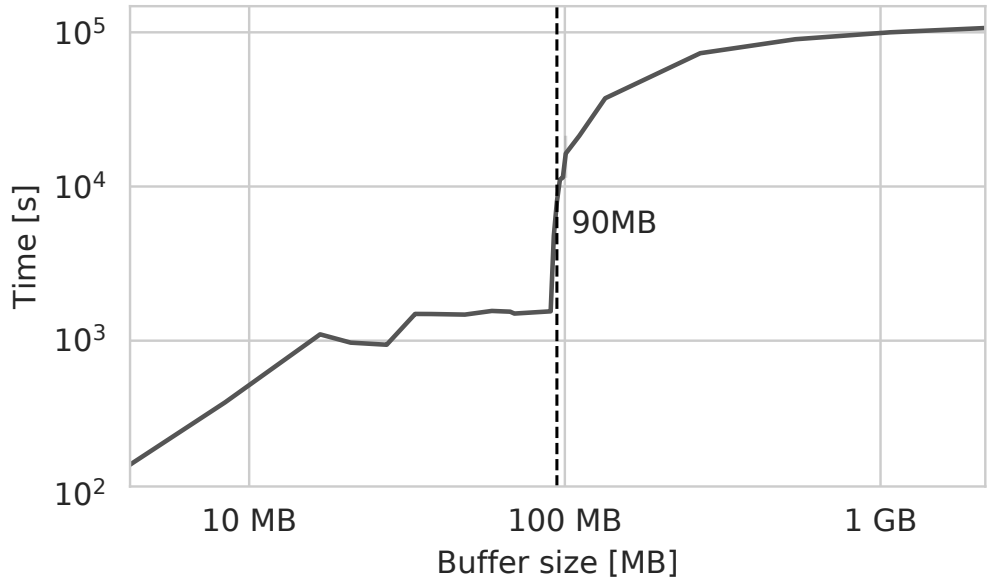

Figure 3.6: Impact on the random accessing time of EPC paging on the MemTable.

the measurement results as a ratio of slowdown to the native SPDK-based RocksDB for different read write ratios, different value sizes, and different number of threads respectively.

Effect of varying workloads In the first experiment, we used different workloads listed in Table 3.1. The workloads were evaluated with 5 million KV pairs each. Each key was 16 B and value was 1024 B. The benchmarks were run single threaded.

We get a throughput of 34.2 k request/second (rps) for Workload A down to 20.8 k rps for Workload C, while RocksDB archived 512.8 krps or 676.8 krps respectively. The results show that SPEICHER overheads 15–32.5 × for different workloads. The overheads in Workloads A and B are mainly due to the operations performed in the MemTable, since SPEICHER has to encrypt the value and generate a cryptographic hash for every write to the MemTable. Furthermore, for each read operation the data has to be decrypted and the hash has to be recalculated and compared to one in the skip list. However, even with AES-NI instructions, this decryption operation takes at least 1.3 cycles/byte for encryption, limiting the maximal reachable performance. The overhead in Workload C is due to reading a very high percentile of the KV pairs from the SSTable files, which uses currently an un-optimized code path for en-/decryption and hash calculations. We expect performance improvement by further optimizing the code path.

We also found in subsequent experiments that we could not fully eliminate

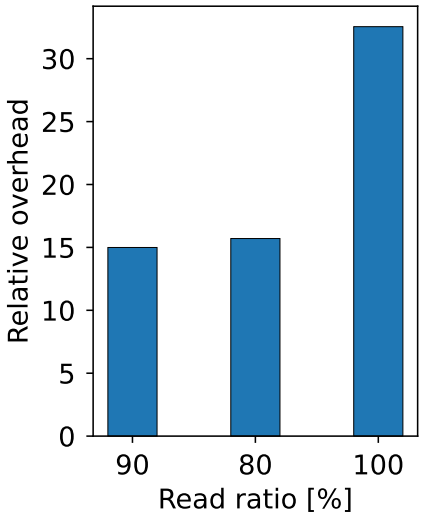

Figure 3.7: SPEICHER's performance normalized to the native RocksDB (with no security) for different workloads with a constant value size of 1024 B and constant number of 1 thread

paging in our setup. This might be further optimized with smaller configured MemTable. More modern TEE implementations do not suffer from the same strict memory limitation, therefore the overhead induced by paging should be drastically reduced in these systems.

Effect of varying byte sizes In the second experiment, we investigate the overheads with varying value sizes, since it changes the amount of data SPEICHER has to en-/decrypt and hash for each request. We used the default Workload A, and changed the value size from 64 B up to 4 KiB.

SPEICHER incurs an overhead of 6.7 × for small value size, i.e. 64 B, up to an overhead of 16.9 × for values of size 4 KiB. As in the previous experiment, the overhead is mainly dominated by the en-/decryption and hash calculation for the values in the MemTable. The benchmark shows a higher overhead for larger value sizes, since the amount of data SPEICHER has to en-/decrypt increases with the size of the values.

Effect of varying threads We also investigated the scaling capabilities of SPEICHER. For that we increased the number of threads up to 8 and compared the overhead to native RocksDB with the default Workload A. Note that the current SGX server machine has 4 physical cores / 8 hyperthread cores.

In the test the overhead increased from around 13.6 × for two threads to 17.5 × for 8 threads. This implies SPEICHER scales slightly worse than RocksDB. This is due

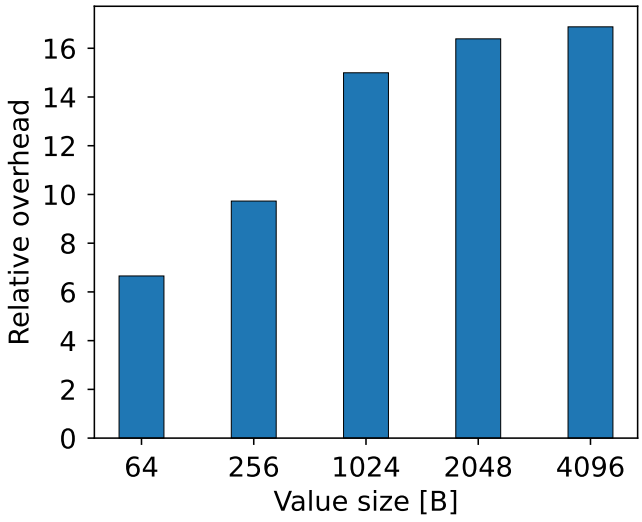

Figure 3.8: SPEICHER's performance normalized to the native RocksDB (with no security) for different value sizes with a constant workload of 90 % read and constant number of 1 thread

to less optimal caching for random memory access in SPEICHER's memory allocator. SPEICHER has to manage two different memory regions (host and EPC) for the MemTable, which leads to sub-optimal caching. We plan to optimize our memory allocator and data structures to exploit the cache locality.

Latency measurements In the benchmarks, SPEICHER has an average latency ranging from 16 µs for single threaded and 64 B value size up to 256 µs for 8 threads and 1024 B value size, native RocksDB had for the same benchmark a latency of 1.6 µs or 14 µs respectively. However, RocksDB's best latencies were in Workload C with an average of 1.5 µs. Note here that the average latency can be smaller than the latency of the underlying storage medium, if the key is in the MemTable.

3.4.5 Performance of the Trusted Counter

The synchronous trusted counter rate of SGX is limited to one increment at every 60 ms. This would limit our approach to only 20 `Put` operations per second since each `Put` has to be appended to the WAL, which requires a counter increment. However, our latency suggest that we have a lot more put operations to deal with. Even in our worse latency case with 256 µs per request we would expect 234.4 request per 60 ms, with a write rate of 10 % this would amount to 23.4 required counter increases every possible sequential counter increase. In practice SPEICHER should reach far higher update rates as this calculation used worst case values from our benchmarks.

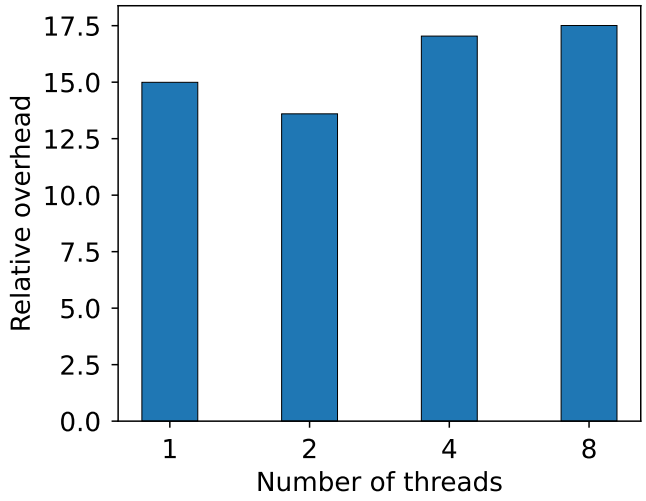

Figure 3.9: SPEICHER's performance normalized to the native RocksDB (with no security) for different number of threads with a constant workload of 90 % read and constant value size of 1024 B

KV store	Default time for persistence (ms)	Configurable
RocksDB	0 (flushing)	✓
LevelDB	0 (non-flushing)	✓
Cassandra	1000	✓
HBase	10 000	✓

Table 3.2: Default time for data persistence in KV stores.

Table 3.2 shows the time before different KV stores guarantee that the values are persisted. We argue that these times can be used to hide the stability time of our asynchronous counters, which is a maximum of 60 ms. This is far less than the maximum time to persist the data in the default configuration of Cassandra and HBase. If the client expects the value is persisted only after a specific period of time, we can relax our freshness guarantees to match to the same time window.

3.4.6 I/O Amplification

We measured the relative I/O amplification increase in data for SPEICHER compared to the native RocksDB. We report the I/O amplification results using the default workload (A) with the key size of 16 B and value size of 4 KiB. We observed an overhead of 30 % for read and write in the I/O amplification. This overhead mainly comes from the footer we have to add to each SSTable as well as from the hashes and counter values we have to add to the log files. This overhead is not only present in

the write case but also in the read, as the additional data has also to be read to be able to verify the files.

3.5 Related Work

I/O for shielded execution To mitigate the I/O overheads in SGX, shielded execution frameworks, such as Eleos [38] and SCONE [40], proposed the usage of an asynchronous system call interface [76]. While the asynchronous interface is sufficient for the low I/O rate applications—it can not sustain the performance requirements of modern storage/networked systems. To mitigate the I/O bottleneck, ShieldBox [93] proposed a direct I/O library based on Intel DPDK [20] for building a secure middlebox framework. Our direct I/O library is motivated by this advancement in the networking domain. However, we propose the first direct I/O library for shielded execution based on Intel SPDK [16] for the I/O acceleration in storage systems.

Trusted counters A trusted monotonic counter is one of the important ingredients to protect against rollback and equivocation attacks. In this respect, Memoir [77] and TrInc [79] proposed the usage of TPM-based [94] trusted counters. However, TPM-based solutions are quite impractical because of the architectural limitations of TPMs. For instance, they are rate-limited (only one increment every 5 seconds) to prevent wear out. Therefore, they are mainly used for secure data access in the offline settings, e.g., Pasture [95].

Intel SGX has recently added support for monotonic counters [78]. However, SGX counters are also quite slow, and they wear out quickly (§3.2). To overcome the limitations, ROTE [85] proposed a distributed trusted counter service based on a consensus protocol. Likewise, Ariadne [89] proposed an optimized technique to increment the counter by a single bit flip. Our asynchronous trusted counter interface is complimentary to these synchronous counter implementations. In particular, we take advantage of the properties of modern storage systems, where we can use these synchronous counters to support our asynchronous interface.

Policy-based storage systems Policy-based storage systems allow clients to express fine-grained security policies for data management. In this context, a wide range of storage systems have been proposed to express client capabilities [96], enforce confidentiality and integrity [97], or enable new features that include data sharing [98], database interface [99], policy-based storage [100, 101], or policy-based

data seal/unseal operations [102]. Amongst all, Pesos [103] is the most relevant system since it targets a similar threat model. In particular, Pesos proposes a policy-based secure storage system based on Intel SGX and Kinetic disks [104]. However, Pesos relies on trusted Kinetic disks to achieve its security properties, whereas SPEICHERtargets an untrusted storage, such as an untrusted SSD. Secondly, Pesos is designed for slow trusted HDDs, where the additional overheads of the SGX-related operations are eclipsed by slow disk operations. In contrast, SPEICHERis designed for high-performance SSDs.

Secure databases/datastores Encrypted databases, such as CryptDB [105], Seabed [106], Monomi [107], and DJoin [108], are designed to ensure the confidentiality of computation in untrusted environments. However, they are primarily for preserving confidentiality. In contrast, SPEICHER preserves all three security properties: confidentiality, integrity, and freshness.

EnclaveDB [109] and CloudProof [110] target a threat model and security properties similar to SPEICHER. In particular, EnclaveDB [109] is a shielded in-memory SQL database. However, it uses the secondary storage only for checkpoint and logging unlike SPEICHER. Hence, it does not solve the problem of freshness guarantee for the data stored in the secondary storage. Furthermore, the system implementation does not consider the architectural limitations of SGX. Secondly, CloudProof [110] is a key-value store designed for untrusted cloud environment. Unlike SPEICHER, it requires the clients to encrypt or decrypt data to ensure confidentiality, as well as to perform attestation procedures with the server, introducing a significant deployment barrier.

TDB [111] proposed a secure database on untrusted storage. It provides confidentiality, integrity, and freshness using a log-structured data store. However, TBD is based on a hypothetical TCB, and it does not address many practical problems addressed in our system design.

Obladi [112] is a KV store supporting transactions while hiding the access patterns. While it can effectively hide the values and their access pattern against the cloud provider, it needs a trusted proxy. In contrast, SPEICHER does not rely on a trusted proxy. Furthermore, Obladi does not consider rollback attacks.

Lastly, in parallel with our work, ShieldStore [113] uses a Merkle tree to build a secure in-memory KV store using Intel SGX. Since ShieldStore is an in-memory KV Store, it does not persist the data using the LSM data structure unlike SPEICHER.

Authenticated data structures Authenticated data structures (ADS) [114] enable efficient verification of the integrity of operations carried out by an untrusted entity. The most relevant ADS for our work is mLSM [115], a recent proposal to provide integrity guarantee for LSM. In contrast to mLSM, our system provides stronger security properties, i.e., we ensure not only integrity, but also confidentiality and freshness. Furthermore, our system targets a stronger threat model, where we have to design a secure storage system leveraging Intel SGX.

Robust storage systems Robust storage systems provide strong safety and liveness guarantees in the untrusted cloud environment [116–118]. In particular, Depot [116] protects data from faulty infrastructure in terms of durability, consistency, availability, and integrity. Likewise, Salus [117] proposed a block store robust storage system while ensuring data integrity in the presence of commission failures. A2M [118] is also a robust system against Byzantine faults, and provides consistent, attested memory abstraction to thwart equivocation. In contrast to SPEICHER, this line of work neither provides confidentiality nor freshness guarantees.

Secure file systems There is a large body of work on software-based secure storage systems. SUNDR [119], Plutus [120], jVPFS [121], SiRiUS [122], SNAD [123], Maat [124] and PCFS [97] employ cryptography to provide secure storage in untrusted environments. None of them protect the system from rollback attacks, and our challenges to overcome overheads of shielded execution are irrelevant for them. Among all, StrongBox [125] provides file system encryption with rollback protection; however, it does not consider untrusted hosts.

3.6 Summary

This chapter presents SPEICHER, a secure persistent LSM-based KV storage system for untrusted hosts. SPEICHER targets all the three important security properties: strong confidentiality and integrity guarantees, and also protection against rollback attacks to ensure data freshness. We base the design of SPEICHER on hardware-assisted shielded execution leveraging Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure enclave memory region to ensure that the security properties are also preserved in the stateful setting of an untrusted storage medium.

To achieve these security properties while overcoming the architectural limitations of Intel SGX, we have designed a direct I/O library for shielded execution,

a trusted monotonic counter, a secure LSM data structure, and associated algorithms for storage operations. We implemented a fully-functional prototype of SPEICHER based on RocksDB, and evaluated the system using the RocksDB benchmark. Our experimental evaluation shows that SPEICHER achieves reasonable performance overheads, between 15 and 32.5 $\times$, while providing strong security guarantees.

While some applications may find the performance overhead of SPEICHER to be too high, it is important to recognize that certain domains, such as medical or financial sectors (e.g., a digital currency used for trading between central banks), may be willing to accept this trade-off. This is particularly true when compared to alternative solutions like homomorphic encryption, which often impose significantly higher overheads. Alternatively, setting up and managing one's own data center brings its own challenges related to scalability, availability, and maintainability.

SPEICHER serves as a fundamental building block that, when combined with other technologies discussed in Chapter §4 and Chapter §5, enables the creation of a modern trusted storage system. By leveraging these complementary projects, a comprehensive and robust storage solution can be constructed.

Chapter 4

AVOCADO: A Secure In-Memory Distributed Storage System

We introduce AVOCADO, a secure in-memory distributed storage system that provides strong security, fault-tolerance, consistency (linearizability) and performance for untrusted cloud environments. AVOCADO achieves these properties based on TEEs, which, however, are primarily designed for securing limited physical memory (enclave) within a single-node system. AVOCADO overcomes this limitation by extending the trust of a secure single-node enclave to the distributed environment over an untrusted network, while ensuring that replicas are kept consistent and fault-tolerant in a malicious environment.

To achieve these goals, we design and implement AVOCADO underpinning on the cross-layer contributions involving the network stack, the replication protocol, scalable trust establishment, and memory management. AVOCADO is practical: In comparison to BFT, AVOCADO provides confidentiality while using f fewer replicas than BFT, but it is also faster — 4.5× to 65× for YCSB read and write heavy workloads, respectively.

4.1 Motivation

In-memory distributed key-value stores (KVS) [24–29, 32–34, 126, 127] have been widely adopted as the underlying storage system infrastructure in the cloud because *(i)* they support latency sensitive applications by keeping data in main memory, and *(ii)* they are able to accommodate large datasets beyond the memory limits of a single server by adopting a scale-out distributed design.

At the same time, the transition to the cloud has increased the risk of security violations in storage systems [65]. In untrusted environments, an attacker can compromise the security properties of the stored data and query operations. In fact, several studies [66, 83, 128] show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems. Further, a malicious cloud operator or co-located tenant, presents an additional attack vector [129, 130].

To address these security threats, hardware-assisted trusted execution environments (TEEs), such as Intel SGX [5], ARM Trustzone [7], RISC-V Keystone [10, 131], and AMD-SEV [9] provide an appealing way to build secure systems. In particular, TEEs provide a hardware-protected secure memory region whose residing code and data are isolated from any layers in the software stack including the OS/ hypervisor. Given this promise, TEEs are now commercially offered by major cloud computing providers [11–13].

Although TEEs provide a promising building block for securing systems against a powerful adversary, they also present significant challenges while designing a *replicated secure distributed storage system*. The fundamental issue is that the TEEs are primarily designed to secure the limited in-memory state of a single-node system, and thus, the security properties of TEEs do not naturally extend to a distributed infrastructure. Therefore we ask the question: *How can we leverage TEEs to design a high-performance, secure, and fault-tolerant in-memory distributed KVS for untrusted cloud environments?*

In this chapter we introduce AVOCADO, a secure, distributed in-memory KVS based on Intel SGX [5] as the foundational TEE that achieves the following properties: (a) strong *security*, in particular, *confidentiality* - unauthorized reads are prevented, and *integrity* — unauthorized changes to the data are detected, (b) *fault tolerance* — the service continues uninterrupted in the presence of faults, (c) *consistency* — strong consistency semantics for a replicated store (linearizability), while protecting against roll back and forking attacks and (d) *performance* — achieving all of these without compromising performance.

To achieve the aforementioned properties, we need to address the following four design challenges pertaining to the network stack, the replication protocol, trust establishment, and memory management in TEEs.

Firstly, in-memory distributed KVSs are increasingly build on high-performance network stacks, where they bypass the kernel using direct I/O [19, 20, 33]. Unfortunately, the prominent I/O mechanism employed by TEE frameworks [38, 40, 132] is based on asynchronous system calls [76], which exhibit significant

overheads [133]. On the other hand, the direct I/O mechanism is fundamentally incompatible with TEEs as the data stored within the protected memory of TEEs cannot be directly accessed by the DMA engine.

To address this challenge, we design a high-performance network stack for TEEs based on eRPC [19] — it supports the complete transport and session layers, while enabling direct I/O within the protected TEE domain. Our network stack outperforms asynchronous syscall by 66 % for `iperf` (§ 4.5.1).

Secondly, in-memory distributed KVSs rely on data replication for fault tolerance. To ensure replicas are consistent in the presence of faults and adversary, a secure replication protocol is deployed. While conventional wisdom requires the employment of BFT protocols [134, 135], they are prohibitively expensive for practical systems [136].

To overcome the limitation, we design a secure replication protocol, which builds on top of any high-performance non-Byzantine protocol [137, 138] - our key insight is to leverage TEEs to preserve the integrity of protocol execution, which allows to model Byzantine behavior as a normal crash fault. Our replication protocol offers linearizable reads and writes, and outperforms BFT [139] by a factor of 4.5–65 $\times$, while requiring f fewer replicas and stronger security properties (§ 4.5.2).

Thirdly, a secure distributed system requires a scalable attestation mechanism to establish trust between the servers and clients. Unfortunately, the remote attestation mechanism in TEEs is designed for establishing root of trust for a single node [140] and it does not provide collective trust establishment across the multiple nodes of a distributed system. Moreover, the attestation itself is based on Intel attestation service (IAS) [86, 141], which suffers from scalability and latency issues. These issues are especially problematic in a distributed KV store, which can get scaled up and down depending on current demand.

To address this, we design a configuration and attestation service (CAS) that ensures scalability and flexibility in a distributed environment. Further, it provides configuration management, and improved performance of 18.3 $\times$ compared to Intel's IAS attestation (§ 4.5.3).

And lastly, an in-memory distributed KVS requires fast access to large amount of main memory on each server for single-node KVS. Unfortunately, TEEs provide a limited secure physical memory, and rely on prohibitively expensive paging mechanism to access data beyond the physical limit.

To address this limitation, we design a novel single-node KVS based on a partitioned skip list data structure, which overcomes the memory limitations of TEEs, while supporting lock-free scalable concurrent updates. Our KVS provides fast lookup speed; 1.5–9 × faster than ShieldStore [113], a state-of-the-art secure KVS for single-node systems (§ 4.5.4).

Based on these aforementioned four contributions, we build AVOCADO as an end-to-end system from the ground-up, and evaluate it using a real hardware cluster using the YCSB [18, 142] workloads. Our evaluation shows that AVOCADO is scalable and performs similar in read heavy and write heavy workloads: AVOCADO suffers only 50 % slowdown compared to a non-secure distributed KVS (§ 4.5.5), which is an order of magnitude better than the state-of-the-art secure distributed storage systems providing strong consistency.

Limitations: AVOCADO requires a large trusted computing base (TCB) compared to other work using TEE to provided secure replication [143–145]. While BFT protocols can handle implementation errors, AVOCADO cannot and requires the TCB to be implemented correctly. Further, we do not aim to protect against side-channel attacks and access or network pattern attacks [82, 83, 146, 147]. Protecting against these attacks is outside of the scope of AVOCADO.

4.2 System Model

AVOCADO divides the key space into shards. Each shard is replicated over a configurable number of nodes, which are connected over a high speed network. A client issuing a `Put`, `Get`, or `Delete` operation selects the shard associated with the key and chooses a *request coordinator* from the list of nodes. The nodes will coordinate with each other to provide proof for the success of the operation. For `Get` operations proofs of integrity, authenticity, consistency and non-/existence need to be provided, too.

Data model AVOCADO provides confidentiality, integrity, authenticity and strong consistency for the stored data. Specifically, a server only acknowledges a request as long as it can prove the following guarantees: 1) an adversary cannot read or manipulate stored data, without the manipulation being detected, 2) the servers can establish trust with each other and the clients and 3) an operation always observes the latest completed operation on the same key, e.g., a `Get` observes the latest `Put`.

Threat model AVOCADO targets an extended threat model beyond the conventional model assumed for single-node shielded execution [39]. In line with the default threat model of SGX, we assume that an adversary has full control over the hardware and software stack of the provided system, including OS and hypervisor. Further, the adversary has the ability to gain full control over the network infrastructure and can drop, delay, or manipulate network traffic. In contrast to BFT protocols, we assume that adversary cannot take advantage of faults in the implementation of SGX or KVS. Moreover, AVOCADO does not protect against side-channel attacks [81, 146–152]. AVOCADO also does not provide mechanisms against access pattern attacks [82, 83]. Lastly, we also do not protect against memory safety vulnerabilities in our implementation [153, 154].

Fault model We assume an asynchronous model with network and crash-stop failures. The network can be manipulated by the attacker, thus, we assume that message transmission delays can be unbounded, network packets can be reordered, lost or duplicated. We do not assume the existence of synchronized clocks. Individual processes might fail by crashing, but do not operate in a Byzantine manner (because of trusted execution in the nodes). Since the network is controlled by the attacker, AVOCADO cannot provide any availability guarantees. However, as long as there is not a denial-of-service attack on the network, AVOCADO will remain available while a majority of processes remain alive (tolerating f failures).

4.3 Design

AVOCADO, as a distributed KVS, runs on a set of nodes, each of which has to continuously guarantee the confidentiality, integrity and authenticity of the stored data as well as the sent/received messages. As shown in Figure 4.1, each node consists of four major components. On the top, a configuration and attestation service (CAS) runs to provide and speed up the trust establishment between the nodes and the clients. Additionally, AVOCADO guarantees fault tolerance as well as consistency between the replicated nodes thanks to an asynchronous replication protocol. We implement this replication protocol using our secure network stack. Further, the network stack securely sends and receives messages, ensuring packet security. Finally, the single-node memory KVS stores the dataset, containing all user provided data and ABD's timestamps.

Following, we will discuss the four major components, i.e. network stack, replication protocol, CAS, and in-memory KVS, in more detail.

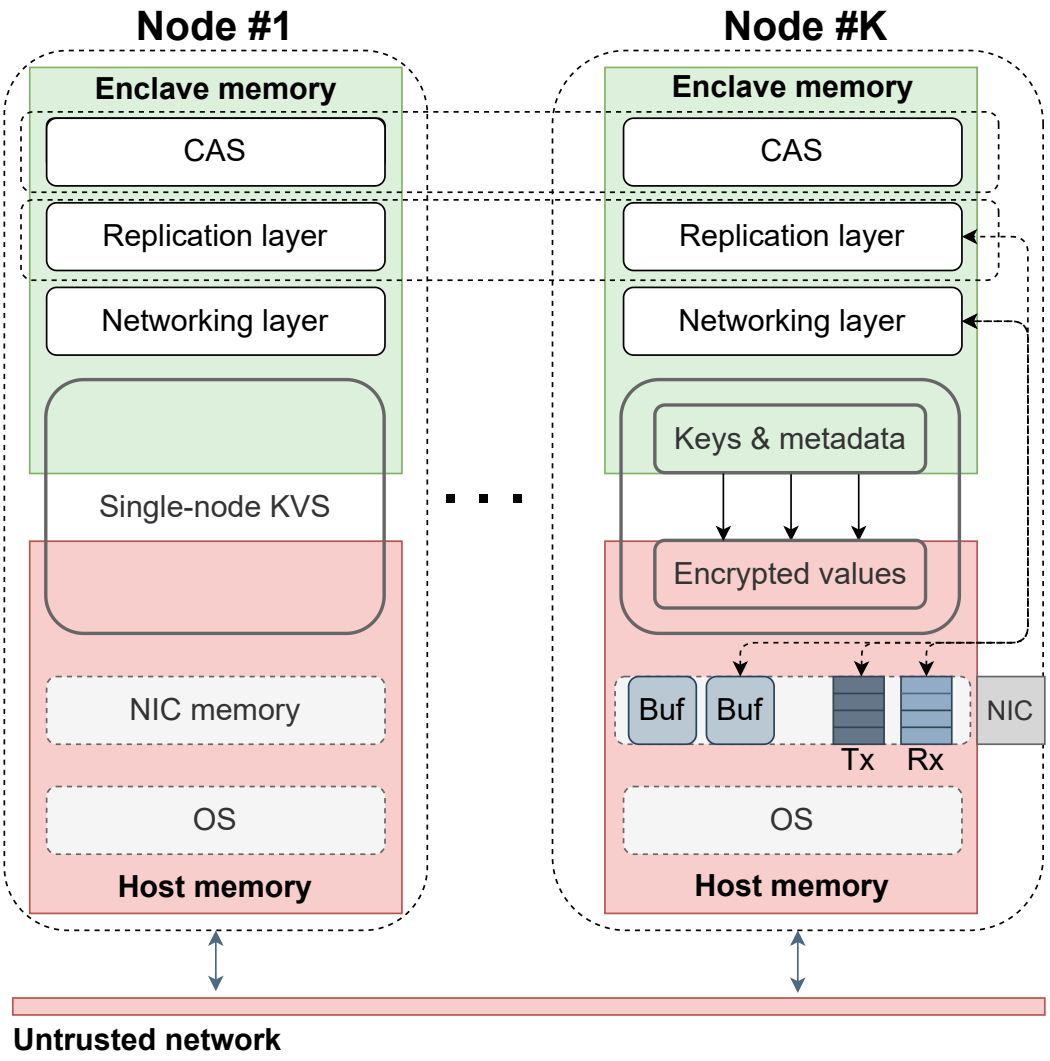

Figure 4.1: AVOCADO's system overview.

4.3.1 Network Stack

Problem High-performance networking based on direct I/O mechanisms (e.g., RDMA and DPDK) is an essential ingredient to design a distributed in-memory KVS. The networking layer is imperative to support high-performance synchronous replication. Unfortunately, mapping devices into TEEs trusted memory is incompatible to the security guarantees, since the device is untrusted. Unfortunately, direct I/O mechanisms are incompatible with TEEs since they require direct DMA accesses from an untrusted source to the protected memory region in TEEs.

Additionally, the synchronous socket syscall I/O is limiting as it requires the expensive world switch in the TEEs (the world switch is around 5.5× more expensive than a kernel context switch; 10 170 cycles compared to 1800 cycles [133]).

To prevent the expensive world switches, asynchronous syscall mechanisms [76]

have been adopted by shielded execution frameworks, such as SCONE [40] or Eleos [38]. Although the asynchronous syscall mechanism helps in mitigating the expensive world switch in TEEs, it is not well-suited for AVOCADO since the system call overhead as well as copying data in/out the enclave memory are not avoided. For example, in our evaluation in Section 4.5.1 we prove that the exit-less asynchronous socket-based networking is poor choice compared to a user space approach for AVOCADO.

Solution To overcome this limitation, we opted for a new network stack based on the user space direct I/O networking approaches (e.g., RDMA and DPDK), offering a secure implementation of the transport and session layer in the OSI model. However, we need to tackle the fact that untrusted resources/memory cannot be mapped into the enclave memory. To address this, our network stack maps the DMA, and therefore, message buffers into the untrusted host memory, which is accessible by the enclave.

Shielded network stack. For our shielded network stack, we use eRPC [19] on top of DPDK [20]. To strengthen AVOCADO's security properties and eliminate world switches, we also map all eRPC's and DPDK's software stack to the enclave address space by leveraging SCONE. Therefore, the logic, i.e., code, *lives* completely within the enclave while the networking buffers (e.g., message buffers, network protocol buffers, Tx and Rx queues) remain in host memory since SGX will not allow accessing enclave memory from the NIC. As a result, we map untrusted host memory to both NIC and network buffers required by eRPC and we utilise `hugepages` memory of 2 MB-pages to boost packet processing (e.g., eliminate page walks, exploit data locality, minimize swapping, and increase TLB hit rate).

As shown in Figure 4.2, the submission and reception of requests and responses mandate the allocation of message buffers. To transmit the message buffer's data, eRPC needs to copy the data to a `rte_mbufs` in the Tx queue which is allocated by DPDK library and also resides in `hugepages` area. However, before that, a header that contains the transport header, and metadata (request handler type, sequence numbers, etc.) is added to the front of the packet. Specifically, eRPC library adds the UDP protocol header while the DPDK library is responsible for the Ethernet protocol header.

Upon a request's reception, a specific handler for the type of the request is invoked. The Rx queue's elements are pointers to the address of the received data. In case the packet is smaller than the `MTU` (1500 B in our case), we perform zero-copy reception by mapping the data address to the message buffer associated for that request. Our networking stack splits big packages ($>$ `MTU`) into a set of ordered `MTU`-size smaller

messages and delivers them in order—we guarantee to order by unique monotonic sequence ids. The first (sub)-message contains all the necessary metadata (e.g. the size of the original message). That way, if a message is lost, our library identifies the missing part and only the lost message is re-transmitted. Lastly, note that each user thread owns a separate RPC object which owns distinct Tx and Rx queues allowing that way multithreaded concurrent operations.

Encryption and message format. To sum up, AVOCADO efficiently eliminates the world switches, establishes a direct communication with the device bypassing the kernel network stack, attacks the limited enclave memory and promotes parallelism. However, by putting these buffers outside the hardware protected area, AVOCADO has to ensure the integrity and confidentiality for all network data. Towards this direction, we implemented an en-/decryption library (using hardware support for AES-GCM-128). Each call or return from eRPC goes through this en-/decryption layer which also checks the integrity of the transmitted data.

Figure 4.3 shows the message format of our AVOCADO-networking layer. For each transmitted packet, the encryption layer builds a payload, which contains a 12 B IV, 8 B operation identifier, 8 B for the key size, 8 B for the size of the entire package then the KV pair with the Lamport clock (§ 4.3.2). The generated payload is followed by a 16 B MAC which is necessary to prove the authenticity and integrity upon reception in the remote host. The operation identifier also contains a unique id for the current request/response, allowing to detect resend packages by an attacker. Replicas are trusted and all replicas authenticate each other in the boot up step. Further, they make a key exchange, therefore we can use this together with the unique id to authenticate each message. By encrypting and authenticating our packages we can deal with security concerns raised for user space networking [155, 156].

Result In Section 4.5.1 we show that our user space shielded network stack based on eRPC outperforms the kernel approach based on sockets up to 1.66×.

4.3.2 Replication Protocol

Problem Distributed systems enforce consistency in the face of faults through replication protocols that establish an order of operations in a replicated environment, preventing data corruption and loss. We strive for linearizability [157], the strongest guarantee from a programmability perspective, which mandates that each request appears to take effect globally and instantaneously at some point between its invocation and completion. Additionally, we strive to provide two often contradictory properties: security and performance. Conventional wisdom suggests

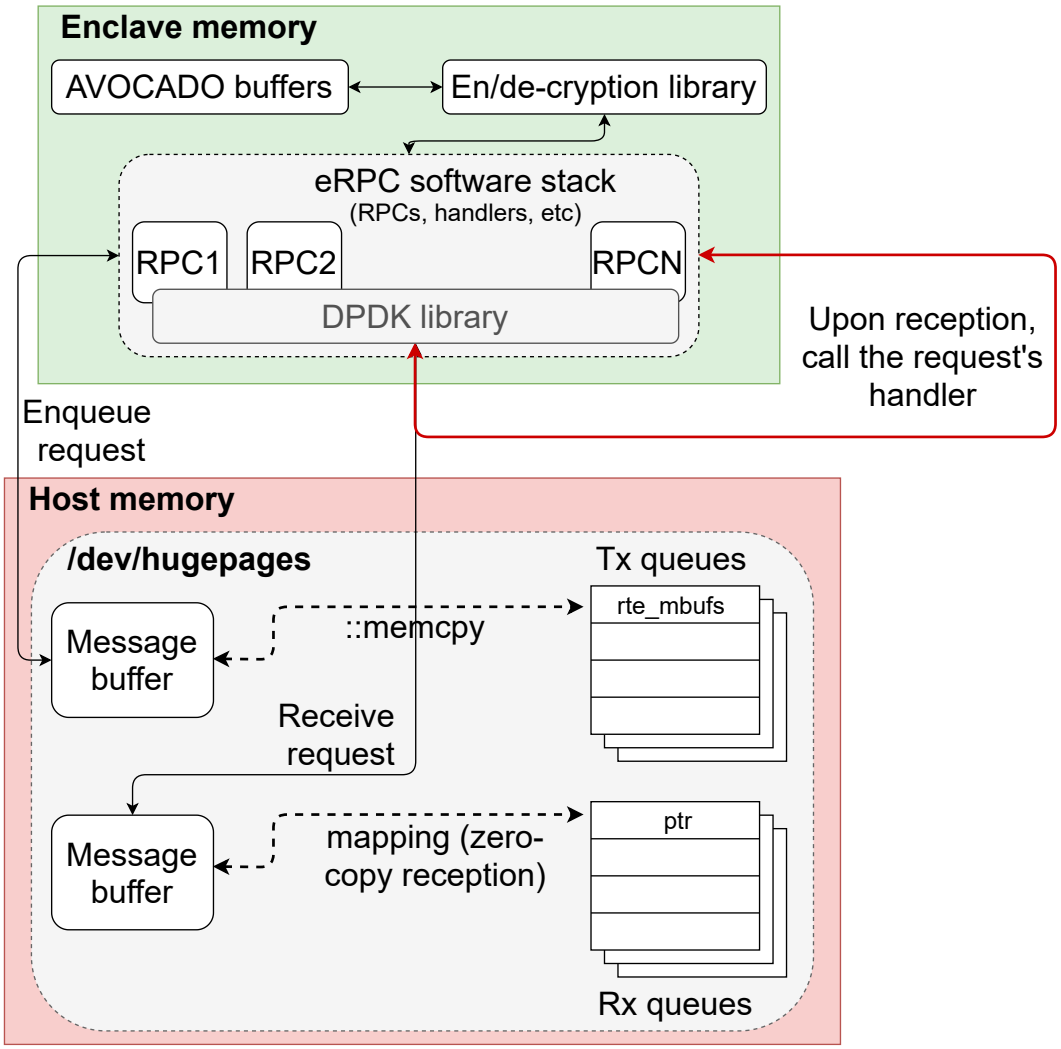

Figure 4.2: AVOCADO's network stack.

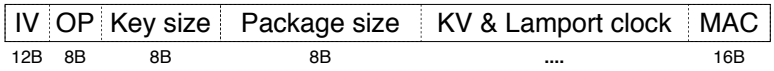

Figure 4.3: AVOCADO's message format.

Protocols	Linearizability	Integrity	Confidentiality	Replication factor	Max compromised nodes
Non-Byzantine	✗	✗	✗	$2f+1$	0
BFT	✓	✓	✗	$3f+1$	f
BFT + TEEs	✓	✓	✓	$3f+1$	All
AVOCADO	✓	✓	✓	$2f+1$	All

Table 4.1: The landscape of replication protocols in the untrusted environment. This table compares theoretical systems with different protocols against AVOCADO. Thereby, all the systems utilize a secure single-node KVS, however only the execution of BFT + TEE is protected. We assume f compromised nodes for linearizability, integrity and confidentiality columns.

the use of BFT protocols [134, 135] since they provide a secure consensus protocol in a malicious environment. However, their performance suffers from their overly pessimistic assumptions. On the other hand, non-Byzantine replication protocols, such as ABD [138, 158], chain replication [137] or Raft [159], perform better than BFT, but cannot tolerate a malicious environment.

Solution Since AVOCADO assumes a malicious environment, BFT [134, 135] protocols could be deployed to deal with malicious responses. Prior work uses trusted components to increase the performance of BFT protocols by detecting equivocation [143, 160]. However, our assumption of the system differs from BFT. In contrast to BFT, we assume that enclaves will respond correctly, preventing equivocation. Furthermore the TEE is able to preserve the integrity of the protocol execution. Therefore, we assume a weaker adversarial model, as the adversary cannot arbitrarily change the execution from a node. This allows us to model a Byzantine behavior as a normal crash fault. As a result, we can adopt a non-BFT replication protocol, which deals with crash faults. Thereby, our design greatly increases the performance by avoiding the additional broadcast rounds required by BFT, while also reducing the required nodes to tolerate f failures. In Table 4.1 we compare security guarantees of different protocols with and without TEEs.

In AVOCADO we build our replication protocol on the well established multi-writer ABD [138] protocol. (From now on "ABD".) By choosing ABD, we can also guarantee protection against forking and rollback attacks. ABD requires a majority of nodes to acknowledge each operation, guaranteeing that at least one replica involved in the operation has observed the most recent operation on the same key. This further guarantees liveness in case of network partitioning as long as a majority of nodes are in the same partition. While we do not change the replication-related behavior of the original ABD protocol, we design a secure

replication protocol based on our network stack (§ 4.3.1). In the following we describe the important operations of AVOCADO.

#I: Put In a `Put` operation the client will determine, by hashing the key and looking up the nodes, the set of nodes responsible for the key. They, then, send the `Put` to a randomly selected replica, which will act as the `Put`'s coordinator.

The chosen request's coordinator will prepare it's own KVS by preparing the local put operation, however it will not make the local put visible for other operations until the replicated `Put` operation is completed. This reduces EPC pressure, since the value doesn't have to be cached in enclave memory before it can be inserted into the nodes KVS. An example of the AVOCADO's `Put` request is shown in Figure 4.4. The coordinator, first, executes the first of two broadcast rounds. All replicas store the key-value along with its Lamport clock to determine an order of operations. The Lamport clock consists of a logical counter and a machine id. This id guarantees that only one machine can generate a specific clock value. In the first broadcast round, the coordinator requests the timestamps that are stored in the replicas for that key. All replicas lookup the key in their in-memory KVS, to find their stored timestamp. Crucially, the replicas do not have to make an authenticity and integrity check on the timestamp, as the Lamport clock is stored as part of the metadata in enclave memory. Upon receiving a majority of the remote timestamps (including its own locally stored timestamp), the coordinator creates the timestamp of the new `Put`, by incrementing the highest of the received timestamps and concatenating its own node-id. Finally, it broadcasts the new KV pair along with its new timestamp to all replicas, which insert the KV pair into their in-memory KVS. Since the put operation does not return the value to the user, and the meta data is protected by the enclave AVOCADO does not have to check the authenticity and integrity of the old value. Upon gathering a majority of acknowledgements it reports completion to the client.

#II: Get The `Get` operation is similar to `Put`; the client sends its request to a randomly selected server, which coordinates it. The server then looks up the KV-pair in its local store.

The chosen request coordinator executes one broadcast round. In certain cases a second, optional broadcast round is required. Similarly, to the first round of a `Put`, the first round of a `Get` finds out the highest timestamp for that key when the majority of replicas has responded. This action guarantees that the `Get` will observe any completed `Put` (recall that a `Put` only completes if it reaches a majority of replicas). The replicas will respond with their locally stored value and corresponding Lamport

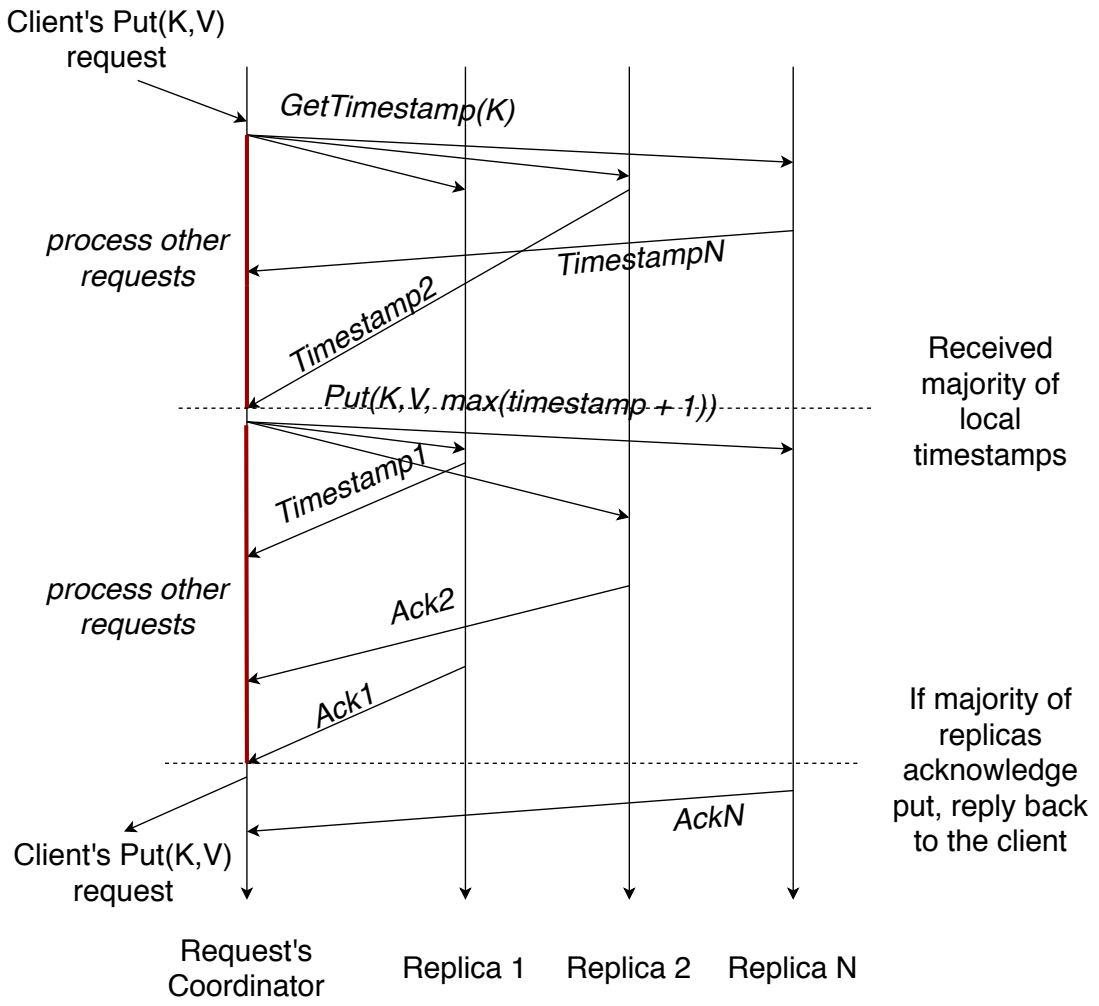

Figure 4.4: Example of `Put` request in AVOCADO protocol.

timestamp to the coordinator, this involves a lookup in the local KVS and decryption together with integrity and authenticity checks of the value. The `Get` always returns to the client the value that corresponds to the highest timestamp found in its first round. However, the coordinator can reply to the client if, based on the replies it received on its first round, it can guarantee that a majority of replicas are aware of this value. Otherwise it must perform a second broadcast round.

The second broadcast round is identical to the second write of a `Put`: it shares the KV-pair along with its timestamp with all replicas. Completion of the `Get` is reported to the client only after gathering a majority of acks. The second round of the `Get`, ensures that a `Get` not only observes the latest completed `Put`, but also guaranteeing that the `Put` will be visible to all subsequent `Gets`.

#III Delete Delete is supported by issuing a `Put` operation with an empty value. This will remove the value from the KVS, but importantly it will not remove the key. We

need to keep the key and the corresponding Lamport clock, to be able to establish an order of operation if a future `Put` operation accesses the same key.

#IV: Fault tolerance AVOCADO remains highly available in the face of machine failures. However, as nodes fail, new nodes must be added, to ensure that the deployment always includes a majority of live nodes. In order to ensure that machines can safely join the configuration, we deploy a recovery algorithm inspired by Hermes [161].

Specifically, when adding a new node all other live replicas are notified of the new node's intention to join the replica group. The new node starts operating as a *shadow replica* that participates in all Put-related broadcast rounds (of remote replicas), but it cannot yet become the coordinator of a client request. Furthermore, the shadow replica does not take part in the Get quorum. In the meantime, the shadow replica reads chunks (multiple keys) from other replicas to fetch the latest values and reconstruct the KVS. To archive this the shadow replica is using the first broadcast round of ABD, but it never executes the second round, because it does not need to notify other replicas of what it read. After reading the entire KVS, the shadow replica is up-to-date and transitions to operational state, whereby it is able to serve client requests.

Result We compare AVOCADO against BFT and Raft in Section 4.5.2. Our evaluation shows that AVOCADO is between 4.5 and 65 $\times$ faster than BFT-Smart [139], a state of the art BFT implementation.

4.3.3 Configuration and Attestation Service

Problem To ensure the integrity of the code and data deployed in the remote hosts with TEEs, TEEs, such as Intel SGX, provide attestation mechanisms. Secrets (e.g., certificates, encryption keys, etc.) are provided only after the attestation. Once an enclave is initialized, an attestation process can be launched to verify the integrity of code and data inside the enclave and proves the enclaves identity to a remote party.

Intel SGX uses an architecture Platform Service Enclave (PSE) called *Quoting Enclave* to sign the report of the loaded enclave [37, 86]. The remote verifier forwards this signed report to the Intel Attestation Service (IAS). Thereafter, IAS confirms or refuses the authenticity of the report to the verifier.

This conventional attestation mechanism using IAS incurs significant overhead in a distributed setting, especially for elastic computing or fault tolerance. The reason is that every time a distributed system (e.g., a distributed KVS) spawns a new enclave,

it needs to perform the remote attestation via IAS which is not necessarily hosted in the same data center, incurring high latency. Lastly, and importantly, cloud providers usually do not want to disclosure their hardware or cluster information, as this information might be confidential.

Intel has also realized the problems occuring over the IAS and offers Intel Data Center Attestation Primitives (DCAP) [162], which allows to create an in data center caching service. This caching service can be used to sign and verify quotes. However, Intel DCAP does not provide a secure configuration service.

Solution In AVOCADO, we designed a decentralized configuration and attestation management system (CAS) for distributed SGX-based applications.

By consolidating and expanding the traditional attestation mechanism of Intel to build our CAS, we automatically and transparently perform the attestation for each node. The key idea behind our design is that we replace the Quoting Enclave in the Intel attestation mechanism by the local attestation service (LAS). The CAS first attests the LAS using the Intel attestation mechanism, thereafter the LAS will operate as the root of trust in our remote attestation mechanism. Note, that we can launch as many LAS instances as required for availability. The LAS performs the local attestation for AVOCADO nodes and provides attestation quotes that can be verified by the CAS. Thus, our mechanism does not need to interact with IAS after the LAS is trusted, this reduces significantly the overhead of the traditional attestation. We achieve the *transparent and automatic properties* by deeply embedding the remote attestation into the AVOCADO runtime. In addition, our CAS only provisions a configuration and secrets to execute AVOCADO once it ensures that all nodes were not manipulated. Each node of AVOCADO can only communicate with others if it can provide a valid certificate provided by our CAS. Therefore, users can just rely on the CAS to control and operate other components of AVOCADO. They only have to attest our CAS before providing secrets to it. The CAS itself also runs inside an enclave, thus users can use the traditional attestation method of Intel to validate it.

Result As shown in 4.5.3 our CAS achieves 18.2 $\times$ lower end-to-end latency in AVOCADO when comparing with IAS.

4.3.4 Single-node KVS

Problem Enclave's memory limitation is in stark contrast to the requirements of designing an in-memory KVS, which requires fast access to large amounts of in-memory data. Unfortunately, enclaves provide only limited physical memory

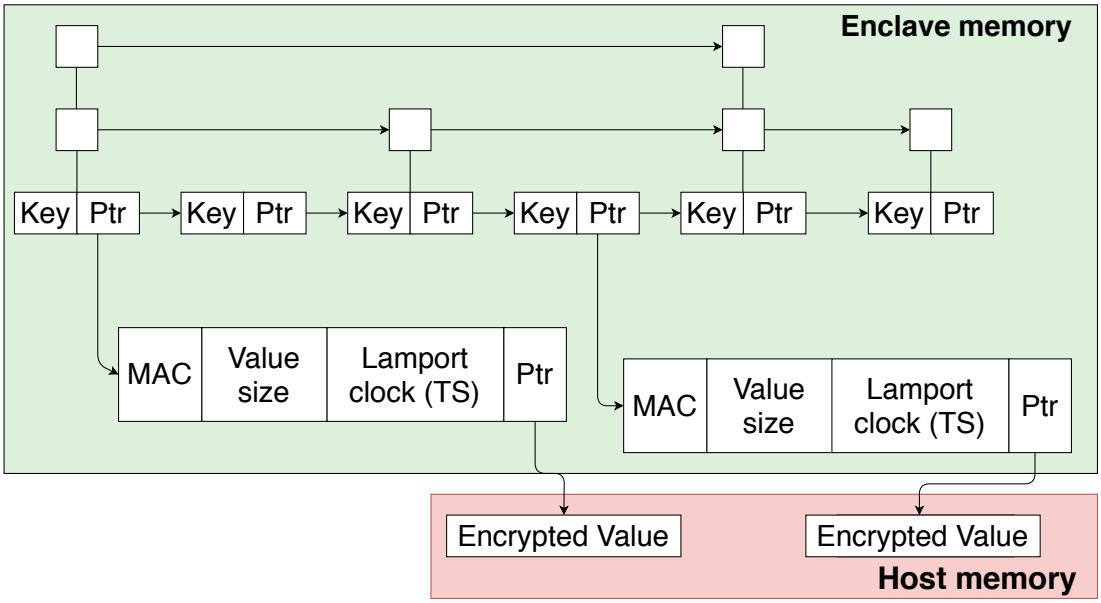

Figure 4.5: AVOCADO's single-node KVS.

(94 MiB) and incur high overheads due to the EPC paging mechanism (2–2000 × [153]) beyond it.

To overcome the limitations of the strawman design, ShieldStore [113], a state-of-the-art secure in-memory KVS for a single-node system, proposed a MerkleTree-like data structure design where the entire KVS resides in the untrusted host memory, except for the security metadata (hash buckets heads). The metadata stored in the enclave memory is used to speed up the look up and to perform authenticity and integrity checks on the KV pairs. However, in our experience, the ShieldStore design suffers from continuous integrity re-calculations. Furthermore, the memory layout prohibits efficient concurrent operations.

Compared to Speicher [1], which also introduced the KV pair separation scheme for enclaves, our KVS is optimized for paging by encountering locality.

Solution To overcome these limitations, we designed and propose our own in-memory concurrent data structure for the single-node KVS. Our KVS is based on a authenticated and confidentiality-preserving skip list [22] which supports secure and fast updates and lookups. We have chosen skip list as our fundamental data structure because it maintains the best features of a sorted array for searching ($O(log_n)$) and of a linked list-like structure for insertion ($O(log_n)$). Our design carefully partitions the key and value space by placing the keys along with metadata inside the enclave memory, while storing the bulk of data encrypted and integrity and authenticity protected in the untrusted host memory. Our partitioned data structure (keys and

values) allow for faster lookups than ShieldStore's hash buckets, while it also reduces the amount of necessary calculations to guarantee the integrity and authenticity. Furthermore, our lock-free data structure supports concurrent operations and it is well-suited for increased parallelism.

As shown in Figure 4.5 the nodes of the skip list reside inside the enclave and contain the key and a pointer to metadata structure. This structure contains the 16 B MAC, for guaranteeing the integrity and authenticity of the value. Furthermore, the data structure also includes the size of the value, which makes checks on the value easier, since we do not need to read any information from the untrusted host memory, to retrieve how many bytes should be read. AVOCADO's consistency protocol uses logical clocks, i.e. Lamport clocks, to establish an order of operations on each key (§ 4.3.2). Therefore, we also store the Lamport clock in the corresponding metadata block, to prevent costly decryptions on the timestamp queries. Lastly, the metadata structure also stores the pointer to the value in the untrusted host memory.

Importantly, separating the metadata and the bulk data (i.e. values) from the skip list allows us to update the skip list *lock free*. Further, it also decreases the EPC pressure when doing a lookup, as nodes can be stored more compact and the metadata can be stored on a different page. However, looking up a value mandates an additional indirection due to keys and metadata separation. Nevertheless, we believe that updating the KVS without acquiring any locks is worth this additional indirection as it allows better multi-threaded scalability. Therefore, in contrast to a HashBucket design like ShieldStore, we never need to stall. In contrast to ShieldStore our approach seems to be limited by the enclave memory, however, assuming we have 1 KiB values and 16 B keys, we achieve a space reduction for enclave memory of 92.8 % compared to a naive implementation. Further, SGX provides a paging mechanism significantly increasing the available trusted memory, therefore increasing the possible size of the KVS. While SGX-paging incurs a high overhead, often accessed keys will eventually resided in EPC.

Result Our evaluation in Section 4.5.4 confirms that our AVOCADO single-node KVS is scalable and more performant; the speedup of AVOCADO single-node KVS compared to ShieldStore increases from 1.6 × in a single threaded benchmark to 5 × when utilizing all 8 available CPU threads.

4.4 Implementation

4.4.1 System Components

AVOCADO network stack The AVOCADO network stack is based on eRPC [19] and DPDK [20]. In particular, we leverage SCONE to build both eRPC and DPDK. We also assure that the device DMA mappings resides in the host memory. The changes to implement the mappings amount to 154 new LoC and 81 removed LoC.

To run eRPC inside the enclave, we accordingly modify the `hugepages` allocation mechanism *(a)* to ensure that all network buffers reside in the host memory, *(b)* to fix a bug regarding the `hugepages`' detection, and *(c)* to alter how the address of the memory region is calculated. We also replace eRPC's allocation algorithm with our own allocator. We notice that the eRPC's native allocation algorithm, which allocates double the space of the previous allocation, quickly reserves all available memory in AVOCADO. Our memory allocator is less aggressive, by not always doubling the space of the previous allocation and by trying smaller increments, the code is available in the avocado repository, and allows us to use our servers' limited huge page memory more efficiently. In total, 80 LoC are added to eRPC, while 28 LoC are removed.

eRPC provides us with its own implementation of the UDP protocol. To secure the network communication, on top of the layer protocol, we use a modified OpenSSL [163] version. These changes allow us to randomly access the encrypted data. We added 55 LoC to OpenSSL. Further, we added another 287 LoC for a shared en-/decryption layer for the AVOCADO single-node KVS store and networking. Lastly, we further extend the shared layer to well-fit with the message format. This adds another 205 LoC.

AVOCADO replication layer We implement AVOCADO replication layer in C++ on top of the AVOCADO network stack (2743 LoC). We implement the protocol from scratch using the eRPC networking library across AVOCADO's different layers, i.e., replication and networking layer.

Configuration and attestation service We implement AVOCADO CAS in Rust [164] for better memory safety (22 730 LoC). To run the CAS inside the Intel SGX, we use SCONE since it transparently supports Rust applications. We make use of an encrypted embedded SQLite [165] to maintain configurations and secrets of AVOCADO inside AVOCADO CAS. To setup the configuration and bootstrap process, we provide configurations scripts, in Bash and Python 3, in total these bootstrap scripts require 709 LoC.

AVOCADO single-node KVS We implement the AVOCADO single-node KVS based on a skip list based partitioned data structure. Particularly, AVOCADO single-node KVS extends Folly's ConcurrentSkiplist [166]. We ported the Folly library to SCONE, which resulted to 167 new LoC and 40 394 removed LoC. In addition, the implementation requires another 190 LoC for the integration of the Boost library [167] to SCONE. Further, we implement an efficient host memory allocator (388 LoC) for our skip list. We share en-/decryption layer based on OpenSSL [163] with the network stack.

4.4.2 Optimizations

[O1] Remove duplicated en-/decryptions In AVOCADO, we use a shared encryption key between all replicas for the network operations. This allows us to replace some encryption calls with memory copies, as we can send the same packets to all replicas without costly re-encrypting the messages. However, this optimization is an optional trade-off between security and performance since one compromised enclave would compromise the entire system.

[O2] Remove locks Separating the metadata from the key allows us to make atomic updates to the skip list, avoiding expensive locks. However, it also allows us to retire values earlier to the host memory; thereby reducing the EPC pressure since the metadata can already be written without being visible to other calls. Further, our host memory allocator supports lock-free operations on our skip list by providing similar atomic allocation and de-allocation primitives.

[O3] Limited number of message buffers We design a rate limiter to allow all current running requests to finish without having to wait for the available resources. While we mostly implement it to prevent eRPC from running out of `hugepages` memory, we also find that it also reduces the stalls between accepting and completing a request.

4.5 Evaluation

Our evaluation answers the following questions.

- How does the AVOCADO network stack perform compared to the alternative networking approaches? (§ 4.5.1)
- How does the AVOCADO replication layer compare with alternative protocols (Raft [159] & BFT [139])? (§ 4.5.2)

- What are the performance overheads of AVOCADO CAS and how it compares with Intel's IAS [141]? (§ 4.5.3)
- How does the AVOCADO single-node KVS perform compared to ShieldStore [113]? (§ 4.5.4)
- What are the overall performance overheads of AVOCADO KVS? (§ 4.5.5)
- How does AVOCADO scale with increasing number of nodes? (§ 4.5.6)

Testbed We perform all of our experiments on real hardware using a cluster of 5 SGX server machines with CPU: Intel(R) Core(TM) i9-9900K each with 8 cores (16 HT), memory: 64 GiB, caches: 32 KiB (L1 data and code), 256 KiB (L2) and 16 MiB (L3), NIC: Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02). They are connected over a 40GbE QSFP+ network switch.

The CPU is designed for workstations, therefore, does not represent a common server CPU. However, at the time of the project the CPU was the most powerful CPU supporting SGX. In a realistic cloud environment the KV store would run either in a VM or container and would map the NIC into the virtual environment. However, as AVOCADO runs entirely in user space, including the network stack, the performance should not be majorly affected by the VM or container, respectively.

Benchmarks For the evaluation, we use the YCSB benchmark [18, 142] with different read/write ratios. Client-server communication over the network is prohibitively expensive from within an enclave (see § 4.3.1). Therefore, we stress-test the performance of AVOCADO by generating the workload within the enclave. This is the worst-case scenario for our system, since a client-server setting will show negligible latency/throughput overheads, due to client-server communication being the bottleneck. We configured AVOCADO to use a shared network key between the replicas (§ 4.4.2 [O1]). For evaluating the network stack, we use `iperf` [168].

4.5.1 Network Stack

Baselines and setup We evaluate the performance of the AVOCADO network stack against three competitive baselines: `eRPC-native`, `sockets-native`, and `sockets-SCONE`. Note that SCONE uses asynchronous syscalls [76] for performance improvements. Further, note that the native (eRPC and sockets) versions do not provide any security.

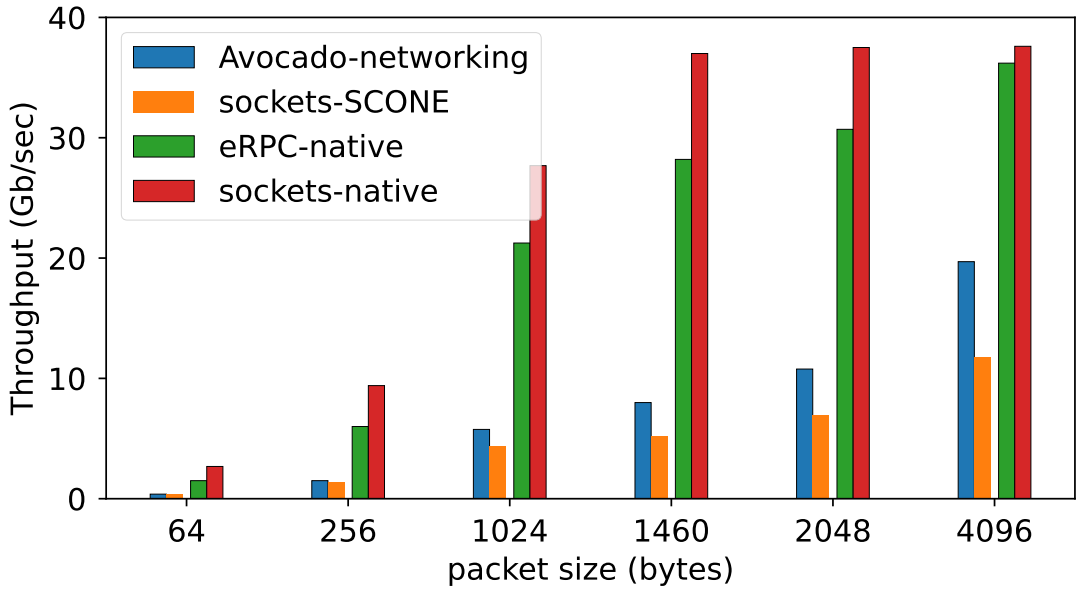

Figure 4.6: Performance comparison of AVOCADO network stack, `eRPC-native`, `sockets-native`, and `sockets-SCONE` for different packet sizes.

For the sockets (native and SCONE), we use `iperf` to measure the throughput. For `eRPC-native` and AVOCADO network stack, we implement a simple server-client model on top of eRPC to simulate `iperf`'s behavior.

In our experiments, we compare the performance with different number of packet sizes, while keeping the number of threads fixed to 4. Note that eRPC supports only UDP while `iperf` supports both TCP and UDP. In our servers, we found that TCP performs better than UDP, so we report `iperf`'s TCP measurements since a designer could always benchmark both protocols and choose the most performant one.

Results Figure 4.6 shows that `eRPC-native` is comparable to `sockets-native`. The reason is that TCP is optimized for high speed bulk transfers while UDP is optimized for low latency in the Linux kernel. This has an impact on buffer sizes and how data is polled and handed over. In addition to this, processing of the entire TCP/IP stack is frequently offloaded to the network controller. Particularly, for very small packets, we do not observe any performance difference since `eRPC-native` is quite fast due to its zero-copy reception/transmission optimization feature. However, for medium packet sizes but still smaller than the MTU (1460 B), `eRPC-native` slows down by 30 %. This performance degradation is due to the difference between send/receive throughput and processing speed. Lastly, for large packets, we also observed that `eRPC-native` is 22 % slower than `sockets-native`.

Based on Figure 4.6 we deduce two core conclusions; (a) SCONE's overhead is not

negligible — SCONE performance degrades ∼4 × and ∼8 × compared to AVOCADO network stack and `sockets-SCONE`, respectively; and (b), due to the number of system calls the sockets' layer is executing, AVOCADO network stack in the context of the secure enclave performs up to 1.66 × faster than `sockets-SCONE`. As discussed, enclave exits and data copies in and out of the enclave deteriorate sockets' performance. This is further supported by the fact that the bigger the packet size is, the worse the performance becomes. Therefore, `sockets-SCONE` is a poor design choice as far as our requirement is concerned, and it justifies our design of the AVOCADO network stack.

4.5.2 Replication Protocol

Baselines and setup We show our system's end-to-end performance in comparison with two state-of-the-art protocols: (a) BFT (BFT-Smart [139]) for the Byzantine setting, and (b) Raft (eRPC-Raft [169]) for the non-Byzantine setting. To the best of our knowledge, there is no secure distributed in-memory KVS; BFT-Smart KVS is the closest baseline in terms of security properties for Byzantine environments, but BFT protocols (or BFT-Smart) still do not preserve confidentiality. Additionally, we compare AVOCADO against eRPC-Raft since it is also built on top of eRPC. This comparison aims to demonstrate the efficacy of eRPC. We compare them with a native version of AVOCADO, AVOCADO-native, which runs without TEEs. We compare AVOCADO and BFT along three parameters, as shown in *(i)* Figure 4.7 for different read/write ratios, *(ii)* Figure 4.8 for different value sizes and *(iii)* Figure 4.9 for different workload threads per machine. We evaluate using the YCSB benchmark [18, 142]. Similarly, we compare AVOCADO against Raft implemented with eRPC. Note that eRPC-Raft is limited to only `PUT` requests and 1 workload thread in total.

Results Our evaluation shows that AVOCADO can achieve 4.5 × to 65 × more operations per second compared to BFT. Our AVOCADO presents similar performance to all four workloads, deducting that it is equivalently performant to both read and write heavy workloads. In addition, we notice that striving for the strictest security guarantees can decrease the performance to half compared to a native, unsecure version of AVOCADO.

Furthermore, we observe that the value size has great impact in the end-to-end performance. For instance, even in a read-heavy workload with value size to be equal to 256 B, the performance of AVOCADO is 6 × higher compared to BFT and 1.83 × lower than the native version. However, for value size to be equal to 1024 B ,

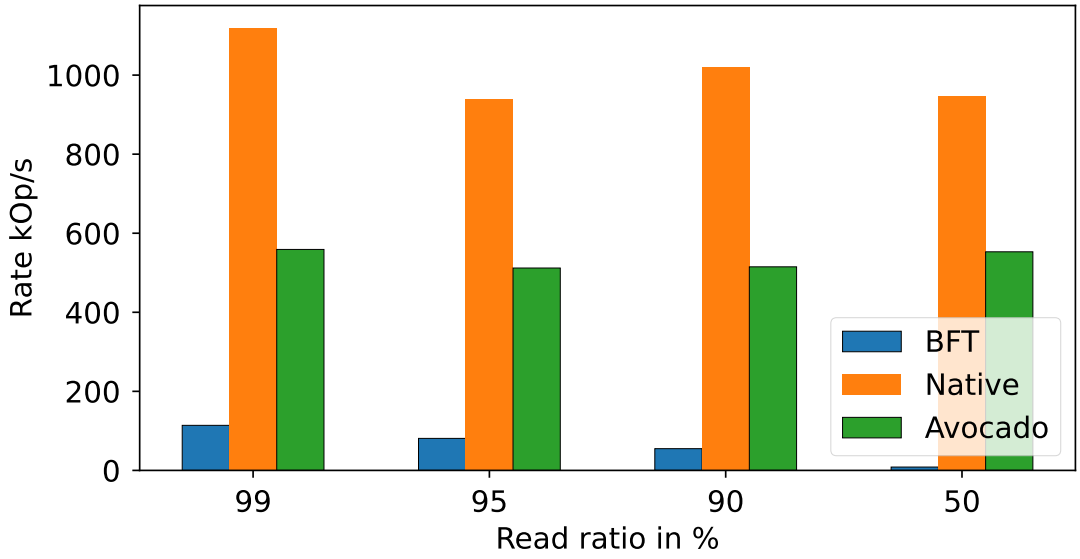

Figure 4.7: End-to-end performance comparison between AVOCADO, AVOCADO-native and BFT for different read write ratios. With a value size of 256 B and 8 threads per node.

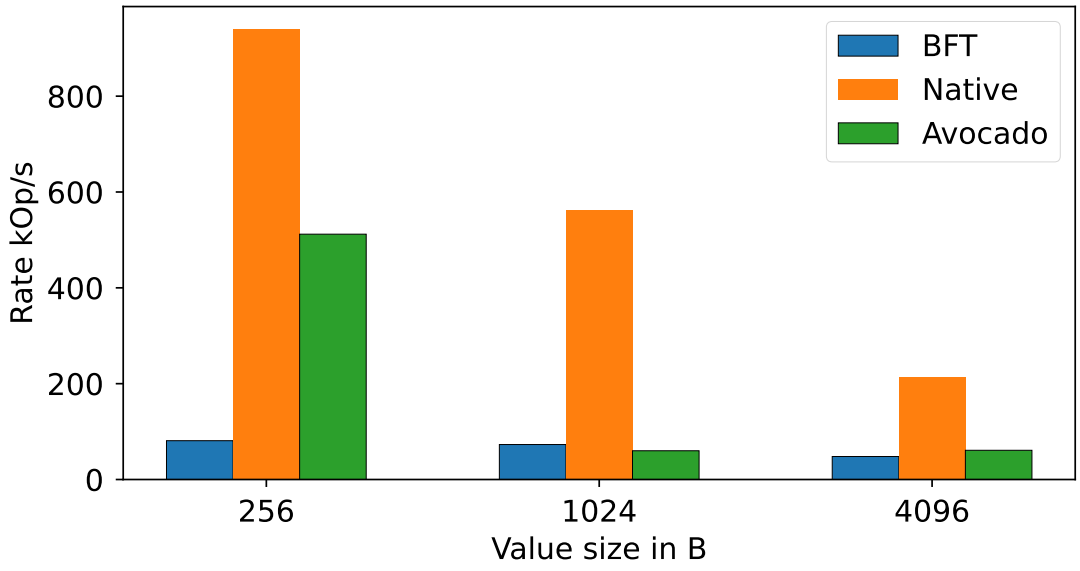

Figure 4.8: End-to-end performance comparison between AVOCADO, AVOCADO-native and BFT for different value sizes. with a fixed read ration of 95 % and 8 threads

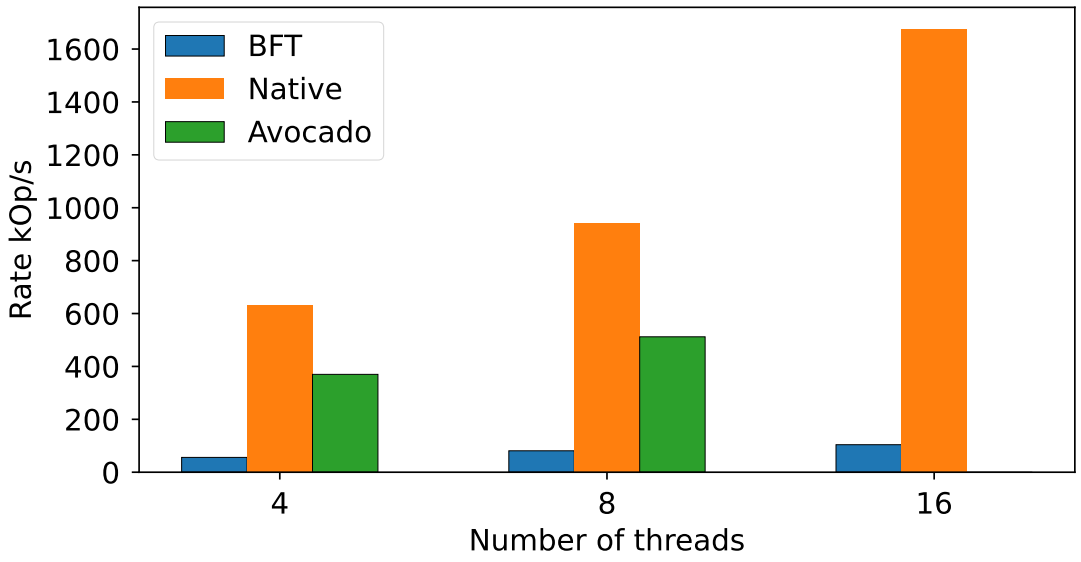

Figure 4.9: End-to-end performance comparison between AVOCADO, AVOCADO-native and BFT for different number of threads with a read ratio of 95 % and a value size of 256 B.

	kOp/s	Speedup
AVOCADO	96	5.05 ×
eRPC-Raft	19	

Table 4.2: Performance comparison between AVOCADO and eRPC-Raft under a 100 % W workload and a single client.

AVOCADO is 20 % slower than BFT and 9 × slower than AVOCADO-native. Similarly, for value size to be equal to 4096 B, AVOCADO is 1.25 × faster than BFT and 3.65 × slower than AVOCADO-native. We discuss the effects of value size on AVOCADO further in section § 4.5.5.

Lastly, AVOCADO scales up with the number of threads; AVOCADO achieves 38 % more operations when the number of threads is increased from 4 to 8 threads. Due to the limitation with the amount of threads inside the enclave, AVOCADO cannot be executed with 16 threads.

Lastly, we compare eRPC-Raft against AVOCADO. AVOCADO under the same settings outperforms eRPC-Raft for 4.8 × as shown in Table 4.2. The reason is that eRPC-Raft does process requests asynchronously while in AVOCADO the time required for the necessary replicas to respond overlaps with processing any outstanding requests.

	Mean / s	SD / s	Speedup
AVOCADO CAS	0.169	0.0195	18.2 ×
IAS	2.913	0.177	

Table 4.3: The end-to-end latency comparison between the attestation mechanisms using AVOCADO CAS and IAS.

4.5.3 Configuration and Attestation Service

Baseline and setup To evaluate the advantage of the attestation mechanism using AVOCADO CAS in comparison to the traditional attestation mechanism of Intel using IAS, we conduct an experiment to measure the end-to-end latency of the attestation process using both mechanisms.

Results The attestation using AVOCADO CAS achieves a speedup of 18.2 × compared to the traditional mechanism using IAS (see Table 4.3). The mechanism using AVOCADO CAS performs the attestation via LAN connections, since AVOCADO CAS is deployed in the same cluster as AVOCADO instances. Meanwhile, the mechanism using IAS performs the attestation via WAN connections since it requires to verify the quotes using IAS that is deployed at Intel. Furthermore, AVOCADO CAS provides transparently provides configuration management features to operate in an distributed environment.

4.5.4 Single-node KVS

Baselines and setup We compare our AVOCADO single-node KVS against ShieldStore [113], a state-of-the-art secure in-memory KVS for a single-node system. For the single-node system evaluation, we use Intel(R) Core(TM) i7-8565U CPU of 8 logical threads and 16 GiB memory. This is due to ShieldStore's dependencies on specific versions of OS, linux-SGX [170] and `tcmalloc` [171], we are not able to run it on our servers. We evaluate AVOCADO single-node KVS and ShieldStore across three dimensions using the YCSB workload: threads (Figure 4.10), value size and read-write ratios (Figure 4.11).

Results Figure 4.10 shows the scaling capabilities of our AVOCADO single-node KVS vs. ShieldStore for two different value sizes. Our AVOCADO single-node KVS, for all number of threads, is 1.6 × to 9.5 × faster than ShieldStore. Regarding the value size, we observe that ShieldStore's performance is highly affected when the value size is increased. For example, with 2 threads, ShieldStore presents a

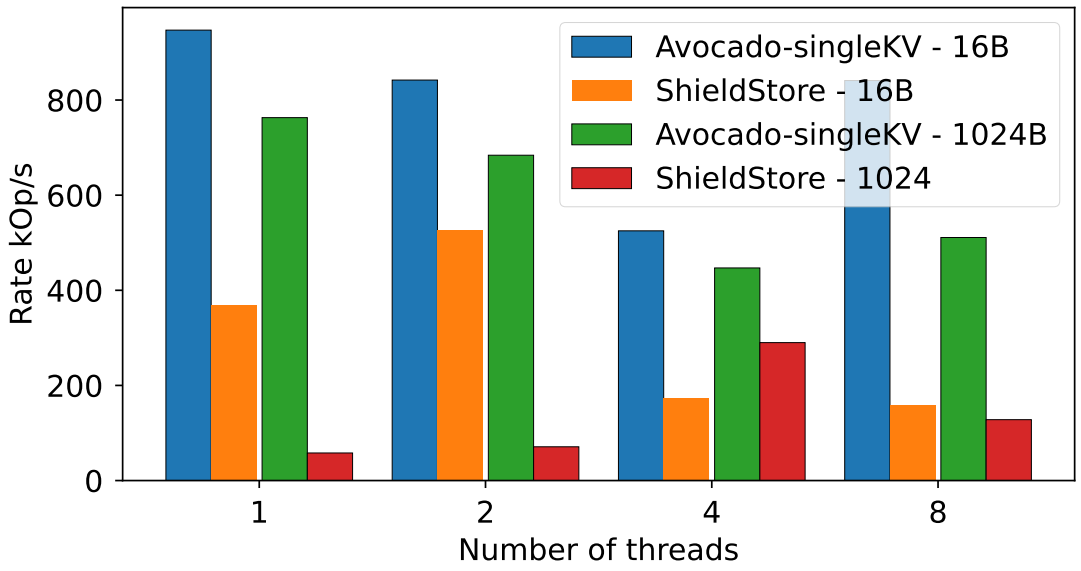

Figure 4.10: Performance comparison between AVOCADO single-node KVS and ShieldStore under a read ratio of 50 % for varying number of threads and value size 16 B and 1024 B.

performance degradation of 7.3 × from 16 B to 1024 B while the same scenario deteriorates AVOCADO single-node KVS's performance only by 1.23 ×. We deduct this to the fact that Shieldstore searches require decrypting the concatenated entry of the key and the value in each visited bucket. As a result, bigger value sizes increase the cipher text that needs to be decrypted leading to higher performance costs. In contrast, AVOCADO stores keys inside protected area and search time is not affected by value sizes and decryption cost.

We observe similar behavior as the number of threads increases. ShieldStore is designed to avoid synchronization costs between threads that are matched to different KVS's areas. However, to achieve this, ShieldStore requires to sort and distribute (through hashing) requests across threads which adds overheads compared to AVOCADO single-node KVS. Specifically, we show that for value size equal to 16 B AVOCADO single-node KVS is 1.6 ×, 3 × and 5 × faster than ShieldStore when using 2, 4 and 8 threads, respectively. Additionally, for value size equal to 1024 B, AVOCADO single-node KVS is 9.5 ×, 1.5 × and 4 × faster than ShieldStore with 2, 4 and 8 threads, respectively.

However, we find that both AVOCADO single-node KVS and ShieldStore have a performance drop, when the number of threads is increased. This trend is visible until the number of threads exceeds the number of physical cores. We attribute this behavior to two main reasons. Firstly, the CPU we are running this experiment on, is

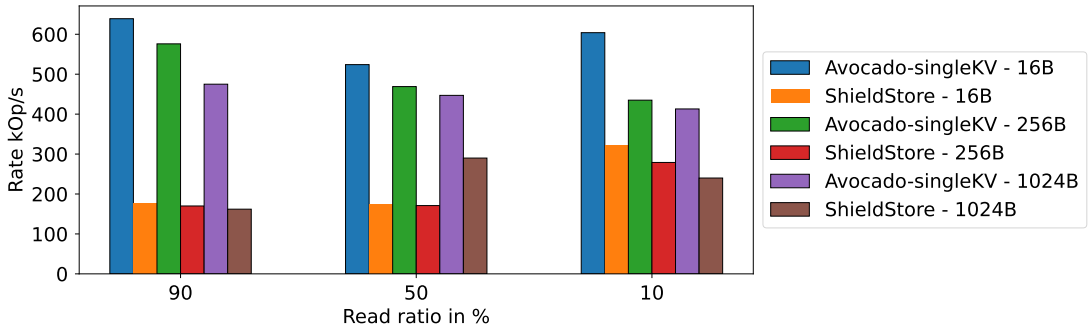

Figure 4.11: Performance comparison between AVOCADO single-node KVS and Shieldstore under three different workloads for varying value sizes.

a lower power CPU, with a low base frequency (1.8 GHz) but a relatively high turbo frequency (4.8 GHz). This high turbo frequency cannot be reached with a high number of threads running, giving a performance boost to low thread numbers, compared to higher thread numbers. Secondly, increasing the number of threads results in a higher rate of cache misses, due to other cores having requested different memory, or having to invalidate the cache lines in lower level caches i.e. L1 and L2. This effect is especially pronounced in a write heavy workload, as presented in Figure 4.10. Increasing the number of threads further, from number of physical cores to logical cores, allows the CPU to schedule a different thread, while it is waiting for a memory access to complete, explaining the increase of performance with 8 threads.

Secondly, we also study how AVOCADO single-node KVS and ShieldStore perform under different value sizes and different read-write ratios. In particular, Figure 4.11 compares the two Key-Value (KV) stores against three different workloads and varying value sizes, where we fix the number of threads across the experiments to 4. For all three workloads as shown in Figure 4.11, AVOCADO single-node KVS achieves better performance than ShieldStore; 3.63 $\times$ to 2.97 $\times$ faster when value size is equal to 16 B, 3 $\times$ to 1.53 $\times$ faster when value size is equal to 256 B and 1.87 $\times$ to 1.56 $\times$ faster when value size is equal to 1024 B.

Lastly, we conclude that AVOCADO single-node KVS is better in read-dominant workload (90 % reads) than in write-heavy workload (90 % writes), since AVOCADO single-node KVS achieves $\sim$5 % to $\sim$30 % better performance.

4.5.5 Distributed KVS

Baselines and setup We evaluate the overhead AVOCADO incurs from running inside an enclave, against running AVOCADO natively, i.e. without SGX. Furthermore, we also evaluate the encryption overheads for in-memory KVS and

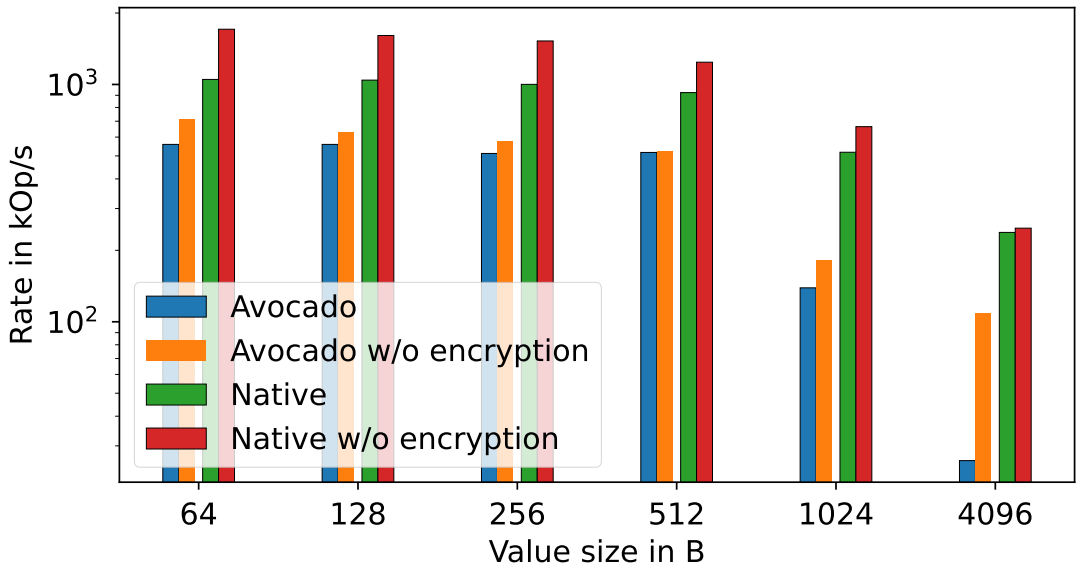

Figure 4.12: Performance comparison of AVOCADO with and without network and KVS encryption, inside and outside of the enclave, with different value sizes with 95 % reads and 8 threads per machine.

network traffic. Thus, we compare AVOCADO with encryption activated and deactivated against the native KVS. Both with encryption for the KVS and network enabled and disabled. We run the YCSB benchmark with 95 % reads with 5 machines and 8 threads per machine, with different value sizes.

Results Figure 4.12 shows the performance influence of SGX and encryption on AVOCADO. The value size comparison shows that for small values, i.e. values under <1 KiB, AVOCADO reaches around half, between 51.2 and 56.0 % of the performance compared to the native KVS. However, with bigger value sizes the difference becomes greater with a slowdown of 3.7 × and 9.0 ×, for 1 or 4 KiB respectively. The sudden drop in performance compared to the native case is mainly due to two reasons: firstly, eRPC has to acquire a lock when allocating DMA for bigger packages size than the MTU, which is configured in our case to 1500 B. While this penalizes native and AVOCADO, it is worse for AVOCADO, since this could result in an enclave exit for yielding. By using jumbo frames instead we could increase the MTU to 9000 B, therefore increasing the value size before eRPC has to acquire locks. Additionally, large sequential de-/encryption operations are faster per byte, then smaller operations, due to setting up and tearing down cost of the de-/encryption operations, and better caching and prefetching behavior for memory. With bigger value sizes, it gets more likely that we have to evict a page from the EPC, when

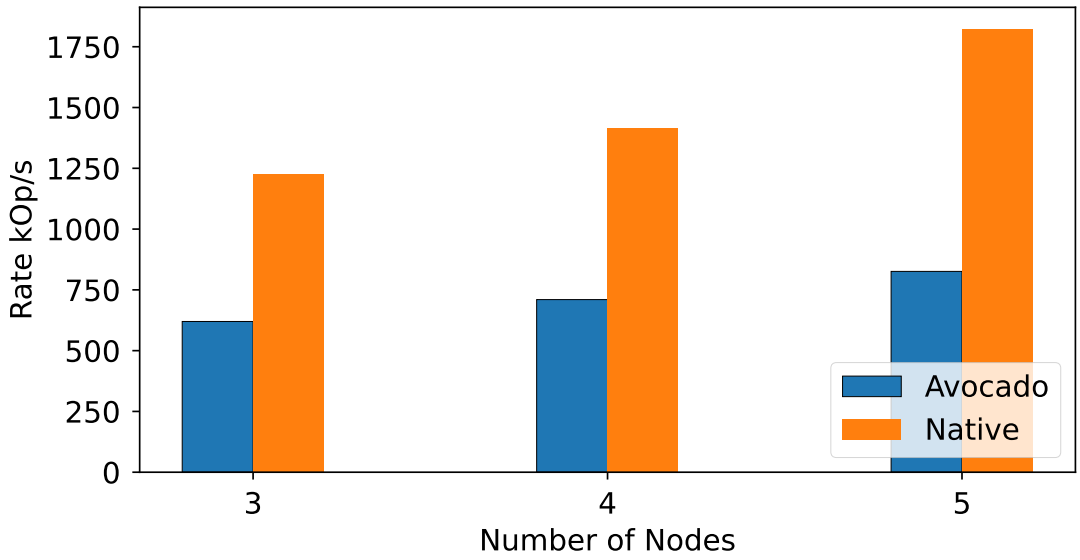

Figure 4.13: Performance of AVOCADO inside and outside of the enclave running on different number of nodes with a value size of 256 B, 95 % reads and 8 threads per machine.

inspecting the network traffic. This might be addressed with a memory buffer, which is reused for all data transfer between untrusted host memory and EPC memory. Due to constant accessing of this buffer, it should rarely get paged out the enclave.

The comparison also shows that encrypting the in-memory KVS and network traffic adds up to 62 % overhead for small values and 4.6 % for large values in the native case. However, we observe a different behavior for AVOCADO. The overhead for small values is more moderate compared to native with around 25 %. However, the overhead does not get smaller with bigger values sizes. In contrast, it peaked with a values size of 4 KiB with 4.1 ×, which is due to EPC paging.

In these experiment we also observed a mean latency of 66 µs. This latency was reached in a fully stressed system. Due to SGX requiring us to make a syscall for taking a timestamp detailed latency analysis was impractical, as the syscall overhead together with the world switch would have easily dominated the benchmark.

4.5.6 Scalability

Baselines and setup We evaluated the scalability of AVOCADO by running it inside and outside (natively) the enclave on a varying number of nodes. We run the YSCB benchmark with 95 % reads on 3, 4 and 5 machines and 8 threads per machine, with a fixed value size of 256 B.

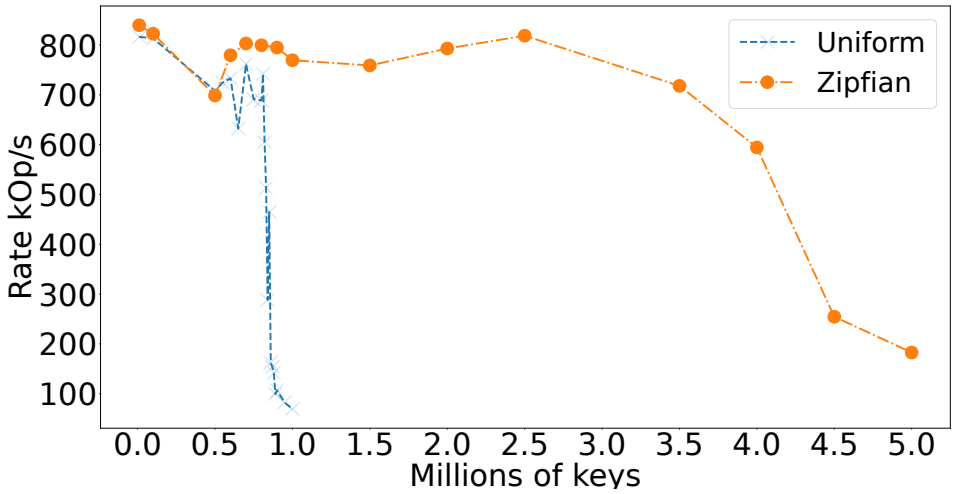

Figure 4.14: Performance of AVOCADO inside the enclave running with different number of distinct keys, with a uniform and Zipfian ($\alpha = 1.0$) distribution with a value size of 256 B, 95 % reads and 8 threads per machine

Results Figure 4.13 shows the scalability numbers for different number of nodes. We are limited in our cluster to 5 nodes. The evaluation shows that replicating the KVS over 5 nodes instead of 3 increases the throughput of the native solution by 49 % and 33 % for AVOCADO.

4.5.7 Number of Keys

Baselines and setup We measure the performance of AVOCADO with increasing number of distinct keys. We run the YCSB benchmark with two different distributions, i.e. uniform and Zipfian, on 5 machines with 8 threads and 95 % reads and a fixed value size of 256 B.

Results Figure 4.14 shows the throughput of AVOCADO with different distribution. Both distributions have a similar performance until 700 k keys, where the performance of the uniform distribution starts to suffer greatly, due to SGX paging overheads. The uniform distribution prevents efficient caching from SGX in the EPC, since it does not generate any hot keys. In a uniform distribution, AVOCADO is therefore restricted by the EPC size. However, this could be alleviated with a multi-level lookup, which stores the lower levels in the host memory.

On the other hand, if the data set is not uniformly distributed AVOCADO can take advantage of the caching of EPC and extend the number of supported keys. In our experiments AVOCADO throughput was stable until 3.5 M keys in the Zipfian

distribution before it starts to suffer from the paging overheads. Similar to the uniform distribution, a multi-level lookup could reduce the paging overheads.

4.6 Related Work

Networking for shielded execution Shielded execution frameworks, like SCONE [40] and Eleos [38] provide an asynchronous system call API [76]. However, the asynchronous syscall mechanism incurs high overheads (due to data copies) and latency. ShieldBox [93] alleviates the issue of asynchronous syscalls by using DPDK [20] as polling user mode driver for a secure middlebox. Unfortunately, Shieldbox network stack targets only layer 2 in the OSI model, and therefore, it is incompatible with the RPC layer required for a distributed KVS. rkt-io provides a library OS in the enclave, and can therefore provide a full network stack [172].

Secure storage systems Secure storage is an important theme for cloud computing systems. A wide-range of systems have been proposed in the community based on different hardware with varying security guarantees and storage interfaces: KVS [1, 103, 113], filesystems [97, 100, 121], databases [105–107, 109, 112], etc. In contrast to these existing systems, AVOCADO proposes a scalable distributed in-memory KV store instead of a single-node system.

For distributed storage system design, Depot [116] and Salus [117] propose secure distributed storages, which provide consistency, durability, availability and integrity. A2M [118] is also robust against Byzantine faults. On top of those properties, AVOCADO also offers confidentiality. CloudProof [110] completely distrusts the cloud provider. However, CloudProof requires the client to guarantee these security properties, which requires major changes to the client, which isn't required by AVOCADO. Furthermore, AVOCADO leverages hardware-assisted shielded execution as the root of trust, we do not need a trusted proxy, which limits the scalability of the system. Microsoft's confidential consortium framework (CCF) [45], which was developed at the same time as AVOCADO, provides replication over a leader based CFT protocol, i.e., Raft [159]. In contrast to AVOCADO, CCF uses the host for networking, decreasing performance. Furthermore, CCF does not handle limited EPC memory. Both of these factors result in CCF being considerably slower than AVOCADO.

4.7 Summary

We present an approach to build a secure, high-performance in-memory distributed KVS system for untrusted cloud environments using TEEs. Our design includes four core contributions involving TEEs in a distributed environment: (a) the first direct I/O RPC network stack for TEEs based on eRPC with the complete support for transport and session layers; (b) a secure replication protocol based on hardening of a non-Byzantine protocol, where we transform a Byzantine behavior to a faulty behavior using TEEs; (c) a configuration and attestation service to seamlessly extend the trust from a single-node TEE to the distributed environment; (d) a secure in-memory single-node KVS based on a novel partitioned skip list data structure, and show that it is well-suited to overcome the memory limitations and support lock-free scalable concurrent updates in the TEEs.

Importantly, we set out to build a practical system without compromising performance — the literature distinctly shows that BFT protocols are typically not adopted in practice due to their high overheads [136, 173]. In contrast to BFT, our system provides stronger security properties (also preserves confidentiality) and improved performance (4.5 $\times$ to 65 $\times$), while using f fewer replicas.

Chapter 5

TOAST: Heterogeneous Memory Management

Modern applications employ several heterogeneous memory types for improved performance, security, and reliability. To manage these heterogeneous memory types, programmers must currently digress from the traditional load/store interface, and instead rely on a range of custom libraries specific to each memory type, thus introducing programmability, portability, and protection challenges.

To overcome these challenges, we propose TOAST, a compiler-based approach that offers *a simplified programming model* based on the established load/store interface combined with an error handling mechanism and a protection mechanism to enforce memory safety.

We implement TOAST in the Clang/LLVM compiler framework accompanied with a runtime library, employing software capabilities and hardware-based protection mechanisms. Our evaluation based on four real-world applications shows that TOAST improves the *programmability*, *portability*, and *protection* of applications, while offering *performance* on par with a hand-optimized version of the application.

5.1 Introduction

Heterogeneous memory is typically understood as memory with varying access latencies, such as NUMA (Non-Uniform Memory Access) [174] and RDMA (Remote Direct Memory Access) [35]. However, we can broaden the definition to include any interface that resembles system memory's load/store behavior but differs in latency or access patterns. This encompasses memory areas accessed by devices through

DMA (Direct Memory Access) and specialized libraries like SPDK (Storage Performance Development Kit) [170] and DPDK (Data Plane Development Kit) [20]. Typically, these libraries offer direct access to the memory buffers in the DMA area for the user. This enables users to interact with the buffers using load/store operations, similar to regular memory accesses. However, to ensure proper communication with the device, such as notifying it of data changes, specific pre- and post-actions need to be executed. As a result, the interface for the user differs from simple memory load/store operations to accommodate these additional requirements.

Modern applications employ multiple heterogeneous memory types for performance, security, reliability, and domain-specific computing [2, 33, 88, 169, 175]. These heterogeneous memory systems span almost all aspects of the stack, e.g., multi CPU (NUMA [174]) network (RDMA [35]/DPDK [20]), storage (SPDK [16]/persistent memory [36]), secure enclaves [5], and accelerators [176].

In theory, heterogeneous memory subsystems are accessible via the memory controller, i.e., allowing read and write directly from and to memory regions. However, in practice, these memory subsystems are accessed via a range of subsystem-specific auxiliary libraries, which force programmers to digress from the traditional *load/store* interface to access byte-addressable memory regions [35, 88]. This library-based approach leads to four significant challenges for heterogeneous memory management ("The 4P challenges"): *(i)* Programmability, *(ii)* Portability, *(iii)* Protection, and *(iv)* Performance.

Programmability challenges arise because programmers must learn and understand the APIs and libraries for each memory technology separately [28].

Moreover, the library-based approach introduces *portability* challenges when the underlying hardware evolves with the introduction of new technologies. To adapt to a new heterogeneous memory subsystem, current approaches require essentially a complete re-design of the software system. Programmers have to rewrite their code to a great extent using different access patterns, libraries and APIs. This is a cumbersome and error-prone task.

Furthermore, heterogeneous memory management also introduces *protection* challenges, as a programmer juggles with different memory regions. A potential error during application development can lead to undesired code behaviours, such as sensitive information leakage to untrusted devices or mistakenly persisting temporary data.

On top of that, application programmers are — as always — pressed to achieve optimal *performance*, which is difficult when they have to deal with different libraries

and their varied interfaces.

To address these challenges, we consider the following problem: How do we define a heterogeneous memory programming interface, which provides *programmability*, i.e., an easy to learn and understand interface, *portability*, i.e., minimizing the effort involved in changing underlying technologies, *protection* against accidental data sharing between different memory regions, while providing *performance* on par with or exceeding existing libraries?

Due to the heterogeneous nature of the devices that leverage the memory interface, designing a unified interface comes with inherent challenges. While heterogeneous memory can be accessed over the cache coherent interconnect, due to device-specific implementations, additional actions might be required, including reading and writing to specific memory addresses before or after the data transfer and additional cache flushes. Furthermore, heterogeneous devices have different sources of errors and faults, leading to vastly different error handling procedures.

To this end, we propose TOAST, *a simplified, generic programming model* based on the established load/store interface combined with an error handling mechanism and a protection library to isolate different memory regions. TOAST consists of a compiler, based on Clang/LLVM [177, 178], and a run-time component. The compiler component is responsible for lowering a high-level load/store interface to our lower-level runtime component. As part of our runtime component, we introduce the abstraction of TOASTPtr, a pointer associated with a specific memory region that is manipulated using a single uniform interface in the programming language. Under the hood, the TOAST runtime transparently translates interface calls to region-specific library calls. TOAST further provides optional programmable error handling callback hooks to enable the programmer to handle memory device-specific errors.

TOASTPtrs lift library/device-specific calls to a device-independent load/store interface at the language level. Thus, TOAST eases *programmability* by reducing the technology-specific knowledge required from the developer. TOASTPtrs are given semantics via a configuration file which can be easily adapted for different technologies, thus improving *portability*. Importantly, once a configuration for a technology is correctly designed, the developer can reap its benefits across applications without additional effort.

Lastly, TOAST incorporates two protection mechanisms to prevent programming errors related to unintended data sharing between memory regions; *(i)* a software-based mechanism designed as a capability storage and *(ii)* a hardware-based mechanism using Memory Protection Keys (MPK) available on Intel processors.

```
1  event_loop(FILE log)
2   int svr, clt;
3   struct sockaddr_in svr_addr,
        clt_addr;
4   svr = socket(AF_INET,
        SOCK_STREAM, 0);
5   svr_addr = init_server_addr();
6   bind(svr, &svr_addr, sizeof(
        svr_addr));
7   listen(svr, 3);
8   clt = accept(svr, &clt_addr,
        sizeof(clt_addr));
9   char buf[msg_sz];
10  for (;;)
11   //Waiting for data to arrive
12   read(clt, buf, msg_sz);
13   if (is_write(buf))
14    //Writing to storage device
15    write(log, buf, msg_sz);
16   //Acknowledge to clt
17   write(clt, rsp, rsp_sz);
```

Listing 5.1: Using POSIX API to write a network stream to a file

```
1  event_loop(uint64_t * log)
2   rx, tx = get_queues()
3   for (;;)
4    //Waiting for data to arrive
5    poll(rx);
6    char * buf = get_buf(rx);
7    if (is_write(buf))
8     //Storage pointer
9     uint64_t * log = next_log(
        buf, log_sz);
10    //Writing to storage
        device
11    *log = buf << 32 | len(buf);
12    //Makes writing visible
13    clflush(log);
14    //Free buffer
15    char * extra = next_free_buf
        ();
16    swap(buf, extra);
17   write_response(tx);
```

Listing 5.2: Accessing network and storage device with DMA [88]

```
1  event_loop(T_log log)
2   rx, tx = get_queues();
3   for (;;)
4    //Waiting for data to arrive
5    T_net buf = get_buf(rx);
6    if (is_write_request(*buf))
7     //Writing to storage device
8     log[idx++] = buf << 32 | len
        (buf);
9    write_response(tx)
```

Listing 5.3: TOAST version of the applications in Listings 5.1 & 5.2.

We evaluate TOAST on four real-world applications that access five different memory types via device-specific libraries: (a) a secure in-memory key-value (KV) store [2], (b) a replication protocol [138, 158], (c) a persistent shared log application [179] and (d) a persistent KV store [180]. Our evaluation shows that TOAST improves *programmability*, *portability*, and *protection* of applications, while incurring a mean *performance* overhead of 4.9 % compared to hand-optimized code.

5.2 Motivation: The 4P Challenges

Modern applications employ multiple DMA-capable devices, and deal with various memory regions with their respective libraries for networking (e.g., DPDK [20]) or storage (e.g., SPDK [16], PMDK [36]) or en-/decryption (e.g., OpenSSL [163]). These devices and libraries expect specific data structures and read/write patterns to reach their full potential. Thus, the adoption of a new technology often requires that developers invest time to learn new device-specific library APIs and leads to the rewriting of major parts of an application to achieve the

desired performance. Such changes are invasive, time consuming and error-prone. Although, the current approaches already try to deal with this issue [176] they are not sufficient since they only target specific device classes.

As an example, Listing 5.1 shows a code path that accepts network packages and writes them to a log on a file system using the POSIX API. Listing 5.2 shows the same program based on non-blocking communication between devices and CPU by introducing heterogeneous memory subsystems, i.e. Remote Direct Memory Access (RDMA) networking and Persistent Memory (PM).

Notably, these modern technologies use very different interfaces, even within the same class of application, e.g, for networking DPDK's interface differs from RDMA's or for storage SPDK's interface from PMEM's, which results in very little code reuse between the implementations. They also have a very different abstraction for the user; on the one hand, POSIX provides common abstractions and, on the other hand, the user has to manually add polling and cacheline flushes to ensure correct code behaviour in the second implementation.

Transitioning from one technology to the other requires addressing the following challenges.

Programmability Programmers are required to familiarize themselves with all the libraries used in their application, as each has its own interface and order of operations. Furthermore, each memory type needs its own memory allocator to make sure that the memory layout follows the specifications of the underlying device. This imposes significant programming challenges as each library creates pointers to its respective memory regions which can be stored and later be reused from other parts of the application. Pointers of two different libraries are *conceptually* distinct, but they are not distinguished by the language's type system and can, therefore, be inadvertently confused by the programmer resulting in catastrophic failures and security vulnerabilities.

Portability Adapting modern systems to use new technologies is challenging. Firstly, the APIs to handle each memory type or device might be designed for or even integrated with the logic of the application. This can be mitigated through an abstraction layer. However, the design of such an abstraction layer is a non-trivial task. On top of that, the deployment of such an abstraction layer needs to be carefully performed by the developers throughout the whole application. In that way, an application gets strongly tied to a specific abstraction layer, hindering portability.

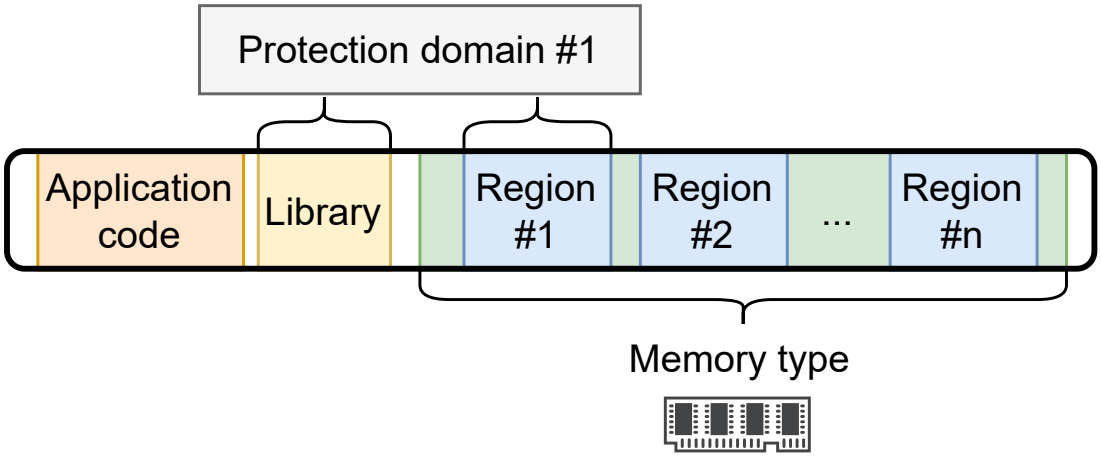

Figure 5.1: Virtual address space layout in TOAST

Performance The appeal of modern heterogenous memory systems is their superior performance compared to conventional abstractions. However, their focus on performance often leads to a tight coupling of the application logic with the technology to take advantage of techniques, like zero copy or asynchronous calls. This requires careful optimizations, which are often tedious and error prone.

Protection Efficient resource access management is important when dealing with multiple memory regions. Currently, developers are responsible for the correct handling of the pointers returned by various libraries. Calling a library with a pointer from another library can lead to information leaks (e.g., revealed keys) or memory corruption (e.g., buffer overflow attacks). Therefore, it is crucial to provide a form of memory region-level isolation to protect against pointer misuse, i.e., a mechanism to ensure correct access to different memory regions only via their respective library pointers.

5.3 Overview

Programming languages offer a well-known abstraction for directly accessing volatile local memory, namely, the *load/store* model. In the context of this chapter we define local memory as memory, which a user can access with load/store operations without having to follow specific access patterns or API calls without it being considered a programming error. The concept of pointers is a central part of this model, providing ease of programmability.

Device libraries often expose pointers to DMA'ed memory regions to programmers to enable zero-copy operations. While heterogeneous memory can be

accessed over these pointers, due to device-specific implementations, additional actions might be required, including reading and writing to specific memory addresses before or after the data transfer and additional cache flushes. However, this requires developers to use specific functions to access memory in a safe way, in contrast of just accessing the memory. This, in turn, results in inconsistencies in the developer's *mental* model, as they have to interact with pointers, which in vastly different ways. Additionally, the idiosyncrasies of each memory type (e.g., persistence granularity) as well as the different access patterns they require (e.g., writing to network queues) make both the unification of the various memory types' interfaces and the memory type-specific runtime error-handling process quite challenging.

5.3.1 The TOAST Approach

To provide similar levels of *programmability*, *portability*, and *performance* as those offered by local memory, we propose TOAST. TOAST improves programmability and portability by providing a well established interface along with a simple configuration setup. Developers do not need to delve into library specific APIs and mechanisms, as they can be provided through easily composable configuration files. In that way, portability becomes easier as well, since the required modifications are restricted to the TOAST configuration files. TOAST specific targets source code portability and not binary portability, therefore a re-compilation is necessary when deploying the application on a new platform.

TOAST introduces the concept of *memory type*, *memory region* and *protection domain*. We define the *memory type* as a set of address ranges in an address space, which is accessed in a uniform way, using a single set of API calls. A memory type may be mapped to *RAM*, *devices*, or *special memory* like PM or trusted enclave memory. A *memory region* is an address range in a memory type, e.g. the Tx/Rx queue for a NIC. A *protection domain* is a set of access rights to address range mappings. Figure 5.1 shows the relationship between the different concepts.

TOAST *refines* (annotates) the pointer type with the memory type, thus creating a separate pointer type for every memory type. It transparently injects code at compile time, which at runtime calls the corresponding TOAST runtime library (see § 5.4.2) for the annotated pointers to support normal dereferencing, while guaranteeing the right order of library calls. On top of this, TOAST provides memory *protection* mechanisms to prevent accidental memory mishandling due to programming errors.

With the definition of memory type and memory region, together with the transformations the TOAST compiler does, TOAST is able to support any memory

Function	Description
write()	Writes data to memory
read()	Fetches data from memory
err_handler(stack,...)	Called by the TOAST runtime library in case of an erroneous memory access

Table 5.1: TOAST runtime library APIs

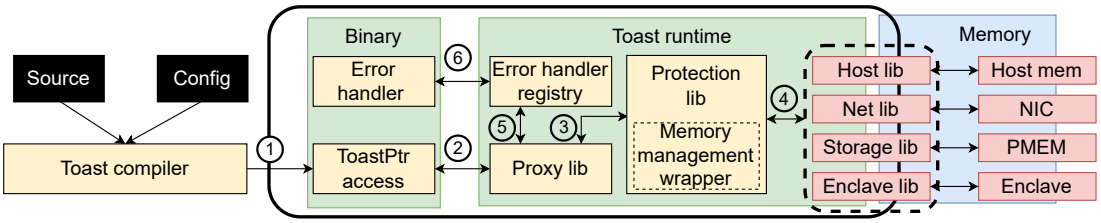

Figure 5.2: Overview of TOAST: The compiler creates the binary and links it with the runtime libraries ①. The binary dereferences a TOASTPtr ②, which results in the proxy library communicating through the protection library ③ with the devices ④. In an error case, the proxy library informs the error handler registry ⑤, which collects information for the error handler and calls it ⑥.

type, which provides a load and store interface, be it through hardware instructions or library calls.

TOASTPtr TOAST's programming model is based on the fact that devices interact with the CPU over the cache coherent interconnect with load/store operations. Most programming languages offer these operations to the programmer in the form of assignments, e.g., the operator '=' in C. However, developers cannot use them directly when they interact with devices, as devices are usually accessed via specific low-level libraries. These libraries implement specific access pattern, to interact with the device, to guarantee the correct behavior, e.g., persistence, atomicity.

A TOASTPtr is a pointer to a memory region. TOASTPtrs contain, in addition to the address, the memory type as well as the protection domain, which enable TOAST to check memory safety violations when a TOASTPtr is dereferenced. The TOAST compiler transforms pointer (de)references to read and write calls to the TOAST runtime library, which acts as a proxy layer between TOASTPtr operations and the low-level runtime library for different devices.

Error handling Error handling is an integral part of any application. Pointers are notoriously bad at communicating errors as they only have two states: the invalid

null and the valid non-null. TOAST allows developers to register error handlers. In case of an error, TOAST calls the respective error handler with a pointer to the program's call stack, a source code position, error information from the underlying device and a pointer to internal memory, *e.g.,* transmission buffers, hash values, etc., all of which can be used in the error handler to recover from an error or to collect debug information.

An error handler returns one of the following states back to TOAST: ***retry*** signals TOAST to just retry the last operation; ***continue*** means that the handler corrected the error in the internal data and TOAST can return the corrected data to the calling code; ***abort*** instructs TOAST to clean up the current action and return an invalid pointer to the caller.

Device configuration In TOAST, the developer defines the device configuration once. This configuration can be reused across different applications seamlessly, provided that the TOASTPtrs referring to this device are annotated correctly in the applications' code. Thus, different projects can simply adopt and combine existing TOAST device configurations, promoting generality and reusability. Precisely, the device configuration includes *(i)* the memory type, *(ii)* the name of the memory type's proxy library, and *(iii)* a list of header files to locate the appropriate library functions.

Workflow Figure 5.2 presents the flow of an application using TOAST. The developer provides the TOAST compiler with the source code and a configuration file containing the necessary information for interacting with memory types.

Table 5.1 shows the API of TOAST after the compiler transformation. During compilation, the pointer (de)reference operations are transformed into calls to the TOAST runtime library ①. A dereference of a TOASTPtr is lowered to a call to the TOAST proxy library ②, which performs necessary checks and preparations. Then, if the TOAST protection mechanism is enabled by the user, the appropriate protection checks and access rights management operations are executed ③ and TOAST calls the underlying library ④.

If the library returns normally, the TOAST proxy provides a pointer to the underlying memory area to the user's code, allowing the user to load or store data in it. If an error occurs, the proxy library informs the error handler registry ⑤, collects the necessary information and triggers an error handling event ⑥. The error handler returns to the error registry either a *retry*, *continue* or *abort* indication, which is forwarded to the proxy library. Finally, the proxy library returns to the user's code with either a valid or an invalid pointer.

TOAST requires a device-specific implementation of its proxy library. However, this effort has to be done once per DMA-capable device and is reusable across all applications.

5.3.2 System Model

Fault model We assume that data is shared between different software components, *e.g.,* libraries. However, not all components are allowed to access all data. We consider each memory region dedicated to a component to be part of a protection domain. We also assume that accidental sharing of information across protection domains, (e.g., without explicit pointer casting) is a critical fault. Importantly, TOAST assumes that programmers do not have malicious intent and only prevents inadvertent programming errors.

Programming model TOAST is designed for heterogeneous memory types with different access semantics. We assume that the underlying system provides a unified address space, *i.e.,* the address itself does not contain information about the type of memory it refers to. In the underlying system, memory is accessed via memory type-specific interfaces, *e.g.,* device-specific library APIs or specific CPU instructions, not via direct, common assignment operations.

5.3.3 Example Revisited

We illustrate the TOAST programming model (Listing 5.3) using the simple example from Section 2 that involves a network and a storage interface. In this example, the programmer has to perform the network and storage management with different interfaces and semantics. We observe that the actual task concerns only copying data received through the network (e.g., sockets, NIC) to the storage (e.g., SSD, PM).

Listing 5.3 shows the example code of Listing 5.1 and Listing 5.2 transformed with TOAST. It abstracts away the POSIX APIs, and the RDMA and PM library calls as well as the device-specific operations (e.g., polling, cache flushing). The intended *logical functionality* becomes unmangled from device-specific library calls, allowing the programmer to focus only on the logical operations when programming and debugging. TOAST relies on the configuration files to rewrite the simplified code and produce the expected, correct binary.

To further emphasize the use cases of TOAST, we present another example of code transformation, specifically showcasing the transformation when utilizing PMEM as the underlying technology.

```
1 write_int(int i, char * path):
2   [[storage]] int * ptr = init(path)
3   *ptr = int
```

Listing 5.4: Example code a programmer would write with toast for persistently storing an int.

```
1 write_int(int i, char * path):
2   ToastPtr<int> ptr = init_pmem(path)
3   ptr.write(i)
4   if ptr.is_persistent():
5     pmem_persist(ptr, sizeof(i))
6   else:
7     pmem_msync(ptr, sizeof(i))
```

Listing 5.5: The example code from Listing § 5.4 after transformation from the TOAST compiler for PMEM.

In Listing § 5.4, we provide a code example that a developer could potentially write using TOAST. In line 3, the user annotates the pointer with the memory type of *storage*. During the compilation step, the user specifies that the storage should be PMEM.

After applying the TOAST compiler transformation, as shown in Listing § 5.5, the code undergoes modifications. The pointer dereference is replaced with an explicit write call, and the generated code ensures the persistence of the data.

5.4 Design

5.4.1 TOAST Compiler

TOAST requires pointer annotations for individual memory types. We extend the list of attributes to support the namespace toast, which contains pointer annotations, and the attribute toast::event for event handler callbacks, i.e., error handling. Each of these attributes takes an additional user-defined parameter, which further specializes the type of the TOASTPtr and event handler. Thus, TOAST lifts knowledge of the pointer's memory type into the type system.

The TOAST compiler initially collects a list of pointers which are annotated by the programmer with an attribute introduced by TOAST. Then, it internally changes the types of these pointers to TOASTPtr. Thereafter, it scans the AST for uses of the pointers, differentiating between read and write accesses. This allows TOAST to insert the corresponding read or write functions and checks for each memory type. Additionally, the compiler understands a set of common memory functions, like memcpy, memset, strcpy, which are replaced with optimized library functions, providing the user familiar standard library functions with optimized implementations.

Changing the types of pointers may have further implications, e.g., the return type or an argument of a function may change. The TOAST compiler tries to infer the necessary changes. However, this is not always possible, since the entire code is not visible to the compiler at the same time. The TOAST compiler identifies these functions and creates a copy for every TOASTPtr type calling the specific function, which is necessary as the libraries contain different read and write calls. This feature also incentivizes code reuse, as the same function can be used for different memory types, as the compiler will create a copy per memory type. The copies of the function are transformed the same way as user-annotated pointer accesses. In cases where the TOAST compiler cannot change the code itself, e.g., when the definition of the function or the caller of the function is in a different compilation unit, TOAST requires additional function signature annotations. The TOAST compiler further registers functions which are annotated as error handlers with the error handler registry in the runtime library.

TOAST allows users to define their own attribute parameters. Thus, a parameter is not coupled to a specific technology and can be as generic as *Network*. The user supplies the TOAST compiler with a configuration file that maps the parameters to technologies. This file provides the compiler with information about which library calls and checks to perform. By re-compiling the whole code base, the compiler can find all code paths where a TOASTPtr of a specific type is used. Then, it scans the AST to find patterns defined in the configuration and adds the necessary function calls and checks.

5.4.2 TOAST Runtime Library

The runtime library implements a unified API for different memory types and inserts necessary runtime checks. The implementation of the runtime library depends on the configuration of the technologies chosen by the system designer.

The API is implemented once for each supported technology and can be reused in different projects. To decrease the implementation effort, TOAST provides templates for commonly used patterns, which can be combined and extended.

Furthermore, the runtime library contains a map of registered error handlers. The runtime library invokes the error handler with appropriate parameters, handles its return code and implements the retry and abort functionalities.

By factoring the low-level implementation into a runtime library (in contrast to implementing it directly in the compiler), we increase the extensibility of TOAST, as this makes the addition of a new device technology easier.

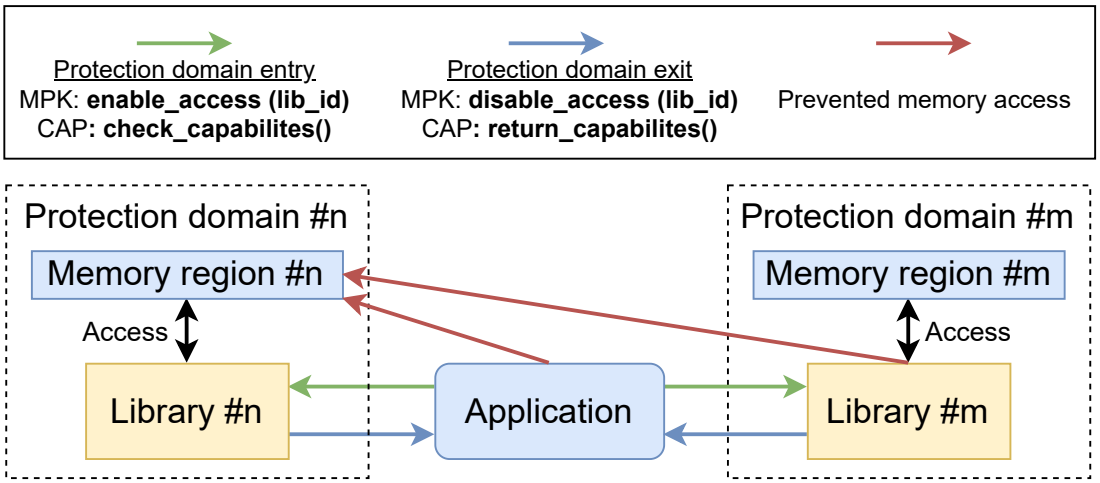

Figure 5.3: TOAST protection mechanism: On a protection domain transition (green and blue arrows) the appropriate capability checks or the enabling/disabling of the access for a protection key are performed. TOAST further prevents access to inappropriate protection domain (red arrows).

5.4.3 Protection Library

TOAST aims to prevent information leaks due to mixups between pointer types and libraries. For this, TOAST defines protection domains inside the virtual address space (VAS) of the application compiled with the TOAST compiler. We implement two different versions of protection libraries in TOAST, shown in Figure 5.3.

The TOAST protection library intercepts every call that triggers a protection domain transition, e.g., library calls, and TOASTPtr dereferences. Note that the mechanism of the actual transition depends on the chosen configuration. The protection library also introduces appropriate checks to determine the validity of each memory access.

Memory safety model At protection domain transitions, TOAST checks pointers for spatial and temporal validity, i.e., the pointer's internal memory type matches the memory type of the pointer's address. Further, the pointer's protection domain should match the protection domain being transitioned into. TOAST prevents the dereference of the provided pointer in case these conditions are not fulfilled. A programmer can explicitly transform any pointer with a pointer type cast, thus supporting zero copy approaches.

TOAST does not enforce memory safety within a library, as this would require instrumenting every dereferencing operation within the library, which can cause significant overheads. However, TOAST can increase the safety guarantee through its

configuration, such as by enabling the MPK protection library, to enforce safety inside the library as well.

At a protection domain transition, TOAST stores the current access rights and sets the new access rights. The access rights are defined by a method-level manifest; TOAST enables implicit access to memory owned by the protection domain as well as to memory which was explicitly provided by the programmer, i.e., explicit cast. Importantly, the transition is not limited by the access right of the caller, i.e. it does not have to be a subset of the access rights of the caller. While not enforcing these policies may seem counter-intuitive, we decided against them for two reasons. First, TOAST does not protect against malicious behavior. Its goal is mainly to prevent programming errors. Second, this allows simpler transitions in cases of call backs.

5.5 Implementation

5.5.1 TOAST Compiler

The TOAST compiler is built into the clang frontend (v. 16). TOAST leverages clang's code generation to hook its plugin and provide warnings and errors in case of a misuse of its attributes. While this currently restricts our implementation to C/C++, the described techniques are language independent.

Since different devices expose different APIs, the compiler has to deal with various interfaces. The programmer provides the TOAST compiler with a json configuration file, which instructs the compiler to replace specific TOAST annotations with calls to their respective libraries. The compiler then can look up the library in an internal but easily extendable database. This database includes the transformation rules and the checks to be injected.

We integrate the *fmt* [181] library into clang for formatting rewrite rules and the *nlohmann/json* [182] library for reading TOAST config files. The TOAST compiler adds 2217 LoC.

5.5.2 TOAST Runtime Library

We implement a set of templates for the TOAST runtime library. Using these, we implement a networking wrapper based on eRPC [19] as well as a socket wrapper. We further implement a wrapper for persistent memory operations provided by PMDK [36] and a mmap wrapper with read and write semantics similar to PMDK. Additionally, we implement normal file I/O and a wrapper for accessing untrusted memory using OpenSSL [163]. To stay compatible with as many code bases as

possible, the runtime library does not use any libraries except the C++ standard library. The entire TOAST runtime library implementation is 1410 LoC.

5.5.3 TOAST Protection Library

On protection domain transitions, TOAST leverages the type system of the programming language to encode information of the target memory type, which allows TOAST to infer the original protection domain. To enable a programmer to pass a pointer to a buffer to a different protection domain than the original, TOAST provides a type cast between protection domains.

In programming languages that support modules [183], TOAST assumes that each module is a separate library. However, for languages without modules (e.g., C/C++), TOAST requires the user to provide information delineating libraries from each other. This information is a list of regular expressions matching header file names for each library.

Software-based capability storage Capabilities are an efficient method to perform resource management and fine-grained access control, suitable for security-critical systems [47, 49–55]. A capability is a reference to an object or resource together with its access rights. When an application attempts to access a resource (e.g., memory, storage) managed by a capability, the system examines the current capability rights and permits the access or aborts the operation based on them.

In TOAST, every pointer in the user code gets transformed into a capability and is represented as a capability object (CapObj). A capability has an *epoch* and an *address*. TOAST leverages the fact that both x86 and ARM require the use of a canonical pointer form and stores the epoch in the higher unused bits. Thus, the user code cannot directly dereference a capability, as it is an invalid memory address. Further, the address is still encapsulated in the capability and allows correct pointer arithmetic operations.

TOAST's capability protection mechanism splits the address into indices to a multi-level table with configurable width, inspired by the design of a multi-level page table. Each level of the table either includes metadata, i.e. a CapObj, to infer the access rights and check whether the memory region was revoked, or a pointer to the next table level. A CapObj contains *(i)* the epoch in which the memory region was created, *(ii)* its access rights, i.e., read and write access, *(iii)* the prefix of the capability to convert it to its canonical form, and *(iv)* a protection domain ID.

For a capability to be valid, its corresponding CapObj must have the same protection domain ID as the capability. The capability's protection domain ID is

stored in its type and therefore does not add extra data to the runtime. Furthermore, the epoch of the capability should be equal to the epoch of the CapObj, and the epoch that is stored with every protection domain. This allows TOAST to perform fast revocation of memory regions, and only requires deleting CapObj from the capability storage in the event of an overflow of the epoch counter, which should happen very rarely. Note that TOAST increments the epoch of a protection domain whenever it is completely removed from the current execution, e.g., unloading the corresponding library or re-initializing it.

Hardware-assisted protection MPK [46] is an x86 ISA extension that allows for page-level access control. It leverages 4 bits of every page-table entry for a *tag*. The allocation and release of a protection key as well as the page tagging operation require elevated privileges and, therefore, are performed via system calls. However, a process can change the granted permissions for the pages tagged with a specific key in user space by updating a special register (PKRU).

TOAST assigns each protection domain its own memory protection key. Additionally, every protection domain has its own unique allocator per application thread. This implies that each library operates in different address ranges. TOAST's MPK-based protection library leverages this region segregation and intercepts the allocation functions (e.g., malloc, realloc, free) as well as the mmap/munmap operations. In this way, TOAST can tag the pages that a library allocates or maps with the appropriate memory protection key.

On a protection domain transition, TOAST identifies the protection key of the new protection domain and enables the access for the memory regions tagged with this specific key while disabling the rest. Protection key 0 is an exception as it is never disabled. It provides metadata essential for the program's execution. With this approach, an access to a memory region belonging to a different protection domain results in a segmentation fault triggered by MPK. Since MPK's access control is thread-local, application threads can legitimately interact with different libraries simultaneously.

Currently, TOAST supports up to 15 protection domains per application, equal to the number of the available memory protection keys excluding the default one. However, this limitation can be lifted with the use of software tools [184].

5.5.4 Allocation Wrapper

TOAST needs to reliably associate specific memory regions with specific protection domains. To achieve that, TOAST makes the assumption that a protection domain

	Memory regions					LoC		
	NIC	Unprotected	PM	Enclave	DRAM	Original	TOAST	Difference
In-memory KVS		✓		✓		110	105	4.5 %
Replication protocol	✓	✓		✓		893	852	4.6 %
Persistent log	✓		✓		✓	123	120	2.4 %
Persistent KVS	✓		✓		✓	225	182	19.1 %

Table 5.2: Toast case-studies (§ 5.6) with memory regions and LoC for the original version compared with the TOAST version.

requests the resources it requires internally. Further, a memory region associated with a specific protection domain cannot switch protection domains. However, TOAST allows it to be unmapped and remapped in a different protection domain.

To this end, TOAST implements a mmap wrapper. The protection library sets a thread local variable with the necessary information about the library, i.e., current epoch, library ID and a callback to add information to a metadata structure (capability storage or the page table, depending on the implementation in use) at every protection domain transition. On a mmap or unmap call, the mmap wrapper reads data from the thread local variable. It then calls the system's mmap or unmap, respectively. In the case of mmap/unmap returning successfully, the TOAST mmap wrapper calls the callback together with the address, size, epoch and library ID. If the callback function returns successfully the mmap wrapper returns to the library code.

Further, most libraries do not use the allocator that is linked to the application, e.g. the libc allocator. This creates an additional challenge for TOAST, as a standard allocator is not aware of different protection domains. Accordingly, TOAST instantiates a separate allocator per protection domain. These instances do not share any data with each other. TOAST wraps the *malloc*, *realloc*, *aligned_alloc* and *free* calls to branch to the appropriate allocator instance.

Our implementation uses instances of the allocator mimalloc [185, 186]. Unfortunately, heaps created by mimalloc cannot be shared among threads. To alleviate this issue, TOAST instantiates thread-local heaps for each protection domain lazily whenever a new thread spawns. Thus, TOAST makes sure that protection domain-specific data is located in pages owned by the respective domain. Note that mimalloc also contains metadata headers that need to be accessible by the application when reclaiming destroyed heaps or during application exit. Therefore, TOAST currently permits access to this data throughout the execution. This is a limitation imposed by mimalloc that can be resolved by using a multi-heap-capable allocator that separates data from metadata [187].

5.6 TOAST Case-studies

To evaluate TOAST, we port four real-world applications. Table 5.2 lists the memory types they use. These applications are chosen as they represent a wide range of applications using different memory areas.

Secure in-memory KVS The adoption of trusted execution environments (TEEs) resulted in a redesign of secure KVSes [1, 2, 113] to place keys and values in different memory areas to alleviate the memory restrictions of TEEs and improve their performance. We port a secure in-memory KVS [2] that accesses both enclave memory in TEEs and untrusted host memory. The in-memory KVS judiciously partitions the keys along with the values metadata (enclave memory) and values (untrusted host memory) using pointer-based data-structures, i.e., skip lists [22]. We replace these pointers, which cannot easily be identified by a programmer as potentially dangerous, with TOASTPtrs to manage the memory accesses of data in the untrusted memory. The TOAST compiler will transform dereferencing of these TOASTPtr into calls to a TOAST wrapper for the OpenSSL library, performaing de-/encryption and necessary security checks on the keys in host-memory. This transformation also changed the format of the metadata, as the details for proving the integrity of stored value was lifted form the programmers responsibility to the TOASTPtr.

Replication protocol Replication is a standard recipe for fault tolerance. To this end, we adapt an implementation of the ABD replication protocol [138, 158], based on the AVOCADO project [2], to TOAST. To provide a secure distributed in-memory KVS, the secure network stack differentiates between the untrusted NIC and trusted enclave memory. It further uses untrusted host memory to store a copy of the requested values. We port the ABD implementation to use TOASTPtr for both the network interface and the untrusted host memory buffers. Thereby, the TOASTPtr is used for two different memory regions. Firstly, for the ring buffers of the NIC, that is read and writes to the DMA'ed memory of NIC. The TOASTPtr, therefore request the memory from the NIC and inform the NIC of new network packages. Secondly, equivalent to the first case-study, the TOASTPtr together with an wrapper around OpenSSL guarantees the confidentiallity and integrity of the stored host memory buffers.

Persistent shared log Shared logs are used to establish the order of operations in distributed systems [188, 189]. The distributed servers are able to read/write entries

from/to the log, which is also replicated over multiple nodes, guaranteeing fault tolerance. Our log application [179] uses sockets for network communication between the system's nodes and PM as storage. We port both to use TOASTPtr. Thereby similar to the second case study we implement a TOASTPtr handling socket communication, i.e. reads will receive data from the network port, while writes to the memory pointed to by TOASTPtr will send the data to a specific network address. To create the TOASTPtr we implemented an allocator like interface for network communication. This allocator interface is not limited to socket communication. For the persistent memory, TOASTPtr are used to append entries to the log. The TOASTPtr call the a pmdk [36] wrapper when being dereferenced. Furthermore, we also implemented a version which uses memory mapped files instead of persistent memory.

Persistent KVS Persistent KVSs are used to store large amounts of data in storage devices (e.g. HDDs, SSDs). On top of that, persistent KVSs require fast networking to communicate with clients [16, 36]. Like storage technologies, network stacks have shifted to user space [20, 33], which requires applications to differentiate between pointers to storage, network devices, and normal memory. As our use case, we port a MICA implementation [180] running with eRPC [19]. Here, we adapt the network stack to use TOAST. Thereby, reads and writes to the TOASTPtr will be translated into calls into send or receive calls into eRPC respectively.

5.7 Evaluation

We evaluate TOASTacross four axes: programmability (§ 5.7.2), performance (§ 5.7.3), portability (§ 5.7.4) and protection (§ 5.7.5).

5.7.1 Experimental Setup

Experimental testbed We perform our experiments on a cluster of 5 machines with Intel(R) Core(TM) i9-9900K CPUs, each with 8 cores (16 HT), 64 GiB memory, 32 KiB (L1D, L1I), 256 KiB (L2) and 16 MiB (L3) caches, and Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02) NICs.

We measure the performance of the in-memory KVS from our case study and run the micro-benchmark for the protection libraries on a machine with an Intel(R) Xeon Gold(TM) 5317 CPU, with 12 cores (24 HT), 256 GiB memory, 512 KiB (L1D), 384 KiB (L1I), 15 MiB (L2), 18 MiB (L3) caches, as the i9-9900K of our networking setup does not support MPK.

Methodology and baseline As explained in Section 5.6, we port four applications to use TOAST. We measure TOAST's overhead by comparing the performance of TOAST versions to that of unmodified versions. We use the YCSB [18, 142] benchmark for the replication protocol and the in-memory KVS. We perform experiments with various read/write ratios (100%, 99%, 90%, 50%, 0% R) and different value sizes (128 B - 2 KiB). For the persistent shared log and persistent KVS, we use benchmarks provided by the applications themselves.

5.7.2 Programmability

Q1: *How easy is it to design applications using the* TOAST*Ptr abstraction?* To answer this question, we count the lines of code (LoC) modified in TOAST compared to the original hand-written version for each TOAST use case (see Table 5.2).

The TOAST version reduces the number of LoC in every application by 2.4 % - 19.1 %. Moreover, TOAST simplifies or eliminates complicated function calls for buffer resizing or message enqueueing.

Q1 takeaway: TOAST significantly reduces the number of LoC of an application. However, our experiments show that TOAST's benefits go beyond simply reducing LoC: TOAST improves programmability by allowing the programmer to manage various DMA-capable devices used in an application with the same interface, instead of having to learn and employ device-specific APIs. Thus, the developer can focus on the code's logic rather than boilerplate library calls.

5.7.3 Performance

Q2: *What is the overhead of using* TOAST *compared to manually optimized code?* To answer this question, we compare TOAST versions of our four applications to the hand-optimized (unmodified original) versions by measuring their throughput (in operations per second, Op/s).

Secure in-memory KVS We run the YCSB benchmark with 400 MOp over 10 M distinct keys following a uniform key distribution with different read-write ratios.

In a read heavy workload (99 % reads), the overhead introduced by TOASTPtr is 2.8 % which increases to 11.9 % for write heavy workloads of 50 % writes and reads. This overhead is mainly due to TOAST not caching decrypted data in the trusted memory, but repeatedly decrypting it on every access. As a write can generate two accesses to the same buffer, this effect is more noticeable in write intensive workloads. TOAST's overhead could be reduced by creating temporary objects with

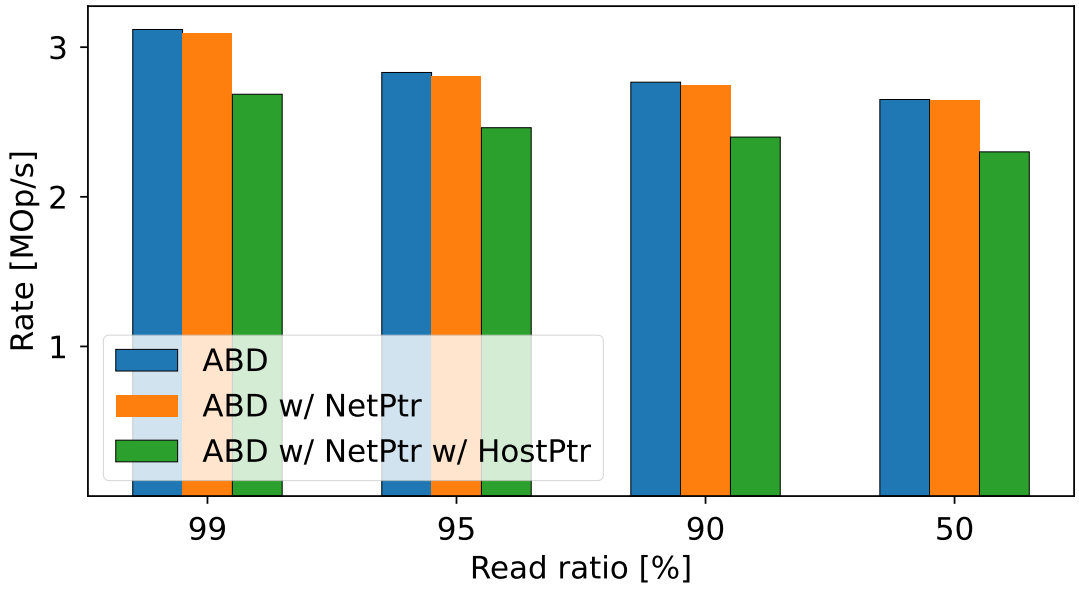

Figure 5.4: Overhead of TOASTPtr on the throughput of the replication protocol with TOASTPtr being used for the networking or networking and unprotected memory in the YCSB benchmark for different read/write ratios.

lifetimes greater than individual operations. However, we did not implement this optimization in TOAST, as it affects synchronization among threads.

Replication protocol We compare the performance of the hand-optimized ABD replication protocol to two TOAST counterparts. One counterpart uses TOASTPtr to access the NIC to perform network communication, while the other additionally uses TOASTPtr to access unprotected memory for its internal KVS. We run the YCSB benchmark, with different read/write ratios (Fig. 5.4) and different value sizes (Fig. 5.5). We run the protocol on all five servers. The benchmarks were configured with 1.2 GOp over 2.5 M distinct keys following a uniform key distribution. We measure the overhead of TOASTPtr on the performance of ABD for the read ratios of 99 %, 95 %, 90 %, and 50 % with a value size of 128 B. The overhead of TOAST for the networking library is 0.84 % for a read ratio of 99 %, which shrinks down to 0.23 % as the write ratio increases to 50 %. Using TOASTPtr also for the unprotected memory increases the overhead to 13.2 % and 13.8 % for read ratios of 50 % and 99 %, respectively. Like the secure in-memory KVS, the increased overhead of TOASTPtr for unprotected memory is due to caching of data in protected memory in the hand-optimized version of the ABD protocol.

The overhead of the NIC-only use of TOASTPtr is generally not affected by the value size and is stable between 0.6 and 0.8 %, until the value size exceeds the MTU

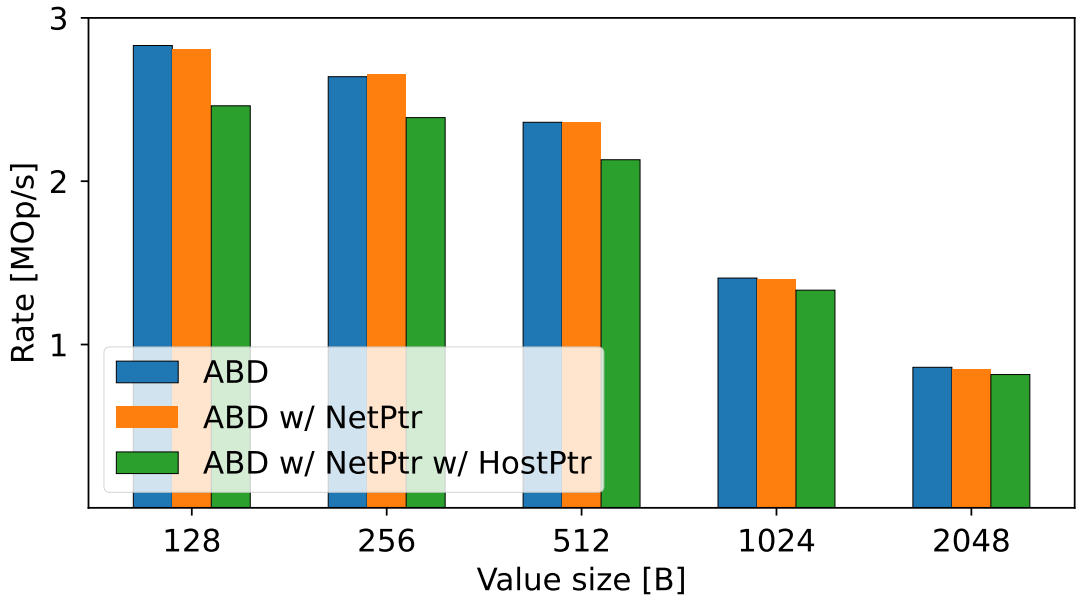

Figure 5.5: Overhead of TOASTPtr on the throughput of the replication protocol with TOASTPtr being used for the networking or networking and unprotected memory in the YCSB benchmark for different value sizes.

size (1500 B) of the network packets, e.g., at value size 2 KiB, the overhead is 1.5 %. Exceeding the MTU size requires making an additional copy of each value (in eRPC) to split it into multiple packets.

Shared log We run the shared log application with 1 server thread and 2 clients each having 8 threads, the largest configuration the benchmark application allowed. The entry size ranges from 64 B to 2 KiB. Figure 5.8 shows the throughput of both the original and the TOAST version. TOAST performs on par with the original version for all log entry sizes, with a mean performance difference of around 1.8 %.

Persistent KVS We measure the throughput of the persistent KVS using a server application with 16 threads and 4 clients each with 16 threads to generate the workload. We used a uniform key distribution and read-ratio of 0 %, 50 %, 100 %. Figure 5.6 shows that the original and TOAST versions have similar performance.

Q2 takeaway: In general, TOAST introduces negligible performance overhead on the ported applications (< 2.8 % relative to the original version). However, since TOAST provides a generic programming model without specialized optimizations (e.g., selective data caching), in some cases, higher overheads can be observed.

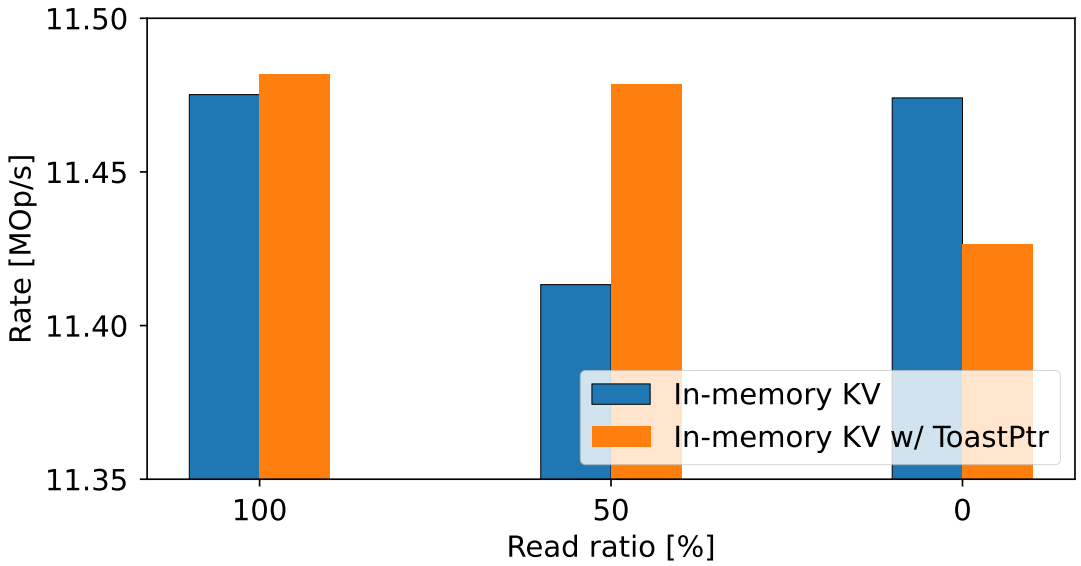

Figure 5.6: Overhead of TOASTPtr version of persistent KVS compared to hand optimized version for different read ratios.

5.7.4 Portability

Q3: *How easy is it to switch underlying technologies with* TOAST*?* To evaluate this, we present two cases studies of the network and storage libraries. Recall that specialized libraries are required to interact with the underlying hardware correctly. While we do not change the underlying hardware in the first case study, we do change the technology from a traditional OS stack approach to a user-level approach and in the second we do switch the underlying hardware from a SSD to PMEM.

Network library We highlight the portability of the network stack from traditional sockets to eRPC [19] for the persistent shared log. The TOAST version requires changing 71 LoC while porting the original code requires changes in 141 LoC. This 50 % decrease occurs due to the different implementations of asynchronous calls between the versions. TOAST's 71 LoC can be reduced further by introducing a unified asynchronous call interface in a future TOAST version.

Storage library We port the the same persistent shared log library from using memory mapped files to using a PM library (PMDK [36]). The TOAST version requires changes to 19 lines, all in the initialization phase. The hand-written port requires the same changes and an additional 20 LoC in the storage backend logic, including changing specialized memcpy and synchronization methods.

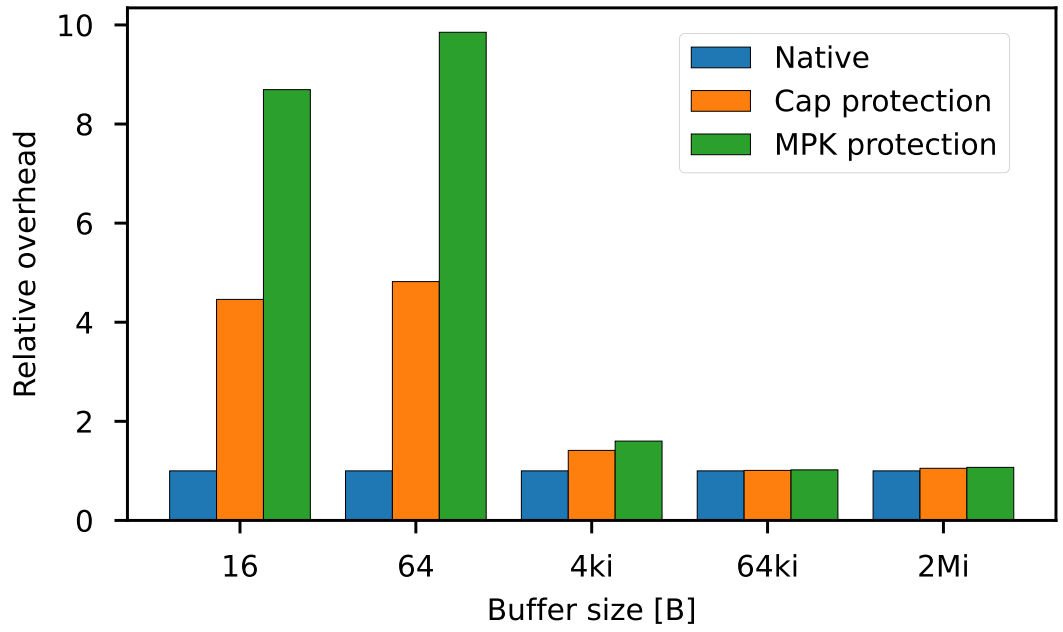

Figure 5.7: Runtime overhead of TOASTprotection mechanisms w.r.t. to the native execution for protection domain transitions performing a *memcpy* with various buffer sizes.

Q3 takeaway: TOAST significantly simplifies the porting process of an application to use a different underlying technology. Our experiments show that TOAST can reduce the number of modified LoC by up to 50 %.

5.7.5 Protection

Q4: *What are the implications and trade-offs, in terms of performance and safety, of* TOAST*'s protection mechanisms?*

To provide an answer, we evaluate the performance overheads introduced by the capability- and Intel MPK-based protection mechanisms provided by TOAST (§ 5.4.3). First, we design a microbenchmark that performs calls to a wrapped *memcpy* function repeatedly through a linked library call, thus performing protection domain transitions. Our microbenchmark uses two protection domains, while we expect to see a slightly larger number of protection domains in real applications. Each *memcpy* function call operates on memory regions accessible from the protection domain of its call site. Additionally, we apply the protections mechanisms to the TOAST skip list and shared log. Note that, like the MPK-based version, the capability-based version is configured to provide protection guarantees at a page-size granularity in our experiments.

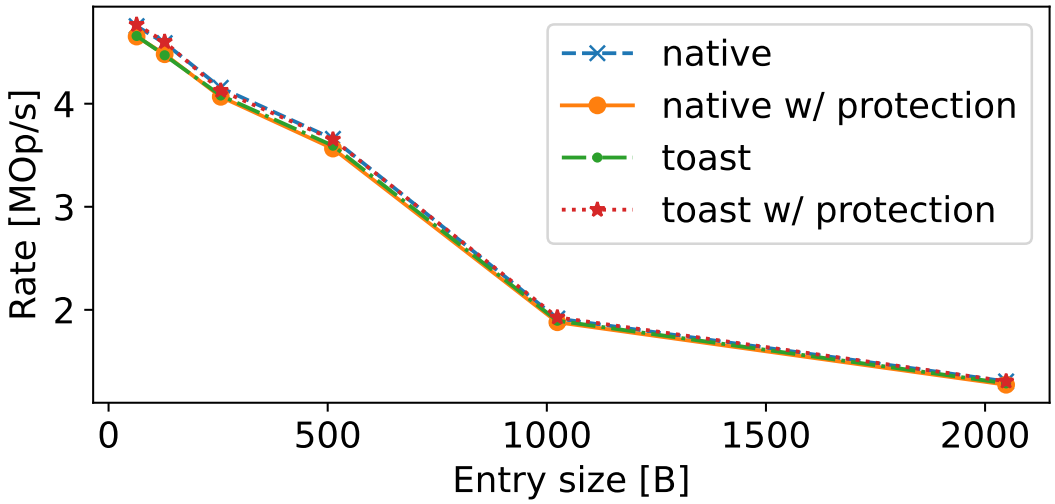

Figure 5.8: Throughput of shared log for different log entry sizes for with and without protection library enabled.

Microbenchmark We configure our microbenchmark to copy 20 GB of data between the protection domains. We can for example assume a file server reading the data from a SSD and sending it to another server. We vary the copied buffer size to highlight the cost of the domain transitions. Experiments with smaller buffer sizes require more *memcpy* operations, and consequently, more transitions to the protection domain of the linked library. We measure the total time required till all the data has been copied. The presented results indicate the mean of 100 runs for each configuration setup.

Figure 5.7 illustrates the relative slowdown of the capability- and MPK-based protection libraries compared to the native execution of our microbenchmark. For small buffer sizes (16 B and 64 B), the capability protection mechanism is 4.46-4.81× slower than the baseline. The respective values for the MPK version are 8.69-9.85×. This large slowdown is caused by the frequent, short-running transitions between the protection domains, which, in turn, results in more checks and pointer cleanups in the capability version, and more costly updates of the PKRU that can lead to pipeline stalls in the MPK version. However, as the buffer size increases, the TOAST protection mechanisms induce lower overheads. When copies are performed at the granularity of a page (4 kB), the overheads are 41 % and 60 %, for the capability- and the MPK-based approach, respectively. Lastly, we observe that for even larger buffers (64 kB and 2 MB), TOAST's protection libraries incur only 1 −7 % slowdown since the domain transition overhead is dominated by the longer *memcpy* operations between domain transitions.

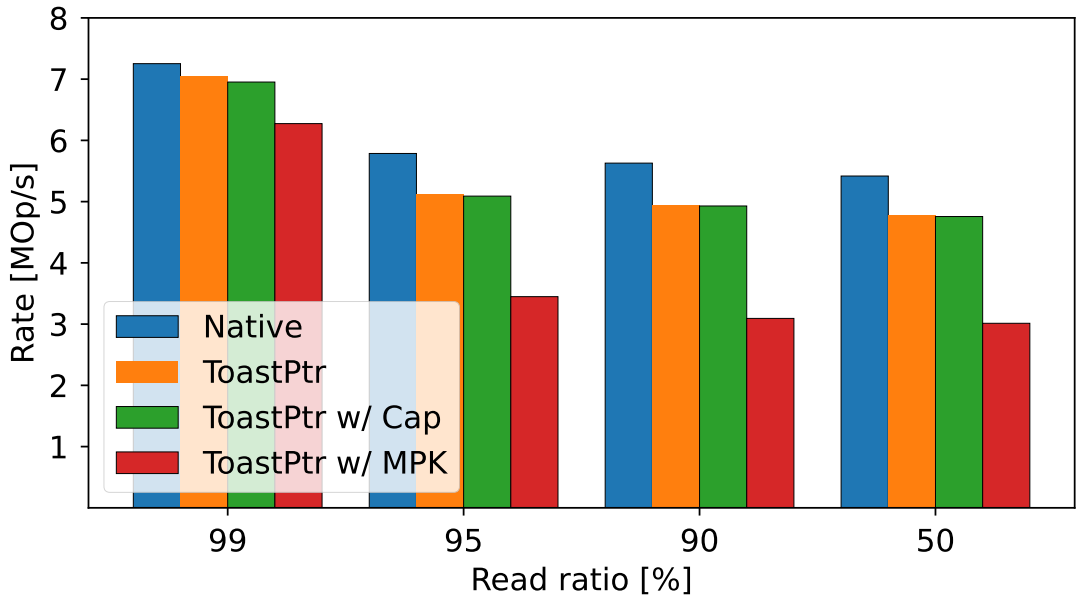

Figure 5.9: Overhead of different protection library implementation for the in-memory KVS for various read ratios.

Shared log We run the shared log application with a setup identical to that of § 5.7.3. We divide the shared log application into a networking and a storage protection domain, with the networking buffer having to pass through the protection domain switch. To evaluate the overhead of the protection domain switch, we run the benchmark in four configurations: the original shared log application, the same application without TOASTPtr but with the protection library, the application with TOASTPtr but without the protection library, and a configuration using both TOASTPtr and protection library. We execute the experiments as described in § 5.7.1.

Figure 5.8 shows the overhead of the capability based protection library in the shared log application. The capability protection library adds 2.2 to 2.5 % overhead to the native solution. The capability protection library version even performs slightly better than the unprotected TOAST version having 1.5 to 2.5 % higher average throughput. The low overhead of the capability version in this application is expected, as most memory is allocated by the user code and then supplied to libraries. As the protection domains in TOAST are designed only to prevent programming errors, not intentional sharing, most checks in this case are optimized away, since the user explicitly allows the access via a cast.

Secure in-memory KVS We run the YCSB benchmark with 400 MOps sampled uniformly from 10 M distinct keys. Figure 5.9 shows the overhead of the different protection libraries for different read-write ratios, with a key size of 8 B and value

size of 128 B. The capability version has an overhead of 1.5 % for the 99 % read workload. With higher write ratios, the overhead shrinks to 0.3 %. The difference in overhead of the capability version in read heavy workloads compared to write heavy workloads is mainly due to read operations having to perform a full capability storage lookup as the read buffer is provided by the library and therefore needs to be transformed into a capability. However, write operations can assume a fast path as the write buffer is allocated in the user code and the user explicitly provides the buffer to the library. This does not require a costly lookup and rewriting of the pointer. The MPK-based protection library version incurs a slowdown of 11.0 % to 37.3 % for the various workloads. We observe that the overhead decreases as the read ratio increases. This is expected as the fewer put operations imply less frequent memory allocations and a smaller application memory footprint which, in turn, leads to fewer page tagging operations, which are expensive. (The number of PKRU register updates depends only on the number of protection domain transitions, and is the same for all read ratios.)

Q4 takeaway: TOAST allows developers to choose which protection mechanism suits their application better depending on the memory access patterns and the desired memory-safety granularity. The overheads of the mechanisms will vary for each case but remain reasonable.

5.8 Related Work

OS memory management The OS provides drivers to communicate with devices [190, 191] on the kernel side and sockets/file descriptors on the user space side. However, modern systems prefer user space libraries to directly communicate with the device and manage heterogeneous memory areas for improved performance, as in DPDK [20], RDMA [33, 35, 175] and eRPC [19] for remote calls, SPDK [16] for SSDs, or PMDK [36] for PM. However, there is no unifying abstraction across these libraries.

Unified API efforts like oneapi [176, 192] focus on specific device classes, e.g., GPUs/FPGAs, and introduce one API per use-case. Likewise, memif [193] provides OS services to speed up memory operations between heterogeneous memory areas. Enclosure [194] uses a combination of a compiler pass and annotations to create packages for legacy code, which gets isolated in contiguous memory regions. Overall, prior approaches rely on a combination of compiler support and runtime libraries. While our idea of unifying APIs is similar, TOAST goes further by lifting the communication completely into the compiler and removing special API calls, while

also considering the safety aspect.

Compiler-based memory management Compiler-based approaches are used in shared memory systems to optimize memory accesses, e.g., UPC [195], OpenMP [196], HPF [197]. Other research has looked into using DMA support in the compiler for heterogeneous compute units [198] or compiler-based approaches for secure memory management [199, 200]. The programming language Verona [201] introduces concurrent access for separate memory types. Static analysis can be used to determine if an object should be stored in slow SRAM or DRAM on cacheless devices [202]. Other work [203] uses compiler transformations on data structures to optimize them for heterogeneous memory types. However, none of these approaches deals with different memory layouts and access patterns based on heterogeneous memory types.

Software-based protection & isolation Traditionally, the OS enforces memory isolation between different processes. Other works propose intra-process memory isolation by introducing capabilities to memory areas and system calls to either threads [204, 205] or objects which can be held by a thread [206]. Two other common techniques to provide isolation are Software Fault Isolation (SFI) [207] and Control-Flow Integrity (CFI) [208]. Other software-based approaches rely on sandboxing to prevent illegal accesses, which cannot be proven correct statically [209–211].

TOAST does not provide strict memory isolation between different components. Instead, it aims at preventing erroneous sharing of sensitive data. It limits code injection to protection domain transitions, and does not require every access to be secured, reducing the amount of injected code and performance overhead, while also being easier to integrate.

Hardware-based protection & isolation CHERI [55], IBM System 38 [212, 213], M-Machine [214] and ARM MTE [215] are examples of hardware support for fat pointers. Other hardware approaches are Page Groups, e.g. HP PA-RISC [216], Intel MPK [217] and ARM Domains [218] that tag memory areas, and Mondrian Memory Protection [219] that separates access rights from translation metadata. These approaches can force access to specific memory regions to go through designated access control gates, thus preventing erroneous accesses. Another hardware-based approach is capability storage systems, such as Intel iAPX 432 [220] and CODOM [221]. This is different from fat pointer schemes, as metadata is stored in multi-level tables. In contrast to these approaches, TOAST capabilities do not require

hardware support. Further, TOAST protection aims to unify the access APIs of different kinds of memory, while maintaining easy-to-use for the programmer.

5.9 Summary

We present TOAST, a compiler-based abstraction for heterogeneous memory management. TOAST builds on the observation that although accesses to heterogeneous memory require different libraries with vastly different interfaces, all interfaces essentially perform the same basic task of loading data from or storing data to a memory region. TOAST makes this uniform for the programmer by introducing the abstractions of memory types and the pointer type TOASTPtr, that work with familiar load and store operations. Further, TOAST provides programmable error handling callbacks as part of the programming model. Lastly, TOAST offers a selection of protection libraries to prevent accidental memory handling errors by developers. Our evaluation on four real-world applications shows that TOAST improves programmability, offers memory safety, and eases the portability to new libraries/memory types, with low to moderate overhead relative to hand-optimized code.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

With the adoption of TEEs into their products, cloud providers have shown an interest into providing a secure and trusted environment for their clients. Modern services have high requirements in availability and scalability of their data storage, which is in odds with the transient single node design of TEEs. This thesis looks at different aspects of storage systems and how to overcome the limitations of TEEs to provide scalable, available and persistent storage.

With SPEICHER we present a secure persistent single node KVS, which provides three important security guarantees: confidentiality, integrity and freshness. SPEICHER achieves these security guarantees by encrypting, integrity protecting and hardening LSM tree data structures for the persistent storage. For freshness SPEICHER provides an asynchronous trusted counter service on top of existing trusted hardware counters, which allows SPEICHER to overlap its freshness guarantees with persistence guarantees. Further SPEICHER introduces a in-memory skip list design, which is split into a trusted and untrusted part, for the lower levels of the LSM data structures, alleviating trusted memory pressure, while at the same time allowing for fast lookup speeds. SPEICHER also provides a fast secure user space direct I/O storage stack to access the persistent storage. These parts, i.e., the secure skip list and the secure storage stack, reduce the performance overhead of SPEICHER to a reasonable 15 to 35 $\times$.

AVOCADO similarly to SPEICHER provides a KVS, however in contrast to SPEICHER it does not provide persistence, instead it focuses on scalability and availability. AVOCADO's in-memory KVS is based on SPEICHER's skip list design, however extends it for multi-threaded lockless writing and provides additional meta-data storage to store a per key Lamport clock. We, further, observed that a

non-BFT replication protocol, MW-ABD, can be harden with TEEs to provide secure replication, since it allows us to model all possible attacks on the protocol as fault crashes. To further harden the replication protocol and to increase the performance, we designed a secure direct I/O network library with complete support for the transport and session layer. Lastly, we build a CAS service which dramatically decreases the latency of attestation for a new node, while also providing a convenient way to provide the trusted nodes with their respectively configuration. AVOCADO performs 4 to 65 $\times$ better than BFT in our configuration while also providing confidentiality and reducing the number of replica by f.

TOAST is a generic compiler-based abstraction for heterogeneous memory management including TEEs' memory models to improve programmability, portability, and memory safety. It unifies the API for different libraries accessing heterogeneous memory, by observing that the libraries fundamentally are based on a load store semantic. TOAST, therefore, introduce the concept of a TOASTPtr which works with the familiar pointer interface and an unified callback based error handling mechanism. Further, TOAST increases portability due to a easily configurable compiler pass and a proxy library. Additionally, TOAST provides memory safety mechanism through a protection library, which prevents accidental data leakage. TOAST performs similar to a hand-optimized version in our evaluation, while improving programmability and portability and offering memory safety.

Each of these projects provides different tools to design a secure storage system. A user can combine these projects, or parts thereof, to build their system depending on their requirements.

6.2 Future Work

SPEICHER is a KVS that provides a limited set of query operations, such as range queries. In order to support more complex query operations and transactions, it may be necessary to redesign the log data structure and implement improved memory management strategies. This is mainly due to two reasons: firstly, because the limited amount of enclave memory available in Intel SGX version 1, on which SPEICHER is based, may not be sufficient to store the temporary data required for tracking transactions, secondly the transactions' data structures have to be harden to provide confidentiality, integrity and freshness.

AVOCADO shows that with the integration of a secure network stack, leaderless fault crash replication protocols can be used in conjunction with TEEs to build a secure distributed KVS. This leads to the assumption that leader-based replication protocols

might also be adoptable to untrusted network.

Similar to SPEICHER, AVOCADO does not support transactions, which limits its usefulness to certain applications and prevents it from being used as a general purpose KVS replacement.

By combining the secure persistence traits of SPEICHER with the replication protocol and advanced in-memory KVS of AVOCADO, a persistent distributed KVS can be designed.

In addition to enhancing the capabilities of SPEICHER and AVOCADO by combining them, there is also potential to extend them to support distributed transactions. TREATY [4], which is based on SPEICHER, has already investigated the use of SPEICHER for secure distributed transactions.

Another interesting area for future development is the support of file system interfaces. SPEICHER's fundamental building blocks are not limited to KVS, and could potentially be used to support the storage and management of files and directories.

Recent developments in TEE designs, such as the increased fast-access trusted memory available in SGX v2, as well as the shift in TEE design philosophy towards a virtual machine (VM) approach, also offer opportunities to reevaluate some of the design decisions made in SPEICHER and AVOCADO. These changes may enable the use of larger in-memory data structures and support for a larger software stack inside the trusted environment, potentially enabling future projects to focus on hardening the software stack rather than designing for low TCB TEEs like SGX.

TOAST currently only supports accessing data on a DMA area. However, more and more specialized accelerators are introduced to speed up the processing of specific tasks, such as GPUs and FPGAs. Many of these accelerators are programmable. TOAST, however, does not have support for programming the accelerators as it is designed with data only in mind. Further, the programmable nature of these devices also results in dynamic access pattern changes, which TOAST does not support. Future projects may explore ways to integrate programmable accelerators into the TOAST environment.

The TOAST interface is based on a pointer abstraction, to instrument dereference operations, but not all languages, such as functional languages, have a pointer semantic. This raises the question of how TOAST could support these languages. One idea might be to instead of using the pointer abstraction to instrument the data itself.

Many of the features explored by TOAST could also be used to provide safety for a multi-tenant systems. For example, the indirection introduced by TOAST could be used to add additional checks to ensure memory safety and prevent unauthorized

access to memory regions. The memory regions defined by TOAST could be extended to support multiple tenants, although a multi-tenant system is much more dynamic than the heterogeneous memory system TOAST is designed for.

6.3 Code availability

For the source code of all three projects.

- SPEICHER's code is based on a port of SPDK [16], DPDK [20] and RocksDB [17], and an internal SCONE [40] version. We did not release SPEICHER's source code, as it contains source code from the SCONE project.
- AVOCADO's source code is available under `https://github.com/mbailleu/avocado`, it depends on a modified DPDK version (available at `https://github.com/mbailleu/SpeicherDPDK`) and a modifies eRPC version (available at `https://github.com/mbailleu/avocado-eRPC`)
- TOAST is based on LLVM/clang [177, 178]. The source code for the TOAST's plugin, runtime, and protection library will be available at `https://github.com/TUM-DSE/toast` after publication of the TOAST paper.

Appendix A

SPEICHER Algorithms

In this appendix, we present the pseudocode for all data storage and query operations in SPEICHER.

Input: KV-pair which should be inserted into the store.
Result: Freshness of MemTable

```
/* Generating a block with the trusted counter                                    */
hash_Block ← hash(KV, counter_WAL + 1);
block ← encrypt(KV, counter_WAL + 1, hash_Block);
/* Writing the block to the persistent storage, before the trusted counter gets incremented */
write_WAL(block);
counter_WAL ← counter_WAL + 1;
/* Generating hash over the KV-pair for the Memtable                              */
hash_KV ← hash(KV);
/* Trying to insert into the memtable, if the memtable is corrupted return a failure */
freshness ← putIntoMemtable(KV, hash_KV);
return freshness
```

Algorithm 1: Put algorithm of SPEICHER

Input: Key in the format of the KV-store
Result: Freshness of the KV-pair and Value
for $level = 0$ **to** $numberoflevels$ **do** /* Check in each level if key-value is existend, from highest to lowest */

- **if** $level = Level_0$ **then** /* First level lookup therefore lookup in MemTable */
  - $path, value \leftarrow$ `lookupMemtable`(key) /* It is possible that the value is empty, however we still have to do a proof of non-existence */
  - **foreach** $node \in path$ **do** /* Validate hash values of the trace to the leaf node */
    - **if** `hash`$(node.left, node.right) \neq node.hash$ **then** /* check that the hash value of the child nodes is equal to the stored hash value */
      - /* The integrity and freshness proof failed */
      - **return** $stale_{MemTable}, value$
    - **end**
  - **end**
  - **return** $fresh, value$
- **end**
- **else** /* Lookup in a level backup by SST files */
  - $SST \leftarrow$ `findSSTFile`$(level, key)$ /* Lookup over authentication structures similar to MemTable */
  - $block, value \leftarrow$ `lookup`$(SSTs_{level}, key)$;
  - **if** `hash`$(block) \neq SST$.`hashBlock`$(block)$ *or* !`freshness`(SST) **then**
    - **return** $stale_{SST}, value$
  - **end**
  - **return** $fresh, value$
- **end**

end

Algorithm 2: Get algorithm of SPEICHER

Input: KV-pair with the lowest key and callback method to the client
/* Build an iterator pointing to the first KV-pair */
$iterator \leftarrow$ `constructIterator`(key_{min});
$next \leftarrow True$;
/* Call the provided function until the iterator is not valid anymore or a freshness proof failed or the client request to end */
while `isValid`$(iterator)$ *and* $state = fresh$ *and* $next$ **do**
- $state, value \leftarrow Iterator.key\_value$;
- $next \leftarrow$ `callback`$(state, value)$;
- $Iterator \leftarrow Iterator.next$;

end

Algorithm 3: Range query algorithm of SPEICHER

Input: Start key
Result: Result of freshness proof or iterator
Function `constructIterator`(key_{min})

```
/* Build an iterator for each level of the LSM pointing to the KV-pair or the next pair in the
   level                                                                                    */
```

foreach $level \in Level$ **do**
  $iterator_{level} \leftarrow$ `lowerBound`$(level, key)$;
  if $iterator_{level}.state \neq fresh$ **then**
    return *state*
  end
  `iterator.add`$(iterator_{level})$;
end
end
Input: iterator
Result: Iterator points to the next KV-pair and freshness of the iterator
Function `next`$(iterator)$

```
/* Forward all iterators pointing to the current key                                        */
```

foreach $iterator_{level} \in iterator$ *where* $iterator_{level}.key = iterator.key$ **do**
  `next`$(iterator_{level})$;
  if $iterator_{level}.state \neq fresh$ **then**
    return $iterator_{level}.state$
  end
end

```
/* Find the level iterator pointing to the lowest key                                       */
```

for $i = 0$ **to** *number_levels* **do**
  $iter \leftarrow iterator[i]$;
  if $iter.state \neq fresh$ **then**
    return $iter.state$
  end
  if $key_{lowest} > iter.key$ **then**
    $key_{lowest} \leftarrow iter.key$;
    $level \leftarrow i$
  end
end
`iterator.currentLevel`(i);
return *fresh*
end

Algorithm 4: Iterator functions of SPEICHER

Input: Manifest File
Result: Restored KV-store

```
/* Get the counter value of the first record in the manifest and check that the first record is an
   inital record                                                                              */
counter ← Manifest.firstCounterValue;
/* Iterate over all records in the Manifest                                                   */
foreach record_encrypted ∈ Manifest do
    record ← decrypt;
    hash ← hash(record);
    /* Check the records hash and counter value, if they do not match, report an error to the client
       */
    if hash ≠ record.hash then
        return Hash does not match
    end
    if counter ≠ record.counter then
        return Counter does not match
    end
    /* If hash and counter match apply the change to the KV-store                             */
    apply(record);
    inc(counter);
end
/* Check if the last counter in the Manifest matches the trusted counter, if not report an error to
   the client                                                                                 */
if counter ≠ tusted_counter_Manifest then
    return Counter does not match
end
/* Get the current WAL and its initial counter value from the Manifest                        */
counter ← Manifest.firstWALCounter;
/* Apply each record of the WAL to the KV if the counter and hash are correct, similar to the
   Manifest                                                                                   */
foreach record_encrypted ∈ WAL do
    record ← decrypt;
    hash ← hash(record);
    if hash ≠ record.hash then
        return Hash does notmatch
    end
    if counter ≠ record.counter then
        return Counter does not match
    end
    apply(record);
    inc(counter);
end
/* Check if the last counter value is the same as the trusted counter                         */
if counter ≠ trusted_counter_WAL then
    return Counter does not match
end
/* KV-store was successfully restored and no integrity or rollbacks problem were found       */
return Success
```

Algorithm 5: Restore algorithm of SPEICHER

Input: SSTable file to be compacted one from level$_n$
Result: Multiple SSTable files for level$_{n+1}$
// Creating an Iterator over the higher level SSTable file create a new file and a new data block
$iterator_n \leftarrow$ `createIterator`$(SSTable_n)$;
$NewSSTable \leftarrow$ `createNewSST`();
$block \leftarrow$ `createNewBlock`();
$last\_key \leftarrow$ `iterator`$_n$`.key` $- 1$;
// As long as their are KV-pairs remaining in the SSTable open the SSTable file in the next level which has the range of the smallest possible next key based on the last key compacted. **while** `has_next`$(iterator_n)$ **do**
    $SSTable_{n+1} \leftarrow$ `findSSTFile`$(n+1, last\_key+1)$;
    $iterator_{n+1} \leftarrow$ `createIterator`$(SSTable_{n+1})$;
    // As long as the currently open SST_{n+1} file has KV-pairs find the smaller next key of SST_n and SST_{n+1} file. If both have the same next key choose from SST_n file.
    while `has_next`$(iterator_{n+1}i)$ **do**
        $iterator_{min} \leftarrow$ `min`$(iterator_n, iterator_{n+1})$;
        // test if the key value is still fresh, that is check the hash of the block compare in the SSTable file hash footer and check against the Manifest
        if $iterator_{min} \neq fresh$ **then**
            // If the key value is not fresh return error to client
            return $iterator_{min}.state$
        end
        // Add key to block, if the block is then over the size limit for blocks calculate a hash add the hash to the footer of the new file and write the block to persistent storage, and create a new block
        `block.add`$(iterator_{min}.kv)$;
        if `size`$(block) > block\_size\_limit$ **then**
            $hash \leftarrow$ `hash`$(block)$;
            $encrypted\_block \leftarrow$ `encrypt`$(block)$;
            `NewSSTable.write`$(encrypted_b lock)$;
            `NewSSTable.addHash`$(hash)$;
            // If the file reaches the size limit after an append, write the footer to the storage and create a new SSTable
            if `size`$(NewSSTable) > SSTable\_size\_limit$ **then**
                `NewSSTable.writeFooter`();
                $NewSSTable \leftarrow$ `createNewSST`();
            end
            $block \leftarrow createNewBlock$;
        end
        $last\_key = iterator_{min}.key$;
        `next`$(iterator_{min})$;
    end
end
// After compaction, flush the block & write the footer. $hash \leftarrow$ `hash`$(block)$;
$encrypted\_block \leftarrow$ `encrypt`$(block)$;
`NewSSTtable.write`$(encrypted\_block)$;
`NewSSTable.addHash`$(hash)$;
`NewSSTable.writeFooter`();
// Write the changes to the Manifest file.
`Manifest.remove`$(\mathtt{SST_n}, \mathtt{SST_{n+1}inrangeofSST_n})$;
`Manifest.add`$(\forall$`NewSSTfile`$)$;

Algorithm 6: Compaction algorithm of SPEICHER

Bibliography

[1] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution". In: *17th USENIX Conference on File and Storage Technologies (FAST)*. 2019.

[2] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. "Avocado: A Secure In-Memory Distributed Storage System". In: *2021 USENIX Annual Technical Conference (ATC'21)*. 2021.

[3] Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzer. "TEE-Perf: A Profiler for Trusted Execution Environments". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019.

[4] Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia. "Treaty: Secure Distributed Transactions". In: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2022.

[5] *Intel Software Guard Extensions (Intel SGX)*. `https://software.intel.com/en-us/sgx`. Last accessed: Jan, 2021.

[6] *Intel Trust Domain Extensions (Intel TDX)*. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html`. Last accessed: Dec, 2022.

[7] *Building a Secure System using TrustZone Technology*. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`. Last accessed: Jan, 2021.

[8] *ARM - Architecture - Confidential Compute Architecture*. `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`. Last accessed: Dec, 2022.

[9] AMD. *AMD Secure Encrypted Virtualization (SEV)*. https://developer.amd.com/sev/. Last accessed: Jan, 2021.

[10] RISC-V. *Keystone Open-source Secure Hardware Enclave*. URL: https://keystone-enclave.org/.

[11] *Introducing Google Cloud Confidential Computing with Confidential VMs*. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms.

[12] Microsoft Azure. *Azure confidential computing*. https://azure.microsoft.com/en-us/solutions/confidential-compute. Last accessed: Dec, 2022.

[13] *TEE-based confidential computing*. https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/tee-based-confidential-computing-tee-based-confidential-computing. Last accessed: Dec, 2022.

[14] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The Log-structured Merge-tree (LSM-tree)". In: *Acta Inf.* 1996.

[15] R. C. Merkle. "Protocols for Public Key Cryptosystems". In: *1980 IEEE Symposium on Security and Privacy*. 1980.

[16] *Intel Storage Performance Development Kit*. http://www.spdk.io. Last accessed: Dec, 2018.

[17] *RocksDB | A persistent key-value store*. https://rocksdb.org/. Last accessed: Dec, 2018.

[18] *YCSB*. https://github.com/brianfrankcooper/YCSB. Last accessed: Jan, 2022.

[19] Anuj Kalia, Michael Kaminsky, and David Andersen. "Datacenter RPCs can be General and Fast". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2019.

[20] *Intel DPDK*. http://dpdk.org/. Last accessed: Jan, 2021.

[21] *LevelDB*. http://leveldb.org/. Last accessed: Dec, 2018.

[22] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Communication of ACM (CACM)* (1990).

[23] *Consistency and Replication Model*. https://docs.hazelcast.com/imdg/4.2/consistency-and-replication/consistency. Last accessed: Dec, 2022.

[24] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. "Sinfonia: A New Paradigm for Building Scalable Distributed Systems". In: *ACM SIGOPS Operating Systems Review (SIGOPS)*. 2007.

[25] Brad Fitzpatrick. “Distributed Caching with Memcached”. In: *Linux Journal* (2004).

[26] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. “Memcached Design on High Performance RDMA Capable Interconnects”. In: *International Conference on Parallel Processing (ICPP)*. 2011.

[27] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur-Rahman, H. Wang, S. Narravula, and D. K. Panda. “Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports”. In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*. 2012.

[28] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. “High-Performance Design of HBase with RDMA over InfiniBand”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*. 2012.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *ACM SIGOPS Operating Systems Review (SIGOPS)* (2007).

[30] Avinash Lakshman and Prashant Malik. “Cassandra: structured storage system on a p2p network”. In: *Proceedings of the 28th ACM Symposium on Principles of distributed computing (PODC)*. ACM. 2009.

[31] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. “TAO: Facebook’s Distributed Data Store for the Social Graph”. In: *USENIX Annual Technical Conference (USENIX ATC)*. 2013.

[32] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. “The RAMCloud Storage System”. In: (2015).

[33] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. “FaRM: Fast Remote Memory”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[34] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*. 2013.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 2016.

[36] *Intel Persistent Memory Development Kit (PMDK)*. `https://pmem.io/pmdk/`. Last accessed: Dec, 2021.

[37] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[38] Meni Orenbach, Marina Minkin, Pavel Lifshits, and Mark Silberstein. "Eleos: ExitLess OS services for SGX enclaves". In: *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*. 2017.

[39] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[40] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[41] Chia-Che Tsai, Donald E Porter, and Mona Vij. "Graphene-SGX: A practical library OS for unmodified applications on SGX". In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 2017.

[42] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "PANOPLY: Low-TCB Linux Applications with SGX Enclaves". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[43] *Asylo: An open and flexible framework for enclave applications*. `https://asylo.dev/`.

[44] *Open Enclave SDK: Build Trusted Execution Environment based applications to help protect data in use with an open source SDK that provides consistent API across enclave technologies as well as all platforms from cloud to edge.* `https://openenclave.io/sdk/`. Last accessed: Jan, 2021.

[45] *CCF documentation*. `https://microsoft.github.io/CCF/master/`. Last accessed: Jan, 2021.

[46] LWN-Jonathan Corbet. *Memory Protection Keys*. `https://lwn.net/Articles/643797/`. Last accessed: Oct, 2022.

[47] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "SeL4: Formal Verification of an Operating-System Kernel". In: *Commun. ACM* (2010).

[48] *L4Re Operating System Framework*. `https://l4re.org/overview.html`. Last accessed: Dec, 2022.

[49] Google. *Fuchsia*. `https://fuchsia.dev/`. Last accessed: Oct, 2022.

[50] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. "A Taste of Capsicum: Practical Capabilities for UNIX". In: *Commun. ACM* (2012).

[51] Genode Labs. *Genode*. `https://genode.org/`. Last accessed: Oct, 2022.

[52] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. "SemperOS: A Distributed Capability System". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019.

[53] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2009.

[54] R. M. Needham and R. D.H. Walker. "The Cambridge CAP Computer and Its Protection System". In: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. SOSP '77. 1977.

[55] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. 2015.

[56] *Fiasco: The L4Re Microkernel*. `http://os.inf.tu-dresden.de/fiasco/`. Last accessed: Dec, 2022.

[57] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems". In: *Proceedings of the*

11th USENIX Conference on Networked Systems Design and Implementationi (NSDI). 2014.

[58] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012.

[59] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010.

[60] Vijay Vasudevan, David Andersen, and Michael Kaminsky. "The Case for VOS: The Vector Operating System". In: *13th Workshop on Hot Topics in Operating Systems (HotOS)*. 2011.

[61] *How long does it take to make a context switch?* `https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`. Last accessed: Jan, 2021.

[62] *Blobstore Programmer's Guide*. `https://spdk.io/doc/blob.html`. Last accessed: Nov, 2021.

[63] *BlobFS: Blobstore Filesystem*. `https://spdk.io/doc/blobfs.html`. Last accessed: Oct, 2020.

[64] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. "Shredder: GPU-Accelerated Incremental Storage and Computation". In: *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. 2012.

[65] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. "Towards Trusted Cloud Computing". In: *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2009.

[66] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2014.

[67] CRN. *The ten biggest cloud outages of 2013*. `https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm`. Last accessed: Dec, 2018. 2013.

[68] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. "An Analysis of Data Corruption in the Storage Stack". In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. 2008.

[69] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. "Availability in Globally Distributed Storage Systems". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[70] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Flavio Junqueira, and Benjamin Reed. "Reliable Data-center Scale Computations". In: *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*. 2010.

[71] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "HAFT: Hardware-assisted Fault Tolerance". In: *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*. 2016.

[72] Dmitrii Kuvaiskii, Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Elzar: Triple Modular Redundancy using Intel AVX". In: *proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016.

[73] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. "Cross-checking Semantic Correctness: The Case of Finding File System Bugs". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. 2015.

[74] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. "IOFlow: A Software-defined Storage Architecture". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[75] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. "SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. 2015.

[76] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-less System Calls". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[77] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. "Memoir: Practical state continuity for protected modules". In: *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*. 2011.

[78] *Intel, "SGX documentation: sgx create monotonic counter"*. `https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter/`. Last accessed: Dec, 2018.

[79] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

[80] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Crash Consistency". In: *ACM Queue* (2015).

[81] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 2018.

[82] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. 2015.

[83] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2017.

[84] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC)*. 2018.

[85] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. "ROTE: Rollback Protection for Trusted Execution". In: *26th USENIX Security Symposium (USENIX Security)*. 2017.

[86] Ittai Anati, Shay Gueron, P. Simon Johnson, and R. Vincent Scarlata. "Innovative technology for CPU based attestation and sealing". In: *Proceedings*

of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP). 2013.

[87] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. "Integrating Remote Attestation with Transport Layer Security". In: (2018). eprint: `arXiv:1801.05863`.

[88] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.

[89] Raoul Strackx and Frank Piessens. "Ariadne: A Minimal Approach to State Continuity". In: *25th USENIX Security Symposium (USENIX Security)*. 2016.

[90] *Botan Library*. `https://botan.randombit.net/`. Last accessed: Jan, 2019.

[91] *RocksDB Benchmarking Tool*. `https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`. Last accessed: Dec, 2018.

[92] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. "Fex: A Software Systems Evaluator". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017.

[93] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. "ShieldBox: Secure Middleboxes using Shielded Execution". In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2018.

[94] Trusted Computing Group. *TPM Main Specification*. `https://trustedcomputinggroup.org/tpm-main-specification`. Last accessed: Jan, 2021. 2011.

[95] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. "Pasture: Secure Offline Data Access Using Commodity Trusted Hardware". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[96] Garth A Gibson, David F Nagle, Khalil Amiri, Jeff Butler, Fay W Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. "A cost-effective, high-bandwidth storage architecture". In: *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1998.

[97] Deepak Garg and Frank Pfenning. "A proof-carrying file system". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy*. 2010.

[98] K. Mast, L. Chen, and E. Gün Sirer. "Enabling Strong Database Integrity using Trusted Execution Environments". In: (2018). eprint: arXiv:1801.01618.

[99] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy compliance for database-backed systems". In: *Proceedings of the 26th USENIX Security Symposium*. 2017.

[100] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. "Guardat: Enforcing data policies at the storage layer". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. 2015.

[101] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. "Thoth: Comprehensive Policy Compliance in Data Retrieval Systems". In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 2016.

[102] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. "Policy-sealed data: A new abstraction for building trusted cloud services". In: *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*. 2012.

[103] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. "Pesos: Policy Enhanced Secure Object Store". In: *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*. 2018.

[104] *Kinetic Data Center Comparison*. https://www.openkinetic.org/technology/data-center-comparison. Last accessed: Dec, 2018.

[105] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. "CryptDB: protecting confidentiality with encrypted query processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[106] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. "Big Data Analytics over Encrypted Datasets with Seabed". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[107] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. "Processing analytical queries over encrypted data". In: *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)*. 2013.

[108] Arjun Narayan and Andreas Haeberlen. "DJoin: Differentially Private Join Queries over Distributed Databases". In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[109] C. Priebe, K. Vaswani, and M. Costa. "EnclaveDB: A Secure Database using SGX (S&P)". In: *IEEE Symposium on Security and Privacy*. 2018.

[110] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. "Enabling Security in Cloud Storage SLAs with CloudProof". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*. 2011.

[111] Umesh Maheshwari, Radek Vingralek, and William Shapiro. "How to Build a Trusted Database System on Untrusted Storage". In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*. 2000.

[112] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. "Obladi: Oblivious Serializable Transactions in the Cloud". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2018.

[113] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. "ShieldStore: Shielded In-Memory Key-Value Storage with SGX". In: *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*. 2019.

[114] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. "Authenticated Data Structures, Generically". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2014.

[115] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. "mLSM: Making Authenticated Storage Faster in Ethereum". In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2018.

[116] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. "Depot: Cloud Storage with Minimal Trust". In: *ACM Transactions on Computer Systems*. 2011.

[117] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. "Robustness in the Salus Scalable Block Store". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[118] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. "Attested Append-only Memory: Making Adversaries Stick to Their Word". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2007.

[119] Jinyuan Li, Mn Krohn, D Mazières, and Dennis Shasha. "Secure untrusted data repository (SUNDR)". In: *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[120] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. "Plutus: Scalable Secure File Sharing on Untrusted Storage." In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. 2003.

[121] Carsten Weinhold and Hermann Härtig. "jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2011.

[122] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. "SiRiUS: Securing Remote Untrusted Storage." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2003.

[123] Ethan L Miller, Darrell DE Long, William E Freeman, and Benjamin Reed. "Strong Security for Network-Attached Storage." In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. 2002.

[124] Andrew W Leung, Ethan L Miller, and Stephanie Jones. "Scalable security for petascale parallel file systems". In: *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. 2007.

[125] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. "StrongBox: Confidentiality, Integrity, and Performance Using Stream Ciphers for Full Drive Encryption". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2018.

[126] Stephen Yang. "SLIK : Scalable Low-Latency Indexes for a Key-Value Store". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*. 2014.

[127] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. "Tailwind: Fast and Atomic RDMA-Based Replication". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC)*. 2018.

[128] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. "Preventing page faults from telling your secrets". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCCS)*. 2016.

[129] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services ". In: *21st USENIX Security Symposium (USENIX Security)*. 2012.

[130] Nuno Santos, Rodrigo Rodrigues, and Bryan Ford. "Enhancing the OS against Security Threats in System Administration". In: *Proceedings of the 13th International Middleware Conference (Middleware)*. 2012.

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

[132] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. *SGX-LKL: Securing the Host OS Interface for Trusted Execution*. 2019. arXiv: `1908.11143 [cs.OS]`.

[133] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves". In: *SIGARCH Comput. Archit. News* (2017).

[134] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* (1982).

[135] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* (2002).

[136] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. "BFT: The Time is Now". In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*. 2008.

[137] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI)*. 2004.

[138] N. A. Lynch and A. A. Shvartsman. "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts". In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FtCS)*. 1997.

[139] J. Sousa and A. Bessani. “From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation”. In: *2012 Ninth European Dependable Computing Conference (EDCC)*. 2012.

[140] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. “Bootstrapping Trust in Commodity Computers”. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*. 2010.

[141] *Intel Corporation. Attestation Service for Intel Software GuardExtensions (Intel SGX): API Documentation.* `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`. Last accessed: Jan, 2021.

[142] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*. 2010.

[143] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. “Hybrids on Steroids: SGX-Based High Performance BFT”. In: *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. 2017.

[144] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. “Worm-IT - A Wormhole-Based Intrusion-Tolerant Group Communication System”. In: (2007).

[145] M. Correia, N.F. Neves, and P. Verissimo. “How to tolerate half less one Byzantine nodes in practical distributed systems”. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 2004.

[146] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P)*. 2019.

[147] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[148] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. `https://sgaxeattack.com/`. 2020.

[149] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *CacheOut: Leaking Data on Intel CPUs via Cache Evictions*. 2020. arXiv: 2006.13353 [cs.CR].

[150] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019.

[151] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *CCS*. 2019.

[152] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *41st IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[153] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory Safety for Shielded Execution". In: *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*. 2017.

[154] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[155] Shin-Yeh Tsai and Yiying Zhang. "A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2019.

[156] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. "Securing RDMA for High-Performance Datacenter Storage Systems". In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2020.

[157] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Transaction of Programming Language and Systems (TOPLAS)* (1990).

[158] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing Memory Robustly in Message-passing Systems". In: *Journal of the ACM* (1995).

[159] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*. 2014.

[160] Dave Levin, John (JD) Douceur, Jay Lorch, and Thomas Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

[161] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. "Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol". In: *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[162] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. "Supporting third party attestation for Intel® SGX with Intel® data center attestation primitives". In: (2018).

[163] *OpenSSL library*. https://openssl.org. Last accessed: Jan, 2022. URL: `https://openssl.org`.

[164] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*. 2014.

[165] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. 2010.

[166] *Folly: Facebook Open-source Library*. `https://github.com/facebook/folly`.

[167] *boost: C++ libraries*. `https://www.boost.org/`. Last accessed: Aug, 2020.

[168] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. `https://iperf.fr/`. Last accessed: Aug, 2020.

[169] *eRPC-Raft*. `https://github.com/erpc-io/eRPC/tree/master/apps/smr`. Last accessed: Jan, 2022.

[170] *Intel Software Guard Extensions SDK for Linux OS*. `https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf`. Last accessed: Dec, 2018.

[171] *TCMalloc*. `https://github.com/google/tcmalloc`. Last accessed: Aug, 2020.

[172] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. "Rkt-Io: A Direct I/O Stack for Shielded Execution". In: *Proceedings of the Sixteenth European Conference on Computer Systems (ACM EuroSys 21)*. 2021.

[173] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. "Visigoth Fault Tolerance". In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*. 2015.

[174] R.P. LaRowe, C.S. Ellis, and M.A. Holliday. "Evaluation of NUMA memory management through modeling and measurements". In: 1992.

[175] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. 2016.

[176] *oneAPI*. `https://www.oneapi.com/`. Last accessed: Dec, 2021.

[177] *Clang: a C language family frontend for LLVM*. `https://clang.llvm.org/`. Last accessed: Jan, 2021.

[178] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: 2004.

[179] *eRPC: a log store application*. `https://github.com/erpc-io/eRPC/tree/master/apps/log`.

[180] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014.

[181] *{fmt}*. https://github.com/fmtlib/fmt. Last accessed: Oct, 2022.

[182] *JSON for Modern C++*. https://github.com/nlohmann/json. Last accessed: Oct, 2022.

[183] *Overview of modules in C++*. `https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170`. Last accessed: Jun, 2023.

[184] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019.

[185] Microsoft. *mimalloc*. `https://github.com/microsoft/mimalloc`. Last accessed: Oct, 2022.

[186] Microsoft. *mi-malloc Documentation*. `https://microsoft.github.io/mimalloc/`. Last accessed: Oct, 2022.

[187] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. "Poseidon: Safe, Fast and Scalable Persistent Memory Allocator". In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. 2020.

[188] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. "CORFU: A Distributed Shared Log". In: (2013).

[189] Minwen Ji, Alistair Veitch, and John Wilkes. "Seneca: Remote Mirroring Done Write". In: *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. 2003.

[190] *Linux* Base Driver for Intel Gigabit Ethernet Network Connections*. `https://www.intel.com/content/www/us/en/support/articles/000005480/ethernet-products.html`.

[191] *Intel Network Adapter Driver for PCIe* 40 Gigabit Ethernet Network Connections under Linux**. `https://www.intel.com/content/www/us/en/download/18026/intel-network-adapter-driver-for-pcie-40-gigabit-ethernet-network-connections-under-linux.html`.

[192] J.Gold Associates. *oneAPI: Software Abstraction for a Heterogeneous Computing World*. `https://jgoldassociates.com/White_Papers/OneAPI_Whitepaper.pdf`.

[193] Felix Xiaozhu Lin and Xu Liu. "Memif: Towards Programming Heterogeneous Memory Asynchronously". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 2016.

[194] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. "Enclosure: Language-Based Restriction of Untrusted Libraries". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21*. Virtual, USA, 2021.

[195] Wei-Yu Chen, C. Iancu, and K. Yelick. "Communication optimizations for fine-grained UPC applications". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005.

[196] *OpenMP: The OpenMP API specification for parallel programming*. `https://www.openmp.org/`.

[197] V. Adve, Guohua Jin, J. Mellor-Crummey, and Qing Yi. "High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes". In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998.

[198] Alexandre Eichenberger, K. O'Brien, Peng Wu, Tong Chen, P. Oden, Dan Prener, J.C. Shepherd, Byoungro So, Zehra Sura, A. Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. "Optimizing Compiler for the CELL Processor". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005.

[199] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. "Glamdring: Automatic Application Partitioning for Intel SGX". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. 2017.

[200] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. "CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019.

[201] *Project Verona: Research programming language for concurrent ownership*. `https://microsoft.github.io/verona/`. Last accessed: Oct, 2022.

[202] Oren Avissar, Rajeev Barua, and Dave Stewart. "Heterogeneous Memory Management for Embedded Systems". In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASE'01)*. 2001.

[203] Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. "Compiler-Driven Data Layout Transformation for Heterogeneous Platforms". In: *Euro-Par 2013: Parallel Processing Workshops*. 2014.

[204] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: (2008).

[205] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-Weight Contexts: An OS Abstraction for Safety and Performance". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[206] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. "Enforcing Least Privilege Memory Views for Multithreaded Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[207] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: (1993).

[208] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity Principles, Implementations, and Applications". In: (2009).

[209] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009.

[210] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. "XFI: Software Guards for System Address Spaces". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006.

[211] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* (2017).

[212] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. "IBM System/38 Support for Capability-Based Addressing". In: *ISCA*. 1981.

[213] Viktors Berstis. In: *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA '80)*. 1980. URL: http://doi.acm.org/10.1145/800053.801932.

[214] William J Dally, Stephen W Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S Lee. *M-Machine architecture v1*. Tech. rep. 0. Technical Report–MIT Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, 1994.

[215] *Armv 8.5 - A: Memory Tagging Extension*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Last accessed: Oct 2020.

[216] *PA-RISC 1.1 Architecture and Instruction Set: Reference Manual*. Hewlett Packard, 1990.

[217] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Last accessed: Oct 2020.

[218] *Domains*. https://developer.arm.com/documentation/ddi0211/k/memory-management-unit/memory-access-control/domains. Last accessed: Oct 2020.

[219] Emmett Witchel, Junghwan Rhee, and Krste Asanović. “Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. 2005.

[220] Ian H. Witten and John G. Cleary. “An introduction to the architecture of the Intel iAPX 432”. In: *Software & Microsystems* (1983).

[221] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. “CODOMs: Protecting software with Code-centric memory Domains”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014.