THE UNIVERSITY *of* EDINBURGH

Trustworthy Distributed Data Management Systems

Dimitra Giantsidi

Doctor of Philosophy

Adviser: Prof. Pramod BHATOTIA

Thesis Committee Members:

Prof. Boris GROT (The University of Edinburgh)

Prof. Martin KLEPPMANN (University of Cambridge)

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2024

Abstract

Distributed data management systems such as transactional databases, fault-tolerant Key-Value stores (KVs), and shared logs are extensively deployed in the third-party cloud infrastructure as the foundational systems to support various online applications (banking systems, e-commerce, serverless computing, healthcare, etc.). From a high-level overview, distributed data management systems are comprised of three layers: the application layer that exposes programming APIs with various semantics (transactions, KVs operations, etc.), the platform layer that implements decentralized protocols to support these APIs and the infrastructure layer that includes the data and storage systems as well as the network infrastructure.

Unfortunately, due to the untrusted nature of the third-party cloud infrastructure, these systems are susceptible to various threats, making it challenging to offer security and privacy for users' data and operations. More specifically, it has been shown that malicious adversaries can gain control, monitor, and tamper with the software system stack of the entire cloud infrastructure (including the OS and hypervisors). Such a powerful attacker can compromise the efficiency (performance and availability), the correctness (consistency), and the security properties of widely used distributed data management systems. Significantly, the vast majority of these systems in the cloud are not shielded against such privileged attackers, allowing for users' private data and operations to be leaked (non-confidential access), compromised by integrity violations, and impersonated (equivocation-based and non-authenticated execution).

In this thesis, we leverage the performance and security capabilities of modern hardware in the cloud infrastructure to design robust and high-performance distributed systems with strict security properties (i.e., confidentiality, integrity, freshness, Byzantine Fault Tolerance (BFT), etc.). Precisely, we explore the potential of the advancements in trusted hardware (i.e., trusted execution environments (TEEs)) and high-performance networking (direct I/O and SmartNICs) to shield and optimize the platform and infrastructure layers in various commonly used distributed systems while exposing powerful semantics to users (application layer).

To this end, we design and build three systems, TREATY, RECIPE and TNIC, leveraging the state-of-the-art modern hardware in cloud, TEEs, direct I/O and SmartNICs. Specifically,

- TREATY is a secure distributed transactional KV store for untrusted cloud environments that offers high-performance serializable Txs with strong security properties: confidentiality, integrity, and freshness against rollback attacks. To achieve that, TREATY implements a secure substrate on top of TEEs and builds a secure

distributed atomic commit protocol based on the two-phase commit and a stabilization protocol for the committed transactions for ensuring crash consistency and rollback resilience across the participating machines.

- RECIPE implements a performant generic hardware-assisted transformation of replication protocols under the crash-fail model for the Byzantine infrastructure that experiences arbitrary failures. We build RECIPE on top of TEEs and direct network I/O to bypass kernel but further avoid the TEE's expensive system calls. In addition, RECIPE can seamlessly extend the security properties handled by traditional BFT systems, offering confidentiality for the executed protocol and accessed data.

- TNIC, inspired by the findings and our experiences from the previous systems, concludes the thesis. TNIC designs a trusted NIC architecture for building trustworthy distributed systems, overcoming the challenges (performance, programmability, and security vulnerabilities) raised by the heterogeneous and untrusted (Byzantine) cloud infrastructure. TNIC implements a minimal, unified, and formally verified trusted computing base (TCB) on top of SmartNICs. TNIC TCB guarantees powerful security properties to ensure a scalable transformation of crash-fail systems to BFT ones.

Lay summary

Distributed data management systems (e.g., transactional databases, fault-tolerant Key-Value stores (KVs), etc.) are massively deployed in the third-party cloud infrastructure as the foundational systems to support various online applications (banking systems, e-commerce, serverless computing, healthcare, etc.).

Unfortunately, the third-party infrastructure is untrusted and prone to a wide range of attacks and failures that can compromise the security properties and the correctness semantics of the executed system. Specifically, powerful adversaries can gain control over the entire system stack of the cloud infrastructure, including the privileged code (OS and hypervisors). Such adversaries can compromise the security properties of users' private data and operations in various ways, i.e., data leakage (non-confidential access), integrity violations, forking and replay attacks, as well as impersonation-based attacks where the adversary attempts to "participate" in the distributed system and convince it to take the wrong decisions.

In this thesis, we leverage modern hardware's performance and security capabilities in the cloud infrastructure to show that the seemingly contradictory objectives of security and performance are no longer mutually exclusive in distributed data management systems. Precisely, we explore the potential of the advancements in trusted hardware (i.e., trusted execution environments (TEEs)) and high-performance networking (direct I/O and SmartNICs) to shield the system (i.e., confidentiality, integrity, freshness, Byzantine Fault Tolerance (BFT), etc.) while offering performance.

To this end, we design and build three systems, TREATY, RECIPE and TNIC, leveraging the state-of-the-art modern hardware in cloud, TEEs, direct I/O and SmartNICs. Specifically,

- TREATY is a secure distributed transactional KV store for untrusted cloud environments that offers high-performance serializable Txs with strong security properties: confidentiality, integrity, and freshness against rollback attacks. TREATY builds a secure distributed atomic commit protocol for secure transactions' execution and a stabilization protocol for crash-consistency and rollback-resilience for the committed data.

- RECIPE implements a generic hardware-assisted transformation of replication protocols under the crash-fail model for the Byzantine infrastructure that experiences arbitrary failures. To achieve that, RECIPE builds a performance efficient distributed trusted computing base (TCB) on top of TEEs and direct network I/O while showing that it can also offer confidentiality, an extra security property that is not offered by traditional Byzantine Fault Tolerant (BFT) systems.

- TNIC builds a trusted NIC architecture for building trustworthy distributed systems for the heterogeneous and untrusted (Byzantine) cloud infrastructure. TNIC implements a minimal, unified, and formally verified trusted computing base (TCB) on top of SmartNICs, exposing generic yet powerful networking primitives. Significantly, TNIC is CPU-host agnostic, while its minimalistic and unified API improves its adoption to the untrusted cloud for transforming crash-fail systems to BFT ones.

Publications

This thesis is based on the following conference papers.

1. TREATY*: Secure Distributed Transactions*
   Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia
   IEEE/IFIP DSN 2022 [109]
   (Best Paper Award Nominee — Among Top 3 Papers)

2. *A Hardware-Accelerated* RECIPE *For Designing Byzantine Fault Tolerant Replication Protocols*
   Dimitra Giantsidi, Emmanouil Giortamis, Maurice Bailleu, Manos Kapritsos and Pramod Bhatotia
   ACM EuroSys 2025 (Under review)

3. TNIC*: A Trusted NIC Architecture*
   Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia
   ACM ASPLOS 2025 (Under review)

Other conference papers during my PhD.

1. FLEXLOG*: A Shared Log for Stateful Serverless Computing*
   Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu and Pramod Bhatotia
   ACM HPDC 2023 [110]

2. AVOCADO*: A Secure In-Memory Distributed Storage System*
   Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia
   USENIX ATC 2021 [38]

3. ANCHOR*: A Library for Building Secure Persistent Memory Systems*
   Dimitrios Stavrakakis, Dimitra Giantsidi, Maurice Bailleu, Philip Sandig, Shady Issa, and Pramod Bhatotia
   ACM SIGMOD 2024 [297]

Acknowledgements

Even though this thesis bears my name, it was only made possible with all my amazing collaborators and mentors. To everyone who contributed to this thesis—even with small things—I am incredibly thankful to all of you!

I will always be grateful to my Ph.D. advisor, Prof. Pramod Bhatotia, who gave me the opportunity to work with him and be part of his research group in TUM (even remotely from the UK), where I was fortunate to have so many fruitful collaborations. I appreciate the countless amounts of time he invested in providing me with guidance and detailed feedback on important aspects of my research, from paper writing to presentation skills, and research ideas. His mentorship throughout this whole journey was crucial for its successful completion. I thank him for all the advice, feedback and guidance he gave me during my preparation for job interviews.

I thank my examiners Prof. Martin Kleppmann from the University of Cambridge and Prof. Boris Grot from the University of Edinburgh for their valuable feedback, discussions and suggestions that significantly improved this thesis.

I would like to express my gratitude to Prof. Natacha Crooks, Prof. Manos Kapritsos, and Dr. Florin Dinu for being outstanding collaborators and for their support, even with the most trivial questions. I am also deeply thankful to Dr. Hugh Leather for his invaluable advice on both personal and professional matters when I needed it most. A special thanks goes to Alex Shamis, Shango Lee, and Marcus Peinado, my internship mentors at Microsoft Research in Cambridge and in Redmond, for giving me the opportunity to work and engage with researchers in such inspiring environments. I am especially grateful to Venki Ramarathnam and Christian König from Microsoft Research in Redmond for the stimulating discussions.

I thank my academic sister, Elena Milkai, for our casual everyday/all-day discussions this summer in Redmond, as well as Vaastav, Jordi, Antonis, Christos and Jovan. It was a blast partying with all of you on the other side of the world! I am very grateful to Dr. Thaleia Doudali for her advice, for having great time with her at EuroSys'24 in Athens and for encouraging me to go on when I needed the most. I also thank Prof. Marios Kogias for his valuable feedback when I was preparing for my interviews' talks.

I cannot thank enough my friend Maurice Bailleu with whom we shared an office for almost five years and worked together on so many projects. I thank him for teaching me all these fancy features of (totally unsafe) C++ and for organizing office drinks once in a while. Both made Edinburgh such an exciting place. Similarly, I want to thank my friend (and collaborator) Dimitris Stavrakakis for his 24/7 availability to chat about anything and everything—he really made Covid quarantine and TEEs debugging bearable. Maurice and Dimitris are both bulletproof systems researchers and I have

learned so much from them—I hope we work together again in the future!

On the “software” side, I thank the distributed systems experts, Antonis Katsarakis, Manos Giortamis, Vasilis Gavrielatos and Heidi Howard; your papers revealed to me a fascinating line of research. I am particularly grateful to Antonis and Manos because they have always been *(highly-)available* to *consistently* chat with me for hours about replication, BFT, and so many other interesting things; I will have never learned about all these without your help and discussions. On the “hardware” side, I thank Felix Gust and Atsushi Koshiba for their valuable contributions in the last project of this thesis.

I also want to thank all members of the Chair of Computer Systems in TUM. Thanks to Jörg Thalheim, Peter Okelmann, Harsha Unnibhavi, Nathaniel Tornow, and Julian Pritzi for their invaluable help with infrastructure and research-related matters. I also thank our admins, Jonathan and Anna. Last but not least, I thank Sophia and Ruth for taking care of all the paperwork and organizing all these nice trips to Germany.

I want to thank my friends Ada, Eleni and Elpida for all the great times we had all these years. I specifically thank Ada for hosting me in London every now and then and for all the fun (and not so fun) times we had (and hope to keep having) in this amazing city.

Lastly, I want to express my gratitude to all the important people in my life. I specifically thank my long-life family friends Eudokia and Marianna for all the laughters, parties and endless midnight discussions at the *Little Park*. My parents, Yannis and Anna, and my brother, Chris, for supporting me unconditionally throughout my whole life. Without all of them, I wouldn’t have reached so far.

Declaration

I declare that this thesis was composed by myself, and that this work has not been submitted for any other degree or professional qualification. I confirm that the submitted work is my own. My contributions and those of the other collaborators 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 IEEE/IFIP DSN 2022 under the title "TREATY*: Secure Distributed Transactions*" by Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia (PhD supervisor). The project described in § 4 has been submitted in EuroSys 2025 with the title "*Modern Hardware and Replication Protocols: A Synergistic Co-Design for Untrusted Cloud Environments*" by Dimitra Giantsidi, Emmanouil Giortamis, Maurice Bailleu, Manos Kapritsos and Pramod Bhatotia (PhD supervisor). Lastly, the research work outlined in § 5 has been submitted in ACM ASPLOS 2025 with the name "TNIC*: A Trusted NIC Architecture*" by Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia (PhD supervisor). The work regarding the formal proofs in sections § 5.4.4 and § 5.9 has been conducted by Julian Pritzi. The design of the remote attestation in section § 5.4.3 has been conducted jointly by Dimitra Giantsidi and Julian Pritzi.

(*Dimitra Giantsidi)*

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Distributed data management systems [50, 71, 82, 216], launched by all major cloud providers (e.g., Google, Amazon, Microsoft Azure, Facebook, etc.), are yet to resolve the challenges of scalability, consistency, availability and efficiency that are raised by the ever increasing amounts of data stored and processed in the cloud. These systems distribute and process the owning data in a decentralized manner across a set of distributed cooperating machines and manifest in the third-party cloud computing infrastructure in various forms ranging from transactional databases [265, 71, 82, 69], to (replicated) Key-Value stores (KVs) and storage systems [271, 186, 88, 250], processing platforms and logs [330, 33, 86, 41, 282], etc.

Distributed (data management) systems commonly follow a layered design. From a high-level overview, they are comprised of three layers: first, **(1)** the applications layer that exposes a programming API to users (e.g., transactions, KVs and network operations, etc.). Right below the application layer, the systems build **(2)** the platform layer which implements distributed algorithms or protocols to coordinate and synchronize the computation between the distributed participating machines. Lastly, the platform layer is built on top of **(3)** the infrastructure layer that is comprised of the data and storage infrastructure as well as the network infrastructure.

These systems are the foundational building block for numerous examples of online cloud applications (e.g., banking systems [319, 111], e-commerce [93], Function-as-a-Service (FaaS) platforms [7, 19, 52], blockchain systems [11, 46], healthcare systems [107, 55], and others [230, 90, 145]). However, their ever-increasing cloud adoption poses serious security challenges. In the third-party cloud infrastructure, the cloud provider has full access to the hardware and software stack, including the operating system (OS) and the hypervisor. This privileged control allows the cloud provider

to intercept all three layers of the cloud system stack by monitoring and controlling all data and network transfers, systems' computation (protocols) as well as users' stored data and operations.

Following this, a potential attacker can tamper and manipulate the entire distributed system not only by disrupting its (correct) execution and causing arbitrary (Byzantine) failures but also by posing security and privacy risks for the users' computation and data. Examples of such malicious attacks include breaches of confidentiality, integrity, and data freshness in storage systems [98, 122, 78, 275, 112, 177]. For instance, hackers breached AT&T's systems, stealing personal data of current and former customers, such as social security numbers, account numbers and passcodes [28]. The data breach is the latest cyberattack AT&T has experienced since a leak in January 2023, that affected nine million users. AT&T is currently facing the threat of multiple class action lawsuits. Along the same lines, these powerful attackers can cause arbitrary Byzantine failures, that, especially in an untrusted cloud environment, they can become the norm. For example, malicious adversaries can deliberately introduce further Byzantine failures by intentionally compromising the system stack, tampering with data, the network traffic and system operations [240], or injecting bugs into the system code [255, 346] and others [42, 226]. Such failures and security violations can have dramatic consequences. For instance, a single malfunctioning NIC at Los Angeles International Airport stalled immigration for more than 12 hours [1], while data breaches—common occurrences [138, 98]—allow unauthorized access to private user data.

Unfortunately, widely adopted data management systems are designed by principle to *blindly* trust the cloud provider and the infrastructure [265, 69, 354, 86, 236, 103]. As such, they cannot maintain the security and correctness properties in the examples we have discussed before. In other words, they are *fundamentally* incapable of addressing the privacy and security concerns as well as maintain their correctness in the presence of privileged attackers that can control and compromise the untrusted cloud infrastructure [274, 122]. To this end, it is imperative to architect secure distributed systems that maintain their performance, scalability, and correctness or consistency guarantees, while also enabling the hosting of sensitive data and workloads in untrusted cloud infrastructures—currently hindered by the broad surface of threats and attacks.

A promising design direction to build robust high-performance and privacy-preserving distributed systems for the untrusted cloud is the use of cloud's modern (trusted) hardware. The state-of-the-art trusted execution environments (TEEs), streamlined by all major CPU manufacturers [24, 23, 148, 15, 184], provide an isolated

secure memory region that remains protected against all types of (privileged) software attacks, opening up opportunities to enforce strict security properties in cloud applications. Similarly, direct network I/O stacks [209, 150] that bypass the OS kernel network stack, as well as the emerging SmartNICs devices [194, 317, 47] that offload network processing in the NIC-level hardware, both present great opportunities for high-throughput and low-latency network operations that is at the core of any distributed system. Given their premises, modern cloud hardware—TEEs [114, 68, 219, 217], direct I/O stacks [36, 34, 246, 161] and SmartNICs [10, 32, 59]—have been adopted by major cloud providers and constitute excellent design choices for improving the performance and the security aspects in distributed systems in the cloud.

However, this modern hardware presents conceptual and architectural challenges when building a distributed system. Importantly, TEEs are designed for local transient processes providing no security guarantees across the system's platform and infrastructure layers that are distributed over the untrusted network. At the same time, while TEEs themselves have a critical performance impact on (syscall-based) networking and storage operations, they are also not compatible with modern direct I/O stacks and have very limited trusted memory availability for storing data. Lastly, SmartNICs are designed for performance but cannot naturally offer foundational security primitives required to *shield* a (networked) distributed system.

To this end, this thesis raises the challenging question of ***how to modernize the distributed data management system stack leveraging the advancements in modern cloud hardware to offer both performance and dependability (security and robustness)***. As such, we identify three questions we seek to resolve: *(1)* How to extend the trust provided by TEEs over the untrusted network and untrusted storage to offer secure distributed transactions, a powerful programming primitive. *(2)* How to provide a generic, high-performance seamless transformation, a recipe, on top of TEEs and the state-of-the-art networking to strengthen the fault model of the widely deployed replicated systems that set the foundations for fault-tolerance and availability. Lastly, *(3)* how to provide a foundational secure, CPU-agnostic, and high-performance network abstraction for the heterogeneous untrusted cloud to allow system designers to build trustworthy distributed (networked) systems.

1.2 Problem Statement

Distributed transactions (Txs) with ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties offer a powerful programming abstraction allowing system designers and users to *transparently* process and store (*commit*) massive datasets. Offloading Txs in

the untrusted cloud causes severe security implications for the stored data and executed operations. As such, re-designing the distributed transactions mechanism with TEEs for building a secure distributed KVs for the untrusted cloud seems a promising solution to offer ACID Txs with confidentiality (i.e., Txs and data can only be accessed by authorized entities), integrity (i.e., unauthorized changes to the operations and data can be detected), and freshness, (i.e., stale state can be detected). However there are two challenges we need to take care of: (1) how to overcome the architectural limitations of TEEs to extend their security properties across the network while targeting performance and (2) how to ensure that distributed committed transactions remain secure and crash-consistent and cannot be reverted in case of rollback attacks.

For handling failures, distributed data stores employ Crash Fault Tolerant (CFT) replication protocols. However, CFT protocols assume that the trusted infrastructure and the participant nodes are trusted and honest. As such, they cannot provide consistent replication when they are hosted in the (Byzantine) untrusted cloud infrastructure that is subject to arbitrary failures. While conventional Byzantine Fault Tolerance (BFT) protocols ensure secure replication in such settings, they are extremely costly and complex to understand. Combining the two prominent technologies, i.e., TEEs for robustness and direct network I/O for performance seems a promising design choice to resolve the tension between security and performance. However, to provide high-performance robust replication protocols for practical deployments in the untrusted cloud, we need to address these two challenges: (1) how to use TEEs along with direct I/O to build an efficient distributed trusted computing base for the executing protocol and (2) how to materialize an approach that is as efficient as generic to aid system designers in easily transforming existing CFT protocols for Byzantine settings without having to be BFT experts.

TEEs are without any doubt a promising solution to build trustworthy distributed systems. However, we have identified three core challenges that complicate their widespread adoption in the cloud. First, cloud TEEs are heterogenous with different programming models and libraries and varying security properties. As such, they introduce programmability and security challenges that make it hard to combine heterogeneous TEEs to build a distributed system. Secondly, TEEs come with large trusted computing bases (TCBs). Unfortunately, TEEs' large TCBs are plagued with security vulnerabilities and cannot be verified. Lastly, based on our empirical experience, we have noticed that TEEs are limited in performance while they require sophisticated optimizations in the lower-level system stack to be used effectively. To this end, we seek to resolve all these challenges by providing a silicon-root-of-trust at the network

interface (NIC) level that offers a minimal TCB with powerful security properties while being CPU-agnostic and offering a unified programming interface.

1.3 Approach and Contributions

In this thesis, we leverage modern cloud hardware, i.e., TEEs and state-of-the-art networking technologies to design and build fundamental distributed data management systems for the untrusted cloud that offer a variety of different programming and consistency semantics with strong dependability, security and performance properties.

Figure 1.1 presents an overview of this thesis contributions. Inspired by the question of how to leverage modern hardware to provide robust and high-performance distributed systems, we design and build three foundational projects: TREATY [109], RECIPE and TNIC. Each of the projects leverages (a combination of) modern hardware technologies to design and build:

- **(1) foundational semantics in the cloud application layer** with strict security properties, e.g., ***(i)*** secure distributed transactions (TREATY), ***(ii)*** secure KVs operations (RECIPE) and ***(iii)*** secure networking operations (TNIC).

- **(2) secure and high-performance distributed protocols and system stacks in the platform layer**, e.g., ***(i)*** a secure distributed atomic commit protocol and a transactional engine (TREATY), ***(ii)*** a generic distributed trusted computing base (TCB) to transform distributed replication protocols for Byzantine settings (RECIPE) and ***(iii)*** a unified, CPU-agnostic, system stack for trusted RDMA operations (TNIC).

- **(3) secure and fault tolerant infrastructure (storage and network) layer**, e.g., ***(i)*** a distributed authenticated log-structured merge tree (LSM) KVs for the untrusted persistent storage (TREATY), ***(ii)*** a (hybrid) in-memory strongly consistent distributed KVs for Byzantine settings with confidentiality guarantees (RECIPE) and ***(iii)*** a trusted NIC architecture that offers a minimalistic and verifiable silicon-root-of-trust on top of SmartNICs (TNIC).

More precisely, **TREATY** is a secure distributed transactional KV store for untrusted cloud environments that offers high-performance serializable Txs with strong security properties: confidentiality, integrity, and freshness against rollback/forking attacks. TREATY builds hardware-assisted secure Txs with Intel SGX. Specifically, it realizes a secure substrate consisting of a secure network library, a secure Tx engine, a memory allocator, and a userland scheduler for low-latency operations. With this substrate, we

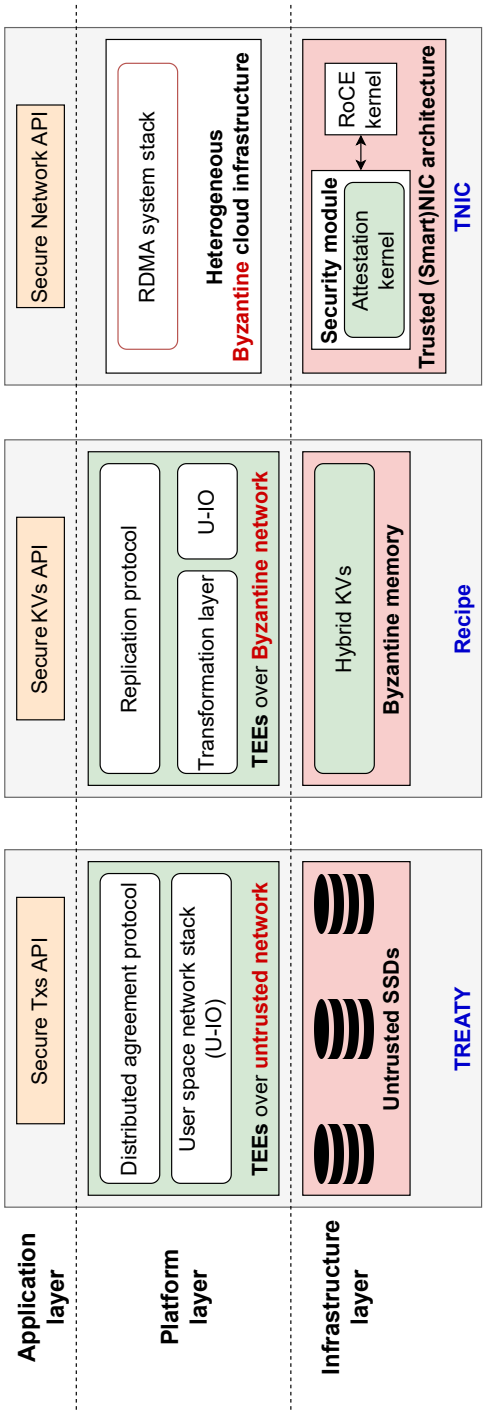

Figure 1.1: Overview of thesis contributions.

build a shielded distributed two-phase commit (2PC) protocol with a direct I/O network library on top of eRPC [162] and DPDK [150] that allows TREATY to bypass the OS kernel for networking and optimize performance. Further, we design a stabilization protocol for transactions to ensure that committed transactions are crash-consistent and rollback-protected. TREATY's stabilization protocol is built on top of two trusted services: an asynchronous trusted counter interface and a distributed attestation service.

We implement TREATY's distributed storage layer extending a secure version of RocksDB [271] (SPEICHER [39]) that served as the underlying storage engine. Our evaluation with the YCSB and TPC-C shows reasonable overheads for TREATY, e.g., up to 15× and 5× for distributed and single-node transactions, respectively, while providing strong security properties.

RECIPE builds a hardware-assisted transformation of Crash Fault Tolerant (CFT) protocols to tolerate (Byzantine) arbitrary failures without any modifications to the core of the protocols, e.g., states, message rounds, and complexity. We realize our approach by building a distributed TCB as a library leveraging modern cloud hardware. We use TEEs (Intel SGX) to guarantee the two security properties of the transferable authentication and the non-equivocation for preventing Byzantine faults. We additionally combine trusted hardware with direct network I/O [209, 150] to improve performance but also propose generic RECIPE APIs for the transformation.

We further show that RECIPE can easily offer confidentiality that is not provided by traditional BFT protocols while we provide a correctness analysis for the safety and liveness of our transformation of CFT protocols operating in Byzantine settings. Lastly, we use RECIPE to successfully transform a range of leader-/leaderless-based CFT protocols enforcing different (total order/per-key) ordering semantics. We present an extensive evaluation of four RECIPE-transformed CFT protocols: Chain Replication, Raft, ABD, and AllConcur. Our evaluation shows that RECIPE achieves BFT while outperforming the state-of-the-art BFT systems up to 24× better throughput.

TNIC is a trusted NIC architecture for building trustworthy distributed systems deployed in heterogeneous, untrusted (Byzantine) cloud environments. TNIC implements a minimal, unified, formally verified, high-performance silicon root-of-trust on top of SmartNICs [317]. Precisely, it builds a host CPU-agnostic verifiable trusted computing base (TCB) to guarantee the two foundational properties of transferable authentication and non-equivocation that suffice to build BFT protocols [67]. We designed TNIC hardware architecture, an associated network stack as well as a generic set of programming APIs and a recipe for building high-performance, trustworthy, distributed

systems for Byzantine settings. We formally verify the safety and security properties of our TNIC system while demonstrating how we leverage TNIC for building four trustworthy distributed systems: A2M, BFT, Chain Replication, and PeerReview—showing the generality of our approach. Our evaluation of TNIC shows up to 6× performance improvement compared to CPU-centric TEE systems while providing a low TCB with provable security properties.

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: TREATY (Chapter 3), RECIPE (Chapter 4) and TNIC (Chapter 5). The thesis will conclude in Chapter 6.

Chapter 2

Background

This chapter summarizes the foundational concepts upon which the thesis projects are built. We design and build various distributed systems, e.g., from transactional storage systems and Key-Value stores (KVs) to replication protocols, that offer improved security and performance properties by leveraging the advancements in modern hardware. We chose to implement these systems as they are widely adopted as generic building blocks for various online services and are offered by major cloud providers (e.g., AWS S3 [14], Google Cloud Storage [113], DynamoDB [82], ZippyDB [353], CockroachDB [69], Spanner [71], etc.). As such, we first present an overview of the distributed data management systems (§ 2.1) that are explored in this work. Next, we discuss the state-of-the-art trusted hardware, i.e., trusted execution environments (TEEs) in § 2.2, which is the foundational building block for providing security in this thesis. Following this, we present an overview of the high-performance networking technologies (§ 2.3). Lastly, we present an overview of the storage system (§ 2.4) that implements the data layer in TREATY project.

Section § 2.1.1 discusses the concepts of the distributed transactions and the transactional Key-Value stores (KVs) on top of which we build TREATY project in Chapter 3. Section § 2.1.2 provides a taxonomy of the replication protocols deployed for fault tolerance in modern data stores on top of which we build RECIPE in Chapter 4.

Section § 2.2 discusses the state-of-the-art trusted hardware, i.e., trusted execution environments (TEEs). We cover the technologies of Intel SGX (§ 2.2.1) and remote attestation (§ 2.2.2) used in TREATY and RECIPE while we also discuss the AMD-sev TEE (§ 2.2.1) which is referenced in TNIC project in Chapter 5.

Section § 2.3 presents an overview of the high-performance networking technologies focusing on direct I/O network (§ 2.3.1) stacks for TREATY and RECIPE as well as the SmartNIC devices (§ 2.3.2) on top of which we build TNIC.

Lastly, we discuss the SPEICHER storage system [39] (§ 2.4.1) on top of which we

build TREATY.

2.1 Distributed Data Management Systems

Due to the increased demand for data storage and processing in the cloud infrastructure, a variety of distributed data management systems have emerged [71, 265, 336, 80]. These systems are crucial to the cloud infrastructure because they efficiently resolve the challenges of reliability and availability, scalability, and performance while hiding all implementation complexities of distribution, consistency, and fault tolerance from the users.

Traditionally, the majority of these systems is comprised of a set of distributed machines or nodes connected over the network that follows a layered design shown in Figure 2.1. The application layer exposes a user interface to external clients with a variety of operations. The platform layer implements protocols that materialize various levels of consistency, availability, and performance across the distributed participant nodes. For example, depending on the level of trust in the cloud provider, the implemented protocols might necessitate additional machines in their system model (Byzantine Fault Tolerance [57]) or costly operations such as data encryption and integrity checks. These can impact scalability, throughput, and latency. Lastly, the infrastructure layer is comprised of storage engines that store the user data and the network infrastructure that implements the network and data link layer.

However, these systems fail to provide the required performance, consistency, and availability guarantees in the modern untrusted cloud infrastructure because they are designed to *unquestioningly* trust the cloud provider and the entire cloud infrastructure. This thesis solves this limitation by presenting distributed data management systems redesigned with TEEs and networking technologies to resolve the tension between performance/scalability and security while remaining correct, i.e., *consistent* when hosted in the untrusted cloud infrastructure.

2.1.1 Distributed Transactional Key-Value Stores

Distributed transactions (Txs). Distributed Key-Value (KV) stores [179, 223, 75, 45, 69, 353] (KVs) reliably store and process large datasets by offering transactional (Tx) APIs. A transaction is a set of operations that atomically processes pieces of data in an "all-or-nothing" manner; either all of the operations complete successfully, or none of them succeeds and the data remains intact. Similarly, a distributed transaction is a set of operations that atomically process data that might be stored on different machines connected over the network.

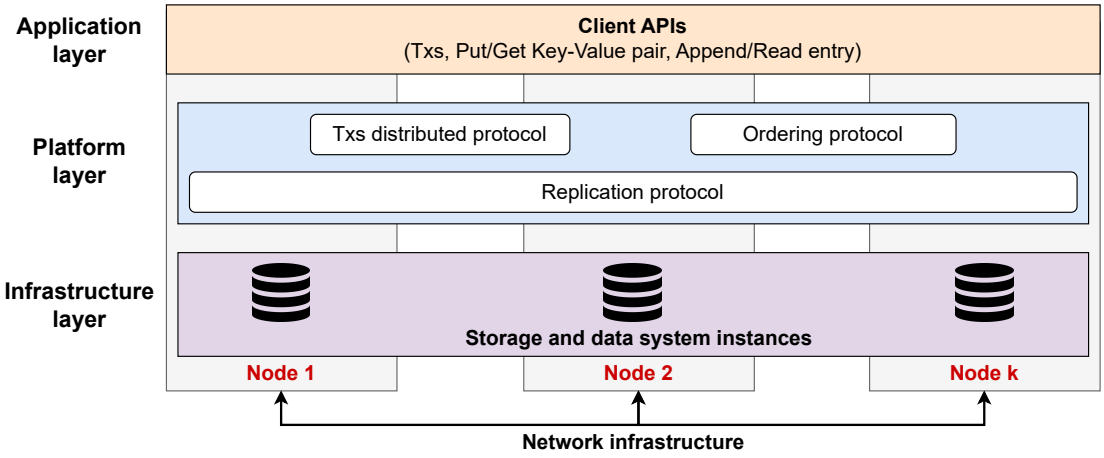

Figure 2.1: Overview of a distributed system hosted in the cloud.

Distributed transactions are an integral part of modern distributed systems [353, 69, 71, 265, 164, 88, 80] because they hide complexities (i.e., parallelism, data races and failures) from the user while they maintain the system's correctness and facilitate programming. Precisely, (distributed) Txs offer **A**tomicity—**C**onsistency—**I**solation—**D**urability (ACID) properties.

Distributed transactions are usually implemented through a distributed atomic commit protocol, e.g., the two (or three)-phase-commit protocols (2PC) where a coordinator node sends the operations to the appropriate participant nodes, *prepares* the Tx to ensure that all involved nodes can successfully commit, and then instructs them to *commit* or *abort* based on the outcome of the Tx prepare phase. In addition they require a concurrency control mechanism to enforce isolation across parallel distributed operations. In this thesis, we built TREATY, a sharded (e.g., the data/key range is divided among the participating machines), non-replicated transactional KV store that maintains all Txs ACID properties in the untrusted cloud infrastructure. Our design ensures shielded execution of the two-phase commit protocol (§ 3.3.1), i.e., the protocol is executed according to its original specification and its execution cannot be compromized by an adversary, whereas we also build a two-phase locking algorithm that enforces isolation as well as strict serializability [248] for correctness. A transaction in the two-phase locking acquires locks as it goes along and releases the locks right after its commit or abort. RocksDB [271], the storage engine on top of which we build TREATY, supports single-node Txs (not distributed). For TREATY we adapt RocksDB's *p*essimistic Txs that acquire locks as they go along. RocksDB's optimistic Txs validate their read and write sets at the commit time and have high abort rates in workloads with significant conflicts.

Tx storage engines. Distributed Tx systems (e.g., ZippyDB [353], CockroachDB [69],

	Leader-based	Leader-less
Total order	**Raft** [236], ZAB [266], Multi-Paxos [322]	**AllConcur** [251], Derecho [159]
Per-key order	**CR** [269], CRAQ [306], PB [235], CHT [62]	**ABD** [201], CP [181], Hermes [168]

Table 2.1: CFT protocols taxonomy.

Spanner [71], etc.) traditionally layer query processing and Txs on top of a per-node storage persistent engine, e.g., RocksDB [271] or LevelDB [186] and others [99]) (for **D**urability) as shown in Figure 2.1. These persistent storage engines are increasingly based on log-structured merge-trees (LSM) [353, 271, 186, 179, 129, 20] due to their high read/write performance. We build TREATY on top of RocksDB [271] where the data is stored in multiple levels, increasing in size. Higher levels (MemTables) are stored in the memory while the bulk of the lower levels (and thus of the data) is stored on disk in (immutable) SSTables. Updates are applied to the MemTable and when it exceeds a maximum size, it is merged into the next lower level (*compaction*) into an immutable SSTable file, in what is known as a compaction event. If this causes the next level to exceed its own maximum size, the compaction cascades further. The system remains correct under failures through a combination of write-ahead logging (WAL) and a MANIFEST file that records all changes in the system.

2.1.2 Replication in Distributed Data Stores

Distributed KV data stores must remain available and operate correctly when failures occur. As such, they usually deploy replication protocols to create multiple consistent copies of the data across a distributed set of cooperating machines or replicas. Replication techniques in distributed databases can vary in how data is accessed and updated, which impacts their consistency and availability guarantees. In this thesis, we explore protocols that enforce either sequential consistency [180] or linearizability [131], also referred to as *strongly consistent* replication protocols. We divide the strongly consistent replication protocols based on their fault model, i.e., Crash Fault Tolerant or Byzantine Fault Tolerance.

Crash Fault Tolerant (CFT) protocols. CFT protocols assume that the infrastructure is trusted. These protocols tolerate only benign faults; replicas can fail by stopping or by omitting some steps [85]. As such, while having low overheads requiring at most $2f+1$ replicas to tolerate up to f benign faults, they are not suitable for modern applications deployed in third-party untrusted cloud infrastructure [17].

We can broadly split strongly-consistent CFT protocols into two categories (see Table 2.1 for the taxonomy): *(i)* leader-based protocols (e.g., Raft [236], Chain Replication (CR) [269], Primary Backup (PB) [235]), where a node, designated as a leader, drives the protocol execution and *(ii)* decentralized protocols (e.g., ABD [201], All-Concur [251], Classic Paxos (CP) [181]), where there is no leader and all nodes can propose and execute requests.

We further divide them based on their ordering semantics. First, protocols with total ordering, where the protocols create a total order of all writes across all keys and apply them in that order. Second, protocols with per-key ordering semantics where the protocol enforces the total order of writes on a per-key basis. This protocol classification has been proven to be the most representative when studying the protocols' performance [106].

In our RECIPE project (Chapter 4), we transform one protocol (shown in bold Table 2.1) for Byzantine settings as a representative system to study each category.

Byzantine Fault Tolerant (BFT) protocols. In contrast to CFT protocols, BFT protocols assume very little about the nodes and the network; faulty nodes may behave arbitrarily while the network is unreliable. To tolerate f arbitrarily faulty processes that may *equivocate* (i.e., make conflicting statements for the same request to different replicas), BFT protocols add f extra replicas to their system model requiring at least $N = 3f + 1$ replicas for safety. As such, BFT protocols exhibit worse scalability compared to CFT protocols (which only require at most $2f + 1$ replicas).

BFT protocols are limited in performance, too. They incur high message complexity ($O(N^2)$) [165, 58, 328], multiple protocol rounds [196, 165, 58, 4, 345] and complex recovery ($O(f^2)$ in view-change) [196, 58, 328, 165]. As an example of this, PBFT [58], a well-known BFT protocol, requires at least $3f + 1$ nodes and executes three *all-to-all* broadcast rounds incurring $O(N^2)$ message complexity.

Thirdly, BFT protocols are complex, introducing burdens to developers. Even if system designers wish to use a state-of-the-art BFT protocol, optimizing it for the specific application settings (e.g., network bandwidth, number of clients and replicas, cryptographic libraries, etc.) can be complicated. Guerraoui et al. [30] found that most protocol implementations consist of thousands of lines of (non-trivial) code, e.g., PBFT [58] and Zyzzyva [172]. Even trivial changes or intuitive optimizations can be extremely hard and might affect other parts of the protocol (e.g., view-change in Zyzzyva).

2.2 Trusted Computing

This thesis presents three secure distributed management systems. To offer security, we build on top of the state-of-the-art hardware advancements in the trusted computing domain, i.e., trusted execution environments (TEEs). Precisely, we build a secure distributed transactions protocol and robust replication protocols in TREATY (Chapter 3) and RECIPE (Chapter 4) projects respectively on top of Intel SGX [148]. In TNIC project (Chapter 5), we compare our trusted NIC architecture with two competitive TEEs, Intel SGX [148] and AMD-sev [15].

2.2.1 Trusted Execution Environments

Trusted execution environments (TEEs) [148, 24, 15, 184, 23, 149] offer a tamper-resistant confidential computing environment that provides hardware-enforced isolation for executing security-sensitive application logic. TEEs can guarantee the integrity and confidentiality of their executing code and data, whereas their content remains resistant against all software attacks, even from privileged code (e.g., a compromised OS and hypervisor) and physical attacks (e.g., memory probes performed on the main memory of the system).

Compared to its "ancestors", i.e., Homomorphic Encryption (HE) [130] and Trusted Platform Modules (TPM) [311], TEEs are superior both in their security guarantees and adaptability. For instance, in contrast to a typical TEE, a TPM is not programmable with arbitrary code. Additionally, the TPM provides secure storage for secrets (e.g., encryption keys, passwords, and digital certificates), but it cannot vouch for the validity (i.e., authenticity and integrity) of the data signed by those keys. On the other hand, a typical homomorphic encryption algorithm can protect arbitrary data but by itself cannot ensure that the correct operations have been done and that the code has not been tampered with. TEEs offer strong isolation for both the data and the code. TEE and TPM's properties rely on the assumption that the hardware manufacturer is trusted. In contrast, HE executes computations on the encrypted data (without having to decrypt them). However, HE is not a generic technology as only some computations on encrypted data are possible [135, 228].

TEEs have been launched by major CPU providers, e.g., Intel SGX/TDX [148, 149], Arm TrustZone [24], Arm Realms [23], RISC-V Keystone [184], AMD-sev(-snp) [15] and have been adopted by major cloud providers [217, 68, 114, 115, 174]. Each of these TEEs enabling technologies offers different degrees of guarantees that can be leveraged to increase the confidentiality and integrity guarantees in applications. Table 2.2 summarizes the popular TEEs security features (explained below):

	Intel SGX	TrustZone	AMD-sev	RISC-V Keystone
Integrity	✓			✓
Confidentiality	✓			✓
Freshness	✓		✓	✓
Local attestation	✓		✓	
Remote attestation	✓		✓	✓
Isolation granularity	Address space	Secure world	VM	Secure world

Table 2.2: Comparison of the state-of-the-art TEEs.

- **Integrity:** The prevention of TEEs trusted memory from being tampered with.
- **Freshness:** The protection of TEEs trusted memory against rollback attacks. In this thesis, we use the term rollback attacks to refer to attackers that strive to revert the system state to a stale, potentially valid, state (e.g., by intentionally shutting the system down and replacing the (persistent) data/state for recovery with an older version of them).
- **Confidentiality:** The encryption of TEEs trusted memory to assure that no unauthorised access or memory snooping of the trusted memory area occurs.
- **Local attestation:** A TEE instance attests genuineness to another instance running on the same system. We elaborate more on the attestation mehanism later in this section.
- **Remote attestation:** A TEE instance attests genuineness to remote parties, such as clients or other nodes in the system (§ 2.2.2).
- **Isolation granularity:** The level of granularity where the TEE operates for providing isolation and attestation of the trusted software.

TEEs categorization. TEEs introduce the idea of a trusted computing base (TCB) that denotes the code and data that are stored within the trusted hardware. In addition, we use the terms untrusted (or normal) and trusted (secure or protected) "worlds" to denote the untrusted and TEE-protected trusted parts of an application, respectively.

We can briefly classify modern TEEs into two categories based on their isolation guarantees. The first category includes TEEs such as Arm TrustZone [24] and RISC-V Keystone [184] that enforce strict isolation between the secure and normal worlds, prohibiting the software stack and data from being shared between them. 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.

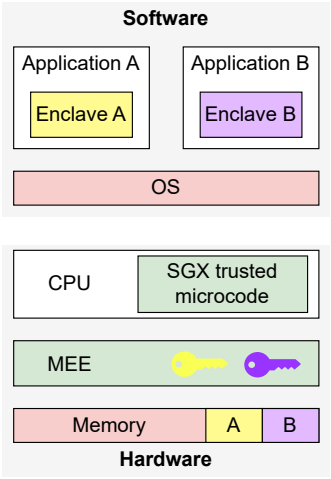

Figure 2.2: Overview of Intel SGX architecture.

The second category of TEEs allows the trusted world to access the untrusted state (e.g., code and data in the normal world). The TEE software can access the unprotected application's memory and even switch to the unprotected, outside-of-the-TEE execution mode, performing a "world switch". These TEEs come with a limited trusted memory area and specific SDKs for programmability. Intel SGX [148] serves as a prime example of this type of TEE, which we analyze later in this section.

Modern TEEs such as Intel TDX [149], AMD-sev-(snp) [15] and ARM Realms [23] port within the trusted hardware the entire virtual machines (VMs) offering confidential VMs (CVMs). Compared to previous TEEs systems (e.g., Intel SGX), VM-based TEEs come with a larger TCB within the trusted environment, but they facilitate easier development since they expose OS-based programming interfaces. Over the past few years, there has been a growing trend towards this category of TEEs.

Intel SGX. Figure 2.2 shows the overview of Intel SGX platform. Intel SGX is a set of x86 ISA extensions for TEE [74] that offer the abstraction of an isolated memory, the enclave. Enclave pages reside in the Enclave Page Cache (EPC)—a specific memory region (94 MiB in v1, 256 MiB in v2) that is protected by an on-chip Memory Encryption Engine (MEE). For larger enclave sizes, SGX implements a rather expensive paging mechanism [26, 39, 242] that encrypts evicted EPC pages and decrypts them when they are brought back.

SGX applications cannot execute outside-of-the-enclave code directly, e.g., system calls, since the OS is considered untrusted. To enable this, SGX enclave threads exit the trusted environment (world switch) and further copy all associated data out of the enclave since the kernel code cannot access it. After the syscall execution, threads have to enter the enclave again and then copy the result of the syscall back to the trusted

enclave. We refer to this as a world switch.

Confidential computing frameworks: SCONE. Confidential computing frameworks leverage TEEs to secure unmodified applications. They can broadly be categorized as libOS-based systems [43, 315, 256, 307], and host-based systems [26, 242, 288]. All of these efforts seek to minimize the number of enclave transitions (world switches) due to their high cost (e.g., TLB flushing, security checks [74]).

In this thesis, we make use of SCONE [26] framework that is built on top of Intel SGX to overcome the performance limitations of the executing syscalls as well as to facilitate the development effort. Particularly, we have built our TREATY and RECIPE systems on top of SCONE [26] that exposes a modified libc library. SCONE combines user-level threading and asynchronous syscalls [293] to reduce the cost of syscall execution.

AMD-sev. Secure Encrypted Virtualization (SEV) is an extension to the AMD-V architecture to support multiple running confidential VMs under the control of a hypervisor. SEV integrates main memory encryption capabilities with the existing AMD-V virtualization architecture to protect the running VMs from physical threats, other VMs, and the hypervisor itself.

SEV hardware tags the VM code and data with its VM identifier. Each VM is associated with a tag and an associated encryption key. The tag is stored with the data at all times when inside the trusted hardware, preventing access from other VMs, including the hypervisor. To protect the VM data residing outside the trusted hardware, SEV employs an AES-128 engine. When data leaves or enters the SOC, it is encrypted/decrypted respectively by the hardware with the associated key. SEV applications require no software modifications, and the technology is particularly applicable to cloud computing, where virtual machines need not fully trust the hypervisor and administrator of their host system.

2.2.2 Remote Attestation

Remote attestation is a security mechanism that verifies that the system state, including the hardware and the software, 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. Remote attestation requires a trusted entity (i.e., *root of trust*), such as a trusted execution environment (TEE), to attest its loaded code and calculate a *measurement*, i.e., secure hashes over the loaded software. First, the root of trust measures the integrity of all software elements (e.g., the bootloader, operating system kernel, hypervisor, software within a TEE). The loaded software then measures additional components, such as programs, and inform the root of trust about their measurements.

The root of trust combines its own calculated hash with the new measurements, producing a new measurement value. A remote entity (challenger) can then request the root of trust to provide the signature of this new hash value (using a hardware key burned in the TEE at the manufacturing process). By inspecting this signed value, the remote process can validate the authenticity of the system's hardware and software. To do so, the challenger verifies the signed measurement of the root of trust by comparing the measurement value (hash value) against a pre-established known value. Remote attestation allows third-party entities, e.g., other nodes or clients, to gain confidence in the integrity and security of remote system stacks.

Intel Attestation Service. To enable remote attestation, Intel offers a Intel Attestation Service (IAS) [141]. In this thesis the CPU Manufacturer is considered trusted, hence IAS, that is managed by Intel, is also trusted. IAS verifies the measurements (or *quotes*) produced by a TEE and provides a verification report upon the TEEs' quote successful verification. This verification report is then sent to the TEE and can be forwarded to the remote party that challenges the TEE. Then, the remote party can verify IAS verification report to gain trust about the genuineness of the TEE and its running software.

2.3 High-performance Networking

High-performance networking is at the core of the distributed systems we study in this thesis. Precisely, in this thesis we leverage the advancements in networking technologies to improve the systems performance as well as guarantee security. TREATY (§ 3.5.1) and RECIPE (§ 4.5.2) leverage and extend a user-space direct I/O library, eRPC [162], on top of DPDK [150] and RDMA [209] which we discuss next in § 2.3.1. Our last project, TNIC (see § 5.4 and § 5.2), extends the scope of the state-of-the-art SmartNIC devices (§ 2.3.2) to guarantee fundamental security properties in the network interface controller (NIC) level.

2.3.1 Direct network I/O

The main mechanism to communicate with devices is through memory-mapped registers and device memory. I/O memory is a region of DRAM-like locations that the device makes available to the processor over the PCIe bus. Conventional applications use syscalls where the network stack and the I/O are handled inside the OS kernel, incurring the overheads of kernel context switches [158, 128, 293, 326, 70]. Recent high-performance distributed systems [168, 88, 164, 162] have shown that it can be beneficial to perform I/O directly to or from user-space buffers. We refer to this technique with the terms direct I/O or kernel-bypass. Direct I/O improves network latency

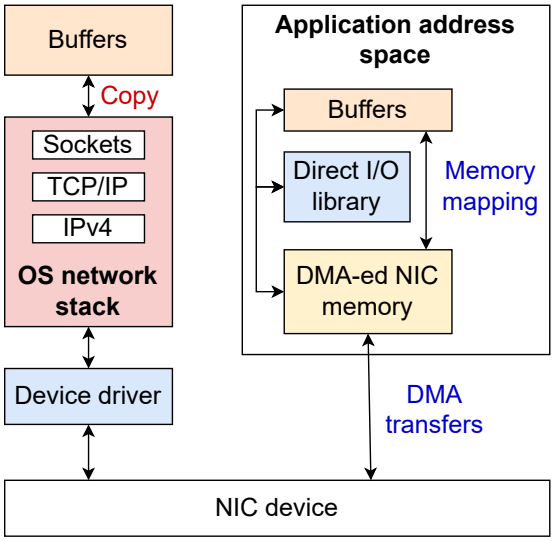

Figure 2.3: Comparison of direct I/O with traditional kernel networking.

and throughput because applications are capable of transferring data directly without an extra copy through kernel space or the overhead of the context switches due to syscalls. Figure 2.3 presents a comparison of the kernel-based networking compared to the direct I/O approach.

The direct I/O or userspace I/O system (UIO) operates by mapping (i.e., associating) the device (i.e., device memory) to a set of userspace addresses. This mapping is typically implemented by the device driver that appropriately creates the application's page table to map the device memory to an application's virtual memory area (VMA). As such, whenever the user program reads or writes in the assigned address range, it is actually accessing the device.

To further optimize performance and the system's resource utilization, modern systems usually implement direct I/O through bus mastering that enables direct memory access (DMA) between host and device memory. DMA allows the network card interface (NIC) and other peripheral components to transfer their I/O data directly to and from main memory without the need to involve the CPU.

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.

However, direct I/O often requires from the application to check for incoming messages (polling) which might increase unnecessarily the CPU utilization and the application's latency. In this thesis we build our systems on top of eRPC library which we discuss later in this section. eRPC builds on top of direct I/O techniques, such as DPDK

and RDMA that do not provide abstractions of the transport layer (TCP/IP, UDP, etc.) whereas the device is bound exclusively to a specific application. On the other hand, the OS-based networking carefully utilizes the CPU by interrupting the execution only when an incoming packet is ready for processing while it offers a portable API supporting TCP/IP and UDP sockets. OS-based networking allows the NIC to be utilized concurrently by multiple applications running on the same host.

DPDK and (one-sided) RDMA. Data Plane Development Kit (DPDK) [150] 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 expects the user to provide their own network stack for the session and transport layer.

RDMA is the abbreviation of remote direct memory access that enables computers in a network to exchange data in the main memory without involving the remote CPU. RDMA supports zero-copy networking by transferring the data directly from the application memory to the NIC buffers and vice versa. RDMA is implemented by the following network protocols: RDMA on Converged Ethernet (RoCE), Internet-wide area RDMA protocol (IWARP), and InfiniBand.

Both RDMA and DPDK [150] are widely favored for high-performance as they *(i)* map a device into the user address space, effectively bypassing kernel, and *(ii)* replace the costly context switches with a polling-based approach. While these technologies have been used to improve throughput in modern distributed systems [168, 164, 88], their naive usage might not always offer performance for free [163]. In addition, these protocols operate in the internet or link network layers shifting the responsibility for implementing the transport layer (TCP/IP, UDP) to the system developer.

eRPC. In this thesis, we adopt direct network I/O because it is even more well-suited to TEEs where syscall execution is extremely expensive [109, 38]. Specifically, we leverage eRPC [162], a general-purpose and asynchronous remote procedure call (RPC) library for high-speed networking for lossy Ethernet or lossless fabrics. eRPC provides us with a UDP stack over multiple transport layers, e.g., RDMA and DPDK. As we explain in each of our projects, we extend eRPC to eliminate its trusted memory footprint and shield the network messages against adversaries in the untrusted network infrastructure.

2.3.2 SmartNICs

Emerging programmable NICs, or SmartNICs [194, 317, 47, 49, 231, 10, 32, 59], represent another promising approach to improve network latency while reducing host CPU workload and power consumption. SmartNIC devices have already shown their

	Performance	Programmability	Security
SoC-based NICs	✓	✓✓✓	✓
FPGA-based NICs	✓✓✓	✓✓	
ASIC-based NICs	✓✓✓		✓

Table 2.3: Comparison of SmartNIC categories.

performance benefits in networked systems [102, 197] and have already been launched by major cloud providers [10, 32, 59].

We can systematically categorize SmartNICs into three different hardware designs: *(1)* ASIC-based NICs [231] that have dedicated hardware for common network processing functions (TCP checksum and segmentation, RSS, etc.), *(2)* (multicore) SoC-based NICs [47, 10, 49, 194] that have an on-chip set of embedded (ARM) cores, and *(3)* FPGAs [317, 292, 59] that introduce a fully programmable hardware "on-path" with the NIC device.

All these hardware designs for SmartNICs present different characteristics in performance and programmability, as shown in Table 2.3. The ASIC NICs provide the highest performance potential, but they suffer from a lack of programmability and adaptability over time [102]. On the other hand, the SoC-based SmartNICs' on-chip cores have full OS support and are fully programmable. Unfortunately, by the time of this thesis writing, we only had access to Bluefield 2 [47] cards where the embedded cores are "off-path" with no support for DMA transfers between the on-chip and host memory. As such, a network operation would involve an extra (intra-host) network operation for forwarding the request from the application to the device cores, increasing the operation's latency. The third category, the FPGA-based SmartNICs, combines the best of both worlds. First, the FPGAs are fully programmable, allowing us to design and implement the exact security properties a system requires on hardware. Secondly, the FPGA resides on-path with the NIC, allowing it to process all incoming/outgoing network traffic on the communication path, optimizing for latency [289].

2.4 Storage Systems and Technologies

This thesis presents distributed systems that need to store data reliably across distributed cooperating machines. TREATY leverages SPEICHER storage system (§ 2.4.1) that is also built on top of Intel SGX and extends the trust to the untrusted storage, providing an authenticated persistent data layer.

2.4.1 SPEICHER Storage System

SPEICHER [39] is a secure storage system based on RocksDB and SGX that offers authenticated and secure LSM data structures. SPEICHER neither supports Txs nor distribution. Clients execute PUTs whose ordering is only secured in a future synchronization point based on an asynchronous trusted counter module. Shutdowns and crashes in the meantime require clients to re-execute the operations, which might change their initial order. In TREATY, we leverage SPEICHER as the underlying storage system, and we extend the following to support Txs processing (§ 3.5.2): controller, buffer management, I/O subsystem, and LSM and logging data structures.

Chapter 3

TREATY: Secure Distributed Transactions

We introduce TREATY, a secure distributed transactional KV storage system that supports serializable ACID transactions while guaranteeing strong security properties: confidentiality, integrity, and freshness. TREATY leverages trusted execution environments (TEEs) to bootstrap its security properties, but *it extends the trust provided by the limited enclave (volatile) memory region within a single node to build a secure (stateful) distributed transactional KV store over the untrusted storage, network and machines*. To achieve this, TREATY embodies a secure two-phase commit protocol co-designed with a high-performance network library for TEEs. Further, TREATY ensures secure and crash-consistent persistency of committed transactions using a stabilization protocol. Our evaluation on a real hardware testbed based on the YCSB and TPC-C benchmarks shows that TREATY incurs reasonable overheads, while achieving strong security properties.

3.1 Motivation

Transactions (Txs) are an integral part of modern cloud computing systems [82, 88, 271, 71]. They hide complexities (data distribution, concurrency, failures, etc.) from programmers and, at the cloud scale, they provide a powerful abstraction to atomically process massive datasets that might be distributed across different machines [14, 216, 113, 84].

While distributed transactional Key-Value (KV) stores are extensively used to build scalable applications with a high degree of reliability and cost-effectiveness, offloading Tx processing in the cloud also poses serious security threats [274]. In untrusted cloud environments, adversaries can compromise the confidentiality and integrity of the data

and application's execution state while they can also violate Txs semantics (isolation, atomicity) by intentionally returning stale/uncommitted data. Prior work has shown that software bugs, configuration errors and security vulnerabilities pose a real threat for storage systems [122, 78, 275, 112, 177]. These security threats are amplified in distributed stores as the state is distributed across machines connected to the untrusted storage and network system stacks.

This work pursues the following question: *How to design a high-performance, serializable, distributed transactional KV store that offers strong security properties?*

A promising direction to this question is to use trusted execution environments (TEEs)—the new trend in confidential computing—to build a secure distributed transactional (Tx) KV store. TEEs provide a secure memory area (enclave) where the residing code and data are protected even against privileged code (e.g., OS, hypervisor). Based on this promise, TEEs are now being streamlined by all major CPU manufacturers [24, 23, 148, 15, 184], and adopted by major cloud providers [68, 217, 114].

However, we cannot use TEEs out-of-the-box to build a secure distributed KV store with Txs. Particularly, we need to address the following three challenges:

First, TEEs only protect a limited volatile memory region (enclave) within a single node. These security properties do not naturally extend to the untrusted persistent storage and network over a distributed set of nodes, which are essential to build a secure distributed transactional KV store.

Secondly, TEEs primarily rely on the expensive syscall mechanism for I/O operations, where the enclave threads need to perform *world switch* to execute the syscall that has been shown to be costly [339]. While modern confidential computing frameworks [26, 288, 315] provide an asynchronous syscall I/O mechanism [293] to alleviate the performance overheads, they are still inadequate for modern distributed storage systems that prominently rely on high-performance networking such as RDMA or kernel-bypass [164, 284, 88, 337]. Unfortunately, it is not trivial to combine high-performance networking with the TEEs because TEEs fundamentally prohibit unauthorized access to the protected enclave via a DMA connection.

Thirdly, distributed stores need to ensure secure and crash-consistent persistency for committed Txs. Secure persistency for distributed systems can be a challenge in an untrusted environment where adversaries can rollback the database state, to a stale yet consistent snapshot violating correctness. While SGX [148] provides hardware-based trusted counters, a fundamental building block for rollback protection, writes incur prohibitively high latency [146, 39]. Further, we need to establish trust between the nodes in the distributed setting to protect against forking attacks. TEEs' attestation mechanisms provide a building block to bootstrap trust. Unfortunately, they cannot

provide collective trust for a distributed set of nodes [121].

To address these challenges, we present TREATY, a distributed KV store with serializable ACID transactions [248] and strong security properties: *integrity*—unauthorized changes can be detected, *confidentiality*—unauthorized entities cannot read the data, *freshness*—stale state of the system can be detected. TREATY embodies three core contributions:

1. **Distributed Tx protocol:** The design of a secure two-phase commit (2PC) protocol for distributed Txs providing strict serializability. Our protocol leverages TEEs for security, and it is co-designed with a kernel-bypass network stack for TEEs to ensure high-performance execution (§ 3.3).

2. **Stabilization protocol:** The design of a stabilization protocol that guarantees secure and crash-consistent persistency of committed Txs. That is, the protocol ensures crash consistency, recovery, and data freshness across rollback and forking attacks in distributed settings (§ 3.4).

3. **Trusted substrate for distributed TXs:** The design of a trusted substrate for distributed Txs—with which we build TREATY—that overcomes the limitations of TEEs. Specifically, we propose *(a)* a secure network library for Txs based on kernel-bypass I/O within TEEs, *(b)* a secure storage engine for Tx processing, *(c)* a userland-scheduler for low-latency requests, and *(d)* a memory allocator for secure Tx buffers management (§ 3.5).

We implement TREATY from the ground-up as a distributed KV store [353, 69], where we layer a distributed Tx layer (2PC) on top of per-node storage engine based on a secure version of RocksDB's [271] storage engine: SPEICHER [39]. Our secure 2PC is co-designed with Intel SGX as the TEE and a secure networking library based on eRPC [162].

We evaluate TREATY with TPC-C [310] and YCSB [349] on a real hardware cluster. Our evaluation shows that TREATY incurs reasonable overheads—6×-15× and 2×-5× for distributed and single-node Txs, respectively—while providing serializable distributed Txs and strong security properties. The overheads derive mainly from TEEs as (1) native runs of TREATY perform similarly to RocksDB, (2) encryption increases the overhead up to 1.4× compared to non-encrypted versions and (3) stand-alone evaluation of TREATY's 2PC shows 2× slowdown w.r.t. a native version of the protocol.

3.2 Overview

3.2.1 Threat and Fault Model

TREATY extends the standard SGX threat model [43] to provide stronger security guarantees even for a distributed setting, where we also consider the untrusted storage and network. An adversary can (1) control the entire software stack outside the enclave (including the network stack, i.e., they can drop, delay, or manipulate network traffic) and, (2) view/modify all non-enclave memory, i.e., untrusted host memory and persistent storage (SSDs). The adversary can perform rollback attacks and revert TREATY nodes to a stale state by intentionally shutting them down and replaying older logs that store the system state for durability and recovery. We assume a *crash-recovery* model: TREATY nodes can crash at any point and will eventually recover. In-memory state is lost upon failure; persistent state (SSDs) is preserved. TREATY guarantees serializability and atomicity in the presence of failures, rollback attacks and other attacks that strive to alter the order of the committed transactions. TREATY also maintains data integrity, confidentiality and freshness. TREATY builds on a fault-tolerant trusted counter service (see § 3.4) which replicates the trusted counters—the core building block for rollback protection [206, 18, 300]—in a set of N SGX-equipped nodes. TREATY trusted counter service requires $N \geq f + 2u + 1$, where f is the maximum compromised SGX nodes (e.g., adversaries can violate the runtime memory, read all enclave secrets and the SGX processor keys) and u is the maximum number of non-responsive (correct) nodes at the time of the counter writing or reading.

We do not protect against side-channel attacks: cache timing, speculative execution [324, 325, 156, 321, 279, 227, 171, 193], access pattern leakage [343, 127], memory safety vulnerabilities [178, 234] or denial of service attacks.

3.2.2 System Overview

Figure 3.1 illustrates our system architecture. TREATY is a sharded transactional KV system, where we layer the Tx layer that implements a secure 2PC protocol (Agreement protocol) on the top of on a persistent KV store (SPEICHER): multiple nodes in the system store subsets of the data and coordinate to maintain consistency. Each node consists of two parts: 1) a trusted set of components that resides in the enclave memory and contains the Txs layer, lock manager, and Txs KV engine, and 2) the untrusted network and storage stack.

Clients communicate with the system through a mutually authenticated channel. Specifically, the system developer *burns* a public-private key pair into the TREATY code that is used by clients to establish a TLS connection with TREATY nodes and proceed to

attestation. The public key is known to the clients, e.g., it can be shared through the (trusted) system developer. The clients establish a TLS connection with their preferred TREATY node and then *attest* this node through TREATY's Configuration and Attestation Service (CAS) that is explained in section § 3.4[1]. After a client successfully attests a node, it can establish a new TLS connection for executing requests using a node's newly generated public key that is shared through the previous TLS connection.

The developer also burns a separate public-private key pair into the TREATY codebase that is used by TREATY nodes to attest each other when needed through CAS. This key pair is secret and it is not known to the clients or the cloud provider. After the nodes have been attested through CAS, they are supplied with the the necessary configuration and secrets as explained in § 3.4.

TREATY exposes a standard transactional API: Txs begin and end through `BeginTxn()` and `TxnCommit()/TxnRollback()` calls, and execute operations through `TxnPut()` and `TxnGet()` operations. More specifically, TREATY's transactions maintain the following properties:

[1]Clients also establish a TLS connection with TREATY's Configuration and Attestation Service the same way they establish secure communication channels with the TREATY nodes.

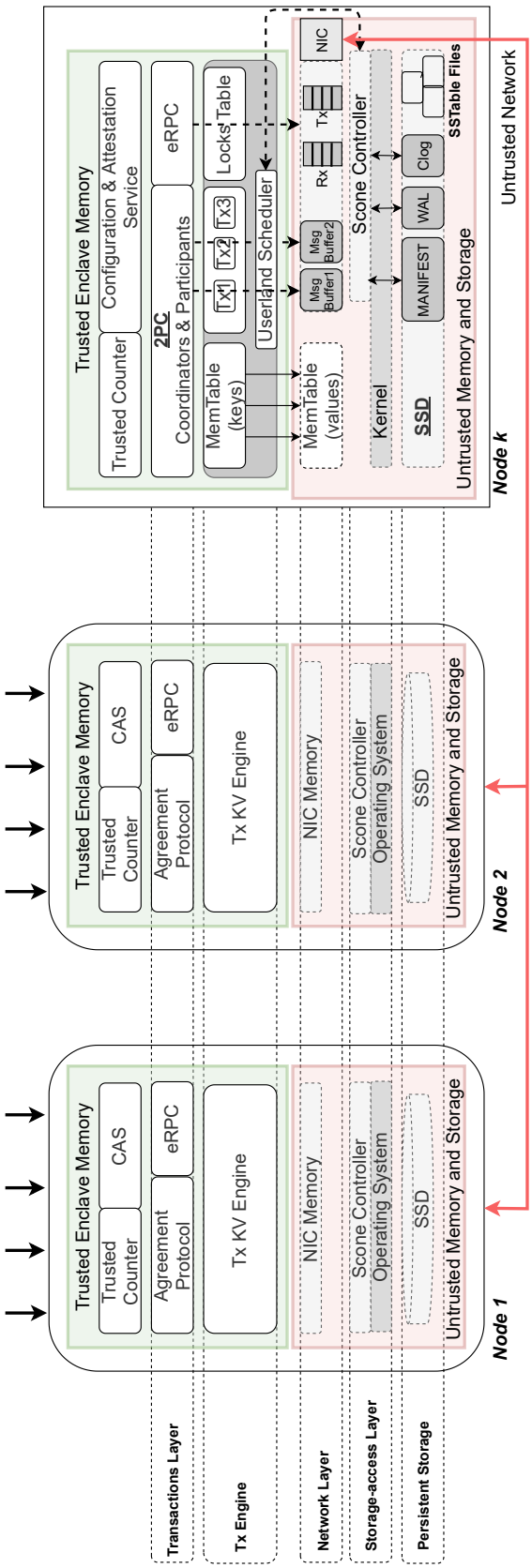

Figure 3.1: TREATY's system architecture.

- *Security.* TREATY guarantees confidentiality, integrity and freshness for all Txs in the presence of untrusted storage and networking over a distributed set of nodes.

- *Programmability.* TREATY offers general serializable ACID Txs which offer the strongest possible correctness guarantees, combined with general purpose, interactive Txs that minimize the programming burden on developers.

- *Performance.* TREATY's careful design minimizes the performance limitations of TEEs (limits on EPC memory, high latency of trusted counter and I/O execution).

TREATY achieves security by designing two protocols: *(i)* a 2PC protocol for the correct and secure execution of distributed Txs (§ 3.3) and, *(ii)* a stabilization protocol for secure and crash-consistent persistence of the committed Txs (§ 3.4). Lastly, TREATY's substrate (§ 3.5) for distributed Txs is designed and implemented with consideration to the TEEs architectural limitations (enclave memory, I/O, scheduling).

TREATY shares the Tx execution workflows of existing systems. Clients start Txs by selecting a transaction coordinator, who is responsible for driving the Tx's execution. Upon receiving a read or write request for a key, the relevant node acquires respectively a R/W lock, storing it in a local lock table. When the Tx is ready to commit, the Tx coordinator initiates a 2PC protocol consisting of a prepare and commit phase. The Tx commits if all involved shards vote to commit. Otherwise, the Tx aborts. In either case, locks are released.

Applications can use TREATY API to store and execute transactions on the stored data with the aforementioned security properties. To further extend TREATY's security properties to the entire application, system designers should also run the application code within the TEE.

3.2.3 Design Challenges and Key Ideas

While designing TREATY, we overcome the following challenges:

#1: Security for distributed transactional KV stores. In the untrusted cloud, adversaries can tamper with the Txs' execution. They can compromise the confidentiality and authenticity of the running Txs, modify the executed operations and its associated data. In addition, they can also tamper with the execution of the distributed Txs protocol itself, i.e., 2PC state. All these attacks can violate the correctness guarantees of the system. In addition, such powerful adversaries are also capable of *illegally* modifying or accessing the KV store's content which includes unauthorized modifications and access to the users, potentially private, data.

Key Idea: A secure distributed transaction protocol. For designing secure distributed Txs for untrusted cloud environments, we can rely on a simple 2PC protocol that leverages the security guarantees of TEEs. Unfortunately, TREATY cannot use a TEE as a black box as its security guarantees are restricted only to the (*limited and volatile*) enclave memory of a single node. In contrast, modern transactional systems like TREATY are distributed, communicate over the network, and store their data on a persistent storage medium (SSDs). To implement distributed Txs with TEEs, TREATY needs to overcome the following system challenges.

- Security and correctness for Txs. TREATY's 2PC needs to ensure confidentiality and integrity along with serializability detecting adversaries that aim to double execute Txs.
- Untrusted persistent storage. TREATY needs to protect the persistent data by detecting unauthorised modifications since attackers can tamper with logs to compromise the history of executed Txs and the 2PC state and/or can delete/modify/access the persistent data.
- Enclave memory. TREATY needs to overcome the limited enclave memory challenge. The limited enclave memory is especially problematic for LSM-based systems which rely on a large MemTable to absorb recent read/write requests (before compacting them to the SSTable). Moreover, the Tx layer on top of LSM storage system must also buffer the uncommitted writes for ongoing Txs. Lastly, network buffers for communication further put pressure on the enclave page cache (EPC).

We discuss TREATY's approach for secure distributed Txs in § 3.3. TREATY offers secure and correct execution of distributed Txs by implementing a secure 2PC (§ 3.3.1) leveraging TEEs and a secure network library (§ 3.5.1). Our TREATY's design also adopts SPEICHER's [39] LSM data-structures as a secure store for the untrusted storage (§ 3.3.2, § 3.5.2), but it extends and adapts SPEICHER's storage engine and data structures for the Tx processing and the design of the 2PC protocol for TEEs.

#2: Secure and high-performance networking for distributed Txs. TREATY's nodes communicate with each other. Traditional kernel-based approaches for network I/O (e.g., sockets) experience high overheads due to context switches that are further deteriorated inside the SGX due to the costly enclave transitions.

Confidential computing frameworks, such as SCONE [26], implement async syscalls to eliminate the expensive world switches, but they still rely on the syscall mechanism for the I/O, which is slow and requires two additional data copies (enclave↔host memory↔kernel). This I/O mechanism is ill-suited for distributed systems [164, 337,

88, 284], like TREATY, that prominently rely on high-performance networking with direct I/O or kernel-bypass. Unfortunately, these direct I/O mechanisms are incompatible with TEEs, since TEEs prohibit enclave memory access via the untrusted DMA connection. Therefore, we need to adapt this mechanism in the context of SGX to use it and design the secure distributed 2PC protocol.

Key Idea: A secure networking library for high-throughput and low-latency direct I/O. TREATY implements a secure network library (§ 3.5.1) through which we build the secure 2PC protocol to enable user-space direct I/O (DPDK [150]) based on eRPC [162]. TREATY's secure network library provides high-performance network I/O overcoming the limitations of SGX. In addition, we ensure confidentiality and integrity properties for the exchanged messages through encryption while we also guarantee freshness, thus protection from network-replay attacks, for transactions.

#3: Secure persistency. TREATY needs to ensure that committed Txs are persisted, remain crash consistent across reboots and are protected against forking and rollback attacks. This requirement is necessary to ensure that a transaction's commit is irreversible. Particularly, TREATY needs to detect violations from attackers that selectively shut down and revert the state of some nodes in order to make the system *undo* parts of a distributed transaction. These attackers primarily seek to violate the ACID properties, thus the correctness, of the system.

Key Idea: A stabilization protocol for the committed transactions. TREATY implements a stabilization protocol to guarantee crash consistency, recovery, and committed Txs freshness in distributed settings. The protocol needs to overcome the following challenges:

- Trust establishment. Remote attestation (RE) ensures that the expected code is running, thus, protecting against forking attacks. SGX's RE, provided by Intel Attestation Service (IAS) [141], verifies a measurement of the enclave. Unfortunately, it is designed for a single-node attestation, not offering collective trust for distributed nodes in a data center, while it incurs high latency (requires explicit communication with the IAS). This can significantly slow down recovery after reboots/migrations, where nodes require re-attestation.

- Crash consistency. Logs are commonly used to persist the state and updates of Txs for durability. As these logs reside in the untrusted storage, recovery needs also to verify their freshness and integrity.

- Distributed rollback protection. Trusted counters are widely used to protect against rollback attacks. TREATY further extends their scope to preserve serializability

where Txs are stored along with a trusted counter value that cannot be overwritten. Consequently, the trusted counter values reveal Txs' order as well as the latest trusted state of the system.

While SGX does provide us with monotonic hardware counters, they suffer from three limitations: 1) high latency (e.g., increments can take up to 250 ms [206]) 2) non-recoverability if the CPU fails—indeed, at high incrementing rates, counters wear out after a couple of days [206][2], and 3) they cannot offer rollback protection to a set of machines as they are private per-node.

TREATY designs this stabilization protocol—incorporated into the 2PC—to ensure crash consistent and secure persistency for Txs (§ 3.4). First, TREATY uses a Configuration and Attestation Service (CAS), which is hosted within the datacenter to avoid the calls to IAS, to attest all its nodes. Secondly, it provides crash consistency for Txs through secured persistent logs. Lastly, we build on an asynchronous trusted counter service to avoid the SGX counter limitations and ensure distributed rollback protection (e.g., all parts of a distributed Tx are securely committed (persisted) to all participant nodes).

3.3 Transaction Protocol

TREATY's 2PC protocol ensures the correct and secure execution of distributed Txs (§ 3.3.1). To achieve this, we leverage TEEs to harden the security properties of the 2PC, which we co-design with a high-performance network library based on kernel-bypass (§ 3.5.1), that guarantees strong security for the untrusted network. To realize distributed Txs, we also design single-node Txs support in SPEICHER (§ 3.3.2).

3.3.1 Secure Distributed Transactions

Distributed design. TREATY partitions data into shards that may be stored on separate machines that fail independently from each other. Each TREATY node runs a transactional single-node KV storage engine built on top of RocksDB/SPEICHER [39], as shown in Figure 3.1. We implement a secure 2PC protocol with the userspace network stack based on eRPC [162] to execute distributed Txs and guarantee security properties. For securing the state of the protocol as well as providing secure recovery we make use of authenticated log files (MANIFEST, Clog and WAL) as introduced in SPEICHER system [39]. More specifically, the entries of these files are encrypted, integrity

[2]Prior research [206] found that the non-volatile memory on top of which these Intel SGX counters are implemented wears out after approximately one million writes. As such, these counters are not suitable for systems like TREATY and other persistent storage systems that handle frequent state updates continuously.

and rollback protected through trusted monotonic counters. The logs' entry format is explained in § 3.4. As such, while being stored in the untrusted persistent storage, unauthorized access is prohibited and *illegal* modifications or rollback attacks are detectable by TREATY. MANIFEST logs the changes in the state of the persistent storage (e.g., compactions, live logs). WAL stores the MemTable updates and the prepared Txs. Lastly, Clog is written by Txs coordinators and keeps the 2PC protocol state.

The system's initialization requires a trusted Configuration and Attestation service to establish trust in the distributed system. TREATY's Configuration and Attestation service also leverages TEEs and distributes to nodes important information about the cluster configuration (e.g., secrets and keys' distribution to nodes, network connections).

Clients access TREATY over the network. For each Tx, a TREATY's node initialises a global Tx handle that is uniquely identified by a monotonically increasing sequence number and the node id. A Tx coordinator interacts with the client and distributes their requests to the involved participant nodes. Participants create local private Txs through TREATY's single-node transactional KV store (§ 3.3.2). To ensure isolation, TREATY's engines own a private (per-node) keys lock table.

Lastly, we leverage the exit-less approach of executing syscalls provided by SCONE for accessing the persistent storage. Prior work [39] has introduced SPDK [154] to access the persistent data in the SSDs. However, we did not use SPDK for two reasons. First, in our experiments the database persistent data fits entirely in the kernel page cache. As such (random) read access was much faster than SPDK because the pages were already in-memory (page cache). In contrast, while SPDK optimized the write path due to bypassing the kernel stack, it only cached in-memory the very recent updates. As such, with SPDK, random read accesses in the persistent data required much more frequent access to the SSD, increasing a Tx's latency. The second reason is that we configured SCONE to best fit TREATY for storage I/O syscall execution.

Integrity, confidentiality and freshness. Each node runs a single modified SPEICHER instance. TREATY engine runs inside the enclave to ensure integrity, confidentiality and freshness for the execution and the resided run-time data (e.g., MemTable, transactions' local buffers, hash values).

To extend the trust to persistent storage, we adapt SPEICHER which offers a secure authenticated SSTable hierarchy. SPEICHER stores encrypted blocks of KV pairs as well as a footer with the blocks' hash values (for integrity checks). TREATY extends the persistent data structures by adding an extra log file, the Clog for the 2PC. Lastly, to ensure crash recovery in TREATY, we defer deleting the old SSTables and logs until MANIFEST's entries for that compactions are stabilized.

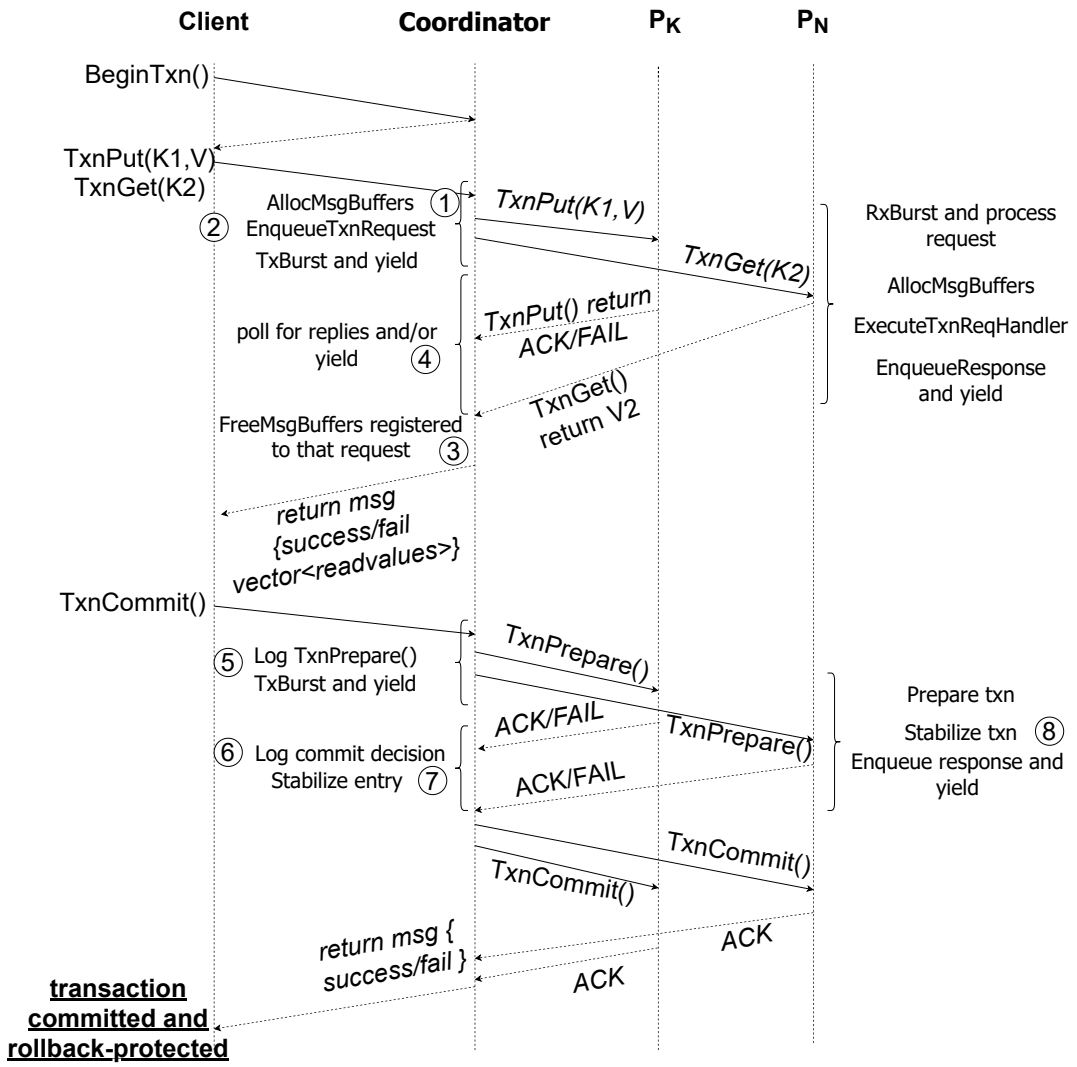

Figure 3.2: TREATY's two-phase commit protocol.

TREATY also extends the trust for the network I/O by constructing a secure message format for Txs (§ 3.5.1). A message encapsulates an Initialization Vector (12 B) and a MAC (16 B) for proving its authenticity and integrity. As explained in § 3.4 the network and storage encryption keys are distributed to a TREATY node by the Configuration and Attestation Service after the node's successful attestation. In addition to Tx's data, we also add some metadata (e.g. node, Tx and operations identifiers) that allows TREATY to protect against duplication of packages by an attacker.

Two-phase commit. TREATY offers serializable distributed ACID Txs with strong security guarantees throughout a secure 2PC protocol implemented over our secure network stack (§ 3.5.1). TREATY also builds a two-phase locking to ensure transactions isolation (and strict serializability [248]). TREATY's transactions acquire locks as they go along and release the locks right after their commit or abort. Transactions' write set uses exclusive locks to ensure that keys are written by a single transaction at a time. The read

set acquires shared locks for performance. Figure 3.2 illustrates the complete protocol design. Clients are connected to TREATY nodes and thereafter, are able to execute transactions. Upon a client's request, the transaction's coordinator node (TxC) initializes a global Tx which is uniquely identified in the entire cluster. The TxC is responsible for driving the secure 2PC protocol to execute the client's request (transaction). Each RPC is strictly owned by one thread, which minimizes shared resources.

The TxC distributes the Tx's requests to the responsible nodes and/or processes its own requests. As shown in Figure 3.2, before forwarding the requests to the participants, each Tx reserves (untrusted) memory for the requests and responses ①. These message buffers have to remain allocated until the entire request has been served ③. To eliminate paging overheads, they reside encrypted in the untrusted host memory.

Once the message is constructed, the TxC enqueues the request ②. Note that enqueuing the request does not transmit the message. In case of multiple requests, coordinators can defer the transmissions until all requests are enqueued. Once the TxC has executed its own-managed requests and has forwarded all requests to the participants, it yields and periodically checks if the participants have replied ④.

At a commit, TREATY first prepares the Tx for a distributed commit accross all parties involved. Every Tx/operation is logged to Clog with its own unique trusted counter value ⑤. Afterwards, all participants prepare their local Tx. Participants delay replying back to the coordinator until the prepare entry in the log is stabilized ⑧. TREATY's stabilization ensures that coordinators will not consider the Tx as successfully prepared until all participants ensure that they are able to recover and commit the transaction after a crash. If not all participants ensure that their prepare phase is stabilized, after a crash this entry cannot be safely recovered. Especially in cases where the participants had already committed the entry but only some of them could recover the committed Tx after a crash, the system would be in a inconsistent state where distributed Txs are partially committed to some, but not all involved, nodes.

The TxC, before committing/aborting, also stabilizes the prepare's phase decision on the Clog ⑥-⑦. If the TxC crashes before this entry is stable, the recovered coordinator will re-execute the prepare phase. Once this is rollback protected, the Tx can commit. We do not need to wait for the commit entry to be stable to reply to the client. Even if the system crashes, this Tx can be committed in the exact same order as before the crash. Permanent failures of the TxC block a transaction's execution. Also, the TxC shard (key space) is not accessible to transactions that are initiated by other node coordinators. For ongoing transactions in the prepare or commit phase the locks on its keys cannot be released and as such the transaction will block infinitely. Prior to prepare-phase, ongoing transactions acquire timed locks to resolve deadlocks. This

behavior is in line with the traditional, non-secure, version of the protocol.

3.3.2 Secure Single-node Transactions

KV Storage engine and single-node Txs. TREATY's storage engine runs inside the enclave for which the security properties are guaranteed. TREATY leverages SPEICHER's data model that offers an authenticated LSM structure for the persistent storage but also optimizes the usage of EPC memory. Particularly, TREATY adapts SPEICHER's MemTable design by separating the keys from the values. We keep keys along with their version number inside the enclave, while we place the encrypted values in the untrusted host. To access values and prove their authenticity we similarly keep a pointer to the value as well as its secure hash value along with the key.

However, SPEICHER cannot support Txs; therefore we extend it to integrate both optimistic and pessimistic Txs exporting an interface to the upper Tx layer to access the LSM-data structure. We preserve the RocksDB's interface and semantics. For the persistent storage, TREATY extends the persistent data structures by adding an extra log file, the Clog for the 2PC. TREATY's distributed Txs can then be viewed as the set of all participants' single node Txs.

Pessimistic Txs take locks on their keys while optimistic Txs use sequence numbers to identify conflicts at the commit phase. For optimistic Txs, each key has a seq. number showing its the latest version and is atomically increased during the commit phase. At commit, Txs log their updates to the WAL and update the MemTable. We only reply to a client after the Tx becomes stable, ensuring that upon a crash, clients will not have to re-execute successfully committed transactions. Thus, conflicting transactions will maintain their initial ordering.

Lock tables. Nodes store a table of locks for their keys that is divided across shards, each protected with a lock, by splitting the key space. TREATY runs with a big number of shards (configurable by the system designer) to avoid locking bottlenecks. Txs that fail to acquire a lock within a timeframe, return with a timeout error.

3.4 Stabilization Protocol

TREATY's stabilization protocol ensures *secure and crash-consistent persistency* for the committed Txs. To achieve this, our protocol relies on three core principles. First, TREATY establishes trust between the nodes based on collective remote attestation. Secondly, after the 2PC's execution (§ 3.3), TREATY ensures crash consistency for the committed Txs. Lastly, once Txs are crash-consistent, TREATY ensures rollback protection in distributed settings. We next explain these three principles.

Distributed trust establishment. Upon startup TREATY bootstraps a Configuration and Attestation Service (CAS) on a TEE in a node in the network to provide scalable remote attestation and authentication. For attestation, the service provider verifies the CAS over Intel Attestation Service (IAS). On success the service provider deploys an instance of TREATY's local attestation service (LAS) on TEEs on all nodes, verified by the CAS over IAS. The LAS replaces the Quoting Enclave (QE)[3], collecting and signing quotes for all TREATY instances, running on the node. After the CAS verified a new instance, it supplies securely the instance with the necessary configuration, e.g., network and storage encryption keys, nodes' IPs, etc. TREATY nodes use a shared symmetric key for network communication. Each node has its own storage key that is different from the network key. CAS and LAS cannot be compromised and leak secrets because they are hosted within a TEE. However, CAS remains a single point of failure; failures in CAS will block the attestation of new or recovered TREATY nodes. LAS instances can also fail and block attestation, however, as long as CAS is operating the system designer can bootstrap new LAS instances.

The CAS is used to attest TREATY nodes and establish trust between TREATY and clients, i.e., clients can attest a TEE and verify the received attestation through CAS.

Crash consistency and recovery. After the 2PC's execution, TREATY ensures crash consistency and recoverability using three persistent log files; MANIFEST, WAL and Clog. As discussed in § 3.3, Clog logs the 2PC states, WAL the committed data and MANIFEST stores the state changes in the SSTables. TREATY relies on each of individual logs being written sequentially; thus, it assigns to each of their entries a unique, monotonic and deterministically increased trusted counter value. Specifically, a log's entry consists of the encrypted data (e.g., KV pair), the assigned trusted counter value (encrypted) , and a cryptographic hash over both as in [39]. The recovery protocol replays the log's entries verifying the authenticity and the integrity of the entry by decrypting them and checking the entry's hash value over the calculated expected hash value. I also relies on the monotonic increments of the assigned trusted counter values to detect rollback attacks or verify freshness and state continuity. Precisely TREATY's recovery verifies that the state of the persistent storage and logged Txs is the most recent (through the verification of the logs) and recovers the most recent stable state.

Upon restart MANIFEST is replayed first; it recovers the SSTable hierarchy and loads metadata (hashes of SSTable's blocks) that will be used to verify the integrity and the freshness of a SSTable upon access into the enclave. Note that TREATY's garbage

[3]The Quoting Enclave (QE) is a *special* enclave on every SGX processor that enables remote attestation through Intel Attestation Service (IAS). It receives measurements (*reports*) from the application enclaves in the same machine, verifies them (local attestation) and signs them with the attestation key producing a *quote* that is verifiable by IAS.

collector only deletes SSTable files when the newly compacted ones refer to stabilized entries in MANIFEST. MANIFEST also recovers all the "live" WAL and Clog files. Similarly, TREATY makes sure that the old versions of the logs are not deleted before their effect to the database has been rollback protected (stabilized). For example, a WAL is marked for deletion as long as the matching MemTable has been successfully compacted and this compaction action refers to a stable entry in the MANIFEST. The Clog is deleted as long as there are no unstable entries and does not contain any unfinished prepared transaction entry.

After the MANIFEST, TREATY replays in order all live WALs to restore the latest MemTables. The WAL also contains the prepared Txs. Therefore, each node will also re-initialize all prepared Txs that are not yet committed. For each prepared Tx, the node communicates with the Tx's coordinator for either committing or aborting.

Lastly, Clog is replayed. TREATY restores the state of the 2PC protocol for all prepared on-going Txs. The coordinator will re-execute the prepare phase, if it cannot guarantee that the Tx will succeed. If the prepare phase decision is logged, then, thanks to the stabilization function of TREATY, these Txs are also prepared in the participant nodes. The coordinator will then instruct the participants to commit. If a node has already committed the Tx, this message is ignored.

Distributed rollback protection. For secure persistency, TREATY provides rollback protection across distributed Txs by leveraging a trusted counter service. While our design is independent of the trusted counter service, we adopt Rote [206], a fault-tolerant distributed system where enclaves preserve the counters freshness with 2 ms average latency.

For each log file, TREATY initializes a unique trusted counter and assigns a monotonically and deterministically increasing counter value to each log entry. TREATY's criterion for freshness is that 1) only log entries with counter value less than the trusted service's value can be recovered, 2) the counter values are deterministically increased—for *state continuity*, e.g., deleted or reordered entries are detected, and 3) last log entry's value match the counter's value.

TREATY accesses the trusted counter service through the network. The communication is asynchronous to avoid blocking and maximize throughput. As discussed in § 3.3 the 2PC incorporates the stabilization protocol ensuring distributed rollback protection—Txs are only considered committed (and clients get notified) after the commit decision has been stabilized in the logs.

In TREATY we use Rote [206] as our system's trusted counter service. In contrast to the single-node persistent trusted SGX counter [146], Rote offers trusted counters by implementing a distributed system of N SGX machines (*protection group*) where

the trusted counters values are kept in the secure enclave memory. Rote targets a threat model that is similar to the SGX threat model, thus to TREATY's threat model. Importantly, Rote overcomes the limitations of the SGX counters, at the cost, however, of extra machines. Assuming that up to f SGX machines can be compromised and up to u machines are unreachable or non-responsive at time of a counter's increment or access, Rote requires N to be at least equal to $f+2u+1$ for the system to proceed (and remain safe). Although Rote requires that at least the quorum (q), where q=$f+u+1$, of the SGX machines are up and running, it successfully overcomes the limitations of SGX counters as follows: firstly, it supports unlimited increments because it keeps the trusted counter values in the TEE secure memory, secondly, it improves latency because the executing distributed protocol (which we discuss next) outperforms the SGX hardware counters increments/writes latency.

Rote implements an *echo broadcast* [268] protocol with an extra confirmation message. A sender-enclave (SE) sends the counter update to all enclaves of the protection group. Receivers-enclaves (REs) send back to the SE an *echo*-message which they store along with the counter value in the protected memory. Once the SE receives echo-messages from the quorum (q $\geq N/2$) it starts a second round of echo-messages. Upon receiving back the echo, each RE verifies that the received counter value matches the one it keeps in-memory and RE replies with a (N)ACK message. After receiving q ACKs, the SE returns the incremented counter value to TREATY nodes.

Secure persistency guarantees. TREATY's attestation and its secure LSM-data structure [39] ensure that TREATY maintains its security properties after a crash as (1) only trusted nodes obtain the encryption keys for the persistent storage, (2) nodes perform integrity checks on accessed persistent data blocks and, (3) at recovery, TREATY verifies the logs' freshness. As the underlying cloud infrastructure is owned by a third-party, TREATY detects but cannot *prevent* unauthorized modifications to persistent state. If TREATY detects security violations in the persistent data, it stops operating as the system state is not recoverable. Consequently, clients lose permanently their access to TREATY's data.

Stabilization protocol correctness. TREATY stabilization protocol remains correct as TEEs guarantee its correct execution on all nodes. Any faults, e.g., crashes or network partitions, can only affect availability. While TREATY's trusted counter offers crash fault tolerance, Configuration and Attestation Service (CAS) can be a single point of failure. In case CAS fails, crashed nodes cannot recover whereas the alive nodes operate as usual.

3.5 Trusted Substrate for Distributed TXs

To support secure Tx processing, we design the following four cross-layer subsystems for our trusted substrate: a secure network library (§ 3.5.1), a secure storage engine for Txs based on Speicher [39] (§ 3.5.2), a userland thread scheduler (§ 3.5.3), and a memory allocator for Tx buffers (§ 3.5.4).

3.5.1 Network Library for Txs

To implement TREATY's 2PC, we build a secure networking library that implements asynchronous remote procedure calls (RPCs) for Txs execution. Our network library relies on eRPC [162], but we had to extend and adapt the codebase to *(i)* overcome the architectural limitations of TEEs (I/O, enclave memory and DMA-ed memory) and, *(ii)* ensure confidentiality, integrity and freshness between TREATY's nodes communication over the network in the presence of malicious attackers.

Architectural limitations of TEEs. To avoid the execution of expensive syscalls for network I/O, we adapt eRPC with DPDK as the transport layer. DPDK offers direct I/O, bypassing the kernel and eliminating the syscalls overheads using userspace drivers and polling.

To secure the software stack, we build eRPC/DPDK with SCONE assuring that the device's DMA mappings reside in the host memory, thus accessible by both enclave and NIC. We achieve this overwriting the `mmap()` of SCONE to bypass its shield layer and allow the allocation of untrusted host memory as well as the creation of memory mappings to the hugepages.

Furthermore, we change the library's memory allocator and we place all message buffers in the host memory (in hugepages of 2 MiB), thus reducing the EPC pressure at the cost of encrypting them. While eRPC by default creates shared memory regions for message buffers in hugepages, a naive port of eRPC with SCONE allocates all of these buffers inside the enclave triggering the costly EPC paging. Lastly, we eliminate `rdtsc()` calls to reduce the number of OCALLs from the hot path by replacing the call with a monotonic counter.

Message layout. TREATY's networking library constructs a secure message to guarantee the integrity and confidentiality of messages through a en-/decryption library based on OpenSSL [239]. Additionally, we ensure freshness, i.e., *at-most once* execution semantics for Txs' execution. The message is comprised of a 12 B Initialization Vector (IV), a payload of 4 B (for memory alignment), a 80 B Tx metadata and Tx data that contains the size of the data and the size of the key and/or value followed by the key and/or value. The message is followed by a 16 B MAC. MAC and IV are necessary to

prove the authenticity and integrity in the remote host. Only the metadata and data are encrypted; in case IV or MAC are compromised the integrity check will fail. The metadata contains the coordinator node's id (8 B) and the Tx id (64 B), monotonically incremented in the coordinator node. Both are necessary for uniquely identifying the transaction in the recipient side. The operation identifier (8 B) is also unique for each Tx request. This unique tuple of the node's, Tx and operation ids ensures that an operation/Tx is not executed more than once. Therefore, along with the two-phase locking which ensures that only one Tx can modify a resource, nodes can verify that no already executed Txs are processed again. Similarly, the participants' reply, except for the ACKs, also include the coordinator's node, Tx id and the operation id.

TREATY' networking protocol enqueues requests, e.g., a user-defined message, that triggers a request handler for this request type in the remote machine. The execution returns after enqueuing the request. The node can enqueue more requests or process received ones. Once the request is processed in the remote machine, the receiver replies back to the sender. A continuation function is triggered in the host machine to notify that the request has been completed. The sender can now deallocate any related resources, e.g., message buffers.

3.5.2 Storage Engine: Extensions to SPEICHER for Txs

To offer persistent Txs in TREATY, we extend SPEICHER's storage engine [39] to support single-node pessimistic and optimistic transactions as discussed in § 3.3.2.

Additionally, we implement an extra persistent log file, the Clog. Clog's entries are similar to MANIFEST and WAL entries format; they are comprised of a counter value, the encrypted Tx data and metadata and a cryptographic hash. Clog's deletions are also logged in the MANIFEST. Clog is thread-safe; coordinators append their entries independently and isolation is ensured with a mutex.

In TREATY, we allow group commits for Txs to flush bigger data blocks to the persistent storage and optimize the SSD throughput. The writer threads in each per-node storage engine that execute the transactions and write to the logs, elect a leader writer thread that merges their and all followers' Txs buffers into a larger buffer. The leader thread is elected based on a first-come-first-serve policy in a lock-free manner. The leader then writes this buffer into WAL and MemTable. This approach also improves system throughput because TREATY requests a trusted counter value for each batch of transactions rather than for each individual transaction. We further defer logging (yield) at commit, allowing us to format group commits of bigger data blocks. For the LSM structures, we implement a MemTable skip list that supports parallel updates for concurrent Tx processing.

Lastly, we change the I/O sub-system of SPEICHER, where we replace the SPDK-based direct I/O for accessing the SSDs with async syscalls to optimize the usage of cores for our eRPC/DPDK-based networking library.

3.5.3 Userland Scheduler

Timer based scheduling in the enclave is extremely expensive, as it involves interrupts that result in world switches. While SCONE implements its own userspace scheduler, it is non-preemptive, relying on threads to either go to sleep or issue syscalls for ensuring progress. This design is not well-suited for TREATY; *(i)* our direct I/O networking library leads to starvation and high latency, and *(ii)* in the presence of multiple clients creating too many threads is inefficient.

We overcome these by implementing a *userland scheduler* on top of SCONE's scheduler. Precisely, each thread spawns one userland thread (*fiber*) for each connected client. Our userland scheduler implements a per-core round-robin (RR) algorithm for fibers' scheduling and a set of queues (run queue and sleeping/waiting queue) for the fibers.

When a fiber needs to block, e.g., acquiring a lock, waiting on condition variables or sleeping, TREATY's userland scheduler places the fiber into a sleeping queue. It picks and schedules the next eligible fiber from the run queue (based on the RR algorithm). Our userland scheduler does not involve interrupts, syscalls and context/world switches when scheduling another fiber. Lastly, we adapted our scheduler to frequently yield threads allowing SCONE to schedule others. Precisely, if no fiber is in a running state, our scheduler sleeps; thereby invoking a syscall. Our scheduler's sleep function yields to another SCONE thread and increases the amount of time before future yields are triggered. In this way, fibers allow us to both maximize CPU utilization and increase scalability.

Our userland scheduler's implementation is based on Boost [48]. We configure SCONE with 8 kernel and 8 application threads each spawning one fiber per client.

3.5.4 Memory Management

We minimize EPC usage or paging; TREATY's in-memory data structures are divided between the enclave and untrusted host memory. All network buffers are kept in host memory at the cost of encryption. Note that transmission is asynchronous so heavy network traffic could exceed EPC limit and trigger paging if the message buffers were allocated in the enclave.

TREATY's engine keeps the updates of uncommitted in-progress Txs into local buffers.

We implement Txs' buffers as a stream of bytes (`std::string`) that allocate continuous memory to eliminate paging.We also explored the case to adopt a design similar to the MemTable for Txs buffers, where we keep only the keys in the enclave (for the read-my-own writes semantics). However, we decided against it as it does not offer any performance improvements; at commit, we still need to perform integrity checks, recollect and encrypt all the KV pairs in the enclave memory for logging. We implement a scalable memory allocator for host and enclave memory that relies on a mempool. It assigns threads to different heaps based on the hash of the `get_id()` and recycles unused memory, drastically reducing the amount of mapped memory.

Implementation details. We implement TREATY in C/C++; 4000 LoC for the 2PC, encryption library and modifications to eRPC, DPDK, boost and SPEICHER codebases. We use Java and Rust for the workload generator and CAS respectively.

3.6 Evaluation

3.6.1 Experimental Setup

Testbed. We perform our experiments on a real hardware testbed using a cluster of 6 server machines. We run TREATY on 3 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). TREATY nodes are connected over a 40GbE QSFP+ network switch. Clients generate workload on 3 machines and are connected with TREATY over a secondary 1Gb/s NIC.

Benchmarks/workloads. We evaluate TREATY's 2PC w/o any underlying storage (§ 3.6.2). For the distributed (§ 3.6.3) and single-node (§ 3.6.4) Txs evaluation, we use YCSB [349] and TPC-C [310]. We configure TPC-C with 10 Warehouses, as in [80]. For distributed Txs, we also run a TPC-C workload with 100 Warehouses. Lastly, we evaluate the network stack (§ 3.6.5) by stress-testing the network using: *(i)* iPerf [155] (implemented w/ kernel-sockets), and *(ii)* our own server/client application, build with eRPC [162], that implements iPerf. Unless stated otherwise, we refer to overheads for throughput (tps).

3.6.2 TREATY's 2PC Protocol

We evaluate TREATY's 2PC protocol designed over eRPC with the YCSB workload (50 %R-50 %W). TREATY's 2PC runs without any underlying storage to isolate the protocol's overheads. We compare two Secure (w/ SCONE) versions of TREATY 2PC with and w/o Enc(ryption) against two Native executions of the protocol with and w/o Enc(ryption)

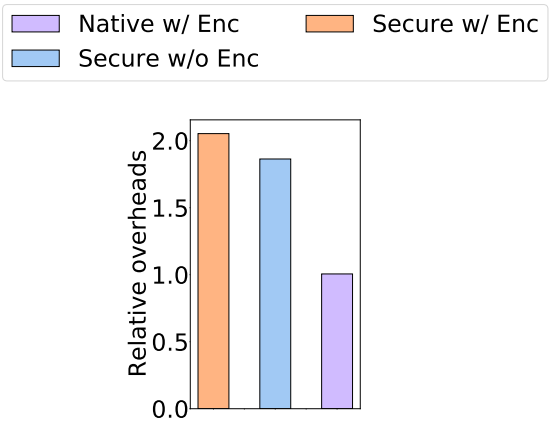

Figure 3.3: Throughput slowdown of three versions w.r.t. Native 2PC.

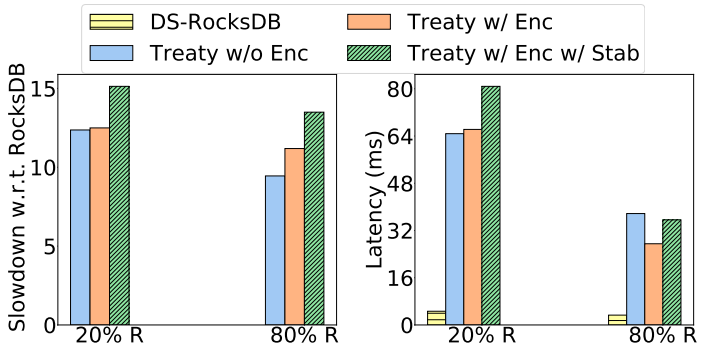

Figure 3.4: Performance evaluation of distributed Txs under a W-heavy (20 %R) and a R-heavy (80 %R) YCSB workload.

respectively. All four versions "saturate" with 300 clients, each of which executes a YCSB workload (10 Ops/Tx, 1000 B value size).

Figure 3.3 shows the slowdown in the throughput of 3 versions of TREATY's 2PC protocol (Native 2PC w/ Enc, Secure 2PC w/o Enc, Secure 2PC w/ Enc) normalized to a native, non-secure version of 2PC. Some Tx's operations might be served by the coordinator node; therefore not all operations are sent thought the network to participants and thus, be en-/decrypted. Our evaluation shows minimal encryption overhead in the native case. Further, TREATY's secure 2PC w/o Enc experiences 1.8× slowdown w.r.t. a native execution while encryption (Secure 2PC w/ Enc) increases the overheads leading to a 2× slowdown in comparison with native 2PC.

3.6.3 Distributed Transactions

Baselines and setup. We evaluate the performance of distributed Txs under two TPC-C workloads, with 10 (approx. 1 GB of data in the persistent storage at loading time) and

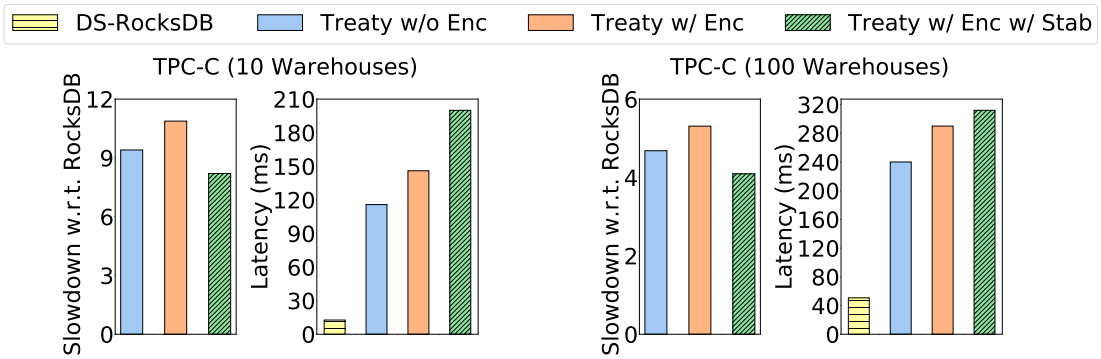

Figure 3.5: Performance evaluation of distributed transactions under two TPC-C workloads with 10 Warehouses and 100 Warehouses respectively.

100 Warehouses (approx. 9.8 GB of data in the persistent storage at loading time), and two YCSB workloads (Zipfian distribution and 10K distinct keys)[4]: read-heavy (80 %R) and write-heavy (20 %R). We show the overheads of TREATY's throughput normalized w.r.t. a native execution of 2PC with RocksDB as the underlying storage (DS-RocksDB). We study the performance behavior of three systems: *(i)* TREATY w/o Enc, *(ii)* TREATY w/ Enc and *(iii)* TREATY w/ Stab(ility) w/ Enc. All three versions run with SCONE and our TREATY's secure storage system.

Results. YCSB. Figure 3.4 (left) shows the throughput slowdown of the three systems with reference to DS-RocksDB. TREATY's performance is 9×—15× worse compared to DS-RocksDB where SCONE overheads fast dominate the performance (TREATY runs w/ and w/o Enc have little differences). For the W-heavy workload, DS-RocksDB achieves 18.5 ktps. All four systems are saturated with 96 clients equally divided across all three machines (each serving 32 clients). Distributed Txs require both participants and coordinator to stabilize their entries and therefore, TREATY rollback protection increases latency further for write-heavy Txs, as shown in Figure 3.4 (right).

For the R-heavy workload, TREATY w/ Enc slows down the execution 11× while the un-encrypted version of the system shows a slowdown of 9.5×, both compared to native DS-RocksDB that achieves 24 ktps. Encryption overheads are reasonable; reading from SSTables requires integrity checks as well as proving the freshness of the entry. All four systems present different scaling capabilities. DS-RocksDB and TREATY w/o Enc scale up to 92 clients while encrypted versions cannot scale more than 60 clients. Therefore, TREATY is over saturated in the benchmark, explaining the higher latency values.

TPC-C (10W). Figure 3.5 (left) shows the throughput overheads and the latencies

[4]TREATY and RocksDB do not support in-place updates, each update appends a new entry in the MemTable. As such the dataset size of the system depends on the experiment duration rather than the initial loaded data in the system. In addition, the dataset size is also affected by the background compaction process which removes duplicate and stale KV pairs. We ran all experiments for 90 seconds; 15 seconds warm-up/warm-down period and 60 seconds runtime.

of three versions of TREATY (all run in SCONE) w.r.t DS-RocksDB under TPC-C with 10 Warehouses. TREATY is 8×—11× slower compared to the native, non-secure DS-RocksDB. This configuration presents heavy W-W conflicts; DS-RocksDB achieves 780 tps. Consequently, DS-RocksDB, TREATY w/o Enc and TREATY w/ Enc cannot scale for more than 10 clients. However, TREATY w/ Enc w/ Stab scales up to 16 clients as the stabilization period (where locks are released) allows the system to serve more requests.

TPC-C (100W). Figure 3.5 (right) shows the throughput overheads and the latencies of three versions of TREATY (all run w/ SCONE) w.r.t DS-RocksDB under TPC-C with 100 Warehouses (total dataset size equals to 10GB divided equally to all 3 nodes). This configuration presents less conflicts than the previous case; DS-RocksDB achieves 1200 tps. Our evaluation shows reasonable overheads (4×-6×) and similar behavior for TREATY w/ Enc and Stab; while all the three other systems (DS-RocksDB, TREATY w/ Enc, TREATY w/o Enc) are saturated with 60 clients, TREATY w/ Enc w/ Stab is saturated with 84 clients.

3.6.4 Single-node Transactions

Baselines and setup. We evaluate the performance of TREATY's pessimistic and optimistic single-node Txs with TPC-C and YCSB. TPC-C is configured with 10 Warehouses as in [80] and YCSB with: 10 ops/Tx, value size to be equal to 1000 B, uniform distribution with 10 k unique keys. For the pessimistic Txs, we measure the performance against read-heavy (80 %R-20 %W) and write-heavy (20 %R-80 %W) workloads, while for the optimistic Txs we use the read-heavy workload. Our experiments stress-test EPC usage since both TREATY and RocksDB do not support in-place updates. We evaluate the throughput (tps) and latency for 6 versions of the single-node TREATY; *(i)* RocksDB, *(ii)* Native TREATY, *(iii)* Native TREATY w/ Enc, *(iv)* TREATY w/o Enc (SCONE), *(v)* TREATY w/ Enc (SCONE) and *(vi)* TREATY w/ Enc w/ Stab (SCONE).

Results. Pessimistic Txs. Figure 3.6 shows the throughput and latency of the TPC-C for the pessimistic Txs. TREATY executed natively (Native TREATY) performs equivalently to RocksDB. Additionally, we deduce that Native TREATY w/ Enc adds minimal overhead compared to the non-encrypted versions. Further, SCONE's overheads are reasonable. TREATY w/o Enc has roughly 1.6× slowdown compared to RocksDB while TREATY w/ Enc has 2× slowdown. Lastly, the stabilization period seems not to have great impact on the overall throughput. We experience a 2.1× slowdown compared to RocksDB. Regarding the latency, we see that all TREATY SCONE versions do not scale as good as the native execution. However, the latency of SCONE systems is equivalent or smaller to the natively executed versions. This behavior is reasonable since the native

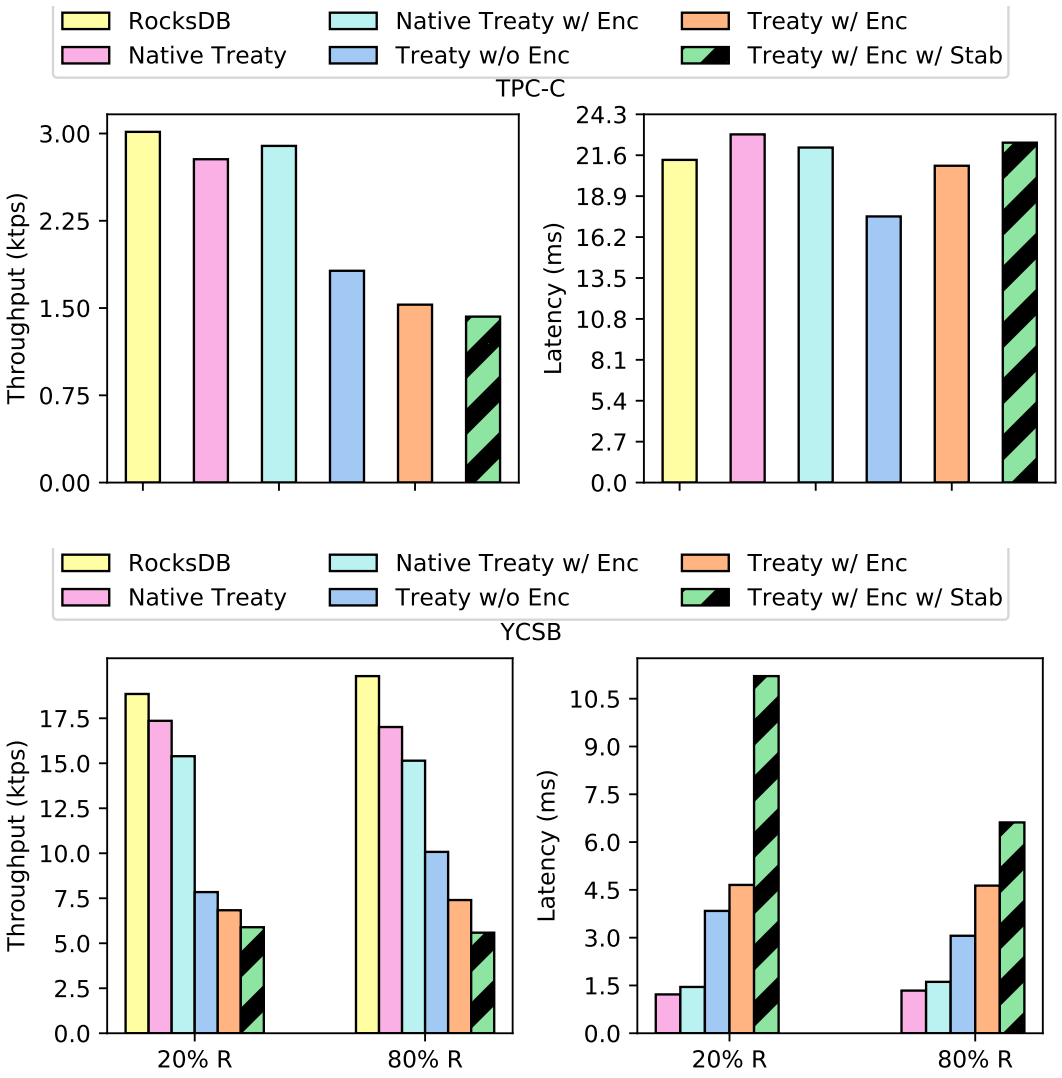

Figure 3.6: Performance evaluation of pessimistic single-node transactions under TPC-C and YCSB benchmarks. YCSB performance is evaluated with a write heavy (20 % reads) and a read heavy (80 % reads) workload.

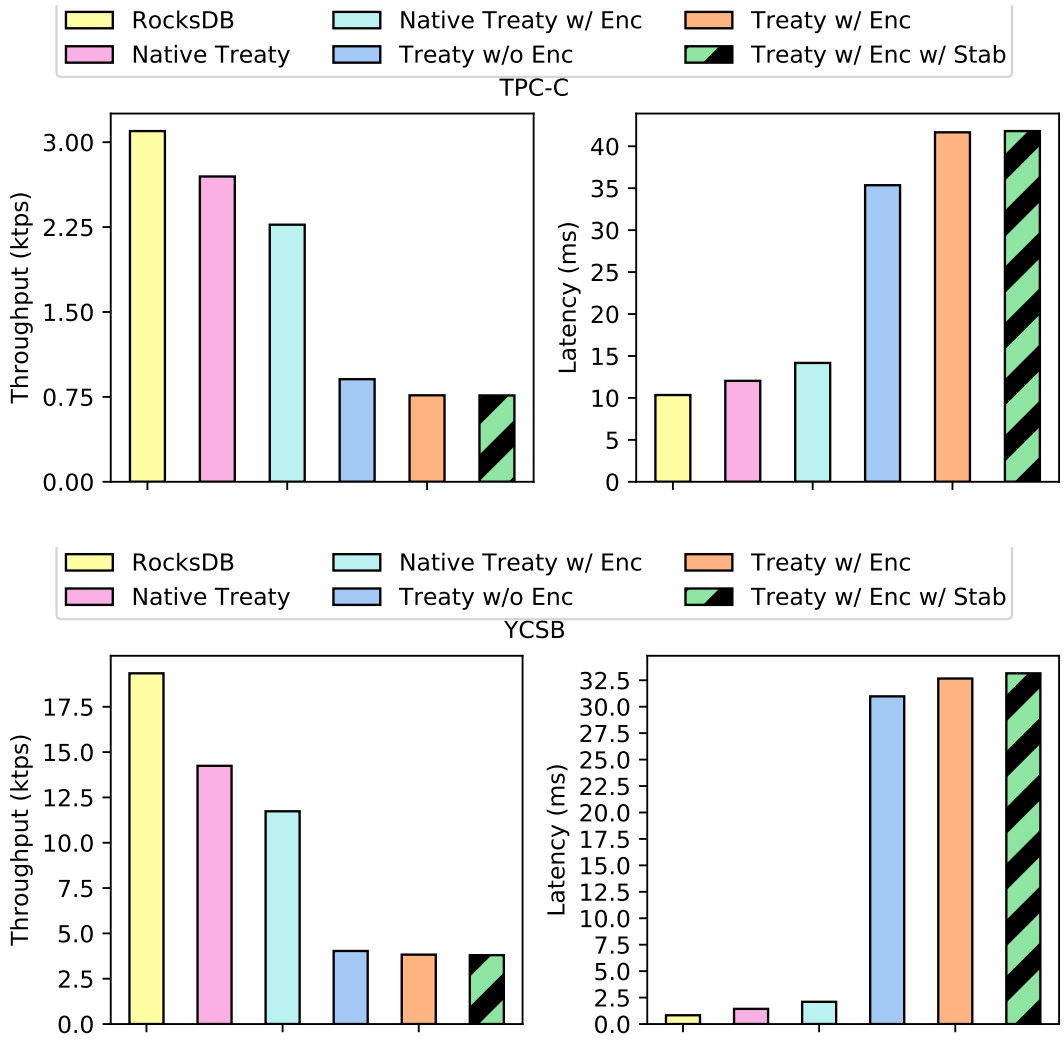

Figure 3.7: Performance evaluation of optimistic single-node transaction under TPC-C and YCSB benchmarks. YCSB performance is evaluated with a read heavy (80 % reads) workload.

versions are "saturated" to 64 clients while the SCONE versions to 32 clients.

Additionally, Figure 3.6 shows the throughput and latency of all 6 systems for the two YCSB workloads. YCSB's configuration, in contrast to TPC-C, present little conflicts. That said, for the read-heavy workload, encryption, adds a throughput overhead of 1.3× and 2.7× compared to native and SCONE versions respectively while the respective overheads for latency are 1.6× and 4.6×. For the write-heavy workload, we have 1.2× and 2.8× slowdown to native and SCONE versions compared with RocksDB. The latency overheads are 1.5× and 4.7× respectively. Similarly to TPC-C, TREATY's stabilization function does not impact performance dramatically. We experience 3.5× slowdown for the read-heavy workload and 3.2× slowdown with respect to RocksDB for the write-heavy workload compared to when de-activating the stabilization mechanism. Further, especially for the read-heavy workload, we find out that TREATY w/ Enc w/ Stab takes advantage of the "idle" (stabilization) time to improve the scalability; TREATY w/ Enc w/ Stab becomes saturated in 64 clients while the other versions are saturated in 32 clients.

Optimistic Txs. Figure 3.7 shows that TREATY w/ Enc w/ Stab performs 5× and 4× worse compared to the native RocksDB for TPC-C and YCSB, respectively. We see that TREATY's stabilization does not incur extra throughput overhead compared to the TREATY w/ Enc as the system, thanks to our userspace fiber scheduler, continues to process requests. TREATY w/ Enc w/ Stab's compared to TREATY w/ Enc experiences roughly 10 % latency overhead. Further, we notice that TREATY w/ Enc w/ Stab's saturation point under YCSB is 128 clients while RocksDB's one is 32. TREATY shows similar overheads as SPEICHER [39] which is the most related system.

3.6.5 Network Library for Txs

We evaluate the performance of TREATY's networking library using iPerf against six competitive baselines: eRPC (SCONE), eRPC (native), iPerf-UDP (native), iPerf-UDP (SCONE), iPerf-TCP (native), and iPerf-TCP (SCONE). All native (eRPC and iPerf) versions do not provide any security. Additionally, SCONE (eRPC and iPerf) versions do not secure network layer; we only use the secure message format for TREATY-networking. Note that iPerf build with SCONE is optimized w.r.t to SGX since SCONE uses the async syscalls [293] for performance.

For the sockets (native and SCONE), we use iPerf to measure the throughput. For the eRPC versions and TREATY-networking, we implement a client-server model with eRPC to implement iPerf. Our experiments saturate network bandwidth where we compare the performance with different packet sizes. iPerf supports TCP and UDP, eRPC supports only UDP.

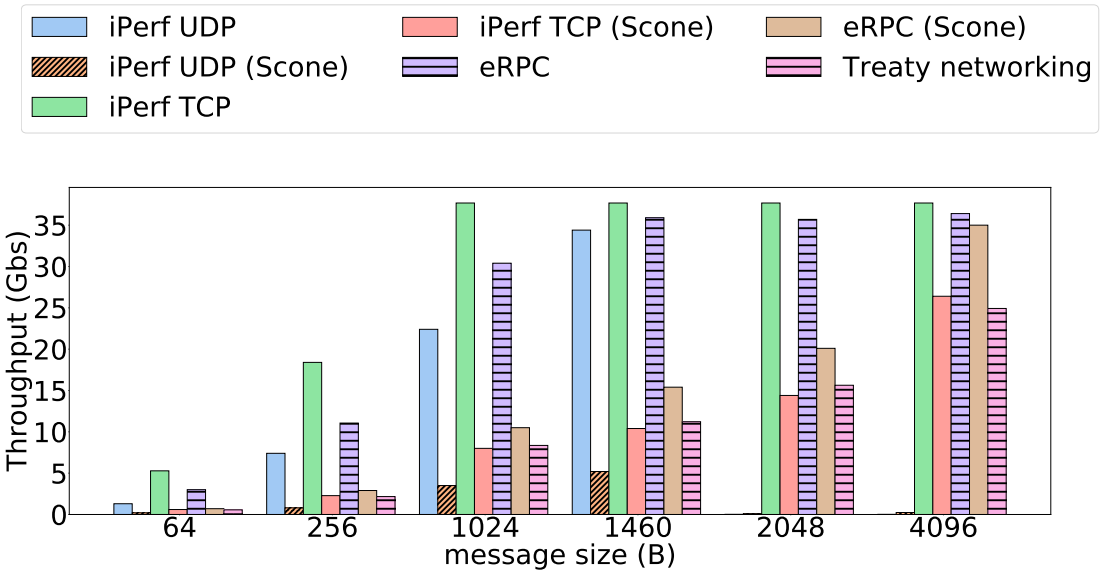

Figure 3.8: Throughput in network bandwidth of TREATY-networking, eRPC (native and SCONE), iPerf-TCP and iPerf-UDP (native and SCONE).

Figure 3.8 shows the throughput in network bandwidth for all seven systems discussed (TREATY networking, eRPC (SCONE), eRPC (native), iPerf-UDP (native), iPerf-UDP (SCONE), iPerf-TCP (native), and iPerf-TCP (SCONE)). We see that eRPC is comparable to iPerf-TCP while iPerf-UDP performs poorly. Especially for large messages ($>$ MTU), UDP throughput equals zero as many messages are dropped. In contrast to UDP, TCP performs equivalently and better than eRPC. We deduct this to the fact that TCP is optimized for high speed bulk transfers and, additionally, the entire TCP/IP stack processing is frequently offloaded to the network controller. For small and medium packets sizes that are still smaller than the MTU (1460 B), we observe performance differences between eRPC and iPerf-TCP. Especially, for packet sizes of 256 B and 1024 B, eRPC shows roughly 30 % and 22 % slowdown respectively compared to iPerf-TCP. For larger messages, both eRPC and iPerf-TCP perform almost equivalently.

Our exaluation shows the following: (a) SCONE's overhead is significant—SCONE deteriorates up to 8$\times$ for iPerf-TCP (SCONE) while up to 4$\times$ for eRPC (SCONE); and (b), due to the amount of kernel syscalls, eRPC in SCONE performs up to 1.5$\times$ faster than iPerf-TCP (SCONE). As discussed, syscalls execution in the enclave incurs heavy overheads. Note the smallest the packet size is, the worse the performance becomes. Lastly, we see that TREATY network stack which also fully secures the network and includes the encryption overheads performs equivalently to iPerf-TCP (SCONE) that do not provide any security. As a result, iPerf-TCP (SCONE) is an inappropriate design.

Version	Slowdown	Version	Slowdown
TREATY w/o Enc	1.5×	TREATY	2.0×

Table 3.1: Recovery overheads w.r.t. native recovery.

3.6.6 Recovery Protocol

We next evaluate the overheads of TREATY recovery w/ and w/o Enc compared with native recovery. We construct logs of 800K entries each that lead to log sizes of 69 MiB and 91 MiB for the non-encrypted and encrypted entries respectively. In this experiment we use relatively small log entries (e.g 100 B per log entry) which is the worse case for TREATY as: *(i)* we have more syscalls, and *(ii)* we have more decryption calls.

Table 3.1 shows that TREATY recovery without decryption costs incurs roughly 1.5× slowdown compared to the native recovery. Further, encryption increases the overheads by up to 2× slower than the native recovery.

3.7 Related Work

Confidential computing frameworks [26, 43, 315, 118] leverage TEEs to build secure systems [173, 39, 278, 257, 312, 314, 313, 259, 37]. TREATY leverages SCONE to build the first secure distributed transactional KV storage system with TEEs that offers integrity, confidentiality and data freshness.

Secure stores with TEEs. Secure systems for cloud computing [247, 253, 80, 316, 257, 95, 22, 352, 40, 205, 214, 38] offer different security properties, interfaces, threat model, and security enforcement mechanisms. EnclaveDB [257] is the most related work. In contrast to TREATY, it (1) is a single-node in-memory system (w/o persistence and distribution), (2) runs in emulated h/w and, (3) assumes unlimited enclaves. TREATY targets a distributed storage system, where we extend the security properties to storage and network and overcome the limitations of TEEs. ShieldStore [170] builds a secure in-memory KV store, providing similar APIs as SPEICHER, using Intel SGX. In contrast to SPEICHER, thus TREATY, ShieldStore offers weak persistency allowing for rollback attacks; they conduct periodic snapshots that are stored persistently along with a hardware monotonic counter (for rollback attacks protection). However, ShieldStore comes with an *untrusted* time window where rollback attacks are possible on the very latest state. Specifically, any updates after the last snapshot are lost, if a crash occurs. Precursor [214] combines SGX with RDMA offloading the cryptographic operations to clients. In contrast, TREATY provides distribution, persistency and Txs.

Secure databases and storage systems. Encrypted databases, such as CryptDB [253],

Seabed [247], Monomi [316] and DJoin [229], focus on ensuring confidential computations in untrusted environments. TREATY, on the other hand, guarantees confidentiality, integrity, and freshness for both the computation, i.e., transactions and the stored data. TDB [205] presents a secure database on untrusted storage by designing a log-structured data store for confidentiality, integrity, and freshness. In contrast to TREATY that implements a hardware-protected trusted computing base (TCB) with Intel SGX, TBD assumes a TCB overseeing many practical challenges arisen in a real-world system.

Obladi [80] supports transactions focusing on hiding access patterns against the cloud provider and incurs performance overheads that are similar to TREATY, i.e., 5×—12×, compared to its non-secure version. Since TREATY provides confidentiality, information about the transactions and the requested data cannot be leaked to the cloud provider.

Other storage systems vary on hardware, security guarantees and interfaces: KV APIs [39, 170, 173] and filesystems [105, 338, 320].

Secure distributed storage systems. Secure distributed storage systems [204, 333, 252] provide consistency, availability, durability and integrity. CloudProof [252], similarly to TREATY, distrusts the cloud provider but it requires (1) clients to guarantee these security properties and (2) a trusted proxy which limits scalability. TREATY leverages TEEs to avoid such limitations. Opaque [352] supports oblivious operators for queries that scan or shuffle full tables and relies on Intel SGX to run a query optimizer. Depot [204] sustains failures in terms of durability, consistency, availability, and integrity. Salus [333] designs a block store robust storage system guaranteeing data integrity in the presence of commission failures. TREATY, on the other hand, further provides confidentiality.

I/O for shielded execution. Other distributed systems [164, 284, 337, 88] deploy RDMA as TREATY. However, we target security which is more challenging; DMA connections for direct I/O are not allowed by TEEs. ShieldBox [313] uses DPDK to overcome this limitation, but it targets only layer 2 in the OSI model which is limiting for distributed systems. SPEICHER [39] uses SPDK [154] for direct I/O to the SSDs. rkt-io [307] provides a library OS in the enclave including a full network stack. We build on these advancements to build a secure direct network I/O mechanism for TEEs with which we design a 2PC protocol.

BFT-based system design without TEEs. TREATY guarantees freshness building on a trusted counter service, Rote [206], that implements fault-tolerant trusted counters even when some of the involved TEEs are compromised (Byzantine). Alternative approaches that do not make use of TEEs could involve the adoption of Byzantine Fault Tolerant (BFT) consensus protocols, such as PBFT [57] and those discussed in the next

Chapter 4, to design fault-tolerant, monotonically increased trusted counters for guaranteeing freshness. Further, for offering confidentiality without TEEs, this alternative system could store the data encrypted in an untrusted KV store.

Such a hypothetical design can provide strong freshness guarantees and confidentiality for the data. However, we argue that leveraging TEEs offers important advantages both in performance and in the security properties of the system. First, by leveraging TEEs, TREATY's counter, Rote, downgrades its replication degree compared to a classical BFT protocol [206] which translates to less required machines and potentially higher system throughput. The next Chapter 4 elaborates more on how to downgrade the replication degree of robust fault-tolerant systems using TEEs in the untrusted cloud.

Secondly, by encrypting the data in the untrusted KV store, the system cannot ensure confidentiality for the operations which might reveal the access patterns on confidential or sensitive data (e.g., banking, healthcare). In addition, since the operations execution is unprotected, the system cannot guarantee integrity flow, i.e., the operations or transactions are executed correctly. In contrast, TREATY on top of TEEs is capable of offering strong confidentiality for all the operations, the transaction protocol and the accessed and stored data. As such, private information or the access patterns cannot be leaked.

Thirdly, for the operations execution, this hypothetical system have two options: (1) first, use homomorphic encryption (HE) and (2) secondly, allowing access to the KV encryption key for decrypting and iterating through the stored entries, e.g., keys. The first approach is limited because HE cannot easily be adapted for transactions. The second approach makes the system *untrusted* since the encryption key can be leaked by the cloud provider. In contrast, the use of TEEs in TREATY guarantees that the storage encryption key remains secure and thus data are accessible only by the current TCB while we are able to support a wide range of operations (e.g., transactions, queries) that due to shielded TEEs' execution will be executed correctly.

Lastly, the use of TEEs in TREATY allows the system to verify and detect integrity and freshness violations by executing all the checks securely within the trusted hardware. In the hypothetical design, such checks could only be conducted in the (trusted) clients' side. While such client-centric designs [79, 252, 172] have been introduced in various domains (database isolation, security enforcement and BFT consensus, etc.), in our hypothetical system, recovery and validation would require clients to retrieve and recover and validate the entire state of the database which is inefficient. Overall, we believe that the use of TREATY presents a *transparent* and *clean* design for offering security.

3.8 Summary

In this chapter, we present TREATY, a secure distributed transactional KV store for untrusted cloud environments. TREATY offers high-performance serializable Txs with strong security properties.We achieve these design goals by building on hardware-assisted secure Txs with SGX and designing a distributed 2PC protocol with a direct I/O network library based on eRPC. Further, we design a stabilization protocol for Txs using an asynchronous trusted counter interface along with a distributed attestation service. We implement an end-to-end secure Tx processing system from the ground-up based on RocksDB/SPEICHER as the underlying storage engine. Our evaluation with the YCSB and TPC-C shows reasonable overheads for TREATY, while it provides strong security properties.

Chapter 4

RECIPE: A Hardware-Accelerated RECIPE For Designing Byzantine Fault Tolerant Replication Protocols

In the previous chapter, we discussed TREATY which implements a secure distributed transactional KV storage system. Unfortunately, TREATY cannot continue operating when failures occur as the key space of the failed nodes cannot be accessed. To address this, distributed systems employ Crash Fault Tolerant (CFT) *replication protocols* [236, 269, 266, 12, 201, 181, 168] to maintain a consistent view of the datasets guaranteeing fault tolerance, i.e., reliability and availability in the presence of failures [353, 82, 179, 265, 271, 186, 103]. However, CFT protocols cannot tolerate arbitrary (Byzantine) failures that occur in the modern untrusted cloud environments.

To this end, we propose RECIPE, a generic approach to transform *existing* Crash-Fault Tolerant (CFT) protocols to tolerate Byzantine failures in untrusted cloud environments. RECIPE leverages the advances in trusted hardware (e.g., trusted execution environments) and direct network I/O to guarantee non-equivocation and transferable authentication, i.e., the ability to establish trust among the distributed nodes over the untrusted network and infrastructure in the presence of Byzantine actors (for example, a node can verify who is the original sender of a message even this message is forwarded from a node other than the original sender). At the same time, RECIPE aims to offer performance and resource overheads on par with CFT protocols. Importantly, RECIPE's APIs are generic and can easily be adapted to existing codebases—we have transformed a range of leader-/leaderless-based CFT protocols enforcing different (e.g., total order/per-key) ordering semantics. Our evaluation based on the transformation of four CFT protocols (Raft, ABD, Chain Replication, and AllConcur) against the state-of-the-art Byzantine-Fault Tolerant (BFT) protocols shows that RECIPE can in-

crease throughput up to 5.9×—24×, while requiring fewer replicas, i.e., $2f+1$ replicas instead of $3f+1$ replicas to tolerate f faults. Lastly, we provide a correctness analysis for safety and liveness properties of our transformation of CFT protocols operating in Byzantine settings.

4.1 Motivation

Distributed data stores are widely used in online cloud services [286, 327, 353, 111, 230, 55, 132]. For performance and fault tolerance requirements, distributed data stores employ a *replication protocol*. As such, they maintain a consistent view of the datasets while improving the performance for read-heavy workloads and, more importantly, guaranteeing fault tolerance, *i.e.,* reliability and availability in the presence of failures [353, 82, 179, 265, 271, 186, 103].

For handling failures, distributed data stores predominately employ Crash Fault Tolerant (CFT) replication protocols [236, 269, 306, 266, 12, 201, 181, 168] to provide consistent replication assuming a *fail-stop fault model*, i.e., replicas are honest and can only fail by crashing [85]. Unfortunately, CFT protocols are *inadequate* for modern untrusted cloud environments, where the underlying cloud infrastructure can be compromised by an adversary, e.g., co-located tenants or even a misbehaving cloud operator that may eavesdrop or actively influence the replicas' behavior. In such an untrusted environment, the surface of faults and attacks expands beyond the CFT fail-stop model, ranging from software bugs and configuration errors to malicious attacks [122, 287, 127]. CFT protocols are fundamentally incapable of providing consistent replication in the presence of non-benign faults in untrusted cloud environments.

To overcome the limitations of CFT protocols, Byzantine Fault Tolerant (BFT) protocols [182] offer important foundations for developing distributed data stores with stronger guarantees in the presence of *Byzantine failures*, i.e., nodes can fail in arbitrary ways. While BFT protocols can tolerate Byzantine failures, *including malicious adversaries*, they are unfortunately *not adopted in practice* because of their performance and replication resource overheads, and implementation complexity [254]. For instance, BFT protocols require at least $3f+1$ replicas [57, 303] instead of the $2f+1$ replicas required by CFT protocols for tolerating f faults. BFT protocols also require additional communication round-trip rounds (at least three execution phases, e.g., `Pre-Prepare`, `Prepare` and `Commit` in PBFT [57], instead of two phases in the CFT model [106]); thus incurring high latency and reduced read/write throughput. Lastly, BFT protocols are extremely complex and hard to understand, let alone to implement correctly and optimize [5]: even intuitive algorithmic optimizations can strongly affect other parts

of the protocol [30].

The "CFT vs. BFT" conundrum creates a fundamental design trade-off between the *efficiency of CFT protocols* for practical deployments and the *robustness of BFT protocols* for Byzantine settings of modern cloud environments. To strike a balance, our work targets the following research question: *How can we design robust and efficient replication protocols for distributed data stores hosted in untrusted cloud environments?*

The key idea. Our work leverages modern cloud hardware for transforming existing CFT protocols for Byzantine settings in untrusted cloud environments. Specifically, our transformation underpins the robustness and efficiency axes. For *robustness*, we leverage trusted hardware available as part of confidential cloud computing to harden the security properties of CFT protocols [148, 184, 15, 149]. In particular, we leverage trusted execution environments (TEEs) which are capable of providing two key properties necessary for successfully transforming a CFT protocol to operate in Byzantine settings [67]: (a) transferable authentication, i.e., the ability to establish trust in nodes in distributed settings by designing a remote attestation protocol, and (b) non-equivocation, i.e., once the trust is established in a node via the remote attestation protocol, the node follows the CFT replication protocol faithfully, and therefore, it cannot send conflicting statements to other nodes.

For *efficiency*, we leverage the modern networking hardware, such as RDMA/DPDK for kernel-bypass, to design a highly optimized communication protocol for replicating the state across nodes in distributed settings [209, 150, 162], while overcoming the I/O bottlenecks in trusted computing [307].

Our proposal: RECIPE. RECIPE leverages TEEs along with direct I/O to resolve the tension between security and performance by building an efficient and practical transformation of unmodified CFT replication protocols for Byzantine settings. RECIPE achieves this by implementing a distributed trusted computing base (TCB) that shields the replication protocol execution and *extends* the security properties offered by a single TEE (whose security properties are only effective in a single-node setup) to a distributed setting of TEEs. Our design is comprised of a transferable authentication phase (§ 4.4.3) for distributed trust establishment, a highly performant network stack for secure communication over the untrusted network (§ 4.5.1) and a memory-efficient KV store (§ 4.5.3).

We realise RECIPE approach as a generic library, RECIPE-lib (§ 4.5), on top of Intel SGX and the SCONE framework [26]. We carefully build RECIPE's high-performance network stack for TEEs, extending eRPC [162] to enable direct I/O, essentially DMA operations, within the protected TEE domain while shielding the exchanged messages (i.e., calculating and sending the message along with its MAC) to maintain authenticity

and non-equivocation across the network. Lastly, our lock-free KV store is designed for the limited trusted area in TEEs [39, 170].

Our evaluation assesses RECIPE's generality and efficiency. Specifically, to show the generality of our approach, we apply and evaluate RECIPE on real hardware with four well-known CFT protocols (from now on, an 'R-' prefix stands for the transformed protocol); a decentralized (leaderless) linearizable multi-writer multi-reader protocol (ABD) [201] (R-ABD), two leader-based protocols with linearizable reads, Raft [236] (R-Raft) and Chain Replication (CR) [269] (R-CR), and AllConcur [251] (R-AllConcur), a decentralized consensus protocol with consistent local reads. To evaluate performance, we compare RECIPE protocols with two competitive BFT replication protocols, BFT-smart [295], the state-of-the-art implementation of PBFT [58] that has been adopted in industry [96], and Damysus [83] a state-of-the-art BFT replication protocol that uses TEEs for reducing its replication degree. Our evaluation shows that RECIPE achieves up to $24\times$ and $5.9\times$ better throughput w.r.t. PBFT and Damysus, respectively, while improving scalability—RECIPE requires $2f+1$ replicas, f fewer replicas compared to PBFT ($3f+1$). We further show that RECIPE can offer confidentiality—a security property not provided by traditional BFT protocols—while achieving a speedup of $7\times$—$13\times$ w.r.t. PBFT and up to $4.9\times$ w.r.t. Damysus.

4.2 System Model

Model sketch. We model the distributed system as a set of N TEEs in N nodes (or replicas) that host either *follower* or *coordinator* processes which execute a CFT protocol and communicate by exchanging messages. We assume that RECIPE's nodes run in a third-party untrusted cloud infrastructure. A coordinator serves client requests by initiating the implemented CFT replication protocol. Upon completion, it replies back to clients. In leaderless protocols, coordinators are selected randomly (any node can act as a follower and/or a coordinator). In leader-based protocols, only the active leader can act as a coordinator, the rest of the nodes are followers.

Communication model. Nodes communicate via a fully-connected, bidirectional, point-to-point and unreliable message-passing network, where messages can be arbitrarily delayed, re-ordered or dropped. In line with previous BFT protocols, we adopt the partial synchrony model [92], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all communications arrive within time Δ.

Fault and threat model. We say that a node is *faulty* if it exhibits Byzantine behavior [182]. The unprotected (*out-of-the-TEE*) infrastructure (e.g., host memory, OS, NIC,

network infrastructure/adversaries) can exhibit Byzantine behavior while we assume that the TEEs can only crash-fail. We say that a node is faulty if one of the following holds true: *(i)* the TEE fails by crashing or *(ii)* the unprotected infrastructure is Byzantine. For liveness we assume that for $N \geq (2f+1)$ nodes up to f can be *faulty*. While the cloud provider and the cloud infrastructure can be Byzantine, we assume that the CPU Manufacturer, hence the provided TEEs hardware, as well as the protocol designer and the codebases are trusted.

4.3 RECIPE Overview

4.3.1 Architecture Overview

Distributed data stores architecture. Figure 4.1 shows the overview of a distributed data store that builds on top of the RECIPE system. Distributed data stores implement a tiered architecture consisting of a *distributed data store layer*, *replication layer*, and *data layer*. In our case, the replication and data layers are provided by RECIPE. The distributed data store layer maintains a routing table that matches the keyspace with the owners' nodes. This layer is responsible for forwarding client requests to the appropriate coordinator nodes (e.g., leader of the replication protocol) for execution. The RECIPE replication layer is responsible for consistently replicating the data by executing the implemented protocol. After the protocol execution, RECIPE nodes store the data in their KV stores (data layer), and they reply to the client [265, 271, 186, 103].

RECIPE architecture. RECIPE design is based on a distributed setting of TEEs that implement a (distributed) trusted computing base (TCB) and shield the execution of unmodified CFT protocols against Byzantine failures. RECIPE's TCB contains the CFT protocol's code along with some metadata specific to the protocol. The code and TEEs of all replicas are attested before instantiating the protocol to ensure that the TEE hardware and the residing code are genuine. All authenticated replicas receive secrets (e.g., signing or encryption keys) and configuration data securely at initialization.

Further, RECIPE builds a *direct I/O layer* comprised of a networking library for low-latency communication between nodes (§ 4.5.1). The library bypasses the kernel stack for performance and shields the communication to guarantee non-equivocation and transferable authentication against Byzantine actors in the network. RECIPE guarantees both properties by layering the non-equivocation and authentication layers on top of the direct I/O layer. In addition, to strengthen RECIPE's security properties and eliminate syscalls, we map the network library software stack to the TEE's address space.

Lastly, RECIPE builds the *data layer* on top of local KV store instances. Our design of the KV store increases the trust to individual nodes, allowing for local reads (§ 4.5.3).

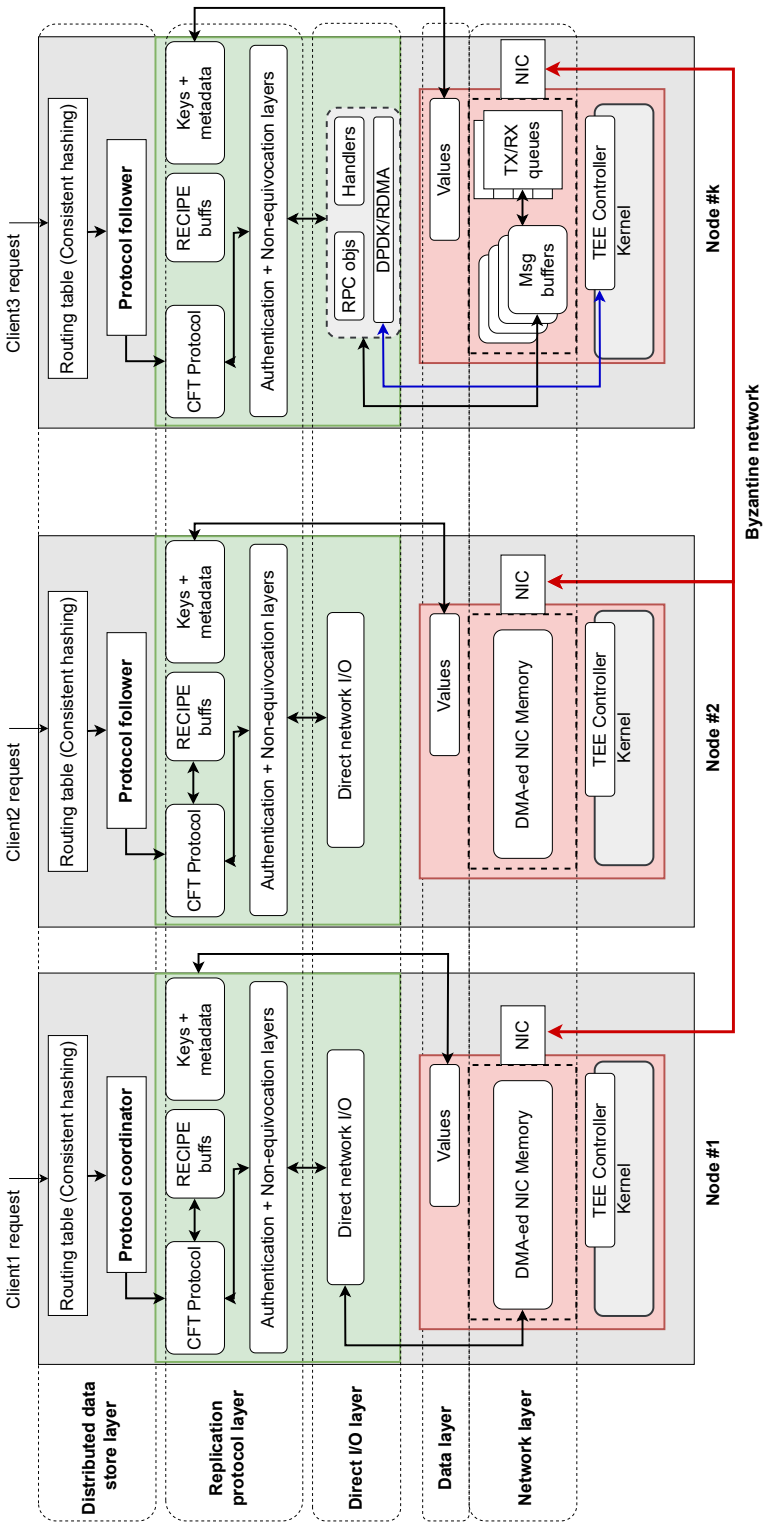

Figure 4.1: RECIPE's system architecture.

Our KV store achieves two goals; first, we guarantee trust to individual replicas to serve reads locally, and secondly, we limit the TCB size, optimizing the enclave memory

usage. As shown in Figure 4.1, RECIPE keeps bulk data (values) in the host memory and stores only minimal data (keys + metadata) in the TEE area. The metadata, e.g., hash of the value, key's timestamps or versions, a a pointer to the value memory address etc., are kept along with keys in the TEE for integrity verification.

4.3.2 Transformation Requirements: CFT to BFT

The basic requirements for transforming a CFT protocol for Byzantine environments are established in a theoretical result published by Clement et al. in PODC 2012 [67]. This seminal paper shows that non-equivocation and transferable authentication are necessary to go from $3f+1$ to $2f+1$ replicas for a reliable broadcast in Byzantine settings. Our work, inspired by this paper, shows that not only can this lower bound be achieved in practice, we can do so while providing high performance by leveraging modern hardware in a cloud environment. In fact, TEEs offer stronger security than these two key properties as their running software is *shielded* from Byzantine actors that might try to tamper with the execution. Next we discuss how RECIPE satisfies these two fundamental requirements, while § 4.3.3 elaborates on how to design *practical* and *efficient* protocols that meet these requirements.

Property 1: The transferable authentication property refers to the authenticity of a received message, requiring that a replica must be able to verify that the supposed sender indeed had sent the message. The authentication is transferable if the original sender can be verified even for forwarded messages.

Mechanism 1: The seminal paper by Clement et al. [67] suggests the use of digital signatures to ensure the transferable authentication property. In contrast, RECIPE employs more performant cryptographic primitives such as MAC for shielding the network messages in combination with an attestation protocol an attestation protocol. The cryptographic primitives ensure that nodes can generate and validate authenticated messages while RECIPE's attestation protocol (§ 4.4.3) ensures that only trusted replicas access the cryptographic keys and execute the protocol.

Property 2: The non-equivocation property guarantees that replicas cannot *execute* conflicting requests for the same round of execution. That implies that RECIPE must detect attacks where adversaries try to compromise the protocol by sending invalid requests or by re-sending valid but stale requests (*replay attacks*).

Mechanism 2: The seminal paper assumes monotonically incremented message counters to ensure that Byzantine nodes cannot *equivocate*, i.e., send different messages to different replicas for the same round of the protocol. In addition, transformed protocols need to replay all previous protocol messages to ensure that the protocol has been executed correctly. In contrast, our work builds on top of TEEs that by design pro-

vide strong security guarantees that combat equivocation and ensure correctness for the execution. In fact, RECIPE prevents equivocation by materializing a distributed TCB that shields the protocol's (distributed) execution, thus the protocol runs as expected, as well as shielding the network communication based on an authenticated message format (§ 4.4.1).

4.3.3 System Design Challenges

Our work shows how to leverage modern hardware to build efficient, robust, and easily adaptable distributed protocols by meeting the aforementioned transformation requirements (see Q1—Q3 below). Motivated by the recently launched cloud-hosted blockchain systems, we also argue that confidential BFT protocols are required to satisfy modern applications' needs for confidentiality (see Q4 below). Specifically, we address the following research questions.

Q1: How to use TEEs efficiently? TEEs are not a panacea: due to their architectural limitations (limited trusted memory[1] and slow syscalls' API) [170, 109], their naive adoption to build practical systems does not necessarily translate into performance improvements. For example, communication in the state-of-the-art BFT protocols [83, 328, 213], which is at the core of any distributed protocol, primarily builds on standardised kernel interfaces (e.g., sockets) suffering from big latencies and not exploiting the full potential of the system [339].

In RECIPE, we carefully address these TEE limitations without introducing an additional burden to developers. We build a secure remote procedure call (RPC) framework on top of a direct I/O network stack for TEEs that achieves three goals. First, it boosts performance by avoiding expensive syscalls within TEEs. Secondly, it extends the transferable authentication and non-equivocation primitives across the untrusted network infrastructure realising the transformation in practice. Lastly, we follow an established RPC-programming paradigm that has proven to be the most effective one for building distributed protocols [168, 88, 164, 162].

Q2: How to use TEEs to transform and build *practical* systems? While Clement et al. show that a translation of a CFT protocol to a BFT protocol *exists*, it adopts an impractical strategy when it comes to building real systems. The entire (transformed) system relies on an expensive mechanism to ensure the correct execution of the underlying CFT protocol. In each round, each replica needs to receive the history of all previous messages, verify their authentication and replay the execution of the protocol's entire history. This way, it is ensured that non-Byzantine replicas rebuild their

[1] At the time of writing this thesis, we had access to Intel SGX v1, which provided approximately 94 MiB of trusted memory [148]. Currently, newer TEEs support larger areas of trusted memory. A discussion on the impact of these advanced TEEs on the systems described in this thesis can be found in section § 6.3.

state correctly and also that the currently executed message is legitimate (i.e., derives from a valid execution scenario of the protocol).

Secondly, the transformed protocol may amplify the native semantics of the original CFT protocol such as linearizable local reads. As in classical BFT protocols, clients cannot trust individuals, instead, they build collective trust by receiving a *quorum certificate*, that is $f+1$ identical replies from different replicas which ensures that at least one correct replica has responded (and verified the correctness of the result).

We design RECIPE to work out-of-the-box to build real systems. RECIPE leverages the properties offered by TEEs to shield the correct protocol execution while our network stack extends the security properties to the network. As a result, our approach does not impose any dependency on the history execution of the protocol, and the original protocol's message complexity is not affected. We also offer an authenticated, per-node, in-memory KV store to allow replicas to store data detecting integrity and authenticity violations and supporting local reads individually (if the implemented protocol supports them). Our approach improves performance, but enables easy adoption as well; developers do not have to worry about maintaining protocols' semantics in Byzantine settings. Note that RECIPE's KV is not required to shield the states and the workflow of the RECIPE's protocols. As explained, it is used to evaluate the performance of RECIPE-transformed systems on top of a distributed data layer (§ 4.7) while preserving the original CFT protocol's operations (e.g., local consistent reads).

Q3: How to realize initialisation, recovery, and failure detection? While the transformation remains agnostic with respect to the transformed CFT protocol in normal operation, the system designers still need to design recovery mechanisms when failures occur. Specifically, Clement et al. do not address how the system *initialises* its state, *detects* failures, and *recovers* from them. Different CFT protocols have different mechanisms for recovery and failure detection. Some protocols continue to operate when failures occur [201, 235] while others rely on accurate timeouts to detect non-responsive leaders and nodes [168, 236, 269]. Unfortunately, TEEs come with neither a trusted initialisation mechanism for distributed systems [141] nor a trusted timer source [224, 151].

RECIPE builds on a secure substrate that overcomes these limitations. We build on a mechanism for collective attestation and a trusted lease mechanism [312] which is a foundational primitive for *trusted timeouts* with which system designers build failure detectors [139], leader election algorithms [101], etc.

Q4: Is BFT enough? The case for confidential BFT protocols. Applications that manage sensitive data (e.g., health-care applications [176]) adopt blockchain solutions for privacy. To this end, cloud-hosted blockchain solutions have been launched [11, 46,

335, 143, 241]. However, these cloud-hosted blockchain systems that fundamentally build on agreement protocols for serialising the ledger [97], jeopardise the blockchain principles of decentralised trust [213]. Indeed, applications are compelled to put too much trust in the cloud provider regarding the integrity and confidentiality of the hosted application.

While BFT protocols offer an important foundation to build trustworthy systems, we argue that more and more modern applications [107, 244, 64, 233, 253, 40, 66] seek confidentiality beyond the scope of the BFT model. RECIPE is a promising step on this direction. Built on top of TEEs, RECIPE transparently offers confidential execution for the distributed protocol. The use of TEEs add significant performance overheads to RECIPE compared to traditional CFT protocols. However, we evaluated four RECIPE protocols against two state-of-the-art BFT protocols and we showed that RECIPE can outperform them (§ 4.7). As shown in our evaluation RECIPE seems to offer better throughput compared to BFT even when offering confidentiality by encrypting the networking messages and the KV store data.

4.4 RECIPE Protocol

Clients in RECIPE execute requests through a `PUT/GET` API. As discussed in § 4.3.1, the request is forwarded to the protocol's coordinator node. Upon the request's execution the node replies back to the client. For the communication between the clients and the RECIPE nodes we assume that the system developer (who wrote the RECIPE protocol) passes through the system initialisation phase (§ 4.4.3) a public-private key pair. As such, clients and RECIPE nodes establish TLS connections similarly to TREATY system in Chapter 3. Figure 4.2 shows as an example a RECIPE implementation of Raft (R-Raft) including all three execution phases of a typical RECIPE protocol: the transferable authentication phase (blue box), the initialization phase (green box) and the normal execution phase where the transformed CFT protocol executes clients' requests (red box)[2]. Prior to the protocol execution, nodes pass through a transferable authentication phase (§ 4.4.3) to prove that the TEEs and loaded code are genuine, followed by initialization and the protocol's normal operation[3].

[2]We implemented and benchmark the original description of Raft where the leader commits and replies to clients once the majority acknowledges the replication phase [236] (one-phase commit) to optimize for latency. However, to discuss in detail the protocol's workflow, we, on purpose, allow the leader to complete the commit phase entirely and then reply to the client. This does not violate the safety of the protocol and does not affect the leader election algorithm.

[3]A node is in *normal operation* when it is not a faulty node and it is not recovering.

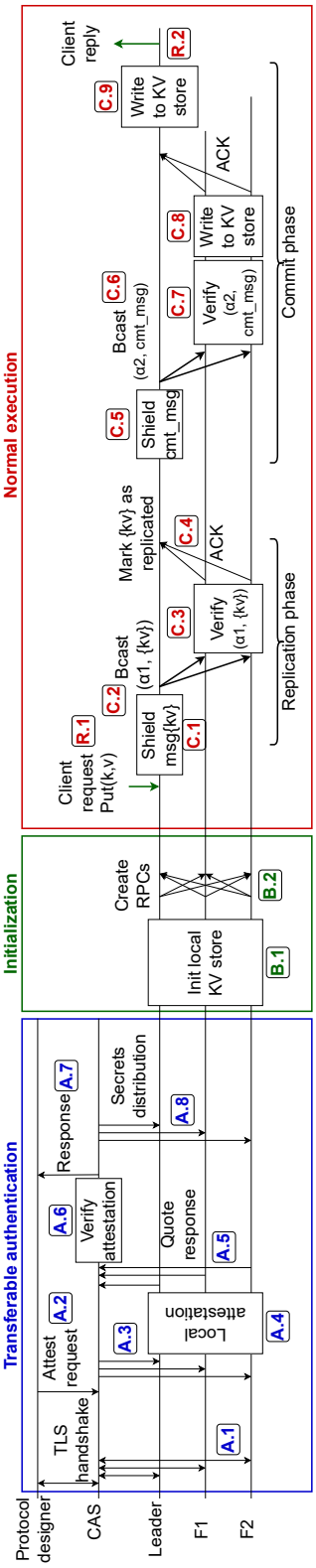

Figure 4.2: Example of the RECIPE version of Raft (R-Raft) execution.

4.4.1 Normal Operation

We first explain the initialization and the normal execution phases, assuming all participant nodes executed the transferable authentication phase successfully. The nodes execute the initialization phase, initializing their own local KV stores (B.1), e.g., allocate host memory. Afterward, each node initiates its network connections (e.g., configures NIC-memory, network ports, etc.) and finally establishes connections with other peers (B.2) based on the configuration it securely received at the attestation process (A.7). The leader then runs the underlying CFT protocol (in our case, Raft (C.1)—(C.9)) to execute the client request (R.1). Upon completion, it replies back to the client (R.2). Next, we discuss the RECIPE abstraction under the normal operation.

We use the notation [$h_{c\sigma_c}$, payload] to denote an *attested* or *shielded* message that is comprised of the signed hash ($h_{c\sigma_c}$) of payload (*certificate*) along with the raw payload data. We use the symbol σ_c to denote that a piece of data is signed with a key c. For simplicity in Figure 4.2, we use the notation (α, `kv`) for an attested message referring to a Key-Value pair `kv` with certificate α.

#1: Clients send to the coordinator their request of the form [$h_{c\sigma_c}$, (metadata, req_data)] (R.1). The req_data is the request's associated data and the metadata might include among others the client's and the request's id, the leader's and term's ids (known to the client).

#2: Nodes receive and process a request after successfully verifying their integrity and authenticity by verifying the received message MAC. In addition, the implemented protocol itself might impose further constraints that are also inherrited in RECIPE version of the protocol. For example, in our Raft, requests that have a wrong view of the term and the leader or have already been processed (e.g., their id is known to the node) are dropped.

#3: Upon the reception of a client's request:

#3.1 The coordinator (leader) verifies the integrity and authenticity of the message using RECIPE's authentication layer. It also verifies the metadata, e.g., the message is invalid if the term and the leader (if any) known to the client are incorrect. The leader updates the client table with the latest processed request for each client.

#3.2 Next, the leader initializes the protocol for that request. In our example, the Raft leader shields the message (C.1), generating a trusted message format (α1, `kv`) where α1 is the certificate of `kv`, and broadcasts the request to the followers (C.2) (replication phase).

#3.3 The messages exchanged between replicas are of the form [$h_{r\sigma_{cq}}$, (metadata, req_data)]. The metadata includes a per-request unique tuple (view, cq, cnt_{cq}) that con-

tains: (1) the view, an identifier that is optionally set by the implementation for every new leader (as such, leaderless protocols might leave it uninitialized) (2) the communication channel id (cq) that is uniquely identified as a tuple containing the connected nodes' ids, and (3) a sequencer id or a message counter (cnt_{cq}) that is assigned to the messages sent over this channel and is increased monotonically for every new message. This unique message sequencer allows nodes to detect stale or missed messages and as such combat network attacks that aim to re-order or double execute requests.

#4: When a replica receives a message from another replica:

#4.1 If the replica is in normal state operation, it verifies the validity of the message. Else, it refuses to process the request.

#4.2 The replica verifies the received sequencer id (cnt_{cq}) to see if it is consistent with its local counter ($recv_{cnt}$). If the cnt_{cq} equals ($recv_{cnt}+1$), the replica executes the request immediately, increases its local counter, acknowledges to the sender node and updates the client table. If the cnt_{cq} refers to a "future" message ($cnt_{cq}>recv_{cnt}+1$), the replica queues the request in the protected area. Periodically, it applies the queued requests eligible for execution and notify coordinators accordingly.

#5: In our example, the followers verify the request (e.g., MAC verification for message integrity and authenticity as well metadata verification to ensure that the request is send from the elected leader and has not been seen before) (C.3), enqueue the uncommitted request in a TEE buffer, and send ACKs back to the leader. The leader, upon receiving the majority of ACKs marks the request as replicated (C.4) and proceeds to the second round of the protocol instructing the followers that replied to commit the update ((C.5)—(C.7)). At this point, each follower instructed to commit applies the request to its local KV store (C.8) and ACKs the commit to the leader. Similarly to the replication phase, the leader finally commits (C.9) when it receives ACKs from the majority.

#6: After the protocol's execution, the coordinator marks the request as committed and notifies the client (R.2).

4.4.2 View Change

While decentralized protocols remain available as long as most nodes are part of the membership, the leader-based protocols do not progress if the leader goes down. To remain available after the leader crashes, the followers need to closely monitor the leader (e.g., heartbeat messages in inactive periods) and, in case it is unreachable, to *elect* a new one, i.e., perform a *view change*.

In line with the CFT protocols, RECIPE leader-based protocols must assign a leader

with a term and the node identifier that is stored securely within the TEE. The term identifier can be seen as an *epoch*, a monotonically increasing counter that uniquely identifies the current view of the system. To continue serving requests after a leader election, the majority of the replicas of the membership (quorum) need to confirm the new leader along with the new term. Since a leader needs to be acknowledged by the majority of the nodes to operate, the latest term will survive in at least one node, ensuring the term's monotonic increments.

Correctness. The correctness condition for leader elections is that every commit must survive into the new leader election in the order selected for it at the time it was executed. RECIPE does not make further assumptions, protocols can rely on their election algorithms [18] as we guarantee that the replicas are trusted (§ 4.6).

Failure detection. CFT protocols [236, 269] often require trusted or accurate timers to detect failures and continue operating. RECIPE builds on top of Intel SGX, which does not secure timers [151, 224]. Unfortunately, OS-timers and software clocks cannot be trusted too. To overcome this, RECIPE builds on top of SCONE that implements a trusted lease mechanism [312]. This mechanism supports all the properties of classical leases [120] in untrusted environments even in the presence of attackers that manipulate the clock. Precisely, the mechanism ensures that (1) the lease terms can be measured securely and accurately by the lease granter (*i.e.*, the process that grands valid leases upon request) and the lease holder (*i.e.*, the process that requests leases from the granter), (2) the underlying timer has not been manipulated and in case of manipulation suspicions the mechanism ensures that the lease term on granter is always a superset of the lease term on the holder and (3) the lease checks and the operations executions can be executed atomically to avoid time-of-check to time-of-use (TOCTOU) attacks. RECIPE users can build trusted timeout systems, failure detectors [139], leader election algorithms [101], etc. However, this is beyond the scope of this thesis.

4.4.3 Transferable Authentication

Before initialization, all participant nodes run the transferable authentication phase (are *attested*). The phase ensures that only authenticated replicas receive configurations and secrets and participate to the protocol, guaranteeing the transferable authentication property and protecting against Sybil attacks [87]. RECIPE materializes this phase using a remote attestation protocol.

The attestation protocol is initialized by the protocol designer (PD) (*challenger*) who establishes TLS connection with the Configuration and Attestation service (CAS) (A.1). CAS is responsible for proving the authenticity of a TEE to others (e.g., clients, recovered nodes). For now, we focus solely on the attestation protocol; we explain the

mechanism of CAS in § 4.5.4. The CAS also establishes secure communication channels with the participant nodes. The secure communication channel establishment is achieved as in section § 3.2.

The PD sends an *attest request* to the CAS(A.2), which is then forwarded to the replicas (A.3). The replicas perform *local attestation* [249], i.e., they calculate a measurement of their code and generate a quote that is uniquely bound to that particular TEE (A.4). The quotes are sent over the TLS channel to the CAS that verifies them. Upon a successful attestation, the CAS notifies the PD that might want to forward configuration data (e.g., singing or encryption keys for the network and the KV store as well as private-public key pairs for the client-node communication), to the replicas (A.7)—(A.8).

4.4.4 Recovery

As nodes fail, new or recovered nodes need to be added to continue operating at peak performance. To add a new node, the membership needs to be reliably updated, following that all other live replicas are notified of the new node's intention to join the replica group. Once the replicas acknowledge this notification, the new node starts operating as a shadow replica that participates as a follower for the writes but does not serve client requests. Additionally, it might read chunks from other replicas to fetch the latest state similarly to [89, 237]. For non-equivocation, recovered nodes always start as fresh nodes and as such are assigned unique (increasing) node ids by the CAS through the attestation phase. As such, RECIPE does not recycle the same (view, cq, cnt_{cq}) for different messages from different nodes. Overall, a new joining node follows the next steps:

#1: A recovering node needs first to be attested before any secrets and membership information are shared. Before the control passes to the CFT protocol, the node sends a join request to a designated node (challenger-node), notifying it about its willingness to join the cluster.

#2: The challenger-node that receives the request inittializes a remote attestation to verify its trustworthiness as discussed in § 4.4.3.

#3: After a successful attestation, the challenger-node shares the network signing or encryption keys and the configuration of the membership as a response to the join-request. In RECIPE all nodes use the same symmetric key for the networking between the RECIPE nodes and the same symmetric key for the KVs operations that is different from the network symmetric key. This design has been decided for simplicity and it is not a restriction imposed by RECIPE; we could configure the system with a unique symmetric key per connection and per local KVs. The challenger-node also broadcasts

a message to the other replicas about the successful attestation of the new node. Once the new joiner acknowledges the response from the challenger-node, it establishes connections with the other replicas.

#4: The new node joins as a shadow replica fetching the state of the system. If the CFT protocol allows, this node can participate in writes while recovering. Once it is synchronized with the system's state, it transits to normal operation and executes the protocol.

4.5 RECIPE Library

This section describes four core components of RECIPE. Table 4.1 summarizes the RECIPE's API for each component.

4.5.1 RECIPE Networking

RECIPE adopts the remote procedure call (RPC) paradigm [36] over a generic network library with multiple transportation layers (Infiniband, RoCE, and DPDK), while also favorable in the context of TEEs where traditional kernel-based networking is impractical [178].

Initialization. System developers that build RECIPE protocols need to initialize the networking layer, e.g., nodes' connections, the types of the available requests and by registering the appropriate (custom) request handlers. In RECIPE terms, a communication endpoint corresponds to a per-thread RPC object (RPCobj) with private send/receive queues. All RPCobjs are registered to the same physical port (configurable). Initially, RECIPE creates a handle to the NIC which is passed to all RPCobjs. Developers need to define the types of the RPC requests, each of which might be served by a different request handler. Request handlers are functions written by developers that are registered with the handle prior to the creation of the communication endpoints. Lastly, before executing the application's code, the connections between RPCobjs need to be correctly established.

send/receive operations. We offer asynchronous network operations following the RPC paradigm. For each RPCobj, RECIPE keeps a transmission (TX) and reception (RX) queue, organized as ring buffers. Developers enqueue requests and responses to requests via RECIPE's specific functions which place the message in the RPCobj's TX queue. Later, they can call a polling function that flushes the messages in the TX and drains the RX queues of an RPCobj. The function will trigger the sending of all queued messages and process all received requests and responses. Reception of a request triggers the execution of the request handler for that specific type. Reception of

Attestation API	
`attest(measurement)`	Attests the node based on a measurement.
Initialization API	
`create_rpc(app_ctx)`	Initializes an RPCobj.
`init_store()`	Initializes the KV store.
`reg_hdlr(&func)`	Registers request handlers.
Network API	
`send(&msg_buf)`	Prepares a req for transmission.
`respond(&msg_buf)`	Prepares a resp for transmission.
`poll()`	Polls for incoming messages.
Security API	
`verify_msg(&msg_buf)`	Verifies the authenticity/integrity and the sequencer id of a msg.
`shield_msg(&msg_buf)`	Generates a shielded msg.
KV Store API	
`write(key, value)`	Writes a KV to the store.
`get(key, &v`$_{TEE}$`)`	Reads the value into v$_{TEE}$ and verifies integrity.

Table 4.1: RECIPE library APIs.

a response to a request triggers a cleanup function that releases all resources allocated for the request, e.g., message buffers and rate limiters (for congestion).

Non-equivocation and authentication layers. RECIPE's networking library embodies a non-equivocation and an authentication layer through two TEE-assisted primitives, the shield_request() and verify_request(), shown in Algorithm 1.

Non-equivocation layer: RECIPE prevents replay attacks in the network with sequence numbers for the exchanged messages. Each replica maintains local sequence tuples of the form (view, cq, cnt$_{cq}$) where view is the current view number, cq is the communication endpoint(s) between two nodes, and cnt$_{cq}$ is the current trusted counter value in that view for the latest request sent over the cq. The sender assigns to messages a unique tuple of the form (view, cq, cnt$_{cq}$) and then increments cnt$_{cq}$ to guarantee monotonicity. Replicas execute the implemented CFT protocol for verified valid requests. Replicas can verify the freshness of a message by examining its cnt$_{cq}$ (verify_request() primitive). The primitive verifies that the message's id (as part of the metadata) is consistent with the receiver's local counter rcnt$_{cq}$ (rcnt$_{cq}$ is the last seen

Algorithm 1: RECIPE's authentication primitives.

```
1  ▷ cnt_cq: the latest sent message id from cq
2  ▷ rcnt_cq: the last committed message id from cq
3  function shield_request(req, cq) {
4      cnt_cq ← cnt_cq+1; t← (view, cq, cnt_cq);
5      [h_σcq, (req,t)] ← singed_hash(req, t);
6      return [h_σcq, (req,t)];
7  }
8  function verify_request(h_σcq, req, (view, cq, cnt_cq)) {
9      if verify_signature(h_σcq, req, (view, cq, cnt_cq)) == True then
10         if view == current_view then
11             if cnt_cq <= rcnt_cq then
12                 return [False, req, (view, cq, cnt_cq)];
13             if cnt_cq == (rcnt_cq+1) then rcnt_cq ← rcnt_cq+1;
14             buffer_locally(req, (view, cq, cnt_cq));
15             return [True, req, (view, cq, cnt_cq)];
16     return [False, req, (view, cq, cnt_cq)];
17 }
```

valid message counter for received messages in cq). RECIPE's replicas are willing to accept "future" valid messages as these might come out of order, i.e., messages whose cnt_{cq} is $>$ ($rcnt_{cq}+1$). These messages are processed and committed according to the CFT protocol.

Authentication layer: For the authentication, we use cryptographic primitives (e.g., MAC and encryption functions when RECIPE aims for confidentiality) to verify the integrity and the authenticity of the messages. Each message m sent from a node n_i to a node n_j over a communication channel cq is accompanied by metadata (e.g., cnt_{cq}, view, sender and receiver nodes id) and the calculated message authentication code (MAC) $h_{cq\sigma q}$. The MAC is calculated over the m and the metadata. The sender node calls into the shield_request(req, cq) and generates such an trusted message for the request req.

API. We offer a create_rpc() function that creates a bound-to-the-NIC RPCobj. The function takes as an argument the application context , i.e., NIC specification and port, remote IP and port, creates a communication endpoint and establishes connection with the remote side. The function returns after the connection establishment. RPCobjs offer bidirectional communication between the two sides. Prior to the creation of the RPCobjs, developers need to register the request types and handlers using the reg_hdlr() which takes as an argument a reference to the preferred handler. The handlers are part of the protocol codebase and as such are executed within the TEE.

For exchanging network messages, we designed a send() function which takes as arguments the session (connection) identifier, the message buffer to be sent, the request type and the cleanup function. This function submits a message for transmission. Upon a reception of a request, the program control passes to the registered request handler where the function respond() can submit a response or ACK to that request. Lastly, the function poll() needs to be called regularly to fetch, process and send the incoming responses or requests and send the queued responses and requests respectively.

4.5.2 Secure Runtime

RECIPE's protocol codebases are executed within the TEE. We have build our codebase in C++ using SCONE to access the TEE hardware. SCONE exposes a modified libc library and combines user-level threading and asynchronous syscalls [293] to reduce the cost of syscall execution. While we limit the number of syscalls, leveraging SCONE's exit-less approach allows us to optimize the initialization phase that vastly allocates memory for networking and the KV store. To enable NIC's DMA operations and memory mappings to the hugepages (for message buffers and TX/RX queues) (§ 4.5.1), we overwrite the `mmap()` syscall of SCONE to bypass its shield layer and allow the allocation of (untrusted) host memory.

For the cryptographic primitives, we build on OpenSSL [239]. Lastly, we build on a lease mechanism [312] in SCONE for auxiliary operations, e.g., failures detection and leader's election.

4.5.3 RECIPE Key-value Store

RECIPE provides a lock-free, high-performance in-memory Key-Value (KV) store based on an in-memory skip-list. We partition the keys from the values' space by placing the keys along with metadata (and a pointer to the value in host memory) inside the enclave and storing values (encrypted) in the host memory. Our partitioned KV store reduces the number of calculations for integrity checks, compared to prior work [170] which implements (per-bucket) merkle trees and re-calculates the root on each update. Importantly, separating the (keys + metadata) and the values between the trusted limited enclave and untrusted unlimited memory decreases the EPC pressure.

RECIPE's KV store design resolves Byzantine errors since the metadata (and the code that accesses them) reside in the enclave. That said, RECIPE allows for local reads as nodes can verify the integrity of the stored values prior to replying to the client. Our partitioned scheme strengthens the system's security properties and can easily offer confidentiality by encrypting the values that reside outside the TEE. Similarly to the

original RECIPE KV store that only offers integrity for the stored data, the KV store encryption keys are shared across initialization at the transferable authentication phase through CAS.

4.5.4 RECIPE Attestation and Secrets Distribution

Remote attestation is the building block to verify the authenticity of a TEE, i.e., the code and the TEE state are the expected [249]. As such, RECIPE provides attest(), generate_quote() and remote_attestation() primitives (Algorithm 2) that allow replicas to prove their trustworthiness to other replicas or clients. The attestation takes place before the control passes to the protocol's code. Only successfully attested nodes get access to secrets (e.g., signing or encryption keys, etc.) and configurations. For RECIPE's attestation we use the same mechanism as in TREATY. In the remainder of the chapter we discuss the attestation primitives (from the TEE perspective) that RECIPE implements.

Algorithm 2: RECIPE's attestation primitive.

```
1  function remote_attestation() {
2      nonce ← generate_nonce();
3      send(nonce, k_pub); DHKE(); quote_σ_kpub ← recv();
4      if verify_signature(quote_σ_kpub) == True then
5          μ_TEE ← decrypt(quote_σ_kpub, k_priv);
6          if (verify_quote(μ_TEE) == True) send_secrets();
7  }
8  function attest() {
9      μ ← gen_enclave_report(); return μ;
10 }
11 function generate_quote(μ, k_pub) {
12     key_hw ← EGETKEY();
13     quote ← sign(μ, key_hw); quote_σ_kpub ← sign(quote, k_pub);
14     return quote_σ_kpub;
15 }
```

The attestation process is initialized by the *challenger*, a remote process that can verify the authenticity of a specific TEE. The challenger executes the remote_attestation() function to send an attestation request to the application—usually in the form a nonce (a random number). The challenger and the application, then, establish a mutual authenticated TLS connection, e.g., passing through a Diffie-Hellman key exchange pro-

cess [212]. The application generates an ephemeral public key which is used by the challenger later to provision any secrets.

When the TEE receives the nonce, it calls the attest() and generates a *measurement* (μ) of its state and loaded code. Following this, the TEE calls into the generate_quote(μ, k_{pub}) to sign μ (quote) with the key_{hw} which is fetched from the TEE's h/w (EGETKEY()). The TEE signs and encrypts the quote $\text{quote}_{\sigma_{k_{pub}}}$ over the challenger's public key k_{pub} which is, then, sent back to the challenger. Upon successful verifications of the $\text{quote}_{\sigma_{k_{pub}}}$, the challenger shares secrets and configurations.

To offer low-latency attestations within the same datacenter that RECIPE runs, we build a Configuration and Attestation service (CAS). The Protocol Designer (PD) deploys the CAS inside a TEE and attests it through the hardware vendor's attestation service—e.g., Intel Attestation Service (IAS [141]). Once the CAS is attested, it is trusted and the PB can upload secrets and configurations.

4.6 RECIPE Analysis

4.6.1 Requirements Analysis

We next show how RECIPE satisfies the non-equivocation and the transferable authentication properties.

Non-equivocation. RECIPE prevents equivocation attacks through a trusted monotonically increasing message counter that assigns *sequence numbers* to the network messages. The sender assigns a monotonically increasing sequence *id* to every message of a given round, guaranteeing a total ordering of all network messages between any two communication endpoints. On the receiving side, it suffices for replicas to verify that the message's counter is *consistent* with their local known sequencer for this communication endpoint. RECIPE's sequencer prevents the repeating of stale (but authenticated) network messages which can be indistinguishable from equivocation. In addition, a Byzantine node may "appear" to not send messages to some (weak non-equivocation) or all (strong non-equivocation) other nodes during a given operation [202]. RECIPE is responsible for neither—we rely on the CFT as both weak and strong non-equivocation are indistinguishable from crash failures [202].

Transferable authentication. RECIPE ensures the following two core properties from its TEE-assisted primitives: property #1: RECIPE distributes the configuration, keys etc. in a secure manner to trusted nodes, and property #2: RECIPE preserves the authenticity and integrity of the network messages.

Transferable authentication is provided implicitly by properties #1 and #2. Property #1 ensures that for every communicating pair of processes, their signing keys

are shared after their successful attestation. This, in turn, follows that only trusted (correct) processes are capable of signing (and generating) valid messages. This also follows that Byzantine adversaries cannot forge "future" messages but, instead, they are only limited to re-playing old messages.

Property #2 ensures that Byzantine actors cannot both alter a message's contents and forge a new valid message by generating a verifiable MAC over the message without the network symmetric key. Lastly, authenticity is transferable and can be verified in the exact same way that any two directly communicating nodes do. The combined power of these properties satisfies the transferable authentication and non-equivocation requirements.

4.6.2 Correctness Analysis

CFT protocols need to provide the following safety properties regarding the messages delivered by the network [133, 134]. We show how these are provided by RECIPE's non-equivocation and (transferable) authentication layers.

Safety. If a correct process p_i *receives and accepts* a message m from a process p_j, then the sender p_j is correct and has executed the `send` operation with m.

Integrity. If a correct process p_i *receives and accepts* a message m, then m is a valid and correct message generated according to the protocol specifications.

Freshness. If a correct process p_i *receives and accepts* a valid message m_{j_x} sent from a correct process p_j, then it will not accept any future message m_{j_y} with $y = x$, $\forall x, y \in \mathbb{N}^+$.

Next, we explain how RECIPE satisfies these properties. Safety and integrity are directly satisfied by our transferable authentication mechanisms. Firstly, recall that only trusted and correct processes can generate valid messages (messages that can be successfully verified). Therefore, a message m accepted by some correct process p_i must have been generated and sent by a correct process p_j. Moreover, correct processes cannot deviate from the protocol's specification in order to generate messages that do not adhere to it. Adversaries can neither forge nor alter messages that do not adhere to it either (§ 4.6.1). In any case, when a correct process p_i receives and accepts a message m it must be a valid message generated according to the underlying protocol's specifications.

Freshness is directly satisfied by our non-equivocation layer that imposes a total order on messages between two communication endpoints. A correct process p_i drops already received messages to sustain replay equivocation attacks.

4.7 Evaluation

4.7.1 How to Apply the RECIPE Library?

Developers can use the RECIPE-lib API to transform their preferred CFT protocol for Byzantine settings without further modifying the core states of the protocol. Listing 4.1 illustrate Raft's transformation using the same example of R-Raft from Figure 4.2. In Listing 4.1, the blue sections show the native Raft code, whereas the orange sections show the modifications introduced by RECIPE.

At a high level, developers need to use the RECIPE's network API and replace the conventional unsecure RPC-API [209, 162, 106] with RECIPE-lib's networking functions (Lines:6, 14, 20-23, 32-25). Some of the RECIPE's API remains equivalent to the native API; typical examples are the poll() (Line: 33) and reg_hdlr() (Lines: 22-23) functions. On the other hand, we introduced some slight modifications in send() operation (e.g., Lines:32, 35) and Initialization APIs (e.g., Lines:19,20). To effectively use them, developers must register a (configurable) amount of the host memory for the KV store data. Lastly, the code must shield the message (Line: 14, 32, 35) before transmission and verify it upon reception (Lines: 4, 10) in the request handler function.

```
1  // ----------- Requests handlers definition -----------
2  void replication_hdlr(Raft_ctx* ctx, Msg* recv_msg) {
3      // verifies recv_msg integrity and counter
4      [msg, cnt] = verify_msg(recv_msg);
5      ... // appends the req to the on-going reqs buffer
6      conn.respond(shield_msg(ACK_repl)); // transmits ACK
7  }
8  void cmt_hdlr(Raft_ctx* ctx, Msg* recv_msg) {
9      // verifies recv_msg integrity (and counter)
10     [msg, cnt] = verify_msg(recv_msg);
11     auto [key, val] = decode(req);
12     // moves val in host mem and stores its certificate in TEE
13     ctx->kv.write(key, val);
14     conn.respond(shield_msg(ACK_cmt)); // transmits ACK
15 }
16 // ----------- Init phase (leader and followers) -----------
17 auto ctx = new Raft_ctx(metadata, node_type); // context object
18 // init local KV with host allocated memory and a cipher
19  ctx->kv = init_store(HostMemSize, cipher);
20  RPC_obj conn = create_rpc(enc_key); // create RPC handle
21 // registers handlers
22  conn.reg_hdlr(&cmt_hdlr);
23  conn.reg_hdlr(&replication_hdlr);
```

```
// ----------- Raft leader -----------
for (auto& node : followers_list) {
    conn.wait_until_connected(node); // connects with followers
}
for (;;) {
    ... // gets client request and marks it as on-going
    for (auto& follower : connections)
    // generates a shielded message and bcast to followers
      follower.send(shield_msg(rep_req), TypeRepl);
    conn->poll(); // periodically polls to flush TX/RX queues
    for (auto& follower : connections) ... // bcast commit
      follower.send(shield_msg(cmt_req), TypeCmt);
    ... // if commit phase finishes, apply changes to local kv
    ctx->kv.write(key, val);
}
```

Listing 4.1: Raft transformation using RECIPE: blue sections (native Raft) and orange sections (RECIPE additions).

4.7.2 RECIPE in Action for CFT Protocols

Experimental setup. We run our experiments on real hardware in a cluster of three SGX machines with CPU (NixOS, 5.15.43): Intel(R) Core(TM) i9-9900K each with 8 cores (16 HT), NIC: Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02) and a 40GbE QSFP+ network switch. For the evaluation, we use the YCSB benchmark [349] (configured with approx. 10K distinct keys with Zipfian distribution) with various R/W ratios and value sizes.

To show the benefits of our approach, we implement four widely adopted CFT protocols (one of each category in § 2.1.2, Table 2.1), with the RECIPE-lib API. We build R-ABD, R-Raft, R-AllConcur and R-CR which are the RECIPE versions of ABD, Raft, AllConcur and CR respectively. We compare these protocols with BFT-smart [295], an optimized version of PBFT [58] and Damysus [83], the state-of-the-art version of HotStuff [4] with $2f+1$ on top of SGX. Next, we discuss the characteristics of protocols categories, our chosen protocol and our evaluation results.

A: Leaderless w/ per-key order. Protocols in this family agree on a per-key order of writes in a distributed manner. All nodes can coordinate a write that is completed in at least two rounds. A typical example is Classic Paxos (CP) that achieves consensus in three broadcast rounds. Several works [225, 94, 137, 168, 201] simplify the complexity of CP to boost performance. Protocols such as [225, 94, 137] can offer consensus in two rounds but fall back to CP if conflicts occur. Others [168, 201] execute writes in

two rounds enforcing all messages to be received by all nodes or relaxing the Read-Modify-Write semantics. These protocols offer linearizable reads by executing quorum reads to consult (at least) the majority. Protocols like [168] where writes need to reach all nodes allow for local reads (at the cost of availability—if a node fails, writes block).

Choice: ABD [201]. We implemented ABD, a multi-writer, multi-reader protocol with RECIPE (R-ABD). R-ABD offers linearizable (quorum) reads using Lamport timestamps (logical clocks) [182] for each Key-Value (KV pair). R-ABD broadcasts requests to all replicas and waits for acks from the quorum.

Writes are executed in two rounds of broadcasts. First, the coordinator asks from all replicas to hand over the key's timestamp (TS), securely stored inside the TEE (KV store metadata). Upon receiving a majority of the timestamps, the coordinator creates a higher TS for that key by increasing the highest received TS. Finally, it broadcasts the new KV pair along with its new timestamp to all replicas which, in turn, insert the KV pair into their KV store. Upon gathering a majority of acks it replies to the client.

R-ABD (usually) executes reads in one round by collecting all values (and their TS) from the majority. If the majority agrees on the latest seen TS, the coordinator replies to the client. Otherwise, the coordinator chooses the highest TS and invokes the second round of the write-path (for availability).

B: Leader-based w/ total ordering. These protocols [236, 266, 322] serialize writes at the leader, offering total order. The write-path usually requires two broadcast rounds; the leader proposes writes to (passive) followers which they ack the proposal. Once the leader collects the acks from the majority it runs the commit round where, the nodes apply the proposed writes in their total order. Since writes are propagated to the majority where the leader is always part of it, it follows that the leader can always know the latest committed write for all keys. As such, leaders can always read locally while followers must forward reads to the leader. Some protocols [266] allow followers to read locally. This is achieved in two ways: they might forego linearizability and downgrade to sequential consistency [29] (w/ the possibility of reading stale values [266]), or ensure that all writes reach all followers at the cost of availability.

Choice: Raft [236]. As a representative protocol of this family we implemented Raft with RECIPE (R-Raft). We target linearizability; all reads are forwared to the leader that also serializes writes. The leader proposes writes to replicas and commits the request when the majority of them acknowledges the proposal.

The leader receives write requests and stores them in an uncommited_queue inside the TEE. We spawn a dedicated (worker) thread for managing this queue and serializing all writes. Then, the worker thread broadcasts the request (or a batch of consecutive requests) to all followers. The follower nodes verify the messages. As an

optimization, followers accept future messages storing them in a separate queue. The followers commit requests respecting leader's total order. The followers send acks for one or more consecutive requests. Leader only commits a request and responds back to the client when it receives a response from the majority.

C: Leader-based w/ per-key order. Protocols in this class use the leader node to only serialize writes to the same key. All writes are steered to the leader node, which ensures that writes to the same key are applied in the same order by all replicas. These protocols can offer linearizability (it is a compositional property) similarly to the leader-based protocols with total order. While writes are propagated to a majority of nodes, reads are propagated to the leader. As the protocols do not respect a total ordering, local reads to followers lead to weak guarantees such as eventual consistency [329]. As before, we can allow for local reads to all nodes when writes are guaranteed to propagate to all followers.

Choice: Chain Replication [269]. As a representative to this category, we implemented Chain Replication (R-CR) with RECIPE. In R-CR, the nodes are organized in a chain, through which writes are propagated from the head of the chain to its tail. Similarly to [106], we consider CR among the leader-based protocols as the head node is the leader to serialize all writes. A write traverses the chain until it reaches the tail where it is considered committed, which guarantees that all writes reach all nodes. As such, we can offer linearizable by reading locally from the tail.

D: Leaderless w/ total ordering. Such protocols do not serialize writes in a central location but rely on a predetermined static allocation of write-ids to nodes. For example, all nodes know that the writes 0 to N-1 will be proposed and coordinated by node-0, the next N writes will be proposed by node-1 and so on. Therefore, in each round each node can calculate the place of each write in the total order based on its own node-id, without synchronizing with any other node. Then, the node broadcasts its writes along with their place in the total order. Typically a commit message is broadcast after gathering acks from a majority of the nodes. Crucially, all nodes must apply the writes in the prescribed total order.

Choice: AllConcur [251]. To study this category, we implemented AllConcur with RECIPE (R-AllConcur), a decentralized replication protocol with total order that relies on an atomic broadcast primitive. Nodes are organized in a digraph (G) [251] where the fault tolerance of the system is given by G's connectivity. For example, to tolerate 1 node failure on a 3-node system, we calculated the vertex-connectivity to be equal to 2, namely, each node is connected to the other 2 nodes. For the writes, all nodes track all messages for each round and commit them in a predefined order without synchronization. We can treat reads as writes (for linearizability) or, we allow for local

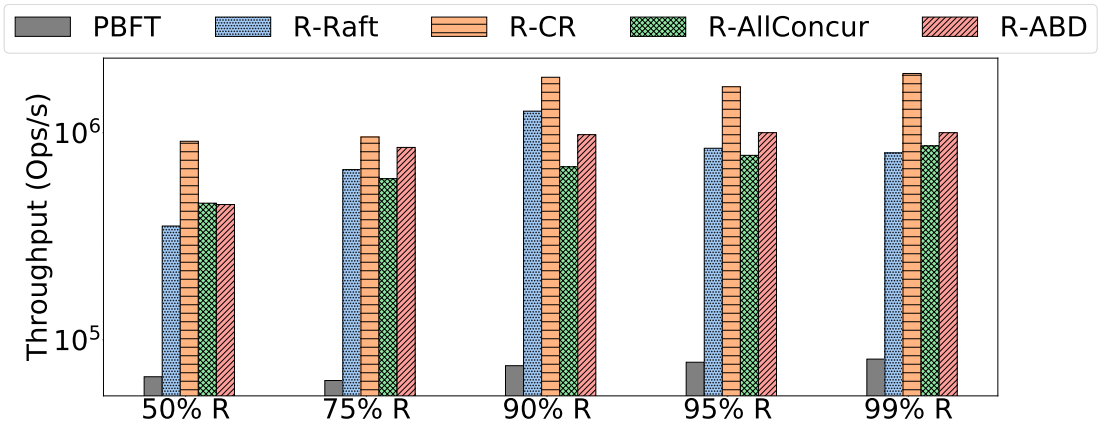

Figure 4.3: Throughput of four protocols with RECIPE compared with PBFT (BFT-smart).

reads to replicas offering sequential consistency [140].

4.7.3 Evaluation Analysis

Reads ratio	R-ABD	R-CR	R-Raft	R-AllConcur
50%	6.5×	13.7×	5.3×	6.8×
75%	13.3×	14.8×	10.05×	9.4×
90%	13×	24×	16.5×	9×
95%	12.8×	21×	10.7×	9.5×
99%	12.3×	23×	9.8×	10.5×

Table 4.2: Speedup in throughput of four protocols with RECIPE compared with PBFT (BFT-smart).

RECIPE vs PBFT. Figure 4.3 shows the throughput (Op/s) and the speedup of the four case studies we implemented with RECIPE compared to BFT-smart [295] (PBFT) for different read/write workloads (and constant value size/payload, 256 B). Our evaluation shows that all four protocols with RECIPE outperform the classical BFT 5× to 24× as shown in Table 4.2. We observe that the local linearizable reads offered by R-CR greatly improve performance. Unfortunately, we see less speedup in read-heavy workloads for the protocols with local reads (e.g., R-Raft and R-AllConcur). We found out that in these protocols, the total ordering was the bottleneck. In case of R-Raft the writer thread that serialized all reqs was slower than the other worker threads (which executed reads or enqueued writes to the writer thread's queue). Additionally, for R-AllConcur we saw that monitoring and waiting for all messages of each round

decreased throughput.

Speedup in R-ABD, R-Raft and R-AllConcur is moderate for write-heavy workloads where writes require two rounds of messages. R-ABD has lighter read-path; reads require the majority to agree on a value which is typically resolved in one round. R-CR outperforms R-ABD as reads are done locally. Lastly, we see that as the workload becomes more read heavy, the performance is not improved due to (1) request rate limiter and (2) single-node bottlenecks.

RECIPE vs Damysus. We compare RECIPE against Damysus [83]. Damysus is an optimized version of HotStuff [4] that, similarly to RECIPE relies on SGX to reduce the number of required replicas to $2f+1$. In addition, Damysus makes use of two trusted services, which we discuss next, to also optimize the protocol latency reducing the protocol phases to two (instead of the three core phases of HotStuff). Damysus is a competitive baseline to RECIPE because it requires similar trusted hardware (Intel SGX), same number of replicas ($2f+1$) and it is also a two phase protocol as R-Raft, R-ABD and R-AllConcur.

We next discuss the HotStuff protocol in comparison to PBFT and the optimizations introduced to this protocol by Damysus. HotStuff is a BFT (streamlined) protocol that rotates the leader on each command to guarantee safety and liveness while eliminating the heavy state transitions of the PBFT-like protocols during the view change. HotStuff compared to PBFT has additional communication phases[4], but it offers linear message complexity as opposed to the quadratic all-to-all communication of PBFT. Damysus is an SGX-based (optimized) derivative of HotStuff that also offers linear message complexity and reduces the core phases of the protocol to two (instead of the three of the classical HotStuff) by implementing two SGX-based services, the Accumulator and the Checker. The Checker guarantees that a Byzantine leader cannot send multiple valid proposals in a given round (equivocation) by assigning to each message a unique identifier based on a monotonically increased counter. In addition the Checker stores information about the prepared and pre-committed (locked) commands. In combination with the Accumulator, that certifies that a command has the highest view among a given set of commands, the two services also guarantee that a leader always proposes to the replicas the latest prepared command.

We compare RECIPE (ran in real SGX hardware) against Damysus [83] with SGX in simulation mode due to the protocol's driver incompatibility with our system. The setup shows the upper bounds for throughput for Damysus that achieves throughput of

[4]HotStuff has in total five execution phases. However, it is considered a three-phase protocol as it has three core phases that follow the propose-vote paradigm: prepare, pre-commit, and commit to agree on blocks. The protocol is complemented by two additional half phases: new-view to submit latest prepared commands, and decide to execute blocks once it is safe to do so.

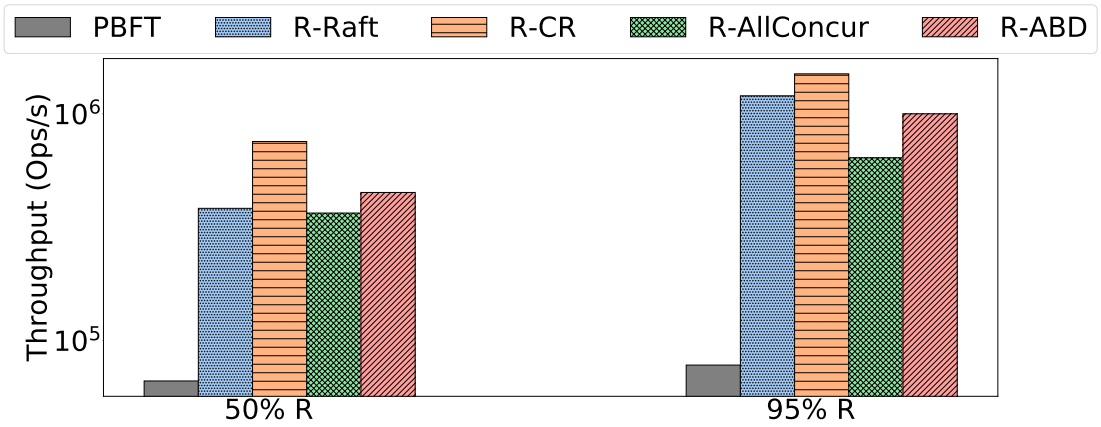

Figure 4.4: Throughput of RECIPE (w/ confidentiality) compared with PBFT (BFT-Smart).

320 kOp/s, 230 kOp/s and 152 kOp/s for payload sizes 0 B, 64 B and 256 B respectively. Our RECIPE (w/ 256 B payload) outperforms 1.1×—2.8× and 2.3×—5.9× Damysus with 0 and 256 B payloads.

RECIPE with confidentiality. Figure 4.4 shows the throughput of RECIPE when we also strive for confidentiality; an extra property that is not offered by classical BFT protocols. We guarantee confidentiality by encrypting all data that leave the TEE (network messages, values residing in the host memory). Briefly, the cost for this extra property is a throughput reduction by a factor of 2. Surprisingly, R-ABD shows minimal degradation compared to R-ABD w/o confidentiality. The reason is that R-ABD quickly saturated all memory resources in our system so throughput was limited mainly by the requests' rate limiter. We see that even with stronger properties, i.e., confidentiality, RECIPE achieves higher throughput than PBFT: on average we calculate 7× and 13× speedup for 50% and 95% workloads respectively.

RECIPE w/ confidentiality boosts throughput up to 4.9× w.r.t. Damysus that does not offer confidentiality (256 B payload).

Value size. Figure 4.5 shows the throughput for different value sizes (under a 90% R workload) for each of the four protocols. The performance drops as the value size is increased due to the EPC's limited size. While RECIPE places the values and network buffers in the untrusted (unlimited) memory, the bigger the allocations are the more we stress test the (limited) enclave memory. R-Raft and R-AllConcur show the greatest slowdown (2× to 7× for 4096 B). We interestingly found out that the batching technique in these protocols with value size of 4096 B deteriorates the performance and, even, crashes the system by consuming all SCONE's memory. For these two protocols with value size 4096 B we depict the numbers with little (< 4) or no batching factor. The

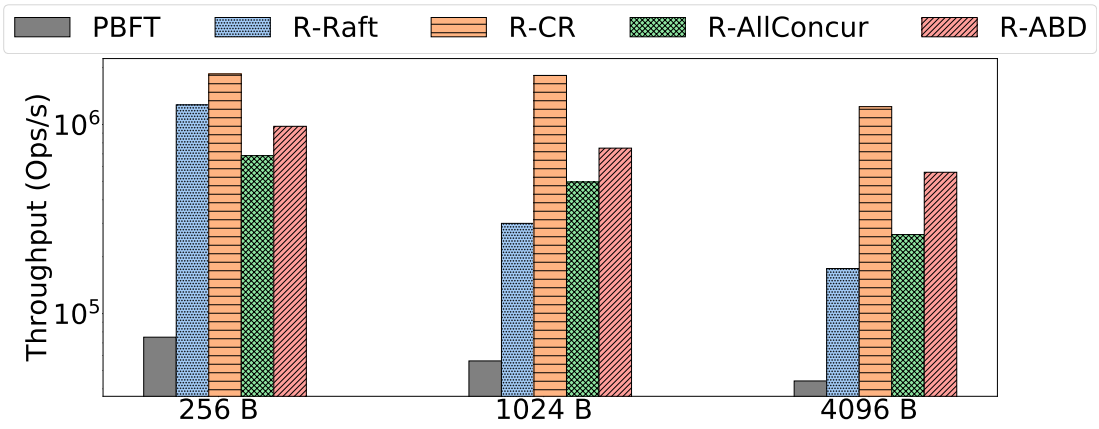

Figure 4.5: Performance of RECIPE for different value sizes.

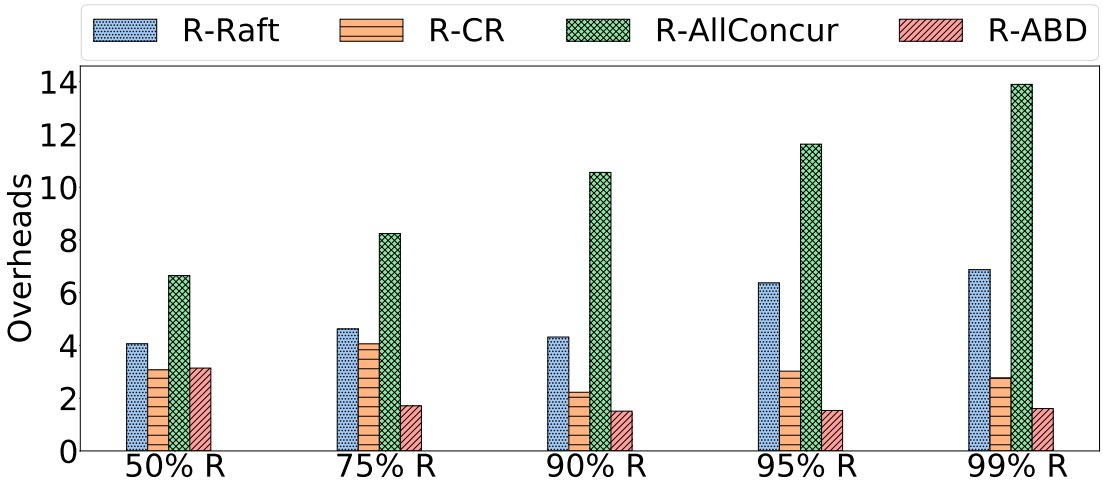

Figure 4.6: Performance overheads of TEEs.

other two protocols, R-ABD and R-CR, also show similar behavior. In these protocols we did not use batching as an extra optimization.

Transformation and TEEs overheads. Figure 4.6 shows the overheads introduced by RECIPE where we compare a native implementation of the protocols with the same network stack without the authentication layer. Overall, an R-CFT protocol experiences 2×—15× slowdown compared to its native execution. The overheads mainly derive from the TEEs. To prove that, we also ran these protocols in `simulation` mode in SCONE where the trusted memory (EPC) is unlimited: we found the throughput to be almost equivalent to the native runs' results. Our observation is also explained from the fact that the higher overheads are for AllConcur and Raft. To optimize these protocols we found extremely helpful the batching. However, batching requires allocations/deallocations of bigger continuous (virtual) memory buffers which stress test SCONE memory subsystem.

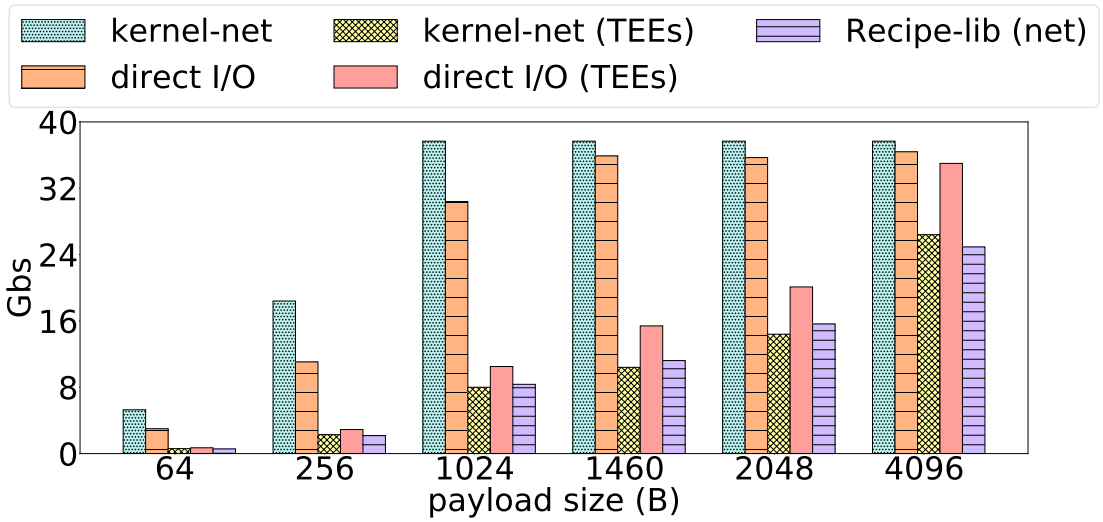

Figure 4.7: Performance comparison of RECIPE network stack (RECIPE-lib (net)), kernel sockets in native execution (kernel-net), kernel sockets within TEEs (kernel-net (TEEs), and kernel-bypass execution using eRPC on top of RDMA/DPDK in native and TEEs execution (direct I/O and direct I/O (TEEs), respectively) for various packet sizes.

RECIPE-lib network performance. Figure 4.7 shows the network throughput (Gbps) of five competitive network stacks: *(i)* a native and a TEE-based network stack on top of kernel sockets [155], *(ii)* a native and a TEE-based direct I/O for networking (RDMA/DPDK) and *(iii)* our TEE-based RECIPE-lib network library. This is to isolate the performance gains of the RDMA-based stack in RECIPE.

We deduct two core conclusions. First, TEEs (SCONE) can degrade network throughput 4×—8× for both kernel-net and direct I/O networking compared to their unprotected (native) runs. Consequently, a naive adoption of TEEs for BFT does not necessarily translate to performance gains. Secondly, RECIPE-lib network performs up to 1.66× faster than the kernel-based networking (kernel-net (TEEs)). As a takeaway the performance speedup (24× w.r.t. PBFT and 5.9× w.r.t. Damysus) for all our four use-cases with RECIPE are primarily due to the transformation (RECIPE) rather than the use of direct I/O.

Attestation. Table 4.3 shows the latencies of Intel's Attestation Service (IAS) [141] and RECIPE CAS. We found that the (mean) average of our CAS is 0.17 s, i.e., 18× faster than the IAS (2.9 s). However, in contrast to IAS that is managed by the hardware vendor, RECIPE's CAS runs in the cloud provider as a single-node system. As such, if the CAS crashes or it is being shutdown (e.g., by malicious adversaries or the cloud provider), RECIPE's availability can be affected because (recovered) nodes cannot join the membership or be attested to new clients.

	Mean / s	Speedup
RECIPE CAS	0.169	18.2×
IAS	2.913	

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

4.8 Related Work

Table 4.4 compares the (most) related work with RECIPE. The resilience is the number of faulty nodes a protocol withstands (for safety and liveness). We divide these works into two categories; the first includes efficient BFT protocols that require $3f+1$ participating replicas [4, 295, 303, 172, 31, 167] and the second category [124, 328, 44, 196, 65, 165] refers to *hybrid* BFT protocols that use trusted modules downgrading the replication degree to $2f+1$. In contrast to our RECIPE, these approaches still require a working understanding of BFT; a task as challenging as it is error-prone [5].

Classical BFT protocols. In the first category, PBFT [58] and its variations [283] run a three-phase protocol. Replicas broadcast messages and transit to the next phases after receiving *quorum certificates* [58] from at least $2f+1$ distinct replicas leading to $O(n^2)$ message complexity. Zyzzyva [172] offloads to the clients the responsibility to correct replicas' state in case of a Byzantine primary. However, prior work [5] found safety concerns in the protocol.

Streamlined protocols [4, 60, 51, 61] avoid heavy state transfers at the view-change by rotating the leader on each command at the cost of additional rounds. HotStuff [4] adds two extra phases to commit the latest blocks. Basil [303] targets *operability* when Byzantine nodes sabotage the execution requiring $5f+1$ replicas.

Trusted hardware for hybrid BFT protocols. The second category includes *hybrid* protocols [83, 328, 124, 109, 38, 219] that leverage trusted hardware to optimize the performance of classical BFT at the cost of generalization and easy adoption. For example, MinBFT [328] (a PBFT derivative optimized with TEEs), Damysus [83] (a HotStuff derivative optimized with TEEs) and Hybster [44] use TEEs to decrease replication factor whereas others [187, 65, 345] utilize trusted counters and logs. Similarly, CheapBFT [165] and FastBFT [196] build on trusted modules to use $f+1$ active replicas but transit to fallback BFT protocols in case of Byzantine failures. HotStuff-TPM [4] uses TPM [249] at the cost of an extra phase.

Similarly to RECIPE, CCF [283] builds within a distributed setting of TEEs a variation of Raft consensus protocol (which operates under the CFT model). CCF in contrast to RECIPE builds a ledger, an append-only log, whereas RECIPE is designed as a generic

Protocols	Active/All Replicas	Resilience	Message complexity	TEEs	Fault Model	TCB
FastBFT [196], CheapBFT [165]	$f+1/2f+1$	0	$O(n), O(n^2)$	Yes	Byz.	Small
MinBFT [328], Hybster [44]	$2f+1/2f+1$	f	$O(n^2)$	Yes	Byz.	Small
PBFT [58], HotStuff [4]	$3f+1/3f+1$	f	$O(n^2)$ $O(n)$	No	Byz.	N/A
CFT	$2f+1/2f+1$	f	depends on the protocol	No	Crash-stop.	N/A
RECIPE	**2f+1/2f+1**	**f**	depends on the protocol	**Yes**	**Byz.**	**Low**

Table 4.4: Related work vs RECIPE.

library to strengthen any CFT replication protocol that does not necessarily offer consensus.

Engraft [332] follows a similar approach to RECIPE implementing a CFT protocol, specifically Raft, inside Intel SGX. Engraft targets the same system model as RECIPE assuming $2f+1$ nodes out of which up to f can exhibit Byzantine behavior and all TEEs (enclaves) are well-behaved. Engraft primarily focuses on ensuring liveness as well as rollback protection for the Raft (persistent) log which are not addressed by RECIPE. In contrast RECIPE targets on providing high-performant BFT leveraging SGX and direct network I/O.

FlexiTrust [124] follows an opposite approach to RECIPE arguing that the classical system model with $3f+1$ is the key to fulfilling the potential of TEEs in BFT. In fact, they identify core challenges in modern hybrid BFT protocols with TEEs such as the lack of liveness, some safety violations due to rollback attacks and performance limitations. As such, FlexiTrust, strives to make minimal use of TEEs while supporting parallel consensus instances and reducing the number of the necessary protocol phases. Compared to RECIPE, it adds f extra replicas to its system model.

Programmable hardware for hybrid BFT protocols. Other works leverage programmable hardware, e.g. FPGAs [165], SmartCards [187] and switches [301] to provide foundational primitives for ensuring BFT. For example, NeoBFT [301] targets the BFT model for permissioned (BFT) blockchain systems [17] by designing an *authenticated ordered multicast* primitive in the programmable switch. To overcome the computation and scalability bottlenecks, they connect to the switch an FPGA device that serves as a cryptographic coprocessor. Compared to NeoBFT where the switch is a single point of failure, RECIPE offloads security into a distributed setting of TEEs providing better availability guarantees while it allows system designers to transform unmodified CFT protocols.

Lastly, Trinc [187] and CheapBFT [165] rely on peripherals to generate attestations for the exchanged message with extremely expensive latencies (50us—105ms) [187, 165]. In addition, similarly to all previous hybrid protocols, they are specifically designed to optimize a particular variation of a BFT system and do not offer a generic methodology for any replication protocol.

Software-based CFT-to-BFT translation. In this chapter, we extensively discussed the theoretical work by Clement et al.[67]. XFT [199] enables the design of efficient protocols that tolerate non-crash faults without relying on trusted hardware. When applied to the Paxos consensus protocol, XFT uses $2f+1$ replicas to tolerate non-crash failures, provided that a majority of the replicas are correct and communicate synchronously with each other.

Similar work, PASC [72], enables CFT protocols to tolerate a subset of Byzantine faults through ASC-hardening. Specifically, ASC-hardening modifies an application by maintaining two copies of the state at each replica. It tolerates Byzantine faults under the assumption that a fault will not corrupt both copies of the state in the same way. However, PASC does not tolerate Byzantine faults that affect the entire replica (e.g., when both state copies are compromised).

To sum up, RECIPE is differentiated from previous systems in two important aspects. First, we implement a generic library to *seamlessly* transform CFT protocols for Byzantine settings. Prior works optimize existing BFT protocols, we harden the CFT protocols to provide performance and scalability, and minimize developers' effort—system designers can now *easily* implement robust replication for the untrusted Byzantine cloud infrastructure. Second, while previous research focuses on algorithmic optimizations sometimes with the help of specialized hardware, we leverage two prominent and widely adopted hardware technologies; RECIPE extends the use of TEEs to distributed settings with direct network I/O and ensures the two key properties, non-equivocation and transferable authentication, improving CFT protocol robustness. Our evaluation with the state-of-the-art BFT protocol, BFT-smart [295] showcases the impact of modern hardware in the context of BFT: we offer up to 23× better performance. Interestingly, our approach can also offer confidentiality (which is not provided by BFT protocols) with a 7×—13× speedup compared to BFT and integrity for the exchanged messages while it also allows for local (trusted and linearizable) reads.

4.9 Summary

In conclusion, we make the following contributions:

- **Hardware-assisted transformation of CFT protocols:** We present RECIPE, a generic approach for transforming CFT protocols to tolerate Byzantine failures without any modifications to the core of the protocols, e.g., states, message rounds, and complexity. We realize our approach by implementing RECIPE-lib leveraging the advances in modern hardware; we use trusted hardware to guarantee transferable authentication and non-equivocation for thwarting Byzantine errors. Further, we combine trusted hardware with direct network I/O [209, 150] for performance.
- **Generic RECIPE APIs:** We propose generic RECIPE APIs to transform the existing codebase of CFT protocols for Byzantine settings. Using RECIPE APIs, we have successfully transformed a range of leader-/leaderless-based CFT protocols enforcing different (total order/per-key) ordering semantics.
- **Confidential replication protocols:** We further show that RECIPE can offer confidentiality, that is an extra security property not provided by traditional BFT protocols.
- **Correctness analysis:** We provide a correctness analysis for the safety and liveness of our transformation of CFT protocols operating in Byzantine settings.
- **RECIPE in action:** We present an extensive evaluation of RECIPE by applying it to four CFT protocols: Chain Replication, Raft, ABD, and AllConcur. We evaluate these four protocols against the state-of-the-art BFT protocol implementations and show that RECIPE achieves up to 24× and 5.9× better throughput.

Chapter 5

TNIC: A Trusted NIC Architecture

In the previous chapter, we introduced RECIPE that offers a seamless transformation of (any) CFT replication protocol to a more robust one for Byzantine settings. While RECIPE allows system designers to offer high-performance robust replication protocols that outperform classical BFT systems, it requires low-level optimizations in the network stack and data layer (KV store) as well as expertise in systems programming; the entire protocol and network stack code needs to "live" within the TEEs. Consequently RECIPE heavily relies on TEEs programmability and security properties, complicating its widespread adoption in commodity cloud environments that are comprised of heterogeneous machines, thus, heterogeneous TEEs.

To this end, we introduce TNIC, a trusted NIC architecture for building trustworthy distributed systems deployed in heterogeneous, untrusted (Byzantine) cloud environments. TNIC realizes this goal by offering a minimal, unified, formally verified, high-performance silicon root-of-trust at the network interface level. In particular, the TNIC architecture strives for three primary design properties: (1) A host CPU-agnostic unified security architecture by providing trustworthy network-level isolation; (2) A minimalistic and verifiable TCB based on a silicon root-of-trust by providing two core properties of transferable authentication and non-equivocation; and (3) A hardware-accelerated trustworthy network stack by leveraging SmartNICs. Based on the TNIC architecture and associated network stack, we present a generic set of programming APIs and a recipe for building high-performance, trustworthy, distributed systems for Byzantine settings. We formally verify the safety and security properties of our TNIC system while demonstrating how we leverage TNIC for building four trustworthy distributed systems: A2M, BFT, Chain Replication, and PeerReview—showing the generality of our approach. Our evaluation of TNIC shows up to 6× performance improvement com-

pared to CPU-centric TEE systems while providing a low TCB with provable security properties.

5.1 Motivation

Distributed systems are integral to the third-party cloud infrastructure [13, 218, 260, 117]. While these systems in the cloud manifest in diverse forms, ranging from storage systems [82, 54, 50, 71, 108, 354, 14] and management services [140, 53] to computing frameworks [33, 35, 116], they all must be fast and remain correct when failures occur.

Unfortunately, the widespread adoption of the cloud has drastically increased the surface area of attacks and faults [122, 287, 127] that are beyond the traditional fail-stop (or crash fault) model [85]. The modern (untrusted) third-party cloud infrastructure severely suffers from arbitrary (*Byzantine*) faults [182] that can range from malicious (network) attacks to configuration errors and bugs and are capable of irreversibly disrupting the correct execution of the system [122, 287, 127, 58].

A promising solution to build trustworthy distributed systems that remain correct even when Byzantine failures occur is based on the *silicon root of trust*—specifically, the trusted execution environments (TEEs) that all major CPU manufacturers offer [74, 23, 15, 270, 149]. While the TEEs offer a (single-node) isolated trusted computing base (TCB), we have identified three core challenges (§ 5.3.3) that complicate their adoption for building trustworthy distributed systems spanning multiple nodes in Byzantine cloud environments.

First, TEEs in heterogeneous cloud environments introduce programmability and security challenges. A cloud environment offers diverse heterogeneous host-side CPUs with different TEEs (e.g., Intel SGX/TDX, AMD SEV-SNP, AWS Nitro Enclaves, Arm TrustZone/CCA, RISC-V Keystone). These heterogeneous host-side TEEs require different programming models and offer varying security properties. Therefore, they cannot (easily) provide a generic substrate for building trustworthy distributed systems. Our work overcomes this challenge by designing a *host CPU-agnostic silicon root of trust* at the network interface (NIC) level (§ 5.4). We provide a generic programming API (§ 5.6) and a *recipe* (§ 5.6.2) for building high-performance, trustworthy distributed systems (§ 5.7).

Secondly, TEEs with a large TCB are plagued with security vulnerabilities, rendering them non-verifiable. With hundreds of security bugs already uncovered [100], TEEs' large TCBs further increase their security vulnerabilities [175, 221], making it impossible to formally verify their security properties. To overcome this, we build a

minimalistic verifiable TCB (§ 5.4.1). Our TCB resides at the NIC-level hardware and is equipped with *the lower bound of security primitives*; we provide only two key security properties of non-equivocation and transferable authentication for building trustworthy distributed systems (§ 5.2.1). Since we strive for a minimal trusted interface, we can (and we did) formally verify the security properties of our TCB (§ 5.4.4).

Thirdly, TEEs report significant performance bottlenecks. TEEs syscalls execution for (network) I/O is extremely costly [339], whereas even state-of-the-art network stacks showed a lower bound of 4× slowdown [38]. We attack this challenge based on two aspects. First, we build a scalable transformation with our minimal TCB's security properties (§ 5.6.2) to transform Byzantine faults ($3f+1$) to much cheaper crash faults ($2f+1$) for tolerating f (distributed) Byzantine nodes. Secondly, we design hardware-accelerated offload of the security computation at the NIC level by extending the scope of SmartNICs with *the lower bound of security primitives* (§ 5.4) while offering kernel-bypass networking (§ 5.5).

To overcome these challenges, we present TNIC, a trusted NIC architecture for building trustworthy distributed systems deployed in Byzantine cloud environments. TNIC realizes an abstraction of trustworthy network-level isolation by building a hardware-accelerated silicon root of trust at the NIC level. Overall, TNIC follows a layered design:

- **Trusted NIC hardware architecture (§ 5.4):** We build a minimalistic, verifiable, and host-CPU-agnostic TCB at the network interface level as the key component to design trusted distributed systems for Byzantine settings. Our TCB guarantees the security properties of non-equivocation and transferable authentication that suffice to implement an efficient transformation of systems for Byzantine settings. We build TNIC on top of FPGA-based SmartNICs [317]. We formally verify the safety and security guarantees of our TNIC system protocols using Tamarin Prover [208].

- **A software-based network stack (§ 5.5) and library (§ 5.6):** Based on the TNIC architecture, we design a HW-accelerated network stack to access the trusted hardware bypassing kernel for performance. On top of TNIC's network stack, we present a networking library that exposes a simple programming model. We show *how to use* TNIC APIs to construct a generic transformation of a distributed system operating under the CFT model to target Byzantine settings.

- **Trusted distributed systems using TNIC (§ 5.7):** We build with TNIC the following (distributed) systems for Byzantine environments: Attested Append-only Memory (A2M) [65], Byzantine Fault Tolerance (BFT) [57], Chain Replication [269], and Accountability with PeerReview [126]—showing the generality of our approach.

We evaluate TNIC with a state-of-the-art software-based network stack, eRPC [162], on top of RDMA [209]/DPDK [150] with two different TEEs (Intel SGX [148] and AMD-sev [15]). Our evaluation shows that TNIC offers 3×—5× lower latency than the software-based approach with the CPU-based TEEs. For trusted distributed systems, TNIC improves throughput by up to 6× compared to their TEE-based implementations.

5.2 Design Requirements for Fast Trustworthy Distributed Systems

We first examine the design requirements for high-performance, trustworthy distributed systems for cloud environments.

5.2.1 Trustworthy Distributed Systems

Byzantine fault model. In the untrusted cloud infrastructure, arbitrary (Byzantine) faults are a frequent occurrence in the wild [122, 287, 348, 347]. To this end, system designers introduced Byzantine Fault Tolerant (BFT) systems that remain correct even under the presence of (a bounded number of) Byzantine failures [182]. While BFT accurately captures the realistic security needs in the cloud [91], it is rarely adopted in practice [294] due to its complexity and limited performance [290, 30]. In contrast, the vast majority of cloud applications operate under the fail-stop (crash fault) model, optimistically *assuming* that the entire cloud infrastructure is trusted and only fails by crashing [85]. While Crash Fault Tolerant (CFT) systems usually offer performance and scalability [106], they are ill-suited for the modern cloud as they are fundamentally incapable of ensuring safety in the presence of non-benign faults.

Security properties for BFT. We seek to offer BFT while reducing its programmability and performance overheads. As such, we materialize the *minimum* security properties required to build trustworthy systems under the BFT model [67]:

- **Transferable authentication** refers to a machine's capability to verify the original sender of a received message, even if it is forwarded by other than the original sender.

- **Non-equivocation** guarantees that a node cannot make conflicting statements to different nodes. Equivocation also manifests as network adversaries or replay attacks that send invalid messages or re-send valid but stale messages.

5.2.2 High-Performance Distributed Systems

The aforementioned two security properties are sufficient to *correctly transform* (any) CFT distributed system to operate in the BFT model [67, 73]. However, a fundamental design trade-off exists between efficiency and robustness for practical deployments in the cloud. Our work aims to resolve this tension.

Trusted hardware for BFT. System designers established trusted hardware, TEEs, as the most effective way to eliminate a system's Byzantine components [38, 328, 44, 83]. While TEEs can be used to offer BFT, prior research illustrated significant performance and architectural limitations in the context of networked systems [38, 83, 44, 328]. Based on performance and security studies [9, 8], TEEs' overheads in the heterogeneous cloud, in addition to their heterogeneity in programmability and security guarantees, are incapable of offering high-performance trusted networking under the BFT model.

SmartNICs for high-performance and BFT. We leverage the state-of-the-art hardware-level networking accelerators, i.e., SmartNICs [194, 317, 47, 49, 231, 10, 32, 59], to address the trade-off between performance and security, overcoming the limitations of TEEs. Our design choice of leveraging SmartNICs is not hypothetical; SmartNIC devices have already been launched by major cloud providers [10, 32, 59], presenting great opportunities for performance thanks to their integrated fully programmable hardware (e.g., ARM cores [47, 10, 49, 194], FPGAs [317, 292, 59]). Precisely, we rely on two promising directions: *(1)* security and network processing offloading at the NIC-level hardware and *(2)* an efficient transformation for BFT.

We extend the scope of FPGA-based SmartNICs [317, 292] by offloading an RDMA protocol implementation to the FPGA and extending its security properties, offering non-equivocation and transferable authentication. Our system not only leverages hardware acceleration for performance, but *seamlessly* offers the foundations of a scalable transformation of distributed systems for BFT. These properties also guarantee that a CFT-to-BFT transformation for state machine replication (SMR) *always exists* with the same replication factor of the original CFT system [67, 73] ($2f+1$), offering better scalability and less message complexity than the traditional BFT ($3f+1$).

5.3 Overview

5.3.1 System Overview

We propose TNIC, a trusted NIC architecture for high-performance, trustworthy distributed systems, formally guaranteeing their secure and correct execution in the het-

erogeneous Byzantine cloud infrastructure. TNIC is comprised of three layers (shown in Figure 5.1): **(1) the TNIC trusted hardware architecture** (green box) that implements trusted network operations on top of SmartNIC devices (§ 5.4), **(2) the TNIC network stack** (yellow box) that intermediates between the application layer and the TNIC hardware (§ 5.5), and **(3) the TNIC network library** (blue box) that exposes TNIC's programming APIs (§ 5.6).

Our TNIC hardware architecture implements the networking IB/RDMA protocol [262] on FPGA-based SmartNICs [317]. It extends the conventional protocol implementation with a minimal hardware module, the attestation kernel, that materializes the security properties of the non-equivocation and transferable authentication. The user-space TNIC network stack configures the TNIC device on the control path while it offers the data path as kernel-bypass device access for low-latency operations. Lastly, the TNIC network library exposes an API built on top of (reliable) one-sided RDMA primitives.

5.3.2 Threat Model

We inherit the fault and threat model from the classical BFT [58] and trusted computing domains [148]. The cloud infrastructure (machines, network, etc.) can exhibit Byzantine behavior. At the same time, it is subject to attackers that can gain control over the host CPU (e.g., the OS, VMM, etc.) and the SmartNICs (post-manufacturing). The adversary can attempt to re-program the SmartNIC, but they cannot compromise the cryptographic primitives [187, 328, 58]. The physical package, supply chain, and manufacturer of the SmartNICs are trusted [351, 165]. The TNIC implementation (bitstream) is synthesized by a trusted IP vendor with a trusted tool flow for covert channels resilience. Lastly, we do not distinguish between different types of untrusted software components. Whether the network library and stack or the application code is compromised, the node is considered faulty (Byzantine) in both cases and must conform to the BFT application system model [58].

5.3.3 Design Challenges and Key Ideas

While designing TNIC, we overcome the following challenges:

#1: Heterogeneous hardware. CPU-based TEEs in the cloud infrastructure are heterogeneous with different programmability [43, 26, 339, 280, 315, 307] and security properties [245, 203, 211] that complicate their adoption and the system's correctness [280]. Prior systems [44, 83, 328, 196] *could* not address this heterogeneity challenge as they require *homogeneous* x86 machines with SGX extensions of a specific version. This is rather unrealistic in modern heterogeneous distributed systems where

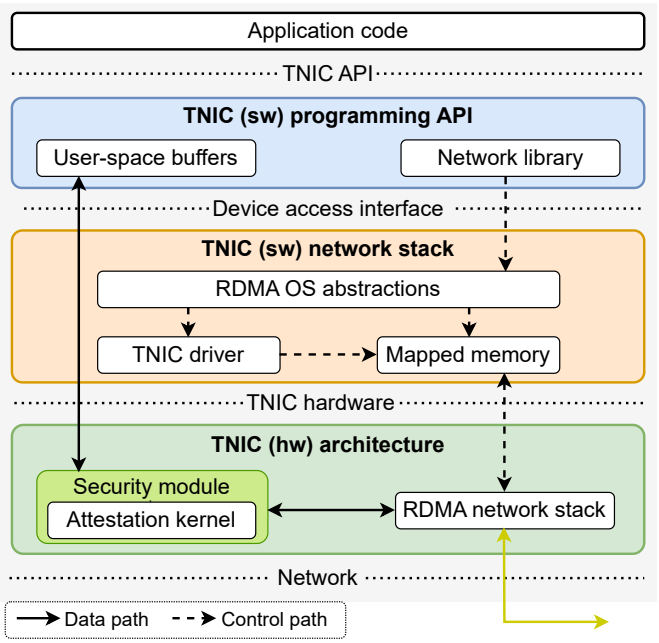

Figure 5.1: TNIC system overview.

system designers are compelled to *stitch heterogeneous TEEs together*. TEE's heterogeneity in programmability and security semantics hampers their adoption and adds complexity to ensuring the system's overall correctness.

Key idea: A host CPU-agnostic unified security architecture based on trustworthy network-level isolation. Our TNIC offers a unified and host-agnostic network-interface level isolation that guarantees the specific yet well-defined security properties of the non-equivocation and transferable authentication. TNIC also offers generic programming APIs (§ 5.6.1) that are used to *correctly* transform a wide variety of distributed systems for Byzantine settings. We demonstrate the power of TNIC with a generic transformation *recipe* (§ 5.6.2) and its usage to transform prominent distributed systems (§ 5.7).

#2: Large TCB in the TEE-based silicon root of trust. TEEs based on a *silicon root of trust* are promising for building trustworthy systems [38, 328, 44, 83]. Unfortunately, the state-of-the-art TEEs integrate a *large* TCB; for example, we calculate the TCB size of the state-of-the-art Intel TDX [149]. The TEE ports within the trusted hardware the entire Linux kernel (specifically, v5.19 [192]) and "hardens" at least 2000K lines of usable code, leading to a final TCB of 19MB. Such large TCBs have been accused of increasing the area of faults and attacks [175, 221] of commercial TEEs that are already under fire for their security vulnerabilities [27, 267, 153, 6, 152]. Importantly, TEE's large TCBs complicate their security analysis and verification, rendering their security properties *incoherent*.

Key idea: A minimal and formally verifiable silicon root-of-trust with low TCB. In our work, we advocate that a *minimalistic silicon root of trust* (TCB) at the NIC level hardware is the foundation for building verifiable, trustworthy distributed systems. In fact, TNIC builds a minimalistic and verifiable attestation kernel (§ 5.4.1) that guarantees the TNIC security properties at the SmartNIC hardware. Moreover, we have formally verified the TNIC secure hardware protocols (§ 5.4.4).

#3: Performance. TEE's overheads are significant in the context of networked systems [38, 109, 328, 83]—the foundational building block in the core of any distributed system. Prior research [38] reported 4×—8× performance degradation with even a sophisticated network stack. Others [83, 44, 328] limit performance due to the communication costs between their untrusted and TEE-based counterparts [165]. The actual performance overheads in heterogeneous distributed systems are expected to be more exacerbated [9, 8]. As such, TEEs cannot *naturally* offer high-performance, trusted networking.

Key idea: Hardware-accelerated trustworthy network stack. Our TNIC bridges the gap between performance and security with two design insights. First, TNIC attestation kernel offers the foundations to transform CFT distributed systems to BFT systems without affecting the number of participating nodes, significantly improving scalability. Second, TNIC user-space network stack (§ 5.5) bypasses the OS and offloads security and network processing to the NIC-level hardware.

5.4 Trusted NIC Hardware

Figure 5.2 shows our TNIC hardware architecture that implements trusted network operations on a SmartNIC device. In this section, we introduce two key components: *(i)* the attestation kernel that guarantees the non-equivocation and transferable authentication properties over the untrusted network (§ 5.4.1) and *(ii)* the RoCE protocol kernel that implements the RDMA protocol including transport and network layers (§ 5.4.2). We also introduce a bootstrapping and a remote attestation protocol for TNIC (§ 5.4.3) and formally verify them (§ 5.4.4).

5.4.1 NIC Attestation Kernel

The attestation kernel *shields* network messages to ensure the properties of non-equivocation and transferable authentication by generating *attestations* for transmitted messages. As shown in Figure 5.2, the attestation kernel resides in the data pipeline between the RoCE protocol kernel that transmits/receives network messages and the PCIe DMA that

transfers data from/to the host memory. The kernel processes the messages as they *flow* from the memory to the network and vice versa to optimize throughput.

Hardware design. The attestation kernel is comprised of three components that represent its state and functionality: the HMAC component that generates the message authentication codes (MAC), the Keystore that stores the keys used by the HMAC module, and the Counters store that keeps the message's latest sent and received timestamp.

The system designer initializes each TNIC device during bootstrapping with a unique identifier (ID) and a shared secret key—ideally, one shared key for each communication channel or *session* that is identified by an id (`c_id`)—stored in static memory (Keystore). The configurations and secrets, e.g., `c_ids` and their matching keys are shared between TNIC instances securely after the device has been attested successfully (see § 5.4.3) and remain unknown to the untrusted parties.

TNIC holds two counters per session in the Counters store: `send_cnts`, which holds the count of the number of the messages sent, and `recv_cnts`, which holds the latest seen counter value for each session. The counters represent the messages' timestamp and are increased monotonically and deterministically after every send and receive operation to ensure that messages are assigned to unique counters for non-equivocation. Consequently, lost, re-ordered, or doubly executed can be detected by TNIC.

Algorithm. Algorithm 3 shows the functionality of the attestation kernel. The module implements two core functions: `Attest()`, which *authenticates* a message with a MAC, and `Verify()`, which verifies the MAC of a received message. Throughout this chapter we also use the term *a*ttestation to refer to a message's MAC. The message transmission invokes `Attest()`, and likewise, the reception invokes `Verify()`.

Upon transmission, as shown in Figure 5.2, the message is first forwarded to the attestation kernel. The attestation kernel executes `Attest()` and generates an *attested* message comprised of the message data and its attestation certificate α. The function calculates α as the HMAC of the message concatenated with the counter `send_cnt` and the device ID (for transferable authentication[1]) with the `key` for that connection (Algorithm 3: L4). It also increases the next available counter for subsequent future messages (Algorithm 3: L2). The function forwards the message with its α to the RoCE protocol kernel (Algorithm 3: L4).

Upon reception, the received message passes through the attestation kernel for verification before it is delivered to the application. Specifically, `Verify()` checks the authenticity and the integrity of the received message by re-calculating the *expected*

[1]The seminal paper of Clement et al. [67] suggested the use of digital signatures to ensure transferable authentication for the messages. However, as digital signatures are expensive, we ensure transferable authentication by assigning to each message a unique per-device ID. Our approach is similar to what practical systems are suggesting [187].

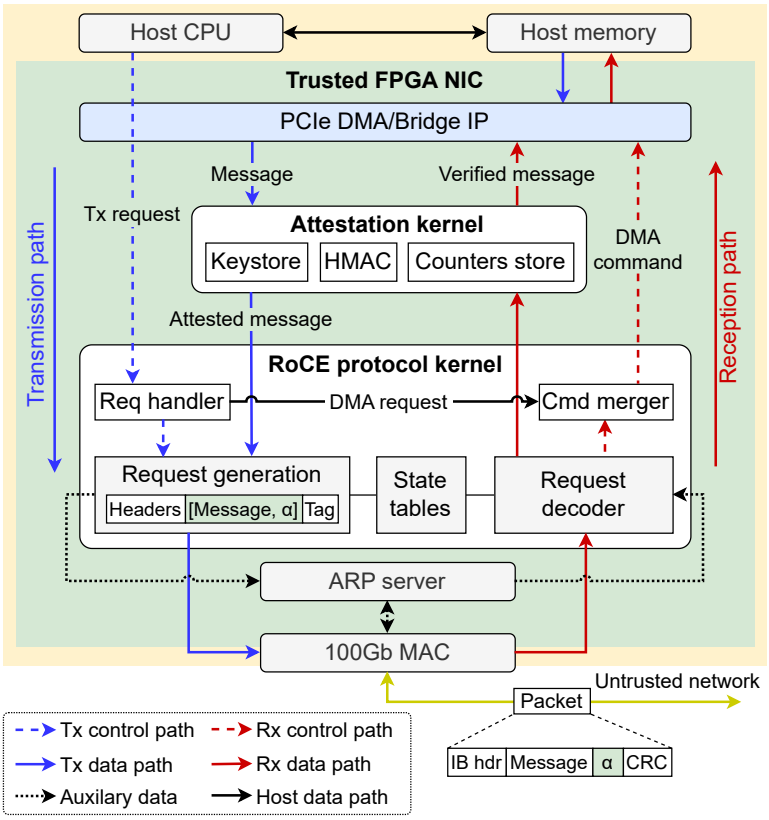

Figure 5.2: TNIC hardware architecture.

attestation α' based on the message payload and comparing it with the received attestation α of the message (Algorithm 3: L7—8). The verification also ensures that the received counter matches the expected counter for that *session*—identified by `c_id` in Algorithm 3—to ensure *continuity*, i.e., that messages are received in the order they are sent and the receiver does not miss any past messages (Algorithm 3: L8).

5.4.2 RoCE Protocol Kernel

The RoCE protocol kernel implements a reliable transport service on top of the Infiniband (IB) Transport Protocol with UDP/IPv4 layers (RoCE v2) [144] (transport and network layers). As shown in Figure 5.2, to transmit data, the `Req handler` module in the RoCE kernel receives the request opcode (`metadata`) issued by the host. The message is fetched through the PCIe DMA engine and passes through the attestation kernel. The request opcode and the attested message are forwarded to the `Request generation` module that constructs a network packet.

Upon receiving a message from the network, the RoCE kernel parses the packet header and updates the protocol state (stored in the State tables). The attested message is then forwarded to the attestation kernel. The message is delivered to the application's

Algorithm 3: `Attest()` and `Verify()` functions.

```
1  function Attest(c_id, msg) {
2      cnt ← send_cnts[c_id]++;
3      α ← hmac(keys[c_id], msg||ID||cnt);
4      return α||msg||ID||cnt;
5  }
6  function Verify(c_id, α||msg||ID||cnt) {
7      α' ← hmac(keys[c_id], msg||ID||cnt);
8      if (α' = α && cnt = recv_cnts[c_id]++ )
9          return (α||msg||cnt);
10     assert(False);
11 }
```

(host) memory upon successful verification.

Hardware design. The RoCE protocol kernel is also connected to a 100Gb MAC IP and an ARP server IP.

100Gb MAC. The 100Gb MAC kernel implements the link layer connecting TNIC to the network fabric over a 100G Ethernet Subsystem [318]. The kernel also exposes two interfaces for transmitting (Tx) and receiving (Rx) network packets.

ARP server. The ARP server has a lookup table containing MAC and IP address correspondences. Right before the transmission, the RDMA packets at the `Request generation` first pass through a MAC and IP encoding phase, where the `Request generation` module extracts the remote MAC address from the lookup table in the ARP server.

IB protocol. The RoCE protocol kernel implements the reliable version of the IB protocol. Similarly to its original specification [262], the kernel implements State tables to store protocol queues (e.g., receive/send/completion queues) as well as important metadata, i.e., packet sequence numbers (PSNs), message sequence numbers (MSNs), and a Retransmission Timer.

Dataflow. The transmission path is shown with the blue-colored axes in Figure 5.2. The `Req handler` receives requests issued by the host and initializes a DMA command to fetch the payload data from the host memory to the attestation kernel. Once the attestation kernel forwards the attested message to the `Req handler`, the module passes the message from several states to append the appropriate headers `IB hdr` along with metadata (e.g., RDMA op-code, PSN, QP number). The last part of the processing generates and appends UDP/IP headers (e.g., IP address, UDP port, and packet length). The message is then forwarded to the 100Gb MAC module.

Similarly, the reception path is shown with the red-colored axes in Figure 5.2. The `Request decoder` extracts the headers, metadata, and attested message. The message is forwarded to the attestation kernel for verification and finally copied to the host

memory.

The message format in TNIC follows the original RDMA specification [262]; the difference between our TNIC and the original RDMA messages is that the attestation kernel *extends* the payload by appending a 64B attestation α[2] and the metadata. The metadata includes a 4B id for the session id of the sender, a 4B ID for the device id (unique per device), and the appropriate `send_cnt`. This payload extension is negligible and does not harm the network throughput. TNIC also offers support for the original RDMA operations that bypass the attestation kernel. We explain TNIC APIs in section § 5.6.1.

5.4.3 Remote Attestation Protocol

We design a remote attestation protocol to ensure that the TNIC device is genuine and the TNIC bitstream and secrets are flashed securely in the device. Although TNIC's attestation protocol has not been implemented, we have verified the security properties of the protocol in the symbolic model (sections § 5.4.4 and § 5.9).

Design assumptions. We base our design on a NIC Controller hardware component that drives the device initialization with no access to confidential information as in [351]. The Controller can be implemented as a soft CPU [215, 232, 351] while after TNIC's initialization, it monitors JTAG/ICAP interfaces to prevent physical attacks [281], ensuring that the bitstream is not modified before use.

Bootstrapping. The Manufacturer (at the construction) burns a secret, unique to the device, key $\texttt{HW}_{key}$. The System designer, who builds the distributed BFT application, shares the configuration and secrets (e.g., session ids, keys, network IPs of the participating devices) with the IP vendor (who implements and provide the bitstreams of TNIC) and instructs the cloud provider to load the (encrypted) FPGA firmware which is then decrypted with the $\texttt{HW}_{key}$. The firmware loads the controller binary $\texttt{Ctrl}_{bin}$, generates a key pair $\texttt{Ctrl}_{pub,priv}$ for the specific device and binary (which we assume they are not compromised or leaked to untrusted parties), and signs the measurement, i.e., the cryptographic hash, of the $\texttt{Ctrl}_{bin}$ and the $\texttt{Ctrl}_{pub}$ with the $\texttt{HW}_{key}$ ($\texttt{Ctrl}_{bin}\texttt{cert}$).

Remote attestation. Figure 5.3 shows our suggested remote attestation design for TNIC. The IP vendor sends a random nonce `n` for freshness to the Controller. The IP vendor's public key $\texttt{IPVendor}_{pub}$ is embedded into the $\texttt{Ctrl}_{bin}$ and as such is known to the Controller. The Controller generates a certificate `cert` over the $\texttt{Ctrl}_{bin}\texttt{cert}$ and `n` (2) which signs with $\texttt{Ctrl}_{pub}$ and sends it to the IP vendor (3).

[2]While 16B are typically sufficient for a MAC, TNIC reserves 64B for the MAC because it builds on top 512-bit AXI4 Stream interfaces for the data path. This is an implementation detail rather than a TNIC restriction.

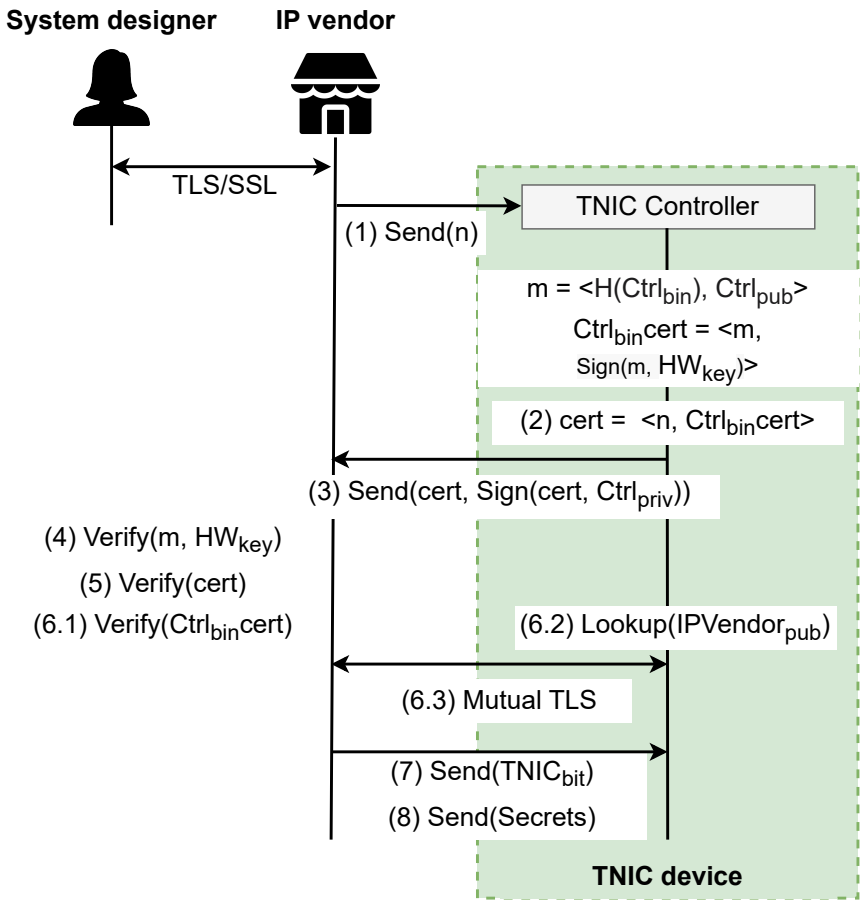

Figure 5.3: TNIC remote attestation protocol.

The IP vendor verifies the authenticity of the `cert` (4)—(5) and establishes a mutually authenticated TLS connection with the Controller. First, the vendor verifies the authenticity of `m` with the $\texttt{HW}_{key}$, ensuring that a genuine $\texttt{Ctrl}_{bin}$ and a genuine device has signed `m` (4). As such, the vendor ensures that the $\texttt{Ctrl}_{bin}$ runs in a genuine TNIC device by verifying that the measurement of the $\texttt{Ctrl}_{bin}$ has been signed with an appropriate device key installed by the Manufacturer. Lastly, the vendor verifies the nonce `n` and `cert` to ensure freshness[3] (with the $\texttt{Ctrl}_{pub}$ included in `m`).

Now, a mutually authenticated TLS connection is established; the IP vendor verifies the authenticity by checking for the desired $\texttt{Ctrl}_{pub}$ and the Controller checks for it's embedded $\texttt{IPVendor}_{pub}$ (6.1)—(6.3). Once the mTLS session is established the IP Vendor sends the Controller the configurations and secrets (provided by the System designer) and the TNIC bitstream, $\texttt{TNIC}_{bit}$ (7)—(8) that are protected by the mTLS guarantees.

[3]We assume that the $\texttt{Ctrl}_{priv}$ is not compromised.

5.4.4 Formal Verification of TNIC Protocols

We formally verify the security and safety properties of TNIC hardware with Tamarin [208], a security protocol verification tool that analyzes symbolic models of protocols specified as multiset rewriting systems. We verify TNIC's bootstrapping and the remote attestation protocol that provides a formal model to argue about non-equivocation and transferable authentication. We next present the verified security properties of TNIC; proofs are available in § 5.9.

Remote attestation. We model the bootstrapping and remote attestation protocols based on Figure 5.3. We define lemmas to ensure the secrecy of the private information involved in the protocol and the main attestation lemma, which holds if and only if: once the last step of the remote attestation protocol is completed, the TNIC device is in a valid, expected state.

Transferable authentication. We extend the model with rules for the network operations that execute `Attest()` and `Verify()` (Algorithm 3) upon the message transmission and reception, respectively. The model extension allows us to reason about transferable authentication by defining an additional lemma: any accepted message was sent by an authentic TNIC device in a valid configuration.

Non-equivocation. We further extend the model by three lemmas that help to reason about non-equivocation as follows: for any message that is accepted, it holds that *(i)* there is no message that was sent before but not accepted, *(ii)* there is no message that was sent after, but accepted before this one, *(iii)* this message has not been accepted before.

To sum up, Tamarin successfully shows that there is no sequence of transitions that leads to any state where our lemmas are violated. Thus, the attestation and transferable authentication lemmas hold for our model, and the counters behave as expected to ensure non-equivocation.

5.5 TNIC Network Stack

We build a software TNIC system network stack that operates as the *middle layer* between the TNIC programming APIs (see § 5.6.1) and the hardware implementation of TNIC. Figure 5.4 shows an overview of the network stack design that is comprised of two core components: *(1)* the TNIC driver and mapped REGs pages that are responsible for initializing the device and configuring host—device communication and *(2)* the RDMA OS abstractions that execute networking operations.

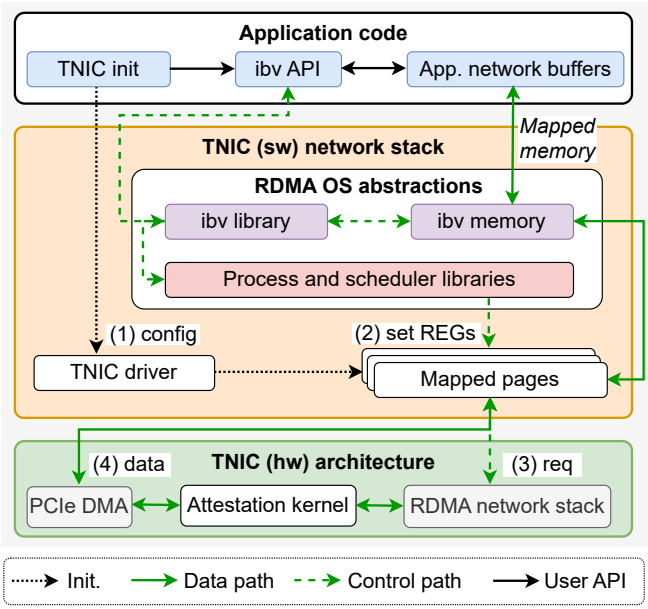

Figure 5.4: TNIC network system stack.

5.5.1 TNIC Driver and Mapped REGs Pages

The TNIC driver is invoked at the device initialization, before the remote attestation protocol (§ 5.4.3), to configure the hardware with its static configuration (the device MAC address, the device QSFP port, and the network IP used by the application).

Additionally, the driver enables kernel-bypass networking—similar to the original (user-space) RDMA protocol—by mapping the TNIC device to a user-space addresses range, the Mapped REGs pages. In our TNIC, we reserve one Mapped REGs page at the page granularity of our system for each connected device that is represented as pseudo-devices in `/dev/fpga<ID>`. Read and write access to the pseudo-device is equal to accessing the control and status registers of the FPGA. As such, the applications can directly interact with the control path of the TNIC hardware at low latency while bypassing the host OS.

5.5.2 RDMA OS Abstractions

The RDMA OS abstractions are a user-space library that enables the networking operations in the TNIC hardware, bypassing the OS kernel for performance. As shown in Figure 5.4, invocations of the TNIC programming API calls into the RDMA OS library are comprised of two parts: *the network (RDMA) library* (colored in purple) that implements the software part of the RDMA protocol and *the OS library* (colored in red) that schedules the TNIC requests.

Network (RDMA) library. The network (RDMA) library includes all the logic and data

(e.g., Tx/Rx queues per connection, local and remote memory addresses, RDMA keys that denote memory access permissions) required to implement the RDMA protocol. It executes the application's networking operations by posting the requests to the hardware. More specifically, it creates an internal representation of the request and the associated data and metadata (i.e., request opcode, remote IP, source/destination addresses, data length, etc.) and writes them into specific offsets in the REGs pages to update the control registers of the TNIC hardware.

As shown in Figure 5.4, the transmission and reception of requests and responses mandate the allocation of application network buffers. We adopt memory management similar to that in widely used user-space networking libraries [162, 150, 209]. Importantly, the network buffers need to be mapped to a specific TNIC-memory, called the ibv memory. The ibv memory area is allocated at the connection creation in the `huge page` area by the application through the ibv library. It resides within the application's address space with full read/write permissions and is eligible for DMA transfers.

OS library. The TNIC-OS library is a lightweight user-space library responsible for scheduling the requests and ensuring isolated access to the mapped REG pages. For performance, the TNIC data path eliminates unnecessary data copies throughout the network stack; the data to be sent is directly fetched by the hardware through DMA transfers. The OS library creates a TNIC-process object to represent each TNIC device. This TNIC-process in TNIC is not a separate scheduling entity (i.e., a thread as in classical OSes). In contrast, it is an object handle, exposed to the ibv library but managed by the TNIC-OS library that acquires locks on the respective REG pages to ensure isolated access to the TNIC hardware.

5.6 TNIC Network Library

We present TNIC's programming APIs (§ 5.6.1), and a generic recipe to transform existing distributed systems (§ 5.6.2).

5.6.1 Programming APIs

TNIC implements a programming API (Table 5.1) that resembles the traditional RDMA programming API [162] used in modern distributed systems[106, 88, 168, 163, 123, 200, 222]. We further extend the security semantics, offering the properties of non-equivocation and transferable authentication (§ 5.2.1).

Initialization APIs. The TNIC application first needs to configure the TNIC system to establish peer-to-peer RDMA connections. The application creates one ibv struct for each connection with `ibv_qp_conn()`, which sets up and stores the queue pair, the

Initialization APIs	
`ibv_qp_conn()`	Creates an ibv struct
`alloc_mem()`	Allocates host ibv memory
`init_lqueue()`	Registers local memory to TNIC
`ibv_sync()`	Exchanges ibv memory addresses
Network APIs	
`local_send/verify()`	Generates/verifies attested messages
`auth_send()`	Transmits an attested message
`poll()`	Polls for incoming messages
`rem_read/write()`	Fetches/writes remote memory

Table 5.1: TNIC programming APIs.

local and remote IP addresses, device UDP ports, and others. The application also invokes `alloc_mem()` to allocate the ibv memory and then register the ibv memory to the TNIC hardware. Lastly, the application synchronizes with the remote machine using `ibv_sync()` to exchange necessary data (e.g., ibv memory address, queue pair numbers).

Network APIs. TNIC executes trusted one-sided, (unreliable) *ordered* RDMA based on the classical one-sided RDMA over Reliable Connection (RC), i.e., a FIFO ordering (per connection), similar to TCP/IP networking. TNIC `auth_send()` can offer reliability and FIFO-ordering guarantees between two nodes in the absence of network adversaries. However, in case of network attacks TNIC detects lost or compromised messages but does not "automatically" re-transmits them.

TNIC offers `auth_send()` to send an attested message with RDMA reliable writes. We support classical RDMA operations for reads and writes: `rem_read()` and `rem_write()`. The remote side runs `poll()` to fetch the number of completed operations; `poll()` is updated only when the message verification succeeds at the TNIC hardware. We offer local operations for generating and verifying attested messages within a single-node setup: `local_send()` and `local_verify()`.

TNIC does not support a hardware-assisted multicast. To implement an equivocation-free multi-casting, system developers can rely on the TNIC's API to generate and verify attested messages locally, e.g., `local_send()` and `local_verify()`. Each multicast group is a associated with a *s*ession identifier, the `c_id`. Nodes can be part of multiple multicast groups each of which is identified by its `c_id`. As such, counters remain consistent across all nodes within a multicast group. As discussed in Algorithm 3,

each node's TNIC attestation kernel rely on this `c_id` to correctly find and check the message's assigned counters. System developers can implement equivocation-free multicast by uni-casting the same attested message, i.e., the sender must generate an attested message with `local_send()` as in [187] and send this attested message to the multicast group through peer-to-peer connections, for example, by using the RDMA's (classical) write operation. Recipients can verify the received attested message by calling into `local_verify()`. The `local_verify()` as before checks for the authenticity of the message. The `send_cnts` and `recv_cnts` of the multicast group are identified by the same `c_id` and as such, equivocation can be prevented; different messages will be assigned different `send_cnts` [187].

5.6.2 A Generic Transformation Recipe

Transformation properties. We show how to use TNIC APIs to transform an existing (CFT) distributed system into one that targets Byzantine settings. Our transformation is defined as wrapper functions on top of the send and receive operations [67]. For safety, our transformation needs to meet the following properties to provide the same guarantees expected by the original CFT system [67, 133, 134]:

- **Safety.** If a correct receiver receives a message m from a correct sender S, then S must have sent with m.
- **Integrity.** If a correct receiver receives and delivers a message m, then m is a *valid* message.

```
1  void send(P_id, char[] msg) {
2      state = hash(my_state);
3      tx_msg = msg || state || receiver_state;
4      auth_send(follower, tx_msg);
5  }
6
7  void recv(recv_msg) {
8      auto [att, msg || state || receiver_state] = deliver();
9      [msg, cnt] = verify_msg(msg);
10     verify_sender_state(state);
11     local_verify(receiver_state);
12     verify_system_view(receiver_state); apply(msg);
13 }
```

Listing 5.1: Generic send and recv wrapper functions using TNIC. TNIC additions are highlighted in orange.

Listing 5.1 shows our proposed `send` (L1-5) and `recv` (L7-13) operations, assuming a two-node scenario where the first node (sender) receives client requests and forwards them to the second node (receiver). For transmission, the sender sends the client message `msg`, its current state (e.g., the sender's action to the `msg`), and the receiver's previous state (known to the sender). The receiver's state is optional depending on the consistency guarantees of the derived system and can be used to ensure that both nodes have the same system view.

Upon receiving a valid message (L8-9), the receiver *simulates* the sender's state to verify that the sender's action to the request is as expected (L10). The receiver also ensures that it does not lag, and both nodes have the same "view" of the system inputs by verifying that the sender has *seen* the receiver's latest state (e.g., message) (L11-12).

Our generic transformation recipe satisfies the requirements listed above. First, TNIC's transferable authentication property directly satisfies the safety requirement. A faulty node cannot impersonate a correct node[4]; if TNIC validates and delivers a message m from a sender, the sender must have sent m. TNIC's network operations ensure liveness properties between correct nodes. In any case, liveness can be ensured by re-trying the message transmission.

Second, our transformation satisfies the integrity property. The property requires the receiver to ensure that a sender's message derives from a valid (correct) execution scenario of the business logic. Our transformation with the transferable authentication mechanism allows the receiver to simulate the sender's output. Consequently, correct receivers can verify the sender's state and that their received message (from the sender) derives from a correct execution scenario of the business logic, namely, that the sender obeyed to the protocol specification.

Consistency property for replication. Our transformation further needs to meet the consistency property [67]: If correct receivers R_1 and R_2 receive valid messages m_i and m_j respectively from sender S, then either (a) Bpg_i is a prefix of Bpg_j , (b) Bpg_j is a prefix of Bpg_i, or (c) $Bpg_i = Bpg_j$ (where Bpg_x is the process behavior, i.e., the execution history of the protocol that supports the validity of the message m_x. Namely, the message derives from a correct execution scenario ensured by the state machine replication and the consensus protocol).

The consistency requirement is enforced through the TNIC's non-equivocation primitive that assigns a (unique) monotonic sequence number to each outgoing message, enforcing a total order on the sender's outgoing messages. Along with the integrity requirement, the total order suffices for consistency. Importantly, TNIC ensures that

[4]Recall that each device is uniquely identified by an ID and the keys to authenticate the messages are only known to the TNIC and cannot be read by the untrusted CPU.

correct receivers cannot miss any past messages. Following this, two followers that receive from the same sender (using the equivocation-free multicast discussed in § 5.8.2) follow the same transition (execution) path.

5.7 Trusted Distributed Systems

Using TNIC as the foundation, we transform the following four distributed systems to operate in Byzantine environments. The implementation details are covered in § 5.10.

Attested Append-Only Memory (A2M). We design an Attested Append-Only Memory (A2M) [65] leveraging TNIC, which can be used to shield and optimize various systems [3, 58, 189, 76]. The original A2M, and hence our implementation over TNIC, builds append-only (trusted) logs, associating each entry with a monotonically increasing sequence number to combat equivocation. While A2M has a large TCB and stores the log within the TEE, our implementation has only a minimal TCB in hardware and it can robustly store the log in the untrusted host memory, improving memory efficiency [187].

As in the original A2M, we build the `append` and `lookup` operations. The `append` calls into TNIC to generate an attestation for the log entry while associating it with a monotonically increased sequence number (`sent_cnt`). The sequence number denotes the entry's position in the log. The `lookup` operation retrieves entries locally without verification.

Byzantine Fault Tolerance (BFT). We design a Byzantine Fault-Tolerant protocol (BFT) using TNIC. The protocol builds a replicated counter as a foundational service for various systems [97, 330, 160, 276, 86]. Our system model considers a network of replicas with at most f Byzantine replicas out of $N = 2f + 1$ total replicas. One replica serves as the leader, and the others act as followers. The system prevents equivocation through TNIC, which enforces and validates the ordering of messages. This reduces the number of replicas required and the message complexity compared to the classical BFT ($3f + 1$).

Clients send increment counter requests to the leader, who performs the requests and broadcasts the change along with a *proof of execution* (PoE) message to followers. The proof of execution is a TNIC-attested message with the original client's request, the leader's counter value, and metadata. The followers leverage their local state machine to detect a faulty leader (or follower) [134]. Subsequently, if and only if a follower has not applied the message before, it applies the incremented counter value to its state machine before forwarding its own PoE message to all other replicas and replying to the client. A quorum of at least $f + 1$ identical messages from different replicas guarantees a correctly committed result for the client.

Chain Replication (CR). We design a Byzantine CR system [323] using TNIC as the replication layer of a Key-Value store. As in the CFT version of CR, the replicas, e.g., head, middle, and tail, are connected in a chained fashion.

Clients execute requests by forwarding them to the head. The head orders and executes the request, creating his own *proof of execution message* (PoE), which is sent along the chain. The PoE consists of the original request and the head's output that TNIC attests. Each node in the chain verifies the previous nodes PoE, executes the request, and creates its own PoE, which consists of the last PoE and the node's output.

Unlike the CFT CR, all operations must traverse the chain as local operations in the tail cannot be trusted (e.g., local reads). Replicas reply to clients with their output after forwarding their PoE message. Clients wait for identical replies from all chained nodes.

Accountability (PeerReview). Lastly, we design an accountability system with TNIC based on the PeerReview system [126] to *detect* malicious actions in large deployments [302, 220].

We detect faults impacting the system's network messages logged into the participant's tamper-evident log. We frame the protocol within an overlay multicast protocol for streaming systems where the nodes are organized in a tree topology. Witnesses assigned to each node audit the node's log to detect faults. The witnesses replay the log entries, comparing them with a reference deterministic implementation to identify inconsistencies. Our TNIC prevents equivocation at NIC hardware efficiently, which eliminates the expensive all-to-all communication of the original PeerReview that does not use trusted hardware [187].

5.8 Evaluation

We evaluate TNIC across three dimensions: *(i)* hardware (§ 5.8.1), *(ii)* network stack (§ 5.8.2) and *(iii)* distributed systems (§ 5.8.3).

Implementation. We implement our prototype of TNIC extending the Coyote [77] codebase on top of Alveo U280 [317]. We build the attestation kernel based on the HMAC module provided by Xilinx with the SHA-384 as the hashing algorithm [342]. We adopt 512-bit AXI4 Stream interfaces for the data paths and AXI4 memory-mapped interfaces for the control paths. For the data transfers, TNIC builds on top of an XDMA IP [340, 341] that enables DMA over PCIe. The 100Gb MAC is implemented with a CMAC IP [318] that exposes two 512-bit AXI4-Stream interfaces to the RoCE protocol kernel for the transmitting (Tx) and receiving (Rx) network paths.

Evaluation setup. We perform our experiments on a real hardware testbed using two clusters: AMD-FPGA Cluster and Intel Cluster. AMD-FPGA Cluster consists of two ma-

System	(host) TEE-free	Tamper-proof
SSL-lib	Yes	No
SSL-server/Intel-x86*/AMD	Yes	No
SGX/AMD-sev	No	Yes
TNIC	Yes	Yes

Table 5.2: Host-sided baselines and TNIC. (*) We use the term SSL-server for this run unless stated otherwise.

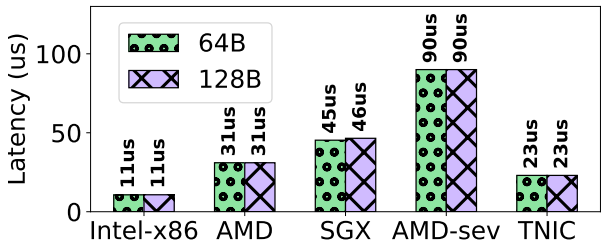

Figure 5.5: `Attest` function latency.

chines powered by AMD EPYC 7413 (24 cores, 1.5 GHz) and 251.74 GiB memory. Each machine also has two Alveo U280 cards [317] that are connected through 100 Gbps QSFP28 ports. Intel Cluster consists of three machines powered by Intel(R) Core(TM) i9-9900K (8 cores, 3.2 GHz) with 64 GiB memory and three Intel Corporation Ethernet Controllers (XL710).

5.8.1 Hardware Evaluation: T-FPGA

Baselines. We evaluate the performance of `Attest()` of the TNIC's attestation kernel (§ 5.4.1) compared with four host-sided systems shown in Table 5.2. For these host-sided versions, we build OpenSSL servers that run natively or within a TEE. The servers attest and forward network messages to the host application. We use the terms Intel-x86 and AMD for a native run of the server process (SSL-server) and SGX and AMD-sev

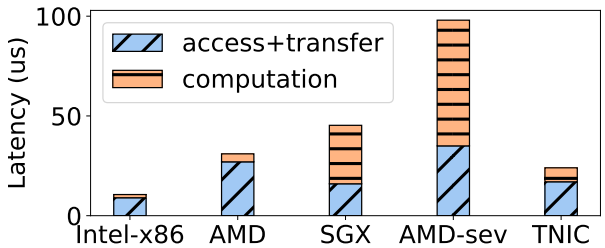

Figure 5.6: `Attest` latency breakdown.

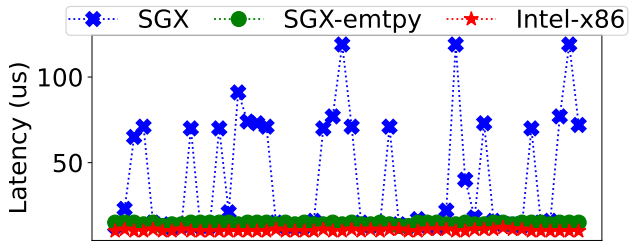

Figure 5.7: Latency over time (SGX).

for their tamper-proof versions within a TEE. The server and host process run in the same machine to eliminate network latency and communicate through TCP sockets. We implement SGX using the SCONE framework [26] while AMD-sev runs in a QEMU VM using the official VM image [16]. In addition, we build (non-temper-proof) SSL-lib, which integrates the `Attest` function as a library.

Methodology and experiments. We use Vitis XRT v2022.2 and the shell `xilinx_u280_gen3x16_xdma_base_1` for the stand-alone evaluation of the TNIC attestation kernel: synchronous data transfers between the host and device (U280). We measure and report the average latency and communication costs by executing an empty function body of `Attest()`.

Results. Figure 5.5 shows the average latency of `Attest()` based on the HMAC algorithm for 64B and 128B data inputs. The latency of `Verify()` is similar, and as such, it is omitted. Our TNIC achieves performance in the microseconds range (23 us) and outperforms its equivalent TEE-based competitors at least by a factor of 2. Importantly, TNIC is approximately 1.2× faster than AMD, which is not tamper-proof.

Figure 5.6 shows the latency breakdown of `Attest()`. Accessing the TNIC device and TEEs can be expensive, ranging from 30% to 90% of the total operation latency among the systems. Regarding TNIC, the transfer time (16us) accounts for 70% of the execution time. We expect that TNIC effectively eliminates this cost by enabling asynchronous (user-space) DMA data transfers. Regarding the TEE-based systems (SGX, AMD-sev), the communication and system call execution costs account for up to 40% of the total execution. To our surprise, this implies that the HMAC computation within any of the two TEEs experiences more than 30× overheads compared to its native run. To analyze TEEs' behavior, we instrument the client's code to measure the operations' individual latency at various periods of time during the experiment accurately.

Figure 5.7 illustrates the individual operation latency, where SGX-empty indicates SGX without HMAC computation. As shown in Figure 5.7, the HMAC execution within the TEE often experiences huge latency spikes. We attribute this behavior to the scheduling effects and asynchronous exitless system calls involved by SCONE [26]. We

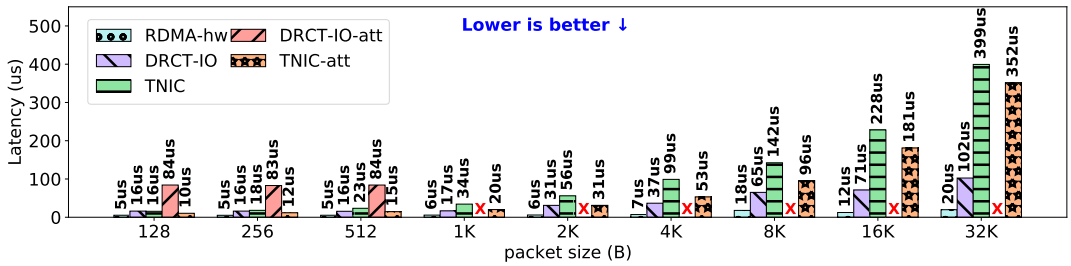

Figure 5.8: Latency of send operations across five competitive network stacks with various security properties.

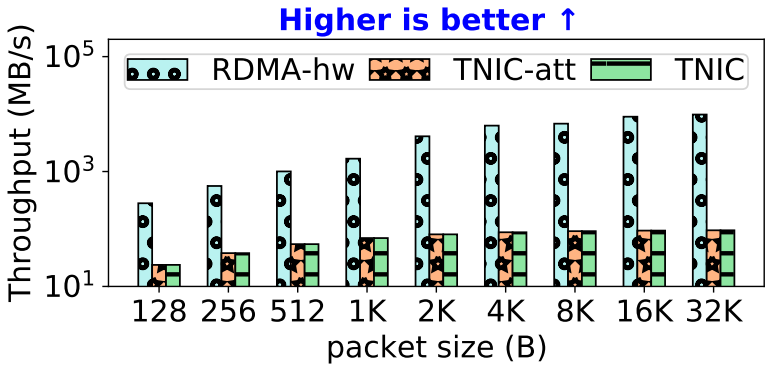

Figure 5.9: Throughput of send operations across the three selected network stacks.

observe similar variations for AMD with latencies spiking up to 200–500 us.

5.8.2 Software Evaluation: TNIC Network Stack

Baselines. To evaluate the TNIC performance, we discuss (1) the effectiveness of offloading the network stack to the TNIC hardware and (2) the overheads incurred by the CFT systems transformation for the BFT model. We compare TNIC across four other software/hardware network stacks with different security properties as follows: *(i)* RDMA-hw, an untrusted RoCE protocol on FPGAs, *(ii)* DRCT-IO (direct I/O), untrusted software-based kernel-bypass stack, *(iii)* DRCT-IO-att, altered DRCT-IO that offers trust by sending attested messages but does not verify them, and *(iv)* TNIC-att, altered TNIC that similarly sends attested messages without verification. We build *(i)* RDMA-hw on top of Coyote [77] network stack similarly to TNIC. For *(ii) (iii)* DRCT-IOs, we base our design on eRPC [162] with DPDK [150] that offers similar reliability guarantees with RDMA-hw. The TNIC-att and the DRCT-IO-att network stacks are evaluated to quantify solely the overhead of the `Attest()` function. The hardware network stacks run on AMD-FPGA Cluster, whereas the rest run on Intel Cluster.

Methodology and experiments. Our experiments measure the latency and throughput for respective network stacks, which run through a single-threaded client-server implementation. For the latency measurement, the client sends one operation at a time, whereas for the throughput measurement, one client can have multiple outstanding operations.

Results. Figure 5.8 and 5.9 show the latency and throughput of the send operation with various packet sizes. First, regarding (1) the effectiveness of network stack offloading, RDMA-hw is 3×—5× faster than DRCT-IO, which indicates that the network offloading boosts performance. Although DRCT-IO offers minimal latency (16-16.6us) for small packet sizes up to 1 KiB due to its zero-copy transmission/reception optimizations [162], they are only effective for up to 1460B (MTU is 1500B, but 40B are reserved for metadata), and RDMA-hw still achieves 3× lower latency (5-5.5us). For bigger data transfers, the RDMA-hw latency increases steadily up to 19 us, whereas DRCT-IO does not scale well with latencies up to 100us.

Second, regarding (2) the TNIC performance overhead, TNIC offers trusted networking with 3×—20× higher latencies than the untrusted RDMA-hw. The latency increase stems from the HMAC calculation of the TNIC hardware. As this algorithm fundamentally cannot be parallelized, the higher the message size, the higher the latency our TNIC incurs. More specifically, for packet sizes less than 1 KiB, doubling the packet size in TNIC results in a 13%—20% increment in the overall latency. For packet sizes bigger or equal to 1 KiB, doubling the packet size increases the latency by 30%—40%. Compared to DRCT-IO-att (82us), TNIC is up to 5.6× faster. Importantly, DRCT-IO-att reports extreme latencies (2000us or more) for packet sizes larger than 521B, which are omitted to avoid plot distortion. We attribute these latencies to the scheduling effects of SCONE [26].

5.8.3 Distributed Systems Evaluation

We implement the four systems of § 5.7 with TNIC in a three-node setup ($N = 3$) except for the single-node A2M system.

Methodology and experiments. We execute all four of our codebases on Intel Cluster, utilizing all its three servers (as the minimum required setup capable of withstanding a single failure, $N = 2f + 1$, where $f = 1$). Due to our limited resources, we cannot install Alveo U280 cards on all these servers. Instead, we build our codebase using the DRCT-IO stack (detailed in § 5.8.2) and inject busy waits to all three servers to emulate the delays incurred by TNIC for generating and verifying attested messages.

We evaluate each codebase using five systems that generate and verify the attestations: *(i)* SSL-lib (no tamper-proof), *(ii)* SSL-server (no tamper-proof), *(iii)* SGX, *(iv)*

	Throughput (Op/s)		Latency (us)	
System	append	lookup	append	lookup
SSL-lib	790K	256M	1.26	0.0039
SGX-lib	380K	3.8M	2.6	0.26
AMD-sev	30K	263M	32.37	0.0038
TNIC	158K	257M	6.34	0.0039

Figure 5.10: Throughput and latency of A2M.

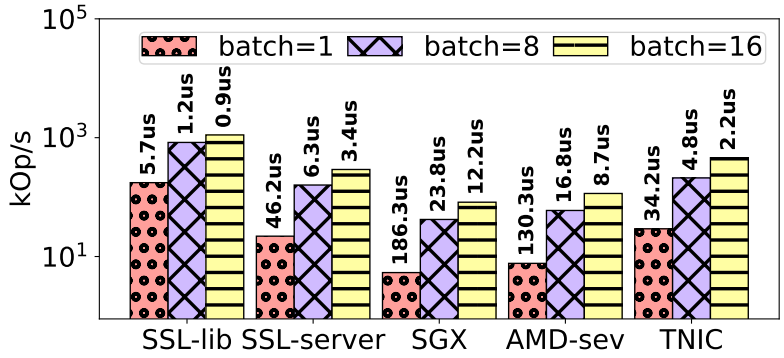

Figure 5.11: Throughput (and latency numbers) of BFT.

AMD-sev, and *(v)* TNIC. To perform a fair comparison, we integrate into our codebases a library that accurately emulates all latencies (measured in § 5.8.1) within the CPU. For the AMD latency, we use 30us, representing the lower bound of the latencies measured in § 5.8.1. We do not emulate the SSL-lib latency.

Given that DRCT-IO, which is used for the emulation, is at least 3× slower than the hardware RDMA network stack (RDMA-hw), the latencies outlined in this section are anticipated to reflect the upper limit for all four systems with TNIC.

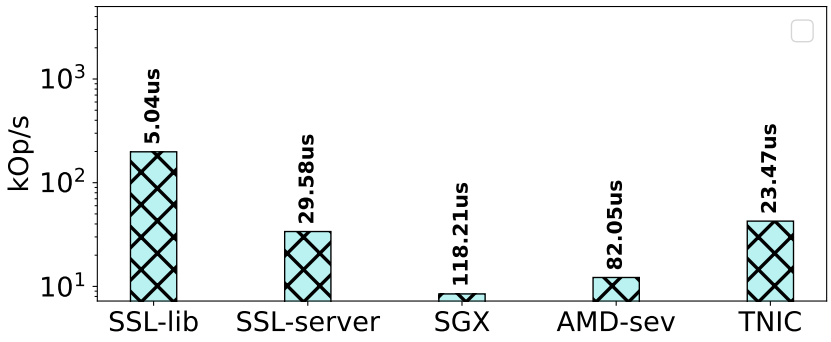

Figure 5.12: Throughput (and latency numbers) of Chain Replication.

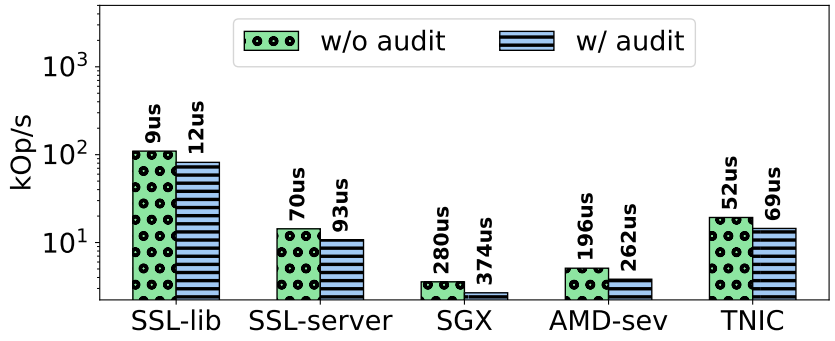

Figure 5.13: Throughput (and latency numbers) of PeerReview.

A2M. We first evaluate our TNIC-A2M system. For SGX, we port the entire log within the TEE, labeled as SGX-lib. This version builds a large TCB that is similar to the original A2M. However, we implemented the system as single SGX process that generates and executes the workload (operations) locally within the trusted area. As such the measured latency and throughput do not consider the inevitable communications costs between the A2M (within SGX) and the untrusted (host) process that would uses this system. We decided this setup to show the upper bound of A2M. All other versions place the attested log in the untrusted host memory, using the trusted systems to generate attestations as in [187]. In the evaluation of these systems we generate the workload in a process in the untrusted host area and as such, by design, their evaluation include the communication costs between the trusted and untrusted components. In this experiment, we construct a 9.3GiB log with 100 million entries and then lookup them sequentially/individually.

Results. Figure 5.10 shows the throughput and mean latency of the append/lookup operations. The native execution (SSL-lib) achieves the highest throughput as it incurs no communication costs. Compared to SSL-lib, SGX-lib experiences only a 2× slowdown because we avoid the costly communication w.r.t. an SGX-based server implementation. On the other hand, AMD-sev, which runs the SSL server, incurs a 15× slowdown. Lastly, TNIC incurs 5× and 2.4× slowdown compared to SSL-lib and SGX-lib, respectively, due to the HMAC calculation.

Regarding the lookup operation, SSL-lib, AMD-sev, and TNIC report similar throughput and latency because they lookup untrusted host memory for the requested entry. However, SGX-lib reports a 66× slowdown due to its trusted memory size constraints and expensive paging mechanism [109]. As a result, while TNIC increases append latencies, it greatly optimizes lookup latencies due to its minimal TCB.

BFT. We evaluate the performance of our BFT protocol with various network batching

factors. We implement network batching as part of the application's message format.

Results. Figure 5.11 shows the throughput and latency of the protocol, which highlights that TNIC significantly outperforms TEE-based versions (SGX, AMD-sev), improving the throughput and latency 4—6×. On the other hand, TNIC incurs 2.4× throughput overhead and up to 7× higher latency compared to SSL-lib. We recall that SSL-lib is not tamper-proof, i.e., it is embedded to the application's process which runs natively without a TEE (Table 5.2), and reduces the communication overheads incurred by other tamper-proof solutions (SGX, AMD-sev).

We also observe that batching improves the throughput and latency proportionally to the number of batched messages. For all except SSL-lib, the batching factors equal to 8 and 16 achieve 7× and 15× higher throughput than without batching, respectively. For SSL-lib, they are moderately effective: approximately 4—6× faster. It is primarily because the native execution of the attestation function is fast enough to saturate the network bandwidth. As such, conventional techniques can drastically eliminate the overheads for BFT and improve TNIC's adoption into practical systems.

CR. In this experiment, we evaluate the performance of our CR. We allocate one message structure per client request comprising 60B context, 4B operation type, and a 32B attestation.

Results. Figure 5.12 shows the throughput and latency of our Chain Replication. We highlight that our TNIC is 5× and 3.4× faster than SGX and AMD-sev, respectively. While TNIC incurs 4.6× overheads compared to SSL-lib, it is 30% faster than SSL-server, which is not tamper-proof. The performance benefit stems primarily from hardware acceleration by the TNIC's attestation kernel on the transmission/reception data path.

PeerReview. We evaluate our PeerReview system's performance by both activating and deactivating the audit protocol. The system uses one witness for the source node that *periodically* audits its log. In our experiments, the witness audits the log after every send operation in the source node until both clients acknowledge the receipt of all source messages.

Results. Figure 5.13 shows the throughput and latency of our PeerReview system with and without enabling the audit protocol. Without the audit protocol, the TEE-based systems (SGX, AMD-sev) result in up to 30× slower throughput than SSL-lib, whereas our TNIC mitigates the overheads: 3—5× better throughput compared to AMD-sev and SGX.

Similarly, with the audit protocol, TNIC outperforms AMD-sev and SGX by 3.7—5×. Importantly, when using TNIC, the audit protocol itself consumes about 25% (17us) of the overall latency, leading to 1.33× performance slowdown. However, even with the audit protocol, TNIC offers 3.7—5.42× lower latency compared to its TEE-based

competitors.

5.9 Formal verification Proofs

We present the detailed security proofs for TNIC security protocols using the Tamarin Prover [208].

Proof artifact. The complete proofs, including the detailed formal models used to generate them, can be found under the following link: [309].

Symbolic model. We prove the security properties of TNIC in a symbolic model. Compared to other approaches, e.g., TLA+ [308] and the computational model [2], Tamarin analyzes protocols in symbolic models and can prove properties by verifying user-defined lemmas. More significantly, TLA+ does not consider security aspects. The computational model is implementation dependant on the cryptographic functions and gives probabilities for the success of and attack. In contrast, we leverage Tamarin's built-in primitives and automated and interactive analysis to verify the correctness of the security protocols.

We impose a set of assumptions on our proofs motivated by Tamarin's symbolic model: *(i)* The symbolic model does not reason about about the individual bits of messages directly. Instead, it assumes a set of atomic terms and functions that operate on these terms. All messages that are part of the model are composed of such atomic terms and functions applied on these terms *(ii)* These cryptographic functions are assumed to be perfect with no side-effects, e.g., hash functions are irreversible, and hash collisions are impossible. This allows for proving lemmas without considering the probabilities of violating specific properties and thus significantly reducing the complexity. The computational model is an alternative to the symbolic model that considers such probabilities. *(iii)* Attackers can read and delete all messages that are sent on the network and modify them in accordance with the set of defined functions.

Tamarin works on symbolic models specified using multiset rewriting rules that operate on the system's state. Different states of the system are expressed as a set of facts with rules capturing the available transitions from one system state to another. Rules are used to model the actions of agents running the protocol and the adversary's capabilities. In addition to the rules, Tamarin also makes use of restrictions. Restrictions further refine the sources of facts in the protocol to improve the efficiency of the proof generation.

Our verification work relies on properties of the already analyzed TLS handshake [304]. It provides a model and lemmas for the security properties of the protocols presented in this chapter.

To prove the correctness of our lemmas, Tamarin computes possible executions for each rule. Tamarin employs constraint solving to refine its knowledge about the sequence of protocol transitions. To check the correctness of the protocol model we also employ sanity lemmas which ensure that there exists a sequence of transitions to reach a predefined valid state. These lemmas ensure that the protocol can be executed as intended.

In the following paragraphs, we give an overview of the rules and lemmas used to model the TNIC protocols.

Rules. The data-structure rules, which create datastructures, that are used by the other rules:

- *create_bitstream*: Models the creation of a bitstream by the IP Vendor.
- *create_secrets*: Models the creation of new communication secrets by the IP Vendor.
- *create_IPVendor_certificate*: Models the creation of a new asymmetric key pair for the IP Vendor.
- *create_controller*: Models the creation of a new controller and embedding of a creator (e.g. IP Vendor) certificate.
- *publish_firmware*: Models the hardware manufacturer publishing a new firmware version.

The bootstrapping rules, in accordance with the bootstrapping steps in § 5.4.3.

- *bootstrapping_1*: Models step (1), the generation and burning of the hardware key by the TNIC manufacturer.
- *bootstrapping_2*: Models step (2), the loading and verification of firmware from the insecure storage medium.
- *bootstrapping_3_4_5*: Models step (3-5), the loading, key and certificate generation of the controller.
- *publish_firmware*: Models the hardware manufacturer publishing a new firmware version.
- *get_tnic_public_key*: Models the retrieval of the public TNIC key for verification.

The attestation rules, in accordance with the attestation steps in § 5.4.3.

- *attestation_1*: Models step (1), the receiving of the configuration data from the protocol designer, and the request for the controller certificates from the TNIC device.
- *attestation_2_3*: Models step (2-3) on the TNIC side, which generates and replies with a signed controller certificate.
- *attestation_4_5_6a*: Models step (4-6) on the IP Vendor side, which comprises the verification of the obtained certificate and start of the mTLS handshake.
- *attestation_6b*: Models step (6) on the TNIC side, the lookup of the IP Vendor certificate and the mTLS handshake.
- *attestation_6c_7_8*: Models step (6-8) on the IP Vendor side, the completion of the mTLS handshake, and the sending of bitstream and secrets in the now encrypted mTLS session.
- *attestation_9*: Models the TNIC device receiving and acknowledging the bitstream and secrets.
- *attestation_10*: Models the final step of the IP Vendor after which the attestation protocol is completed.

The communication rules, in accordance with the functions provided in § 5.4.1.

- *init_ctrs*: Models the initialization of the send and receive counters for each session. Is restricted to guarantee the uniqueness of the session counters.
- *send_msg*: Models sending an arbitrary message by attesting it before sending it over the secure channel. Is restricted to guarantee the session counters are increased.
- *recv_msg*: Models receiving an arbitrary message by only accepting it after a successful verification.

The compromising rule, which model an attacker compromising arbitrary sensitive information, including: TNIC private key, controller private key, IP vendor private key, bitstreams and communication secrets.

Lemmas. The sanity lemmas, which ensure the protocol can be executed as intended:

- *sanity* (verified in 20 steps): Ensures that the protocol allows for successfully completing the bootstrapping & attestation phase, such that the IP Vendor and uncompromised TNIC device are in an expected state.

- *send_sanity* (verified in 63 steps): Ensures that the protocol allows for successfully verifying a message sent during the communication phase after two TNIC devices are successfully initialized.

The attestation lemmas, which ensure the bootstrapping & attestation phase behaves as expected:

- *sensitive_info_stays_secret* (verified in 202 steps): Ensures all sensitive information stays secret and can not be obtained by an attacker through means of the protocol. A special case are the bitstream and communication secretrs, as those can only be obtained if the attacker was able to compromise the private keys of either IP Vendor, controller or the TNIC device.
- *fresh_info_is_not_reused* (verified in 48 steps): Ensures that the symmetric key, which is established during the attestation is fresh, as long as the keys of the IP Vendor and the controller are not compromised.
- *initialization_attested* (verified in 154 steps): Ensures that after the IP Vendor finished the attestation during the initialization phase, the TNIC device is in an expected state and loaded the correct configuration.

The transferable authentication lemma:

- *verified_msg_is_auth* (verified in 92 steps): Ensuring that each message that is successfully accepted by a TNIC device is sent by a genuine TNIC device, assuming the hardware of the TNIC devices was not compromised.

The non-equivocation lemmas:

- *no_lost_messages* (verified in four steps): Ensures that for all messages that are successfully accepted by a genuine TNIC device, there are no messages that were sent before but not accepted by the same TNIC device.
- *no_message_reordering* (verified in 307 steps): Ensures that for all messages that are successfully accepted by a genuine TNIC device, there are no messages that were sent after that message but accepted before.
- *no_double_messages* (verified in 614 steps): Ensures that a genuine TNIC device does not accept the same message multiple times.

5.10 Protocols Implementation

We next present the implementation details of four distributed systems shown in Table 5.3 using TNIC, presented in § 5.7.

System	N	f $(N=3)$	Byzantine faults
A2M	1	0	Prevention
BFT	$2f+1$	$f=1$	Prevention
Chain Replication	$f+1$	$f=2$	Prevention
PeerReview	$f+1$	$f=2$	Detection

Table 5.3: Properties of the four trustworthy distributed systems implemented with TNIC.

5.10.1 Clients

Clients in a TNIC distributed system execute requests by sending singed request messages to TNIC nodes through the network. TNIC assumes Byzantine (untrusted) clients; as such, its installed shared keys cannot be outsourced. We assume that at the initialization, the system designer also loads to TNIC devices a (per-device) key pair $C_{pub,priv}$ where the C_{pub} is distributed to clients. TNIC then replies to a client by verifying the (under transmission) attested message and signing it with C_{priv}. As such, TNIC is restricted to only sending valid attested messages to clients where clients can prove the transferable authentication and validity of the message. The only attack vector open to a Byzantine machine is to try to equivocate by sending a stale, valid, attested message that does not reflect the current execution round. However, clients can detect this by verifying that the original request is theirs.

5.10.2 Attested Append-Only Memory (A2M)

We designed a single-node trusted log system based on the A2M system (Attested Append-Only Memory) [65] using TNIC. A2M has been proven to be an effective building block in improving the scalability and performance of various classical BFT systems [189, 58, 3]. We show the *how* to use TNIC to build this foundational system while we also show that TNIC minimizes the system's TCB jointly with the performance improvements demonstrated in § 5.8.

System model. Our TNIC version and the original A2M systems are single-node systems that target a similar goal; they both build a trusted append-only log as an effective mechanism to combat equivocation. The clients can only append entries to a log; each log entry is associated with a monotonically increasing sequence number. Each data item, e.g., a network message, is bound to a unique sequence number, a well-known approach for equivocation-free operations [67, 44].

A2M was originally built using CPU-side TEEs—specifically, Intel SGX— whereas

we build its TNIC derivative. While the original A2M system keeps its entire state and the log within the TEE, we use TNIC to keep the (trusted) log in the untrusted memory. As such, in contrast to the original A2M, TNIC effectively reduces the overall system's TCB. Our evaluation showed that naively porting the application within the TEE has adverse performance implications in lookup operations.

Execution. Similarly to A2M, we expose three core operations: the `append`, `lookup`, and `truncate` operations to add, retrieve, and delete items of the log, respectively. A2M stores the lowest and highest sequence numbers for each log. Upon appending an entry, A2M increases the highest sequence number and associates it with the newly appended entry. When truncating the log, the system advances the lowest sequence number accordingly. We next discuss how we designed the operations using TNIC APIs.

Algorithm 4: Attested Append-Only Memory (A2M) using TNIC.

```
1  function append(id, ctx) {
2      [α,i,ctx] ← local_send(id,ctx);
3      log[id].append(log_entry(α,i,ctx));
4      return [α,i,ctx];
5  }
6  function lookup(id, i) {
7      return log[id].get(i);
8  }
9  function truncate(id, head, z) {
10     [α,tail,ctx] ← append(id, TRNC||id||z||head);
11     e ← append(MANIFEST,[α,tail,ctx]);
12     return e;
13 }
14 function verify_lookup(id, e, head, tail) {
15     assert(e.i>=tail);
16     local_verify(id, e);
17 }
```

Append operation. The `append(id,ctx)` operation takes a data item, `ctx`, and appends it to the log with identifier `id`. A log entry at index `i` is comprised of three items: the sequence number of that entry (`i`), the context of the entry (`ctx`), and the *authenticator* field, namely the digest of the `ctx||i` as in [187]. In our implementation, we additionally support the original A2M *authenticator* format calculated as the cumulative digest `c_digest[i]` for that entry which is calculated as `c_digest[i]=hash(ctx||i||c_digest[i-1])` where `c_digest[0]=0`. The sequence number `i` is then increased to distinguish any entry that will be appended in the future.

Lookup operation. The `lookup(id, i)` retrieves the log entry at index `i` of log with identifier `id`. Compared to A2M, where lookups are compelled to access the trusted hardware, TNIC-log only performs a local memory access. The function does not verify whether the entry is legitimate. Developers need to implement the `verify_lookup(id, entry, head, tail)` to verify the attestation. The boundaries of the log (i.e., `head` and `tail`) can constantly be retrieved by replaying a specific log, which keeps the state changes, the MANIFEST. We explain how MANIFEST works in the next paragraph.

Truncate operation. The `truncate(id, head, z)`, where `z` is a nonce provided by the client for freshness, "forgets" all log entries with sequence numbers lower than `head`. A non-Byzantine client can never successfully verify a forgotten log entry. To do that, TNIC-log uses an additional log MANIFEST, which keeps the logs' state changes. First, the operation attests to the *tail* of the log by appending a specific entry, which includes the nonce for a correct client to be later able to verify the operation. Then, the algorithm will append the last attested message of the log to the MANIFEST log and return the attested message for the second append. To retrieve the boundaries of a log, clients can always attest to the tail of the MANIFEST and read backward until they find a TRNC entry.

System design takeaway. TNIC minimizes the required TCB in the A2M system while offering faster lookup operations than its original version.

5.10.3 Byzantine Fault Tolerance (BFT)

As a second example of TNIC applications, we build a Byzantine Fault Tolerant protocol (BFT) that implements a robust counter based on *state machine replication* (SMR). Clients send increment counter requests to the SMR and receive the updated value of the counter. Despite its simplicity, this particular system can represent an ordering service, which is a fundamental building block of various distributed applications ranging from event logging and database systems to serverless and blockchain [97, 330, 160, 276, 86]. Our BFT combats equivocation by leveraging the attestation kernel of TNIC. As such, via TNIC, it reduces *(i)* the number of replicas and *(ii)* the message complexity (and latency) required by classical BFT.

System model. We consider a system of $N = 2f + 1$ replicas (or *nodes*) that communicate with each other over unreliable point-to-point network links. At most f of these replicas can be Byzantine (aka *faulty*), i.e., can behave arbitrarily. The rest of the replicas are *correct*. Recall that classical BFT protocols require an extra set of f replicas, in total $3f + 1$, to handle f Byzantine failures. One of the replicas is the *leader* that drives the protocol, whereas the remaining replicas are (passive) followers. There is only a single active leader at a time.

For liveness, we assume a partial synchrony model [92, 63]. We have only explored deterministic protocol specifications; the correct replicas begin in the same state, and receiving the same inputs in the same order will arrive at the same state, generating the same outputs. Lastly, as in classical BFT protocols, we cannot prevent Byzantine clients who otherwise follow the protocol from overwriting correct clients' data.

Execution. We implement BFT with TNIC as a leader-based SMR protocol for a Byzantine model that stores and increases the counter's value. The leader receives clients' requests to increment the counter. The leader, in turn, executes the protocol and applies the changes to its state machine—in our case, the leader computes and stores the next available counter value. Subsequently, the leader broadcasts the request along with some metadata to the passive followers. The metadata includes, among others, the leader's calculated output in response to the client's command, namely, the increased counter value the leader has calculated.

The followers, in turn, execute and apply the incremented counter value to their state machines. However, they first attest to the leader's (and other followers') actions to detect misbehavior. Importantly, followers validate if the state (counter) of the replicas (including the leader and all other replicas) match the expected value.

After a follower applies the increments to its local counter, it replies to the leader. It also forwards the leader's request to every other replica to ensure that all correct replicas will eventually receive and apply the same command. Once the majority acknowledges the reception it replies to the client. Replicas that have already applied the request ignore it; otherwise, they validate it and apply it. The leader, upon successful validation, will also reply to the client. The client can trust the result if they receive identical replies from a majority quorum, i.e., at least $f+1$ identical messages from different replicas (including the leader). This guarantees that at least 1 correct replica have verified the correct result.

Failure handling. Our strategy to verify the replica's execution jointly with the primitives of non-equivocation and transferable authentication offered by TNIC shields the protocol against Byzantine behavior. The leader cannot equivocate; even if it attempts to send different requests for the same round to different followers, executing the `local_send()` will assign different counter values, which healthy followers will detect. As such, a leader in that case will be exposed.

Likewise, the equivocation mechanism allows correct followers to discard stale message requests sent through replay attacks on the network. If a follower is Byzantine, a healthy leader or replica can detect it. For $f \geq 2$, it is impossible for a faulty leader and, at most, $f-1$ remaining Byzantine followers to compromise the protocol. Either these faults will be detected by a healthy replica during the validation phase, or the protocol

will be unavailable, i.e., if the leader in purpose only communicates with the Byzantine followers. This directly affects BFT correctness requirements; a client will never get at least $f+1$ matching replies. Even in the extreme case of a network partition or a faulty leader that purposely excludes some healthy replicas from its multicast group, when the network is restored, these replicas will not accept any future messages unless they receive all missed ones. Suppose the leader fails in the middle of the broadcast. In that case, the last step in the follower's protocol ensures that if a correct replica accepts a request, all correct replicas will eventually apply the same request. Since the reliability aspect and FIFO ordering are implemented in hardware, healthy replicas will ultimately receive and deliver all past messages in the proper order. For protocols to progress in the case of a faulty leader, they must pass through a recovery protocol or view-change protocols similar to those described in previous works [328, 58]. Recovering is beyond the scope of this work, and as such, we did not implement it.

System design takeaway. TNIC optimizes the replication factor and the message rounds compared to classical BFT.

5.10.4 Chain Replication (CR)

We implement a Byzantine Chain Replication using TNIC that represents the replication layer of a Key-Value store. Chain Replication is a foundational protocol for building state machine replication and initially operates under the CFT model using $f+1$ nodes to tolerate up to f failures. We show *how* to use TNIC to shield the protocol without changes to the core of the algorithm (states, rounds, etc.) while keeping the same replication factor.

System model. We make the same assumptions for the system as in the previous BFT system. For error detection and reconfiguration, we assume a centralized (trusted) configuration service as in [323] that generates new configurations upon receiving reconfiguration requests from replicas. Recall that the classical Chain Replication under the CFT model relies on reliable failure detectors [269] as well. For liveness, we also assume that the configuration service will eventually create a configuration of correct replicas that do not intentionally issue reconfiguration requests to perform denial of service attacks.

Clients send requests to `put` or `get` a value and receive the result. The replicas (e.g., head, middle, and tail nodes) are chained, and the requests flow from the head node to the tail through the intermediate middle replicas.

Malicious primaries, i.e., the head that does not forward the message intentionally, are detected on the client's side and trigger reconfiguration [58, 328].

Execution. To execute a request `req`, e.g., `put/get`, a client first obtains the current

Algorithm 5: BFT using TNIC.

```
1  function leader(req) {
2      output ← execute(req);
3      msg ← req||output;
4      attested_msg ← local_send(msg);
5      rem_write(FOLLOWERS[:], attested_msg);
6      upon reception of ack from FOLLOWERS:
7          [α || f_attested_msg || f_output || f_id]
8              ← upon_delivery(ack);
9          assert(validate_follower(f_attested_msg,
10             f_output));
11         incr_req_acks_if_not_incr_before(f_id);
12     auth_send(CLIENT,msg);
13 }
14 function follower() {
15     upon reception of attested_msg:
16         [α || req || output] ←
17             upon_delivery(attested_msg);
18         assert(validate_sender(req, output));
19         if (in_order_not_applied(req))
20             current_output ← execute(req);
21             f_attested_msg ← local_send(req||current_output);
22             ack ← f_attested_msg
23             auth_send(LEADER, ack);
24             auth_send(FOLLOWERS[:],f_attested_msg);
25             upon reception of f acks:
26                 auth_send(CLIENT, f_attested_msg);
27         else
28             auth_send(NODE, reception_ack);
29 }
```

Algorithm 6: Chain Replication using TNIC.

```
1  function head_operation(req) {
2      output ← execute(req);
3      msg ← req||output;
4      auth_send(MIDDLE,msg);
5      auth_send(CLIENT,msg);
6  }
7  function middle_tail_operation(msg) {
8      assert(validate_chain(msg));
9      output ← execute(req);
10     chained_msg ← msg||output;
11     if (!TAIL)
12         auth_send(MIDDLE,chained_msg);
13     auth_send(CLIENT,req||output);
14 }
15 function validate(msg) {
16     len ← sz;
17     [req, out, cmt] ← unmarshall(msg[0:len]);
18     assert(memcmp(req, out));
19     assert((cmt == expected_cmt));
20     for (i = 1; i < NODE_ID; i++) {
21         [out, cmt] ← unmarshall(msg[len:len+sz]);
22         assert(memcmp(req, out));
23         assert((cmt == expected_cmt));
24         len ← len + sz;
25     return True;
26 }
```

configuration from the configuration service and sends the `req` to the head of the chain. The head orders and executes the request, and then it creates a *proof of execution message*, which is sent along the chain. The proof of execution includes the `req` and the leader's action (`out`) in response to that request. In our case, the leader sends the `req` along with the assigned commit index. The message is then sent (signed) to the middle node that follows in the chain.

The middle node checks the message's validity by verifying that the head's output is correct, executes the `req`, and forwards the request to the following replica. Similarly, every other node executes the original request, verifies the output of all previous nodes, and sends the original request and a vector of all previous outputs. A replica must construct a *proof of execution message* that achieves one goal. TNIC allows the following replicas in the chain to verify all previous replicas. As such the messages are of the form $< .. <<\texttt{req}, \texttt{out}_{leader}>_{\sigma_0}, \texttt{out}_{middle1}>_{\sigma_1}, .., \texttt{out}_{tail}>_{\sigma_N}$. The tail is the last node in the chain that will execute and verify the execution of the request.

In contrast to the CFT version of the Chain Replication protocol, local operations in the tail, `get` or `ack` in a `put` request cannot be trusted. As such, the replicas in the chain need to reply to the clients with their output after they have forwarded their proof of execution message. Clients can wait for all $f+1$ replicas replies to collide. Clients can execute the `get` requests similarly to `write` requests, traversing the entire chain, or clients to improve latency can consult the majority and broadcast the request to $f+1$ replicas, including the tail.

Failure handling. By the protocol definition, all nodes will see and execute all messages in the same order imposed by the head node. As such, all correct replicas will always be in the same state. In addition, network partitions that may split the chain into two (or more) individual chains that operate independently cannot affect safety: the clients must verify at least $f+1$ identical replies. Suppose a correct replica or a client detects a violation (by examining the proof of execution message or having to hear for too long from a node). In that case, they can expose the faulty node and request a reconfiguration.

System design takeaway. TNIC *seamlessly* shields the Chain Replication system for Byzantine settings with the same replication factor as the original CFT system.

5.10.5 Accountability (PeerReview)

We implement an accountability protocol based on the PeerReview system [125, 126]. Compared to the previous three BFT systems that prohibit an improper action from taking effect, accountability protocols [134, 125, 126] slightly weaken the system (fault) model in favor of performance and scalability. Specifically, our protocol *allows* Byzan-

tine faults to happen (e.g., correct nodes might be convinced by a malicious replica to permanently delete data). Still, it guarantees that malicious actions can always be detected. Accountability protocols can be applied to different systems as generic guards that trade security for performance [126], e.g., NFS, BitTorrent, etc.

The original version of the system did not use trusted components and as such it incurs a high message complexity, i.e., *all-to-all* communication to combat equivocation. We use TNIC to improve that message complexity.

System model. We only detect faults that directly or indirectly affect a message, implying that *(i)* correct nodes can observe all messages sent and received by that node and *(ii)* Byzantine faults that are not observable through the network cannot be detected. For example, a faulty storage node might report that it is out of disk space, which cannot be verified without knowing the actual state of its disks.

We further assume that each protocol participant acts according to a deterministic specification protocol. As such, detection can be accomplished even with a single correct machine, requiring only $f+1$ machines. This does not contradict the impossibility results for agreement [92] because detection systems do not guarantee safety.

Execution. The participants communicate through network messages generated by TNIC. In addition, each participant maintains a *tamper-evident* log that stores all messages sent and received by that node as a chain. A log entry is associated with an entry index, the entry data, and an authenticator, calculated as the signed hash of the tail of the log and the current entry data.

We frame our protocol in the context of an overlay multicast protocol [56] widely used in streaming systems. The nodes are organized as a tree where the streaming content (e.g., audio, video) flows from a source, i.e., *root* node, to clients (*children* nodes). To support many clients, each can be a source to other clients, which will be connected as children nodes.

In our implementation, we consider nodes in a tree topology. The tree's height equals one, comprising one source node and two client (children) nodes connected to the source. When the source sends a context (executes the `root()` function), it implicitly includes a signed statement that this message has a particular sequence number (generated by TNIC). The clients execute the `child()` function that validates the received message, logs the received message, executes the result, and responds to the source.

Each node is assigned to a set of *witness* processes to detect faults. Similarly to the original system, we assume that the set of nodes and its witnesses set *always* contain a correct process. The witnesses audit and monitor the node's log. To detect destructive behaviors (or expose non-responsive nodes), the witnesses read the node's log and

Algorithm 7: PeerReview using TNIC.

```
1  function root(ctx) {
2      auth_send(CHILD,ctx);
3      upon reception of response:;
4          assert(validate_reception(response));
5          log(response);
6  }
7  function child(α||cmd||seq) {
8      assert(validate_reception(α||cmd||seq));
9      log(α||cmd||seq);
10     result ← execute(cmt);
11     response ← log(result||cmd);
12     auth_send(ROOT, response);
13 }
14 function log_audit() {
15     while last_id < log_tail {
16         entry ← validate_log_entry_at(last_id);
17         last_id++;
18         assert(replay(entry));
19     }
20 }
```

replay it to run the participant's state machine. As such, they ensure the participant's state is consistent with proper operation.

Specifically, each witness for a participant node keeps track of n, a log sequence number, and s, the state that the participant should have been in after sending or receiving the message in log entry n. It initializes n to 0 and s to the initial state of the participant.

Whenever a witness wants to audit a node, it sends its n and a nonce (for freshness). The participant returns an attestation of all entries between n and its current log entry using the nonce. The witness then runs the reference implementation, starting at state *s*, and progressing through all the log entries. If the reference implementation sends the same messages in the log, then the witness updates n, which is the state of the reference implementation at that point. If not, then the witness has proof it can present of the participant's failure to act correctly.

The original PeerReview system requires a receiver node to forward messages to the original sender's witnesses so they can ensure this message is *legitimate*, i.e., it appears in the sender's log. No other conflicting message is sent to another peer (equivocation).

As such, a peer must communicate with the witness set of any other peer, leading to a quadratic message complexity. TNIC eliminates the overhead; a participant that sends or receives a message needs to attest and append the message and its attestation in each log. A participant can process received messages only if they are accompanied by attestations generated by the sender's TNIC hardware.

System design takeaway. TNIC can be used to optimize the message complexity in accountable systems.

5.11 Related work

Trustworthy distributed systems. Classical BFT systems [58, 303, 4, 60, 51, 61, 295, 31] provide BFT guarantees at the cost of high complexity, performance, and scalability overheads. TNIC bridges the gap between BFT and prior limitations, designing a *silicon root-of-trust* with generic trusted networking abstractions that materialize the required security properties for BFT.

Trusted hardware for distributed systems. Trustworthy systems [83, 328, 124, 83, 109, 38, 219] leverage trusted hardware to optimize the performance of classical BFT at the cost of generalization and easy adoption. The systems suffer from high latencies (50us—105ms) [187, 165], build large TCBs [109, 38], and rely on specific TEEs [328, 44]. In contrast, our TNIC aims to offer performance and generality, while our minimalistic TCB is verifiable and unified in the heterogeneous cloud.

SmartNIC-assisted systems. Networked systems offer fast network operations with emerging (programmable) SmartNIC devices [194, 317, 47, 49, 231, 10, 32, 59]. Some of them [21, 102, 119, 142, 183, 191, 210, 261, 298, 299, 285, 291] offload the network functions to the hardware and reduce the host processing and energy overheads. In contrast, others [197, 169, 198, 207, 258, 243, 277, 190, 188, 195, 273, 185] build generic execution frameworks to optimize a wide variety of distributed systems. Our TNIC follows a similar approach by building a high-performance unified network stack with SmartNICs and offloading security properties to the NIC hardware.

SmartNIC-assisted network stacks. SmartNICs effectively provide high-performance network stacks. Another line of research [77, 136, 289, 272, 334, 104, 238] re-designs generic networking protocols, from RDMA/RoCE to TCP/IP network stacks, on top of FPGA-based SmartNICs for performance. Our TNIC further extends its security semantics with the properties of non-equivocation and transferable authentication.

Programmable HW for network security. Programmable hardware, SmartNICs, and switches are used to shield networking. Recent systems [301, 350, 166, 344, 331] leverage programmable switches and FPGAs to offload security processing and boost

performance in the context of blockchain systems [301] or security functions (e.g., access control, DNS traffic inspection) [350, 166, 344]. Our TNIC similarly offloads security into the hardware, but it carefully uses SmartNICs to overcome the processing bottlenecks of the switches.

5.12 Summary

In this chapter, we present the design and implementation of TNIC, a trusted NIC architecture. TNIC proposes a host CPU-agnostic unified security architecture based on trustworthy network-level isolation. We materialize the TNIC architecture using SmartNICs by exposing a minimal and formally verified silicon root-of-trust with low TCB, relying on just two fundamental properties of transferable authentication and non-equivocation. Using TNIC, we implement a hardware-accelerated trustworthy network stack that efficiently transforms a range of distributed systems under the fail-stop operation model for untrusted (Byzantine) cloud environments. Notably, we realize the TNIC architecture-based FPGA-based SmartNICs on Alveo U280 cards; we believe other commercial SmartNIC vendors can implement our NIC-level interface.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

With major cloud providers providing as part of their computing infrastructure the state-of-the-art hardware for trusted computing (i.e., TEEs) and high-performance networking (i.e., direct I/O and SmartNICs) there are opportunities to improve the performance as well as the security aspects of widely deployed distributed data management systems in the cloud. This thesis explores the synergies between modern cloud hardware and the design of general-purpose distributed data management systems to resolve the trade-off between performance, scalability, and security.

We addressed this challenge by building the following systems.

First, we build TREATY, a distributed persistent KVs for the untrusted cloud infrastructure, that offers serializable distributed (ACID) transactions (Txs) with strong security properties for the operations and the stored data, i.e., confidentiality, integrity, and freshness. We achieve these design goals by building on top of TEEs to offer a secure substrate for distributed Txs that extend TEEs trust across the untrusted network and storage and overcome their architectural limitations. TREATY guarantees the correct execution of the distributed transactions by implementing a secure version of the two-phase commit protocol and a secure storage engine on top of TEEs and direct I/O networking. In addition, it makes use of two trusted services, a trusted counter and a configuration service to guarantee secure persistency, i.e., committed transactions remain crash-consistent and rollback-resilient across reboots. TREATY incurs reasonable overheads, 5× to 15× that derive from the TEEs' usage as well as encryption and decryption operations.

Secondly, we build RECIPE, a distributed trusted computing base (TCB) implemented as a library that exposes shielded remote procedure calls RPCs and put/get

KVs APIs, offering a fast, scalable transformation of Crash Fault Tolerant (CFT) protocols for Byzantine settings in the untrusted cloud. RECIPE builds its distributed TCB combining TEEs, direct I/O, and an attestation service that materializes the transformation and resolves the engineering and architectural (performance) challenges of these technologies. RECIPE's TCB prevents equivocation (i.e., sending conflicting statements for the same round to different nodes) and ensures (transferable) authentication; these two properties are the lower security bounds for such a transformation. Compared to TREATY that assumes a crash-fail recovery model with no availability guarantees in case of failures, RECIPE offers fault tolerance and availability. RECIPE is generic and easy to use, we transformed four CFT protocols without important alterations to the core protocol specification while we outperformed the state-of-the-art BFT by a factor of 5×—24×.

Lastly, we build TNIC, a trusted NIC architecture that offers high-performance trusted network operations for building trustworthy (networked) distributed systems for the Byzantine and heterogeneous cloud infrastructure. Similarly to RECIPE, TNIC materializes a trusted silicon-root-of-trust that exposes the properties of (transferable) authentication and non-equivocation. However, TNIC builds on top of SmartNICs to effectively address RECIPE's limitations. First, in contrast to RECIPE that relies on CPU-sided TEEs, TNIC is CPU-agnostic and exposes a unified network library to ease programmability in the modern cloud infrastructure that is comprised of thousands of hundreds of heterogeneous machines with security properties and SDKs. Secondly, TNIC offloads these two fundamental properties on the NIC hardware, extending the scope of the SmartNIC devices to also offer security. As such, compared to RECIPE, TNIC has a minimalistic TCB that is fully verifiable. Lastly, TNIC is generic and offers better programmability and performance in heterogeneous settings compared to the RECIPE approach, which relies on sophisticated optimizations at the TEEs system stack. We applied TNIC to build and optimize BFT systems for the untrusted cloud, showing its generality and simplicity as well as its superiority in performance compared to a TEE-based approach.

6.2 Critical Analysis

In this thesis, we advocate and prove that leveraging modern cloud hardware is an efficient design choice to address the current challenges and trade-offs in distributed data management systems regarding their security, performance, and scalability requirements. As such, we studied and implemented four systems leveraging the advancements in trusted computing (TEEs), high-performance networking (direct I/O, RDMA)

and SmartNICs.

We now critically review our design decisions with the benefit of hindsight.

Firstly, while we argued that TEEs' offer a powerful building block for security without giving up on performance and scalability—as we showed in TREATY and RECIPE—the truth is that we had to build a lot of software and engineering techniques to overcome their limitations regarding their trusted enclave memory area and syscalls execution. In addition, we explored Intel SGX as our foundational TEE to offer trust. The security semantics, as well as the architectural and programming libraries, can possibly vary between different TEEs. As such, we conclude that TEEs naive usage does not necessarily translate to performance efficiency and generality in distributed systems.

Our experiences and observations from building TREATY and RECIPE with CPU-based (distributed) TEEs motivate our work on TNIC. TNIC offers minimalism (and it is verifiable) by offloading only the necessary security processing to the specialized NIC hardware rather than the entire business logic of the system. In TNIC, we argue that the heterogeneity in CPU-based TEEs can complicate the development of trustworthy systems because these TEEs expose different programming APIs and security properties to system developers. The need for unified security protocol implementations has also been recognized in the industry. For example, there is an ongoing effort to unify the remote attestation protocol implementations for heterogeneous TEEs [263]. However, due to significant differences in TEEs hardware implementations, this approach often relies on software modifications. In contrast, TNIC designs a remote attestation protocol that leverages the existing hardware security mechanisms of commodity FPGAs [351], e.g., Xilinx [317] and Intel [264] FPGAs. Additionally, in contrast to heterogeneous CPU-based TEEs, TNIC exposes a unified RDMA-based network API that is consistent across heterogeneous FPGA-based NICs. The network stack (bitstream) is implemented once and provided by a trusted vendor, eliminating the need for system designers to program the FPGA. Consequently, TNIC does not require system designers to be experts in system programming or to introduce complex low-level optimizations or sophisticated data structures.

Finally, while TNIC guarantees that a CFT-to-BFT protocol scalable transformation always exists, it does not provide it *automatically* as RECIPE does. System designers with TNIC are still required to carefully think about the system's business logic specifications.

6.3 Future work

In this thesis, we studied the design of distributed (data management) systems when deployed in the untrusted third-party cloud infrastructure that is equipped with TEEs, SmartNICs, and RDMA-based NICs. As these technologies are becoming more mature, there are many other directions to advance this field of general-purpose, highly available, and trustworthy distributed systems, which is why it remains a very active area of research. Below, we briefly discuss some possible directions, and finally, we discuss how this work can impact future directions.

Trusted VMs. Recent developments in trusted hardware involve trusted virtual machines (TVMs) on top of advanced TEE designs [15, 147, 25]. Based on our programming experiences with AMD Secure Encrypted Virtualization technology (AMD-sev) [15] as part of the TNIC project, which offers an isolated encryption-protected (AMD-SEV-SNP also offers integrity and confidentiality) Linux-based VM, we believe that these advancements offer opportunities to reevaluate some of the design decisions in TREATY and RECIPE. Importantly, trusted VMs allow the use of larger in-memory data structures and TCBs. While this might increase the surface of possible vulnerabilities, we believe that future directions of our work mitigate the implementation and low-level optimization burdens and focus more on techniques to harden and verify the TCBs.

Trusted disaggregated architectures. The emergence of SPDM/TDISP protocols [296, 305] to secure I/O devices in conjunction with confidential virtual machines (CVMs) as offered by a new generation of CPUs, e.g., AMD-SEV-SNP [15], Intel TDX [147], and Arm CCA [25], could adapt our work by augmenting TNIC with an "SPDM-broker" to build an end-to-end secure trusted domain design. Since TNIC already supports remote attestation and authentication, it can also support an encrypted channel between a CVM and the target NIC. Additionally, given the premises of PCIe-enabled CXL standard [81], we believe TNIC's architecture can significantly influence the design of future "secure bridges" because our minimalistic interface can also be realized on a CXL-expander card.

Network devices. Programmable (Smart)NICs and switches are becoming the norm in the cloud infrastructure [59, 32] whereas they are used to accelerate parts of existing systems or even motivate the design of new ones [195, 301, 157]. We believe that the minimal yet powerful properties of TNIC could be integrated into ASIC-based NICs, leading to even better performance while also simplifying the remote attestation process. In addition to this, future network devices could offer hardware-assisted trusted networking group-communication primitives as the fundamental building block

of most state machine replication protocols. An example of this would be the design of a reliable, equivocation-free, multicast algorithm at the NIC level.

Bibliography

[1] Contingency planning, for technology and terrorism. https://www.washingtonpost.com/wp-dyn/content/article/2007/08/15/AR2007081502282.html.

[2] ABADI, M., AND ROGAWAY, P. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of cryptology 15* (2002), 103–127.

[3] ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., REITER, M. K., AND WYLIE, J. J. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[4] ABRAHAM, I., GUETA, G., AND MALKHI, D. Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil.

[5] ABRAHAM, I., GUETA, G., MALKHI, D., ALVISI, L., KOTLA, R., AND MARTIN, J.-P. Revisiting Fast Practical Byzantine Fault Tolerance, 2017.

[6] Intel SGX: Not So Safe After All, AEPIC Leak. `https://thenewstack.io/intel-sgx-not-so-safe-after-all-aepic-leak/`.

[7] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2020), USENIX Association.

[8] AKRAM, A., AKELLA, V., PEISERT, S., AND LOWE-POWER, J. SoK: Limitations of Confidential Computing via TEEs for High-Performance Compute Systems. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)* (2022).

[9] AKRAM, A., GIANNAKOU, A., AKELLA, V., LOWE-POWER, J., AND PEISERT, S. Performance Analysis of Scientific Computing Workloads on General Purpose

TEEs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021).

[10] Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution. https://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution_593986.

[11] Alibaba Cloud: Blockchain as a Service. https://www.alibabacloud.com/product/baas.

[12] ALSBERG, P. A., AND DAY, J. D. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)* (1976).

[13] Amazon EC2. https://aws.amazon.com/pm/ec2.

[14] AMAZON. Amazon S3 Cloud Object Storage. https://aws.amazon.com/s3.

[15] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[16] AMDSEV. https://github.com/AMDESE/AMDSEV.

[17] ANDROULAKI, E., BARGER, A., BORTNIKOV, V., CACHIN, C., CHRISTIDIS, K., DE CARO, A., ENYEART, D., FERRIS, C., LAVENTMAN, G., MANEVICH, Y., MURALIDHARAN, S., MURTHY, C., NGUYEN, B., SETHI, M., SINGH, G., SMITH, K., SORNIOTTI, A., STATHAKOPOULOU, C., VUKOLIĆ, M., COCCO, S. W., AND YELLICK, J. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys)* (2018).

[18] ANGEL, S., BASU, A., CUI, W., JAEGER, T., LAU, S., SETTY, S., AND SINGANAMALLA, S. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 193–208.

[19] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2018).

[20] Apache AsterixDB. https://asterixdb.apache.org/.

[21] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

[22] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal security with cipherbase. In *CIDR* (2013).

[23] ARM. ARM Confidential Compute Architecture. `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`.

[24] ARM. Building a Secure System using TrustZone Technology. `https://developer.arm.com/documentation/PRD29-GENC-009492/latest/`.

[25] ARM. Introducing Arm Confidential Compute Architecture. `https://developer.arm.com/documentation/den0125/0300`.

[26] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2016).

[27] ARS TECHNICA. New Spectre-like attack uses speculative execution to overflow buffers. `https://arstechnica.com/gadgets/2018/07/new-spectre-like-attack-uses-speculative-execution-to-overflow-buffers/`.

[28] AT&T Addresses Recent Data Set Released on the Dark Web . `https://about.att.com/story/2024/addressing-data-set-released-on-dark-web.html`.

[29] ATTIYA, H., AND WELCH, J. L. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems* (1994).

[30] AUBLIN, P.-L., GUERRAOUI, R., KNEŽEVIĆ, N., QUÉMA, V., AND VUKOLIĆ, M. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems 32*, 4 (jan 2015).

[31] AUBLIN, P.-L., MOKHTAR, S. B., AND QUÉMA, V. RBFT: Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013), pp. 297–306.

[32] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[33] AWS Lambda. https://aws.amazon.com/lambda/.

[34] DPDK on Azure VMs. https://learn.microsoft.com/en-us/azure/virtual-network/setup-dpdk?tabs=redhat.

[35] Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[36] BAI, W., ABDEEN, S. S., AGRAWAL, A., ATTRE, K. K., BAHL, P., BHAGAT, A., BHASKARA, G., BROKHMAN, T., CAO, L., CHEEMA, A., CHOW, R., COHEN, J., ELHADDAD, M., ETTE, V., FIGLIN, I., FIRESTONE, D., GEORGE, M., GERMAN, I., GHAI, L., GREEN, E., GREENBERG, A., GUPTA, M., HAAGENS, R., HENDEL, M., HOWLADER, R., JOHN, N., JOHNSTONE, J., JOLLY, T., KRAMER, G., KRUSE, D., KUMAR, A., LAN, E., LEE, I., LEVY, A., LIPSHTEYN, M., LIU, X., LIU, C., LU, G., LU, Y., LU, X., MAKHERVAKS, V., MALASHANKA, U., MALTZ, D. A., MARINOS, I., MEHTA, R., MURTHI, S., NAMDHARI, A., OGUS, A., PADHYE, J., PANDYA, M., PHILLIPS, D., POWER, A., PURI, S., RAINDEL, S., RHEE, J., RUSSO, A., SAH, M., SHERIFF, A., SPARACINO, C., SRIVASTAVA, A., SUN, W., SWANSON, N., TIAN, F., TOMCZYK, L., VADLAMURI, V., WOLMAN, A., XIE, Y., YOM, J., YUAN, L., ZHANG, Y., AND ZILL, B. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 49–67.

[37] BAILLEU, M., DRAGOTI, D., BHATOTIA, P., AND FETZER, C. TEE-Perf: A Profiler for Trusted Execution Environments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019).

[38] BAILLEU, M., GIANTSIDI, D., GAVRIELATOS, V., QUOC, D. L., NAGARAJAN, V., AND BHATOTIA, P. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), USENIX Association, pp. 65–79.

[39] BAILLEU, M., THALHEIM, J., BHATOTIA, P., FETZER, C., HONDA, M., AND VASWANI, K. Speicher: Securing LSM-Based Key-Value Stores Using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)* (2019).

[40] BAJAJ, S., AND SION, R. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *In Proceedings of the 2011 international conference on Management of data* (2011), ACM, pp. 205–216.

[41] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBLER, T., WEI, M., AND DAVIS, J. D. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, Apr. 2012), USENIX Association, pp. 1–14.

[42] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: a tale of two systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[43] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[44] BEHL, J., DISTLER, T., AND KAPITZA, R. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (2017).

[45] BINDSCHAEDLER, L., GOEL, A., AND ZWAENEPOEL, W. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)* (2020).

[46] Blockchain on AWS. `https://aws.amazon.com/blockchain/`.

[47] NVIDIA BlueField Data Processing Units. `https://www.nvidia.com/en-gb/networking/products/data-processing-unit/`.

[48] boost: C++ libraries. `https://www.boost.org/`.

[49] Broadcom Stingray SmartNIC Accelerates Baidu Cloud Services.

[50] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (USENIX ATC)* (2013).

[51] BUCHMAN, E., KWON, J., AND MILOSEVIC, Z. The latest gossip on BFT consensus. *CoRR abs/1807.04938* (2018).

[52] BURCKHARDT, S., GILLUM, C., JUSTO, D., KALLAS, K., MCMAHON, C., AND MEIKLEJOHN, C. S. Durable Functions: Semantics for Stateful Serverless. In *OOPSLA* (October 2021), ACM, pp. 133:1–133:27.

[53] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes. *Communications of the ACM* (2016).

[54] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[55] Case Study: GE Healthcare Takes DynamoDB On-Premises with ScyllaDB's 'Project Alternator'. https://www.scylladb.com/users/case-study-ge-healthcare-takes-dynamodb-on-premises-with-scyllas-project-alternator/.

[56] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: High-Bandwidth Multicast in Cooperative Environments. *ACM SIGOPS Operating Systems Review* (2003).

[57] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (USA, 1999), OSDI '99, USENIX Association, p. 173–186.

[58] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* (2002).

[59] Project Catapult. `https://www.microsoft.com/en-us/research/project/project-catapult/`.

[60] CHAN, T.-H. H., PASS, R., AND SHI, E. PaLa: A Simple Partially Synchronous Blockchain. *IACR Cryptol. ePrint Arch. 2018* (2018), 981.

[61] CHAN, T.-H. H., PASS, R., AND SHI, E. PiLi: An Extremely Simple Synchronous Blockchain. *IACR Cryptol. ePrint Arch. 2018* (2018), 980.

[62] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2016), PODC '16, Association for Computing Machinery, p. 325–334.

[63] CHANDRA, T. D., AND TOUEG, S. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM) 43*, 2 (mar 1996), 225–267.

[64] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX Security Symposium* (2007).

[65] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).

[66] 3 Pillars of Data Security: Confidentiality, Integrity and Availability. `https://mark43.com/resources/blog/3-pillars-of-data-security-confidentiality-availability-integrity/`.

[67] CLEMENT, A., JUNQUEIRA, F., KATE, A., AND RODRIGUES, R. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2012), PODC '12, Association for Computing Machinery, p. 301–308.

[68] CLOUD, A. Alibaba Cloud's Next-Generation Security Makes Gartner's Report. `https://www.alibabacloud.com/blog/alibaba-clouds-next-generation-security-makes-gartners-report_595367`.

[69] CockroachDB. `https://www.cockroachlabs.com/`.

[70] 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`.

[71] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. In *ACM Transactions on Computer Systems (TOCS)* (2013).

[72] CORREIA, M., FERRO, D. G., JUNQUEIRA, F. P., AND SERAFINI, M. Practical hardening of Crash-Tolerant systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association, pp. 453–466.

[73] CORREIA, M., NEVES, N., AND VERISSIMO, P. 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).

[74] COSTAN, V., AND DEVADAS, S. Intel SGX Explained, 2016.

[75] Couchbase. https://www.couchbase.com/.

[76] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (USA, 2006), OSDI '06, USENIX Association, p. 177–190.

[77] Coyote: OS for FPGAs. https://github.com/fpgasystems/Coyote.

[78] CRN. The ten biggest cloud outages of 2013. https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, 2013.

[79] CROOKS, N. *A client-centric approach to transactional datastores*. PhD thesis, The University of Texas at Austin, 2019.

[80] CROOKS, N., BURKE, M., CECCHETTI, E., HAREL, S., AGARWAL, R., AND ALVISI, L. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2018).

[81] Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[82] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review (SIGOPS)* (2007).

[83] DECOUCHANT, J., KOZHAYA, D., RAHLI, V., AND YU, J. DAMYSUS: Streamlined BFT Consensus Leveraging Trusted Components. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022), EuroSys '22, Association for Computing Machinery, p. 1–16.

[84] DELL. Elastic Cloud Storage. https://www.dellemc.com/en-us/storage/ecs/.

[85] DELPORTE-GALLET, C., FAUCONNIER, H., FREILING, F. C., PENSO, L. D., AND TIELMANN, A. From Crash-Stop to Permanent Omission: Automatic Transformation and Weakest Failure Detectors. In *Distributed Computing* (2007), A. Pelc, Ed., Springer Berlin Heidelberg.

[86] DING, C., CHU, D., ZHAO, E., LI, X., ALVISI, L., AND RENESSE, R. V. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 325–338.

[87] DOUCEUR, J. R. The Sybil Attack. In *Peer-to-Peer Systems* (Berlin, Heidelberg, 2002), P. Druschel, F. Kaashoek, and A. Rowstron, Eds., Springer Berlin Heidelberg, pp. 251–260.

[88] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014).

[89] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 54–70.

[90] Under the bonnet: Dropbox architecture overview. `https://www.dropbox.com/en_GB/business/trust/security/architecture`.

[91] DUAN, S., REITER, M., AND ZHANG, H. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018).

[92] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)* (1988).

[93] Architecting a Highly Available Serverless, Microservices-Based Ecommerce Site. https://aws.amazon.com/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/.

[94] ENES, V., BAQUERO, C., REZENDE, T. F., GOTSMAN, A., PERRIN, M., AND SUTRA, P. State-Machine Replication for Planet-Scale Systems (Extended Version). `https://arxiv.org/abs/2003.11789`, 2020.

[95] ESKANDARIAN, S., AND ZAHARIA, M. ObliDB: Oblivious Query Processing for Secure Databases. In *Proceedings of the VLDB Endowment (VLDB)* (2019).

[96] Hyperledger Fabric Ordering Service. `https://github.com/hyperledger/fabric/tree/main/orderer#service-types`.

[97] FABRIC, H. Ordering service implementations. `https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html`.

[98] Facebook Is Receiving Sensitive Medical Information from Hospital Websites. https://themarkup.org/pixel-hunt/2022/06/16/facebook-is-receiving-sensitive-medical-information-from-hospital-websites.

[99] FAY CHANG AND JEFFREY DEAN AND SANJAY GHEMAWAT AND WILSON C. HSIEH AND DEBORAH A. WALLACH AND MIKE BURROWS AND TUSHAR CHANDRA AND ANDREW FIKES AND ROBERT E. GRUBER. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[100] FEI, S., YAN, Z., DING, W., AND XIE, H. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv. 54*, 6 (jul 2021).

[101] FETZER, C., AND CRISTIAN, F. A highly available local leader election service. *IEEE Transactions on Software Engineering 25*, 5 (1999), 603–618.

[102] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 51–66.

[103] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal* (2004).

[104] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An Open-Source 100-Gbps Nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2020), pp. 38–46.

[105] GARG, D., AND PFENNING, F. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).

[106] GAVRIELATOS, V., KATSARAKIS, A., AND NAGARAJAN, V. Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols. In *Proceedings of the 16th European Conference on Computer Systems EuroSys'21* (United States, Apr. 2021), Association for Computing Machinery (ACM), p. 245–260. 16th

ACM EuroSys Conference on Computer Systems, EuroSys 2021 ; Conference date: 26-04-2021 Through 28-04-2021.

[107] GHAYVAT, H., PANDYA, S., BHATTACHARYA, P., ZUHAIR, M., RASHID, M., HAKAK, S., AND DEV, K. CP-BDHCA: Blockchain-Based Confidentiality-Privacy Preserving Big Data Scheme for Healthcare Clouds and Applications. *IEEE Journal of Biomedical and Health Informatics 26*, 5 (2022), 1937–1948.

[108] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 20–43.

[109] GIANTSIDI, D., BAILLEU, M., CROOKS, N., AND BHATOTIA, P. Treaty: Secure Distributed Transactions. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2022), pp. 14–27.

[110] GIANTSIDI, D., GIORTAMIS, E., TORNOW, N., DINU, F., AND BHATOTIA, P. FlexLog: A Shared Log for Stateful Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2023).

[111] How are databases used in banking? `https://globalbanks.com/how-are-databases-used-in-banking/`.

[112] GOODSON, G., AND SCHROEDER, B. An Analysis of Data Corruption in the Storage Stack. In *6th USENIX Conference on File and Storage Technologies (FAST 08)* (San Jose, CA, Feb. 2008), USENIX Association.

[113] GOOGLE. Cloud Storage. `http://www.cloud.google.com/storage`.

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

[115] Dataproc Confidential Compute. `https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/confidential-compute`.

[116] Google Functions. `https://cloud.google.com/functions`.

[117] Google Compute Engine. `https://cloud.google.com/`.

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

[119] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 681–693.

[120] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)* (1989).

[121] GREGOR, F., OZGA, W., VAUCHER, S., PIRES, R., QUOC, D. L., ARNAUTOV, S., MARTIN, A., SCHIAVONI, V., FELBER, P., AND FETZER, C. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)* (2020).

[122] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. 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).

[123] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16, p. 202–215.

[124] GUPTA, S., RAHNAMA, S., PANDEY, S., CROOKS, N., AND SADOGHI, M. Dissecting bft consensus: In trusted components we trust! In *Proceedings of the Eighteenth European Conference on Computer Systems* (New York, NY, USA, 2023), EuroSys '23, Association for Computing Machinery, p. 521–539.

[125] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. The Case for Byzantine Fault Detection. In *Proceedings of the Second Conference on Hot Topics in System Dependability* (USA, 2006), HotDep'06, USENIX Association, p. 5.

[126] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).

[127] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (ATC)* (2017).

[128] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012).

[129] Apache HBase. `https://hbase.apache.org/`.

[130] Homomorphic Encryption. `https://homomorphicencryption.org/`.

[131] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[132] Hess Corporation Case Study. `https://aws.amazon.com/solutions/case-studies/hess-corporation/`.

[133] HO, C., DOLEV, D., AND VAN RENESSE, R. Making Distributed Applications Robust. In *Principles of Distributed Systems* (2007), E. Tovar, P. Tsigas, and H. Fouchal, Eds.

[134] HO, C., RENESSE, R. V., BICKFORD, M., AND DOLEV, D. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (San Francisco, CA, 2008), USENIX Association.

[135] Homomorphic Encryption. https://chain.link/education-hub/homomorphic-encryption.

[136] HOSEINZADEH, M., AND SWANSON, S. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, p. 429–442.

[137] HOWARD, H. *Distributed consensus revised*. PhD thesis, University of Cambridge, 2018.

[138] Data Breach Chronology. https://privacyrights.org/data-breaches.

[139] HUANG, P., GUO, C., LORCH, J. R., ZHOU, L., AND DANG, Y. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 18) (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 1–16.

[140] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (ATC)* (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.

[142] IBANEZ, S., SHAHBAZ, M., AND MCKEOWN, N. The Case for a Network Fast Path to the CPU. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2019), HotNets '19, Association for Computing Machinery, p. 52–59.

[143] IBM Blockchain. https://www.ibm.com/blockchain.

[144] InfiniBand Architecture Specification. https://www.infinibandta.org/ibta-specification/.

[145] Instagram Search Architecture. https://instagram-engineering.com/search-architecture-eeb34a936d3a.

[146] Intel, SGX documentation: sgx create monotonic counter. https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter/.

[147] Intel SGX vs TDX: what is the difference? https://www.canarybit.eu/intel-sgx-vs-tdx-what-is-the-difference/.

[148] INTEL. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[149] Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[150] Intel DPDK. http://dpdk.org/.

[151] SGX Monotonic Counters not supported. https://github.com/intel/linux-sgx/issues/424.

[152] Latest SGAxe and CrossTalk Attacks Leak Sensitive Data and Expose New Intel SGX Vulnerability. https://news.hackreports.com/sgaxe-crosstalk-attacks-intel-sgx-vulnerability/.

[153] AEPIC Leak is an Architectural CPU Bug Affecting 10th, 11th, and 12th Gen Intel Core CPUs. https://wccftech.com/aepic-leak-is-an-architectural-cpu-bug-affecting-10th-11th-and-12th-gen-intel-core-cpus/.

[154] Intel Storage Performance Development Kit. `http://www.spdk.io`.

[155] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. `https://iperf.fr/`.

[156] ISLAM, S., MOGHIMI, A., BRUHNS, I., KREBBEL, M., GULMEZOGLU, B., EISENBARTH, T., AND SUNAR, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium (USENIX Security 19)* (2019).

[157] ISTVÁN, Z., SIDLER, D., ALONSO, G., AND VUKOLIC, M. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 425–438.

[158] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. 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).

[159] JHA, S., BEHRENS, J., GKOUNTOUVAS, T., MILANO, M., SONG, W., TREMEL, E., RENESSE, R. V., ZINK, S., AND BIRMAN, K. P. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems 36*, 2 (apr 2019).

[160] JIA, Z., AND WITCHEL, E. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 691–707.

[161] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR-RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP)* (2011).

[162] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[163] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.

[164] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[165] KAPITZA, R., BEHL, J., CACHIN, C., DISTLER, T., KUHNLE, S., MOHAMMADI, S. V., SCHRÖDER-PREIKSCHAT, W., AND STENGEL, K. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, Association for Computing Machinery, p. 295–308.

[166] KAPLAN, A., AND FEIBISH, S. L. Practical handling of DNS in the data plane. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2022), SOSR '22, Association for Computing Machinery, p. 59–66.

[167] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 237–250.

[168] KATSARAKIS, A., GAVRIELATOS, V., KATEBZADEH, M. S., JOSHI, A., DRAGOJEVIC, A., GROT, B., AND NAGARAJAN, V. 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).

[169] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High Performance Packet Processing with FlexNIC. *ACM SIGARCH Computer Architecture News* (2016).

[170] KIM, T., PARK, J., WOO, J., JEON, S., AND HUH, J. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)* (2019).

[171] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P)* (2019).

[172] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (jan 2010).

[173] KRAHN, R., TRACH, B., VAHLDIEK-OBERWAGNER, A., KNAUTH, T., BHATOTIA, P., AND FETZER, C. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (2018).

[174] Encrypt workload data in-use with Confidential Google Kubernetes Engine Nodes. `https://cloud.google.com/kubernetes-engine/docs/how-to/confidential-gke-nodes`.

[175] KUO, H.-C., CHEN, J., MOHAN, S., AND XU, T. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst. 4*, 1 (may 2020).

[176] KUO, T.-T., KIM, H.-E., AND OHNO-MACHADO, L. Blockchain distributed ledger technologies for biomedical and health care applications. *Journal of the American Medical Informatics Association 24*, 6 (09 2017), 1211–1220.

[177] KUVAISKII, D., FAQEH, R., BHATOTIA, P., FELBER, P., AND FETZER, C. HAFT: Hardware-Assisted Fault Tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, Association for Computing Machinery.

[178] KUVAISKII, D., OLEKSENKO, O., ARNAUTOV, S., TRACH, B., BHATOTIA, P., FELBER, P., AND FETZER, C. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)* (2017).

[179] LAKSHMAN, A., AND MALIK, P. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM Symposium on Principles of distributed computing (PODC)* (2009), ACM.

[180] LAMPORT. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (1979), 690–691.

[181] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems* (1998).

[182] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* (1982).

[183] LE, Y., CHANG, H., MUKHERJEE, S., WANG, L., AKELLA, A., SWIFT, M. M., AND LAKSHMAN, T. V. UNO: uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, Association for Computing Machinery, p. 506–519.

[184] LEE, D., KOHLBRENNER, D., SHINDE, S., ASANOVIĆ, K., AND SONG, D. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)* (2020).

[185] LETTIERI, G., FAIS, A., ANTICHI, G., AND PROCISSI, G. SmartNIC-Accelerated Stream Processing Analytics. In *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2023), pp. 135–140.

[186] LevelDB. `http://leveldb.org/`.

[187] LEVIN, D., DOUCEUR, J. J., LORCH, J., AND MOSCIBRODA, T. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[188] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 137–152.

[189] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[190] LI, J., LU, Y., WANG, Q., LIN, J., YANG, Z., AND SHU, J. AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 951–966.

[191] LIN, J., PATEL, K., STEPHENS, B. E., SIVARAMAN, A., AND AKELLA, A. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 243–259.

[192] Ubuntu kernel lifecycle. `https://ubuntu.com/kernel/lifecycle`.

[193] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[194] LiquidIO II Smart NICs. `https://www.marvell.com/products/infrastructure-processors/liquidio-smart-nics/liquidio-ii-smart-nics.html`.

[195] LIU, J., DRAGOJEVIC, A., FLEMMING, S., KATSARAKIS, A., KOROLIJA, D., ZABLOTCHI, I., CHEUNG NG, H., KALIA, A., AND CASTRO, M. Honeycomb: ordered key-value store acceleration on an FPGA-based SmartNIC, 2023.

[196] LIU, J., LI, W., KARAME, G. O., AND ASOKAN, N. Scalable Byzantine Consensus via Hardware-assisted Secret Sharing. *CoRR abs/1612.04997* (2016).

[197] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2019), SIGCOMM '19, Association for Computing Machinery, p. 318–333.

[198] LIU, M., PETER, S., KRISHNAMURTHY, A., AND PHOTHILIMTHANA, P. M. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 363–378.

[199] LIU, S., VIOTTI, P., CACHIN, C., QUEMA, V., AND VUKOLIC, M. XFT: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 485–500.

[200] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 773–785.

[201] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FtCS)* (1997).

[202] MADSEN, M. F., AND DEBOIS, S. On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (New York, NY, USA, 2020), PODC '20, Association for Computing Machinery, p. 159–168.

[203] MAENE, P., GÖTZFRIED, J., DE CLERCQ, R., MÜLLER, T., FREILING, F., AND VERBAUWHEDE, I. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers 67*, 3 (2018), 361–374.

[204] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud Storage with Minimal Trust. In *ACM Transactions on Computer Systems* (2011).

[205] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. 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).

[206] MATETIC, S., AHMED, M., KOSTIAINEN, K., DHAR, A., SOMMER, D., GERVAIS, A., JUELS, A., AND CAPKUN, S. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security)* (2017).

[207] MATUS, F. Distributed Services Architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)* (2020), pp. 1–17.

[208] MEIER, S., SCHMIDT, B., CREMERS, C., AND BASIN, D. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)* (2013).

[209] MELANOX. RDMA Aware Networks Programming User Manual.

[210] MELLETTE, W. M., DAS, R., GUO, Y., MCGUINNESS, R., SNOEREN, A. C., AND PORTER, G. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 1–18.

[211] MÉNÉTREY, J., GÖTTEL, C., KHURSHID, A., PASIN, M., FELBER, P., SCHIAVONI, V., AND RAZA, S. Attestation Mechanisms For Trusted Execution Environments Demystified. In *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings* (Berlin, Heidelberg, 2022), Springer-Verlag, p. 95–113.

[212] MERKLE, R. C. Secure Communications over Insecure Channels. *Commun. ACM 21*, 4 (apr 1978), 294–299.

[213] MESSADI, I., BECKER, M. H., BLEEKE, K., JEHL, L., MOKHTAR, S. B., AND KAPITZA, R. SplitBFT: Improving Byzantine Fault Tolerance Safety Using Trusted Compartments. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (New York, NY, USA, 2022), Middleware '22, Association for Computing Machinery, p. 56–68.

[214] MESSADI, I., NEUMANN, S., WEICHBRODT, N., ALMSTEDT, L., MAHHOUK, M., AND KAPITZA, R. Precursor: A Fast, Client-Centric and Trusted Key-Value Store Using RDMA and Intel SGX. In *Proceedings of the 22nd International Middleware Conference (Middleware)* (2021).

[215] MicroBlaze Soft Processor Core. `https://www.xilinx.com/products/design-tools/microblaze.html`.

[216] MICROSOFT. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs.

[217] MICROSOFT AZURE. Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/`.

[218] Microsoft Azure. `https://azure.microsoft.com/en-gb`.

[219] MICROSOFT RESEARCH. The Confidential Consortium Framework. `https://microsoft.github.io/CCF/main/research`.

[220] MISLOVE, A., POST, A., HAEBERLEN, A., AND DRUSCHEL, P. Experiences in building and operating ePOST, a reliable peer-to-peer application. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), EuroSys '06, Association for Computing Machinery, p. 147–159.

[221] MISRA, S. C., AND BHAVSAR, V. C. Relationships between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. In *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI* (Berlin, Heidelberg, 2003), ICCSA'03, Springer-Verlag, p. 724–732.

[222] MITCHELL, C., GENG, Y., AND LI, J. 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).

[223] MongoDB. https://www.mongodb.com/.

[224] Unable to find Alternatives to Monotonic Counter Application Programming Interfaces (APIs) in Intel Software Guard Extensions (Intel SGX) for Linux to Prevent Sealing Rollback Attacks. https://www.intel.co.uk/content/www/uk/en/support/articles/000057968/software/intel-security-products.html.

[225] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 358–372.

[226] MUKHERJEE, S., EMER, J., AND REINHARDT, S. The soft error problem: an architectural perspective. In *11th International Symposium on High-Performance Computer Architecture* (2005), pp. 243–247.

[227] MURDOCK, K., OSWALD, D., GARCIA, F. D., VAN BULCK, J., GRUSS, D., AND PIESSENS, F. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)* (2020).

[228] NAEHRIG, M., LAUTER, K., AND VAIKUNTANATHAN, V. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2011), CCSW '11, Association for Computing Machinery, p. 113–124.

[229] NARAYAN, A., AND HAEBERLEN, A. DJoin: Differentially Private Join Queries over Distributed Databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 149–162.

[230] The history of databases at Netflix and how they use CockroachDB. https://www.cockroachlabs.com/blog/netflix-at-cockroachdb/.

[231] Netronome. https://www.netronome.com/.

[232] Nios Soft Processor Series. https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor.html.

[233] OF MEDICINE, N. L. Beyond the HIPAA Privacy Rule: Enhancing Privacy, Improving Health Through Research. https://www.ncbi.nlm.nih.gov/books/NBK9579/.

[234] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., FELBER, P., AND FETZER, C. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[235] OLIVEIRA, R., PEREIRA, J., AND SCHIPER, A. Primary-backup replication: from a time-free protocol to a time-based implementation. In *Proceedings 20th IEEE Symposium on Reliable Distributed Systems* (2001).

[236] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)* (2014).

[237] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 29–41.

[238] AMD OpenNIC Project. `https://github.com/Xilinx/open-nic`.

[239] OpenSSL library. https://openssl.org.

[240] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)* (Seattle, WA, Mar. 2003), USENIX Association.

[241] Oracle Blockchain. `https://www.oracle.com/blockchain/`.

[242] ORENBACH, M., MINKIN, M., LIFSHITS, P., AND SILBERSTEIN, M. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)* (2017).

[243] PACÍFICO, R. D. G., DUARTE, L. F. S., VIEIRA, L. F. M., RAGHAVAN, B., NACIF, J. A. M., AND VIEIRA, M. A. M. eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF. *IEEE/ACM Transactions on Networking* (2023), 1–14.

[244] PADILHA, R., AND PEDONE, F. Belisarius: BFT Storage with Confidentiality. In *2011 IEEE 10th International Symposium on Network Computing and Applications* (2011).

[245] PAJU, A., JAVED, M. O., NURMI, J., SAVIMÄKI, J., MCGILLION, B., AND BRUMLEY, B. B. SoK: A Systematic Review of TEE Usage for Developing Trusted

Applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security (ARES'23)* (2023).

[246] Enable DPDK on ESXi. `https://docs.paloaltonetworks.com/vm-series/10-1/vm-series-deployment/set-up-a-vm-series-firewall-on-an-esxi-server/performance-tuning-of-the-vm-series-for-esxi/enable-dpdk-on-esxi`.

[247] PAPADIMITRIOU, A., BHAGWAN, R., CHANDRAN, N., RAMJEE, R., HAEBERLEN, A., SINGH, H., MODI, A., AND BADRINARAYANAN, S. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[248] PAPADIMITRIOU, C. H. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM) 26*, 4 (Oct. 1979), 631–653.

[249] PARNO, B., MCCUNE, J. M., AND PERRIG, A. Bootstrapping Trust in Commodity Computers. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)* (2010).

[250] pmem-rocksdb. `https://github.com/pmem/pmem-rocksdb`.

[251] POKE, M., HOEFLER, T., AND GLASS, C. W. AllConcur: Leaderless Concurrent Atomic Broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2017).

[252] POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., AND ZHUANG, L. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)* (2011).

[253] POPA, R. A., REDFIELD, C., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[254] PORTO, D., LEITÃO, J. A., LI, C., CLEMENT, A., KATE, A., JUNQUEIRA, F., AND RODRIGUES, R. Visigoth Fault Tolerance. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)* (2015).

[255] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, Association for Computing Machinery, p. 206–220.

[256] PRIEBE, C., MUTHUKUMARAN, D., LIND, J., ZHU, H., CUI, S., SARTAKOV, V. A., AND PIETZUCH, P. SGX-LKL: Securing the Host OS Interface for Trusted Execution, 2019.

[257] PRIEBE, C., VASWANI, K., AND COSTA, M. EnclaveDB: A Secure Database using SGX (S&P). In *IEEE Symposium on Security and Privacy* (2018).

[258] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014).

[259] QUOC, D. L., GREGOR, F., ARNAUTOV, S., KUNKEL, R., BHATOTIA, P., AND FETZER, C. SecureTF: A Secure TensorFlow Framework. In *Proceedings of the 21st International Middleware Conference (Middleware)* (2020).

[260] Rackspace. `https://www.rackspace.com/en-gb`.

[261] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC : Scalable NIC for End-Host Rate Limiting. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[262] A Remote Direct Memory Access Protocol Specification. https://datatracker.ietf.org/doc/html/rfc5040.

[263] Unifying Remote Attestation Protocol Implementations. https://confidentialcomputing.io/2023/03/06/unifying-remote-attestation-protocol-implementations/.

[264] Intel Stratix 10 FPGA and SoC FPGA. `https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/item.html`.

[265] Redis. `https://redis.io/`.

[266] REED, B. C., AND JUNQUEIRA, F. P. A simple totally ordered broadcast protocol. In *LADIS '08* (2008).

[267] REGISTER, T. Boffins show Intel's SGX can leak crypto keys. `https://www.theregister.com/2017/03/07/eggheads_slip_a_note_under_intels_door_sgx_can_leak_crypto_keys/`.

[268] REITER, M. K. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)* (1994).

[269] RENESSE, R. V., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)* (2004), USENIX Association.

[270] RISC-V. Keystone Open-source Secure Hardware Enclave. `https://keystone-enclave.org/`.

[271] RocksDB, A persistent key-value store. https://rocksdb.org/.

[272] RUIZ, M., SIDLER, D., SUTTER, G., ALONSO, G., AND LÓPEZ-BUEDO, S. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019), pp. 286–292.

[273] SADOK, H., ATRE, N., ZHAO, Z., BERGER, D. S., HOE, J. C., PANDA, A., SHERRY, J., AND WANG, R. Enso: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 1005–1025.

[274] SANTOS, N., GUMMADI, K. P., AND RODRIGUES, R. Towards Trusted Cloud Computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing* (USA, 2009), HotCloud'09, USENIX Association.

[275] SANTOS, N., RODRIGUES, R., AND FORD, B. Enhancing the OS against Security Threats in System Administration. In *Proceedings of the 13th International Middleware Conference (Middleware)* (2012).

[276] SAUREZ, E., BALASUBRAMANIAN, B., SCHLICHTING, R., TSCHAEN, B., HUANG, Z., NARAYANAN, S. P., AND RAMACHANDRAN, U. METRIC: A Middleware for Entry Transactional Database Clustering at the Edge. In *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets* (New York, NY, USA, 2018), MECC'18, Association for Computing Machinery, p. 2–7.

[277] SCHUH, H. N., LIANG, W., LIU, M., NELSON, J., AND KRISHNAMURTHY, A. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 740–755.

[278] SCHUSTER, F., COSTA, M., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3 : Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (2015).

[279] SCHWARZ, M., LIPP, M., MOGHIMI, D., VAN BULCK, J., STECKLINA, J., PRESCHER, T., AND GRUSS, D. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS* (2019).

[280] SCOPELLITI, G., POUYANRAD, S., NOORMAN, J., ALDER, F., PIESSENS, F., AND MÜHLBERG, J. T. POSTER: An Open-Source Framework for Developing Heterogeneous Distributed Enclave Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 2393–2395.

[281] Security Monitor IP. `https://www.xilinx.com/support/documents/product-briefs/security-monitor-ip-core-product-brief.pdf`.

[282] Serverless Message Queues and How to Leverage the Best. `https://dzone.com/articles/serverless-message-queues-and-how-to-leverage-the`.

[283] SHAMIS, A., PIETZUCH, P., CANAKCI, B., CASTRO, M., FOURNET, C., ASHTON, E., CHAMAYOU, A., CLEBSCH, S., DELIGNAT-LAVAUD, A., KERNER, M., MAFFRE, J., VROUSGOU, O., WINTERSTEIGER, C. M., COSTA, M., AND RUSSINOVICH, M. IA-CCF: Individual Accountability for Permissioned Ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 467–491.

[284] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)* (2019).

[285] SHAN, Y., LIN, W., KOSTA, R., KRISHNAMURTHY, A., AND ZHANG, Y. SuperNIC: A Hardware-Based, Programmable, and Multi-Tenant SmartNIC, 2022.

[286] SHARMA, A., JIANG, J., BOMMANNAVAR, P., LARSON, B., AND LIN, J. GraphJet: Real-Time Content Recommendations at Twitter. *Proceedings of the VLDB Endowment 9*, 13 (sep 2016), 1281–1292.

[287] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCCS)* (2016).

[288] SHINDE, S., LE TIEN, D., TOPLE, S., AND SAXENA, P. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[289] SIDLER, D., WANG, Z., CHIOSA, M., KULKARNI, A., AND ALONSO, G. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[290] SINGH, A., DAS, T., MANIATIS, P., AND ROSCOE, T. BFT Protocols Under Fire. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (San Francisco, CA, Apr. 2008), USENIX Association.

[291] SIVARAMAN, A., MASON, T., PANDA, A., NETRAVALI, R., AND KONDAVEETI, S. A. Network architecture in the age of programmability. *SIGCOMM Comput. Commun. Rev. 50*, 1 (mar 2020), 38–44.

[292] Alveo sn1000 smartnic accelerator card.

[293] SOARES, L., AND STUMM, M. 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).

[294] SONG, Y. J., JUNQUEIRA, F., AND REED, B. BFT for the skeptics.

[295] SOUSA, J., AND BESSANI, A. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference (EDCC)* (2012).

[296] Security Protocol and Data Model (SPDM) Specification. `https://www.dmtf.org/dsp/DSP0274`.

[297] STAVRAKAKIS, D., GIANTSIDI, D., BAILLEU, M., SÄNDIG, P., ISSA, S., AND BHATOTIA, P. Anchor: A Library for Building Secure Persistent Memory Systems. *Proceedings of the ACM on Management of Data* (2023).

[298] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 33–46.

[299] STEPHENS, B., AKELLA, A., AND SWIFT, M. M. Your Programmable NIC Should be a Programmable Switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)* (2018).

[300] STRACKX, R., AND PIESSENS, F. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 875–892.

[301] SUN, G., JIANG, M., KHOOI, X. Z., LI, Y., AND LI, J. NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering. In *Proceedings of the ACM SIGCOMM 2023 Conference* (2023), ACM SIGCOMM '23, p. 239–254.

[302] SUN MICROSYSTEMS, I. NFS: Network File System Protocol Specification. `https://www.ietf.org/rfc/rfc1094.txt`.

[303] SURI-PAYER, F., BURKE, M., WANG, Z., ZHANG, Y., ALVISI, L., AND CROOKS, N. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[304] Tamarin TLS handshake proof. `https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/classic/TLS_Handshake.spthy`.

[305] TEE Device Interface Security Protocol (TDISP). `https://pcisig.com/tee-device-interface-security-protocol-tdisp`.

[306] TERRACE, J., AND FREEDMAN, M. J. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference (ATC)* (2009).

[307] THALHEIM, J., UNNIBHAVI, H., PRIEBE, C., BHATOTIA, P., AND PIETZUCH, P. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems (ACM EuroSys 21)* (2021).

[308] The TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html.

[309] T-NIC Protocol Verification. `https://github.com/julianpritzi/tnic_proofs`.

[310] TPC-C. `http://www.tpc.org/tpcc/`.

[311] Trusted Computing Group. `https://trustedcomputinggroup.org/`.

[312] TRACH, B., FAQEH, R., OLEKSENKO, O., OZGA, W., BHATOTIA, P., AND FETZER, C. T-Lease: A Trusted Lease Primitive for Distributed Systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)* (2020).

[313] TRACH, B., KROHMER, A., GREGOR, F., ARNAUTOV, S., BHATOTIA, P., AND FETZER, C. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (2018).

[314] TRACH, B., OLEKSENKO, O., GREGOR, F., BHATOTIA, P., AND FETZER, C. Clemmys: Towards Secure Remote Execution in FaaS. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)* (2019).

[315] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2017).

[316] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)* (2013).

[317] Alveo U280 Data Center Accelerator Card.

[318] UltraScale+ Integrated 100G Ethernet Subsystem. `https://www.xilinx.com/products/intellectual-property/cmac_usplus.html`.

[319] UNGER, C., MURTHY, D., ACKER, A., ARORA, I., AND CHANG, A. Examining the evolution of mobile social payments in Venmo. In *International Conference on Social Media and Society (SMSociety)* (2020).

[320] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., MEHTA, A., GARG, D., DRUSCHEL, P., RODRIGUES, R., GEHRKE, J., AND POST, A. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)* (2015).

[321] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. 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).

[322] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos Made Moderately Complex. *ACM Comput. Surv.* (2015).

[323] VAN RENESSE, ROBBERT AND HO, CHI AND SCHIPER, NICOLAS. Byzantine Chain Replication. In *Principles of Distributed Systems* (2012), Springer Berlin Heidelberg.

[324] VAN SCHAIK, S., KWONG, A., GENKIN, D., AND YAROM, Y. SGAxe: How SGX Fails in Practice. `https://sgaxeattack.com/`.

[325] VAN SCHAIK, S., MINKIN, M., KWONG, A., GENKIN, D., AND YAROM, Y. CacheOut: Leaking Data on Intel CPUs via Cache Evictions.

[326] VASUDEVAN, V., ANDERSEN, D., AND KAMINSKY, M. The Case for VOS: The Vector Operating System. In *13th Workshop on Hot Topics in Operating Systems (HotOS)* (2011).

[327] VENETIS, P., HALEVY, A., MADHAVAN, J., PAŞCA, M., SHEN, W., WU, F., MIAO, G., AND WU, C. Recovering Semantics of Tables on the Web. *Proceedings of the VLDB Endowment* (2011).

[328] VERONESE, G. S., CORREIA, M. P., BESSANI, A. N., LUNG, L. C., AND VERÍSSIMO, P. Efficient Byzantine Fault-Tolerance. *IEEE Transactions on Computers* (2013).

[329] VOGELS, W. Eventually Consistent: Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability. *Commun. ACM 52*, 1 (jan 2009), 40–44.

[330] WANG, G., KOSHY, J., SUBRAMANIAN, S., PARAMASIVAM, K., ZADEH, M., NARKHEDE, N., RAO, J., KREPS, J., AND STEIN, J. Building a Replicated Logging System with Apache Kafka. *Proceedings of the VLDB Endowment 8*, 12 (Aug. 2015), 1654–1655.

[331] WANG, T., YANG, X., ANTICHI, G., SIVARAMAN, A., AND PANDA, A. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 1289–1305.

[332] WANG, W., DENG, S., NIU, J., REITER, M. K., AND ZHANG, Y. ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery.

[333] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the Salus Scalable Block Store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[334] WANG, Z., HUANG, H., ZHANG, J., WU, F., AND ALONSO, G. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), USENIX Association.

[335] Web3. `https://azure.microsoft.com/en-us/solutions/web3/#overview`.

[336] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., AND MALKHI, D. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), USENIX Association.

[337] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).

[338] WEINHOLD, C., AND HÄRTIG, H. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2011).

[339] WEISSE, O., BERTACCO, V., AND AUSTIN, T. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[340] DMA for PCI Express (PCIe) Subsystem. `https://www.xilinx.com/products/intellectual-property/pcie-dma.html`.

[341] XILINX. DMA/Bridge Subsystem for PCI Express v3.1. `https://docs.xilinx.com/v/u/3.1-English/pg195-pcie-dma`.

[342] Vitis Security Library. `https://github.com/Xilinx/Vitis_Libraries/tree/main/security/L1/include/xf_security`.

[343] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015).

[344] YAN, J., TANG, L., LI, J., YANG, X., QUAN, W., CHEN, H., AND SUN, Z. UniSec: a unified security framework with SmartNIC acceleration in public cloud. In *Proceedings of the ACM Turing Celebration Conference (ACM TURC)* (2019).

[345] YANDAMURI, S., ABRAHAM, I., NAYAK, K., AND REITER, M. Brief Announcement: Communication-Efficient BFT Using Small Trusted Hardware to Tolerate

Minority Corruption. In *35th International Symposium on Distributed Computing (DISC)* (2021).

[346] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)* (Seattle, WA, Nov. 2006), USENIX Association.

[347] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[348] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems* (2006).

[349] YCSB. `https://github.com/brianfrankcooper/YCSB`.

[350] YOU, M., NAM, J., SEO, M., AND SHIN, S. HELIOS: Hardware-assisted High-performance Security Extension for Cloud Networking. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)* (2023).

[351] ZHAO, M., GAO, M., AND KOZYRAKIS, C. ShEF: Shielded Enclaves for Cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2022).

[352] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[353] How big is RocksDB adoption? https://rocksdb.org/docs/support/faq.html.

[354] ZippyDB a strongly consistent, geographically distributed key-value store at Facebook. `https://engineering.fb.com/2021/08/06/core-data/zippydb/`.