Dependable Persistent Memory Architectures

Dimitrios STAVRAKAKIS

Doctor of Philosophy

Adviser: Prof. Pramod BHATOTIA

Thesis Committee Members:

Prof. Antonio BARBALACE (The University of Edinburgh)

Prof. Kaveh RAZAVI (ETH Zurich)

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2024

Abstract

The modern cloud computing infrastructure is shifting towards disaggregated system architectures, primarily due to the emergence of the Compute Express Link (CXL) technology. In such setups, byte-addressable persistent memory (PM) is anticipated to become a fundamental building block as it provides opportunities for high-volume pools of low-latency, non-volatile memory. While offloading critical data management operations to the cloud is advantageous, at the same time, it is imperative to provide reliable and trustworthy services in untrusted cloud environments. However, the incorporation of PM in the cloud providers' system stack, despite its performance benefits, introduces new challenging dependability issues, especially in the context of safety and security.

Unfortunately, in the cloud infrastructure, the underlying storage, network, and computing stacks are entirely managed by untrusted third-party providers. In such settings, a powerful adversary, e.g., a malicious system administrator, can control the entire software system stack (including the OS/hypervisor layers) and even perform physical attacks (e.g., memory probes) aiming to compromise the security properties of both PM data and storage operations. Moreover, attackers can gain control over the network stack and tamper with network traffic, amplifying the vulnerability vectors in untrusted cloud environments.

Additionally, memory safety bugs in the deployed software constitute another critical source of reliability and security issues. Memory safety bugs are also prevalent in PM because, similarly to its volatile counterpart, it employs a byte-addressable programming model. To this end, uncontrolled memory accesses can result in various software bugs and severe vulnerabilities that potentially lead to information exposure or leakage of secrets, thus posing security and privacy concerns.

In this thesis, we aim to provide system designers with the means to build an end-to-end, dependable persistent memory architecture. Precisely, we design two memory safety solutions tailored for PM (SafePM, SPP), targeting both debugging and production environments. We base our design on effective, well-tested approaches that have been extensively applied for volatile memory systems. On top of that, we combine modern hardware advancements in trusted computing, high-performance networking, and byte-addressable PM to build an end-to-end secure PM data management system (Anchor). More specifically,

- SafePM is a memory safety mechanism for PM-based applications that detects spatial and temporal memory safety violations. SafePM employs a shadow memory approach augmented with crash-consistent data structures, ensuring safety across reboots and crashes. It is based on Google's AddressSanitizer (ASan) and offers comprehensive memory safety with reasonable overheads, uncovering real-world bugs in the Persistent Memory Development Kit (PMDK), the de-facto programming framework for PM.

- SPP is a low-overhead, tagged pointer-based memory safety solution for PM. SPP protects against buffer overflows in PM applications. It enhances persistent pointers with memory safety metadata, which is embedded in native tagged pointers during runtime. SPP is built on the LLVM compiler infrastructure and is integrated with the PMDK. To promote applicability in existing toolchains, SPP maintains the PMDK API intact and requires no source code modifications. SPP incurs low runtime and space overheads while preserving the crash consistency property both for the PM data and its memory safety metadata.
- ANCHOR enables building end-to-end secure PM data management systems with reasonable performance implications. ANCHOR extends the hardware-assisted security properties — confidentiality, integrity, and freshness — of trusted execution environments (TEEs) to PM. Precisely, ANCHOR secures PM data leveraging confidential and authenticated data structures, preserves crash consistency with a formally proven secure logging protocol, and allows for secure remote operations via a secure network stack based on kernel-bypass networking. It further offers a formally verified attestation protocol for trust establishment and exposes intuitive APIs tailored for secure data management on byte-addressable storage devices.

Collectively, these systems contribute to establishing security and reliability in modern untrusted cloud infrastructures that incorporate persistent memory as their storage medium.

Lay summary

Cloud infrastructures host a wide range of services that manage and store private, security-critical user data. Typical examples include streaming services, financial and banking services, and healthcare applications. To improve the quality of their services, cloud providers constantly update their infrastructures using state-of-the-art technologies. This trend brings in a new type of storage called persistent memory (PM), which is fast and preserves the data across power cycles. Using PM in cloud computing has many benefits but also comes with new safety and security challenges.

More specifically, cloud services are often managed by third-party providers who might not be fully trustworthy. This means that someone with bad intentions, like a malicious system administrator, could potentially control the software on their systems or even physically tamper with the system components and the stored data to extract valuable, potentially private information. Consequently, data stored in PM are no exception and can also be vulnerable to such types of attacks.

Apart from that, erroneous or buggy software can open up further holes that can be exploited by attackers to achieve their goals. In the context of PM, many common security-critical bugs are caused by so-called *memory safety* errors. Such bugs allow attackers to access parts of the PM device that they are not supposed to, and unfortunately, this uncontrolled access can lead to data leaks and security breaches.

To this end, this thesis aims to provide system designers with tools to build reliable and secure PM systems for the cloud. We introduce two memory safety solutions for PM, SAFEPM and SPP, that present different trade-offs between their memory safety guarantees and performance overheads. These solutions are based on well-tested methods and tools for memory safety but have been carefully adapted considering the idiosyncrasies of PM. In simple terms, SAFEPM is capable of detecting all types of memory safety errors in PM but incurs considerable overheads, which restricts its usage in testing environments. On the other hand, SPP's detection capabilities are limited to *spatial* memory safety bugs, a popular subcategory of memory safety errors, but come with minimal overheads constituting it suitable for production deployments.

On top of that, we also develop a secure data management system for PM, called ANCHOR, that combines modern hardware security features, high-speed networking techniques, and PM. It provides strong security properties, namely data confidentiality, integrity, and freshness, supports secure remote operations on PM, provides methods for trust establishment between remote system entities, and exposes intuitive interfaces to facilitate application development to materialize secure PM data management.

These systems, cumulatively, aim to improve the safety and security of untrusted cloud infrastructures that use persistent memory as a storage medium.

Publications

This thesis is based on the following conference papers:

1. SAFEPM*: A Sanitizer for Persistent Memory*
   Kartal Kaan Bozdoğan*, Dimitrios Stavrakakis*, Shady Issa, Pramod Bhatotia
   ACM Eurosys 2022 [30] **(Honorable Mention: Best Artifact Award)**
   ** Equal contribution to the paper.*

2. ANCHOR*: A Library for Building Secure Persistent Memory Systems*
   Dimitrios Stavrakakis, Dimitra Giantsidi, Maurice Bailleu, Philip Saendig, Shady Issa, Pramod Bhatotia
   ACM SIGMOD 2024 [285]

3. SPP*: Safe Persistent Pointers for Memory Safety*
   Dimitrios Stavrakakis, Alexandrina Panfil, MJin Nam, Pramod Bhatotia
   IEEE/IFIP DSN 2024 [286]

Other conference papers during my PhD:

1. TOAST*: A Heterogeneous Memory Management System*
   Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Soham Chakraborty, Deepak Garg, Pramod Bhatotia
   ACM PACT 2024 [24]

2. GRAMINE-TDX*: A Lightweight OS Kernel for Confidential VMs*
   Dmitrii Kuvaiskii*, Dimitrios Stavrakakis*, Kailun Qin, Cedric Xing, Pramod Bhatotia, Mona Vij
   ACM CCS 2024 [173]
   ** Equal contribution to the paper.*

3. CAGE*: Hardware-Accelerated Safe WebAssembly*
   Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, Pramod Bhatotia
   ACM CGO 2025 [74]

4. *Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX*
   Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, Pramod Bhatotia
   SIGMETRICS 2025 [204]

Acknowledgements

I know, a simple 'thank you' is not enough...
I could write page after page to properly thank everyone who contributed, directly or indirectly, to this thesis. To save some trees, I will keep it short and sweet, but you should know that I am **deeply thankful** to all of you!

First and foremost, this thesis would not have been possible without my Ph.D. advisor, Prof. Pramod Bhatotia. He opened the door to the academic world for me, providing me with the opportunity to work in a fantastic environment with excellent collaborators and fellow students. Of course, each new beginning is challenging. Therefore, I want to express my gratitude for the time invested and the patience he gave me, which was crucial for shaping my research mindset. His never-ending energy, constructive feedback, and extensive guidance on every aspect of my research throughout the years have been invaluable. He was always there to help me discover and develop a wide range of skills, from generating new research ideas to enhancing my writing and presentation abilities. I am deeply grateful for his support and dedication in bringing out the best of me and showing me what I am capable of.

Further, I want to thank Dr. Shady Issa, Prof. Deepak Garg, and Prof. Soham Chakraborty for their insights and valuable advice, which significantly contributed to shaping exciting research projects. Additionally, I extend my gratitude to my internship mentors at Intel Labs, Dmitrii Kuvaiskii and Mona Vij, for allowing me to work on great projects with amazing people and helping me improve my technical skills.

This research journey would have been impossible without my friend and collaborator (soon-to-be Dr.) Dimitra Giantsidi. She was my go-to person for any advice or help – both personal and work-related. Despite our often significant time difference, she was available to listen to me anytime. She relieved me from academic stress countless times and helped me endure the tough times of the pandemic and TEE debugging madness without losing my mind. Apart from being a great friend, Dimitra is a bright system researcher with remarkable capabilities and even greater ideas.

I am also really thankful to Maurice Bailleu, who was a senior PhD student when I first joined my program. His experience, solid technical skills, and research attitude played a significant role in my development. I really enjoyed our collaboration and discussions – many times beyond research, especially during traveling – throughout these years. Maurice, apart from a C++ master, is a great person and researcher, to say the least.

However, a successful series of studies would have been impossible without a great environment and people. Therefore, I must thank the whole Systems Research Group at TUM for providing that. I would like to especially thank Jörg Thalheim and Peter

Okelmann for all the times ($\rightarrow \infty$) that they helped with infrastructure management and technical matters. Additionally, I would like to take a moment to thank the remarkable students Kartal, Philip, Alexa, Julian, and Martin, with whom I had the honor to collaborate. I am sure they all have a bright future ahead! Lastly, I also owe a lot to Sophia, who helped with every administrative matter during my Ph.D. and made everything look simple – and you know that bureaucracy in Germany is not.

This whole period of my life would have been unbearable (pandemic, stress, constant annoying rain, you name it) without the support from my friends, Harsha(vardhan) Unnibhavi, (Prof. Dr.?) Redha Gouicem, Manos Giortamis and Michalis Georgoulakis. I want to thank all of them for our fun moments and activities that made my time enjoyable. I appreciate them being there for me, listening to my problems, and providing valuable advice and unconditional help. I also would like to express my gratitude to my friend (and exceptional researcher) Giorgos Vavouliotis for the numerous calls we had during my studies, the emotional and academic support, and the great time whenever we managed to meet (Barcelona, Zurich, and Crete)!

I also want to sincerely thank my childhood friends, Giorgos (Troulon), Astrinos (Psilos), Kostas (Kapten), and Alekos (White liar), although I doubt that they ever understood what I was working on all this time. I hope the title of this thesis gives them an idea. They were always there for me, both at the good and bad times. They support me in every step of my life, no matter the circumstances. I cannot thank them enough for all the fun times, our trips, our laughs, our silly, endless discussions, and many more experiences that kept me happy and sane through this journey. These people have shaped me as a person. I consider myself extremely lucky to call them friends!

Lastly, but **most** importantly, I thank my beloved parents, Giorgos and Katerina, and my brother, Konstantinos (Tsikis). Despite being far away, I could always feel their constant care and support. At times, I could see they were more stressed for me than myself — Mum, Dad... **We** made it! Their boundless love and care were, and will always be, the most significant motivating factor in my life. Although it is probably the smallest piece of appreciation, I dedicate this work to them.

To conclude, once more, all of you should always know:
I am incredibly thankful to each one of you!

The journey continues...
Dimitris

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. Work that has been formed as a part of jointly-authored publications is also included. 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 § 3.1 was previously published in ACM Eurosys 2022 under the title "SAFEPM*: A Sanitizer for Persistent Memory*" by Kartal Kaan Bozdoğan*, Dimitrios Stavrakakis*, Shady Issa, and Pramod Bhatotia (PhD supervisor). The project described in § 3.2 was accepted for publication in IEEE/IFIP DSN 2024 with the title "SPP*: Safe Persistent Pointers for Memory Safety*" by Dimitrios Stavrakakis, Alexandrina Panfil, MJin Nam, and Pramod Bhatotia (PhD supervisor). Lastly, the research work outlined in § 4.1 was published in ACM SIGMOD 2024 with the name "ANCHOR*: A Library for Building Secure Persistent Memory Systems*" by Dimitrios Stavrakakis, Dimitra Giantsidi, Maurice Bailleu, Philip Sändig, Shady Issa, and Pramod Bhatotia (PhD supervisor).

(*Dimitrios* STAVRAKAKIS)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud computing infrastructures are constantly evolving towards cutting-edge disaggregated system architectures [67, 98, 152, 156, 323], where computational units (e.g., CPU, GPU, FPGAs), memory (e.g., DRAM), and storage (e.g., SSDs) resources are decoupled and can be managed and utilized separately. They become increasingly favorable due to their ability to optimize resource allocation and enhance scalability, allowing cloud providers to scale their system components independently. This is particularly beneficial for data-intensive tasks and applications constrained by the available memory capacity or storage bandwidth (e.g., HPC) [9, 335]. A significant driving force behind this evolution is the emergence of Compute Express Link (CXL) technology [47], an open standard for high-speed communications supported by industry leaders that enables building large pools of memory and storage within rack [314] while cross-datacenter interconnectivity is achieved through high-performance networking solutions (e.g., via Ethernet).

On the storage frontier of disaggregated cloud systems, the technology of Non-Volatile Dual In-line Memory Module (NVDIMM) or Persistent Memory (PM) [113, 192, 201] is pitched as a prominent storage medium to bridge the gap between the volatile main memory and the SSDs. PM is a non-volatile storage medium, accessible at a byte granularity with `ld/st` instructions. Its performance characteristics are similar to DRAM [143], while also ensuring data persistence across application or system failures/reboots. Several high-performance data management systems have been developed [37, 117, 189, 268] or are being adapted to incorporate PM in their design [253, 256].

While byte addressable storage [90, 113, 192, 201, 260, 271] is projected as a fundamental building block for the next generation of data management systems in the cloud, it also presents trade-offs between security and performance. Precisely, cloud computing comes with strict scalability and performance requirements [11, 32, 85, 316]. Integrating byte-addressable storage devices in the system stack is a progressive step towards meeting these needs due to their unique performance characteristics. However, introducing new device types with their programming nuances, despite their performance benefits, can serve as a new attack vector for malicious adversaries. Thus, it can potentially lead to critical safety, security, and privacy issues, particularly in untrusted cloud environments. For instance, a powerful attacker can access, leak, or tamper with the persistent data, compromising their security properties.

On top of that, PM is also susceptible to *memory safety* issues in a similar manner as traditional volatile memory is, especially in the context of memory-unsafe languages (e.g., C/C++). More specifically, PM content is directly mapped into an application's address space and can be manipulated at a byte granularity via native pointers. Uncontrolled memory accesses can be a source of various software bugs and security vulnerabilities. It has been proved that *memory safety* is the root cause of many reliability and security issues in unsafe programming languages [289]. As a matter of fact, three out of the top ten most dangerous software weaknesses are memory bugs (even in 2023!), according to the MITRE ranking [205]. Additionally, apart from several well-known memory safety violations [94, 96, 284], several big software vendors' products, such as Chromium [42], Android [7], and Microsoft Windows [206] have reported that the majority of their systems' vulnerabilities (70-75%) are related to memory safety problems. Memory safety issues are exacerbated in the context of PM, where data is durable across reboots. Importantly, a memory safety bug can lead to *permanent* information leaks or corruptions and vulnerabilities that remain exploitable even after system restarts.

Likewise, in virtualized cloud infrastructures, the underlying storage, network, and computing stacks are owned and operated by an untrusted third-party provider. Under these circumstances, an adversary, such as a malicious system administrator or co-located tenants, can potentially compromise the security properties of both persistent data and query operations [261, 262]. In fact, prior work has shown that software bugs, configuration errors, and security vulnerabilities pose a real threat to data management systems [52, 72, 80, 171, 262]. More specifically, in the context of PM, attackers can tamper with the persistent state and data operations, violating the *confidentiality* and *integrity* security properties. They are also able to arbitrarily rollback the PM data into a stale but valid state, violating the *freshness* property [8, 25, 31, 229, 287]. Further, PM *crash consistency* mechanisms constitute another vulnerability vector, where the required logs, i.e., for recoverability in case of a crash, are susceptible to these security violations. On top of that, adversaries can also manipulate the untrusted network infrastructure and interfere with the transmitted network packets, thus making them eligible to remotely compromise the data management operations. For instance, an attacker might block network packets or intercept and alter a packet carrying a command to update records, filling it with invalid data to overwrite or delete critical data.

To this end, the need for effective memory safety and security solutions for PM is paramount. Therefore, this thesis strives to tackle the challenging problem: **how to design a dependable persistent memory architecture that ensures both memory safety and security while guaranteeing crash consistency and performance within the realms of existing persistent memory programming paradigms?**. To achieve this overarching goal, we break it down into three key questions: *(i)* How can we design an effective memory safety solution for PM that not only provides complete, comprehensive memory safety but also maintains the correctness and crash consistency properties of both the application's data and the memory safety metadata? *(ii)* Is it feasible to design a memory safety solution that goes beyond debugging purposes and incurs minimal performance overheads, constituting it suitable for production deployments? *(iii)* How can we design a secure PM data management system for untrusted cloud environments while preserving the crash consistency property and minimizing its performance implications?

1.2 Problem Statement

Designing effective approaches to enforce memory safety is a well-explored area of research, including both software [3, 68, 92, 170, 172, 220, 264] and hardware-based solutions [14, 69, 223, 224, 322]. There has been extensive use of such memory safety approaches in commercial software products, both during development [137, 198, 264] and production [14, 101, 280] phases.

However, existing memory safety solutions are largely tailored for volatile main memory and are inadequate for byte-addressable persistent memory (PM) devices. Addressing memory safety issues for PM is challenging, especially due to the idiosyncrasies of its unique programming model [239, 276]. More specifically, unlike volatile memory, which uses native volatile pointers, the PM programming model introduces a PM pointer representation [243], and the persistent memory heap is handled by specialized, crash-consistent memory allocators designed for PM [57, 120]. Additionally, although the types of memory safety vulnerabilities on PM remain the same as those on volatile memory (e.g., buffer overflows, use-after-free), memory safety needs to be further ensured for the recovery paths that are executed after a potential crash and/or a restart. Unfortunately, state-of-the-art approaches for memory safety on PM either require adopting a new programming language [99] or are limited to offline testing due to prohibitive performance overheads [240].

Nonetheless, providing tools to prevent memory safety vulnerabilities on PM is not enough, as PM devices are meant to reside and be utilized within the untrusted infrastructures provided by cloud providers. Under this setting, a powerful malicious adversary can manipulate data stored in PM in the cloud, resulting in security and privacy risks. Therefore, it is imperative that we provide an end-to-end PM data management system that is resilient to attacks from such powerful adversaries.

One promising direction for building a secure PM data management system destined for untrusted cloud environments is to leverage Trusted Execution Environments (TEEs). TEEs offer a hardware-protected memory area that protects both code and data from all system layers, including the operating system and hypervisor. TEEs are now included in several commodity CPUs [5, 12, 15, 132, 138, 177] and are available from major cloud providers [44, 82, 103, 203]. However, despite their strong security guarantees, TEEs are fundamentally incompatible with PM. Precisely, TEEs are designed to protect solely volatile memory regions [50] and do not naturally extend their security properties to untrusted PM storage, where data persists across reboots and shutdowns because TEEs cannot provide their guarantees for data once it leaves the secure volatile memory region making it generally vulnerable to tampering or exposure on untrusted persistent storage devices. Additionally, the protected memory region is typically limited in size, and its backup paging mechanism can be detrimental to the performance of the deployed applications [16, 25, 226]. On top of that, their provided trusted counters, which are essential to building mechanisms against rollback attacks, usually cannot keep up with the high-performance demands of modern applications.

Another challenging aspect for PM systems is the provision of crash consistency guarantees. This challenge is magnified when considering the need to maintain consistency for both data and security metadata. To this end, a carefully designed, secure crash consistency mechanism is mandatory to provide the necessary atomicity guarantees without compromising the security properties.

Lastly, to build an end-to-end system, networking is an essential component. Unfortunately, conventional network I/O methods, such as kernel-sockets, introduce significant overheads, particularly in the context of TEEs, due to context switches between trusted and untrusted environments [51, 145]. Kernel-bypass networking can optimize network operations but is incompatible with TEEs because untrusted DMA operations are prohibited in protected memory [23]. On top of that, providing security properties and ensuring crash consistency when accessing PM remotely presents additional challenges [152].

In summary, we identify four core challenges in designing an end-to-end *memory safe* and *secure* PM data management system that we aim to tackle in this thesis:

1. **TEE limitations:** The security properties of TEEs do not extend to the untrusted stateful PM storage since TEEs are primarily designed to protect only the volatile enclave region. On top of that, TEEs present two architectural limitations in our context: *(i)* the hardware-protected memory is limited and often impractical for modern storage systems and *(ii)* the trusted counters, a fundamental building block for rollback resilience, are not suitable for high-performance applications.
2. **Safe & secure crash consistency:** While crash consistency is already a major issue in PM systems due to the non-atomic and out-of-order architectural interface between the CPU cache and PM, this issue is exacerbated in our setting because we not only need to ensure the crash consistency of the data but also of the associated memory safety- and security-related metadata. Additionally, PM applications are designed to be able to recover from abrupt crashes. This requires special code paths that restore the PM to a consistent state. Thus, memory safety and data security properties have to be ensured for the recovery paths of an application as well.
3. **High-performance secure networking:** Conventional approaches for network I/O (e.g., kernel-sockets) incur great overheads — especially in the context of TEEs due to switches between the trusted and untrusted world. While kernel-bypass networking vastly optimizes network operations, it is incompatible with TEEs, as untrusted DMA operations are prohibited in the enclave memory. On top of that, ensuring security and crash consistency when accessing PM remotely is another major challenge [100].
4. **Performance and transparency:** Aside from its memory safety and security properties, a dependable data management system should strive for high performance and be minimally intrusive to be practical. An effective solution should follow the established PM programming model, require minimal or no source code modifications, and be compatible with commodity hardware infrastructures. The induced performance, storage, and memory overheads should also be minimized within acceptable levels for efficient testing and debugging purposes and, ultimately, for production deployment, if possible.

1.3 Approach and Contributions

In this thesis, our overarching goal is to provide an end-to-end, dependable persistent memory architecture for untrusted cloud infrastructures. To this end, we design and build three systems that target different aspects of the dependability property; a *complete* memory safety solution for PM in SAFEPM, a *low-overhead* memory safety tool for PM targeting production environments in SPP, and an end-to-end PM data management system with *strong security properties*, namely data confidentiality, integrity and freshness, in ANCHOR.

We base the design of our memory safety tools for PM on well-established solutions for volatile memory. We explore two approaches for ensuring memory safety for PM applications, using either *shadow memory* (SAFEPM) or *tagged pointers* (SPP). These approaches present a trade-off between memory safety guarantees and memory and performance overheads. While the former allows for detecting both spatial and temporal memory bugs, it incurs considerable memory and performance overheads and is mainly targeting debugging environments. On the other hand, the latter comes with minimal memory and performance overheads but is capable only of detecting spatial memory violations. We provide the developer with the option to choose between those two variants and decide upon the required level of safety and the deployment target.

On the security frontier, we leverage modern hardware extensions for trusted computing, i.e., TEEs, and state-of-the-art network approaches, i.e., kernel-bypass networking, to build an end-to-end PM data management system (ANCHOR). We extend the trust of the TEE to untrusted PM by designing secure and authenticated PM data structures. Further, we preserve the security properties for both the PM data and our security metadata across system reboots/crashes through a secure logging protocol. Lastly, to facilitate remote PM operations, we propose a secure network stack designed with TEEs and PM in mind, along with a remote attestation protocol for trust establishment.

More specifically, **SAFEPM** [30] is a shadow-memory-based mechanism that provides comprehensive memory safety for PM-based applications. SAFEPM is based on AddressSanitizer (ASan) [264], a robust, well-tested tool for memory safety on volatile memory systems. It reserves a PM region (*persistent shadow memory*) where it places its memory safety metadata and utilizes ASan's compiler instrumentation and runtime libraries. SAFEPM also incorporates its own runtime library that instruments the PM management functions to reflect their modifications in the persistent shadow memory region in a crash-consistent manner. Thus, SAFEPM is able to detect both spatial and temporal memory safety bugs. Importantly, it is transparent to the application; it requires no modifications to the source code of the application, enabling easy integration into existing toolchains. Our evaluation shows that SAFEPM offers complete — spatial and temporal — memory safety for PM with reasonable performance overheads, e.g., $1.20-2.62\times$ slowdown for a PM KV store [117].

Safe Persistent Pointers (SPP) [286] is a spatial memory safety solution that protects against PM buffer overflows. SPP introduces the first tagged pointer scheme specially designed for PM. Safe Persistent Pointers enhance the durable representation of persistent pointers with memory safety metadata which gets embedded in the native tagged pointers during runtime. SPP's design relies on the pointer tag representation, which implicitly indicates whether a PM pointer has surpassed its boundary and enforces lightweight bound checks without the need for additional PM accesses. Its compiler instrumentation and runtime library further ensure that crash consistency is preserved for both PM data and memory safety metadata. On top of that, SPP's pointer encoding scheme is configurable. This option enables SPP to meet the PM management requirements of every PM application. Overall, SPP is a practical approach that detects PM buffer overflows, while its memory overheads and performance implications are kept minimal ($6 - 23\%$ slowdown on average), based on our conducted evaluation on several PM indices and a specialized PM KV store [117].

Anchor [285] is an end-to-end secure data management system for byte-addressable PM storage. It strives to provide strong security properties — confidentiality, integrity, authenticity, and freshness — for the data residing on PM and the (local/remote) PM data management operations. Anchor combines three non-trivially compatible, recent hardware advancements: TEEs, PM, and kernel-bypass networking. Our design encapsulates confidentiality-preserving data structures to extend the trust beyond the hardware-protected volatile memory. Further, Anchor leverages a trusted counter interface to provide rollback resilience while also preserving crash consistency through its formally verified secure logging protocol. Apart from that, our proposed system extends its scope by introducing a TEE-compatible network stack for PM based on kernel-bypass networking. Anchor also provides means to verify the authenticity of its instances through a formally verified remote attestation protocol. Our system exposes APIs for secure data management, networking operations, and remote attestation within the realms of the established PM programming model [114]. Our evaluation using YCSB workloads shows that Anchor incurs reasonable overheads considering its strong security properties. Thus, Anchor can be used to develop trusted applications both in a single-node setup as well as in distributed setups.

1.4 Thesis Outline

The chapters of this thesis are organized as follows:
Initially, we present the required technical background information (Chapter § 2). Then, we provide an in-depth look into our approaches to ensure memory safety solutions for persistent memory (Chapter § 3), where we present SafePM (Section § 3.1)

and SPP (Section § 3.2). Following, we highlight the security problems in persistent memory systems (Chapter § 4), and introduce our solution, ANCHOR (Section § 4.1) Finally, we conclude this thesis (Chapter § 5) and also discuss future research directions.

Chapter 2

Background

The main goal of the thesis is to design tools to build dependable persistent memory systems. To this end, this chapter provides the required background information about the technologies and system design concepts that the projects of the present thesis are built on. We commence our background section with a description of byte-addressable persistent memory (§ 2.1), which is the core target device for all projects in this thesis, namely SAFEPM, SPP and ANCHOR. Following, we provide an overview of memory safety (§ 2.2) to better understand the problem tackled by the SAFEPM and SPP projects. As a last part of this chapter, we describe modern, trusted computing technologies § 2.3 and high-performance networking techniques § 2.4, which are leveraged in ANCHOR to design a secure persistent memory system.

More specifically, section § 2.1 explains the architectural characteristics and idiosyncrasies of byte-addressable persistent memory. It presents the established byte-addressable persistent memory programming model § 2.1.1 and its nuances compared to the conventional volatile memory programming model. Then, we delve deeper into the core libraries and programming concepts of the Persistent Memory Development Kit (PMDK) [114], such as the persistent pointers, software transactions, and PM management, as they constitute the base layer for the projects described in this thesis.

Section § 2.2 highlights the importance of memory safety in modern security-critical systems and identifies the types of memory safety issues in low-level unsafe languages. We present and categorize existing software-based approaches for dealing with memory safety issues in § 2.2.1. Additionally, we summarize the proposed hardware-based approaches in § 2.2.2. Next, in § 2.2.3, we provide further design details about AddressSanitizer [264], a tool that serves as a foundation for our SAFEPM project. Lastly, we describe the core design principles of tagged-pointer-based approaches for memory safety, as our SPP project falls into this category, in § 2.2.4.

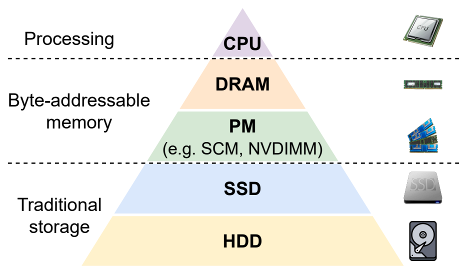
Section § 2.3 starts by explaining the state-of-the-art hardware extensions for trusted computing and analyzes their security guarantees in § 2.3.1. Then, we provide an analysis of the Intel SGX technology [138], followed by a detailed description of the attestation mechanisms for trusted execution environments in § 2.3.2. Lastly, we present an overview of shielded execution frameworks in § 2.3.3 that ease the application deployment in TEEs. We make a special reference to SCONE, the framework that acts as a base for ANCHOR.

Lastly, Section § 2.4 discusses the need for high-performance networking in modern disaggregated cloud systems. It emphasizes the functionalities and benefits of kernel-bypass networking techniques (e.g., DPDK [66], RDMA [155]). This section concludes with the presentation of the eRPC [153] framework, where ANCHOR's network stack is based.

2.1 Byte-addressable Persistent Memory

Persistent memory (PM) is a byte-addressable memory type with performance properties close to DRAM while providing durability [113, 192], aiming to bridge the gap between volatile memory and traditional storage. PM devices reside on the memory bus [219], providing access latency similar to DRAM. Figure 2.1 presents the system stack hierarchy, including representative capacity and latency values that highlight the order-of-magnitude differences in terms of capacity and latency between each layer.

The recently-emerged and evolving Compute Express Link (CXL) technology[90] further allows these devices to be attached to the PCIe bus, and with the addition of a network interface (e.g., RDMA-capable NIC), they can be exposed over the network enabling the creation of large pools of byte-addressable storage.



System Stack Hierarchy Characteristics		
	Capacity	Latency
Registers	8-512 B	< 1 ns
Caches	≈ 10 MB	< 10 ns
DRAM	≈ 100 GB	< 100 ns
PM	≈ 500 GB	≈ 400 ns
SSD	≈ 1 TB	≈ 10 µs
HDD	≈ 10 TB	≈ 10 ms

Figure 2.1: System stack hierarchy

PM can be used in two distinct modes [4, 127]; *(i) memory* and *(ii) app-direct* mode. In the former, the PM device acts as a volatile memory extension, while in the latter, data persists in PM even when the system is powered off. In the app-direct mode, PM interfaces with the OS via PM-aware direct access file systems (DAX) [105, 161]. DAX eliminates the page cache from the I/O path and allows `mmap(2)` to establish direct mappings to PM [112].

Thus, PM content can be directly accessed as memory-mapped files in an application's address space. PM devices are accessed with typical `ld/st` instructions. Applications use pointers to access PM, which must be reconstructible and consistent across reboots or crashes.

Importantly, writes to persistent memory are not guaranteed to persist until they reside in the power failure-protected domain. To ensure that, the application must flush the associated modified cache lines. The failure atomicity boundary is 8 B. Any update that spans a PM range larger than 8 B is not considered fail-safe and can be torn in case of a system crash. Fence placement (e.g., after the cache line flush) also plays a crucial role, as cache lines can be written back to PM out-of-order. Such architectural features have to be considered to avoid PM inconsistent states in case of system failures [254].

2.1.1 Persistent Memory Programming Model (PMDK)

Persistent Memory Development Kit (PMDK). Intel has developed a collection of libraries in Persistent Memory Development Kit (PMDK) [239] to support and facilitate application development for PM. The included libraries cover a wide range of applications and provide different ways to manage PM, ranging from low-level primitives [180] to a persistent transactional object store [120, 129] or an extension of volatile memory [122, 123], and a persistent key-value store [117]. It also exposes interfaces to create PM resident logs [119] or to treat PM as an array of persistent, crash-consistent blocks [118]. To enable developers to build end-to-end systems, PMDK further provides networking support through *librpma* [133]. It allows for accessing remote PM over Remote Direct Memory Access (RDMA). Lastly, PMDK provides additional tools for PM pool management [121, 124].

libpmemobj. The `libpmemobj` [120] library is a core component of PMDK. It contains a PM allocator [125] and implements software-based transactions to provide support for atomic updates to PM data leveraging redo and undo logging. It handles PM files as flexible object stores and exposes intuitive non-transactional as well as transactional APIs [109, 110] for PM management, similar to the conventional malloc/free API. `libpmemobj` organizes PM in files called PM *pools*, which are mapped into contiguous regions in the application's virtual address space. The organization of a PM pool is shown in Figure 2.2. PM pools contain a pool header, a section dedicated to the redo and undo logs required for transactions support, called *lanes*, and the persistent heap, which hosts the PM objects allocated by the application as well as heap metadata.

Persistent pointers. The operating system kernel can map PM pools to different regions of the address space in different runs. To maintain consistent persistent object

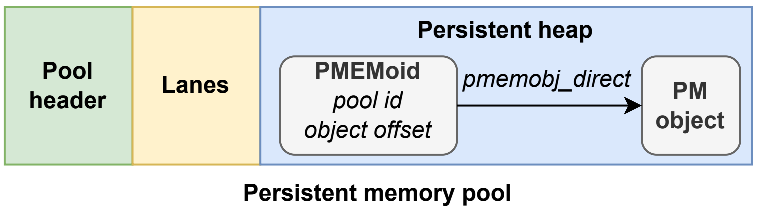


Figure 2.2: PM pool organization

references across restarts, `libpmemobj` introduces the concept of *persistent pointers*. In contrast to volatile pointers, persistent pointers are durable, crash-consistent data structures that are used to reconstruct native pointers to PM objects across application restarts or crashes. Precisely, persistent pointers employ a fat-pointer scheme [243], where each object is identified by a 16 B structure, called *PMEMoid*, demonstrated in Figure 2.2. A *PMEMoid* contains a *pool_id* field (8 B) and an *offset* (8 B) relative to the beginning of the pool. `libpmemobj` exposes the `pmemobj_direct()` function to construct the native pointer for an object based on its offset within the pool and the virtual address where the pool is mapped.

PMDK transactions. PMDK introduces software transactions to ensure the crash consistency of PM data updates exceeding the failure atomicity boundary of 8 B. In particular, *libpmemobj* exposes a transactional API [115] with durability, consistency, and atomicity semantics. For each transaction, a *redo* log stores the heap metadata updates while an *undo* log maintains snapshots of the PM objects involved in the transaction. After a crash, PMDK replays any live redo/undo logs to recover PM to a consistent state. Note that PMDK transactions do not provide data isolation; applications need to resolve any data races themselves.

Persistent memory management. The `libpmemobj` library provides a PM-optimized, crash-consistent allocator to manage the PM heap. It focuses on efficiency while minimizing fragmentation and avoiding PM leaks. `libpmemobj` splits the PM heap into smaller parts, called *chunks* and *runs*, and maintains heap-related data structures in volatile memory to improve its performance. It supports the common memory operations (e.g., alloc, realloc, free). It further offers both a non-transactional atomic [110] and a transactional API [109] for which the fail-safety is ensured with the use of redo logging.

2.2 Memory Safety

Low-level unsafe languages (e.g., C/C++) allow applications to directly interact with the system's memory. While this is powerful for optimizing performance, the poor built-in memory protection capabilities of such languages can lead to memory safety

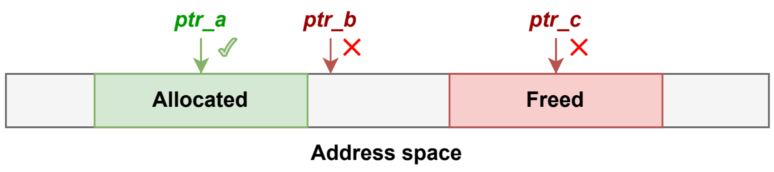


Figure 2.3: Memory safety errors: `ptr_a` *performs legitimate access to the allocated object;* `ptr_b` *tries to perform access outside the allocated bounds of the memory object resulting in a spatial memory safety error;* `ptr_c` *attempts to access a memory object after its deallocation which leads to a temporal memory safety error.*

violations with dire consequences when exploited by malicious attackers [197, 308]. There are two broad categories of memory safety bugs, *spatial* and *temporal*, illustrated in Figure 2.3. *Spatial* memory safety bugs refer to accesses beyond the intended boundaries of the targeted memory region (e.g., buffer overflows, stack overflows), whereas *temporal* memory safety bugs occur when a memory region is accessed before its allocation or after its release (e.g., dangling pointers, use-after-free, double free). These bugs constitute one of the main targets for attackers to get access to unintended memory regions and, thus, be able to hijack the control flow or leak sensitive data [197, 308].

To prevent such severe vulnerabilities, illegal memory accesses outside the allocated memory regions must be prevented. This requires a suitable instrumentation of an application to perform bound checks on memory accesses. Towards this direction, several memory safety techniques have been proposed based on software implementations [3, 68, 92, 170, 172, 220, 264] or on hardware modifications [14, 69, 223, 224, 322]. *Deterministic dynamic bounds-checking* is widely regarded as the only way of defending against all memory safety attacks [211, 289]. Bounds-checking techniques augment the original unmodified program with metadata (bounds of live objects or allowed memory regions) and insert checks against this metadata before each memory access.

2.2.1 Software-based approaches

Software-based approaches against memory safety bugs typically leverage different techniques, such as a compiler pass instrumentation accompanied by runtime libraries and compact representation of upper and lower pointer bounds, needed to perform the appropriate checks. They aim to minimize performance and memory overheads while maintaining compatibility and efficiency. More specifically, the proposed approaches can be classified into three broad categories according to their metadata granularity [211]:

(a) Trip-wire or shadow memory based approach. These approaches use a part of the available memory to store whether or not each fixed-size chunk of memory is accessible. This memory part is called *shadow memory*. Allocated memory regions are surrounded by guard regions, marked as inaccessible, allowing for the detection of memory safety violations [55, 88, 92, 93, 264].

(b) Object-based approach. Such approaches check all the pointer operations to ensure that the resulting pointer is not out-of-bounds with respect to the object it points to [3, 60, 61, 68, 69, 71, 147, 259]. They track metadata on a per-object level, i.e., they store the object bounds information on each object's allocation. Thus, they can verify that the pointer's object is not altered and the pointer remains valid. Relaxing this requirement, such as in SAFECode [61], allows for efficient optimizations and simplification of the runtime bounds checks.

(c) Pointer-based approach. Approaches of this category [146, 170, 212, 213, 216, 218, 275] store metadata on a pointer-based level. This metadata indicates the bounded memory region (i.e., upper and/or lower bounds) that a pointer is permitted to access. On every memory access, a bound check is performed to query the validity of the operation. Note how SoftBound [212] associates metadata not with an object but rather with a pointer to the object. This allows pointer-based techniques to detect intra-object overflows (e.g., one field overflowing into another field of the same struct) by *narrowing the bounds* associated with the particular pointer.

2.2.2 Hardware-based approaches

There also exists a large body of work that enforces memory safety for volatile memory using hardware extensions [14, 59, 69, 166, 209, 224, 322]. The introduction of hardware modifications primarily aims to achieve a lower performance overhead. Lowfat pointers [69] enforce spatial safety by associating the pointer with its bounds. In particular, Lowfat pointers approach is based on encoding the bounds information (size and base) directly into the native bit representation of a pointer itself. This bounds information can then be retrieved at runtime and be checked whenever the pointer is accessed, thereby preventing out-of-bounds accesses. It contains gate-level implementations of the logic for updating and validating the compact fat pointers. Cheri [322] ensures memory safety bug detection with the support of hardware capabilities. Cheri uses a fat pointer which includes bounds information and permission bits. Upon every memory access, this embedded metadata is checked. However, the implementation requires modifications to the entire system, including the pipeline stages, compiler, language runtime, and the OS. Intel MPX [224] is a hardware-assisted pointer-based mechanism. It provides ISA extensions of Intel x86-64 architecture for memory protection. Arm MTE [14] is an ARM extension that enables hardware-assisted memory tagging to detect both temporal and spatial memory safety bugs. Nevertheless, the

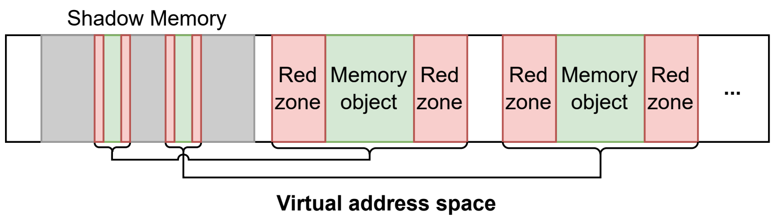


Figure 2.4: Shadow memory-based approach, enforced by ASan.

complexity of the required hardware modifications for these approaches and their inability to be applied to commodity hardware are a burden towards their incorporation in the production of established systems.

2.2.3 ASan: a Shadow Memory-based approach

Among all the memory safety approaches, the shadow memory-based approach, as adopted by AddressSanitizer (ASan) [264], is the most popular memory safety technique [289]. It is widely used by Google and other organizations to detect memory safety violations. ASan supports both GCC [76] and Clang/LLVM [43]. It consists of a compiler instrumentation module and a runtime library. The design of this approach is shown in Figure 2.4.

In particular, ASan reserves a part of the address space for *shadow memory*, where it keeps metadata indicating the state (i.e., accessible or not) of the memory regions of an application, including the stack, heap, and global variables. ASan updates the shadow memory whenever an object is created, freed, or moved. It surrounds objects with memory regions, called *red zones*, which are marked as inaccessible (poisoned) in the shadow memory. On each memory access, ASan checks if the requested address is addressable. These checks are added via ASan's compiler instrumentation module. Any access to an unallocated region or red zone is detected, usually resulting in the crash of the program before its state can be corrupted or sensitive information leaks. Additionally, to ensure temporal memory safety, ASan implements a quarantine zone for recently freed objects, which prevents their regions from being allocated for some time.

Further, ASan is shipped with a runtime library, which is responsible for the initialization of the shadow memory. The size of the shadow memory is 1/8th of the virtual address space. ASan's runtime library also replaces the memory management functions (e.g., malloc, free) to perform the required red zone allocations and to mark both the red zones and the entire freed objects as inaccessible in the shadow memory during runtime.

However, ASan also comes with some known limitations. The additional memory

(≈12.5%) and runtime overhead (≈70% [264] introduced by red zones, shadow memory, and frequent memory access checks can slow down applications, especially those with intensive memory usage. Moreover, ASan primarily detects spatial and, probabilistically, temporal memory errors but may miss other certain types of code injection or control-flow attacks, i.e., when they exploit vulnerabilities outside of strictly allocated regions. As a result, while ASan is highly effective for memory safety violations, it may not fully protect against all classes of memory safety vulnerabilities.

2.2.4 Tagged pointer approaches

Tagged pointer approaches [14, 170, 172, 216, 326] encode memory safety metadata within the pointer representation. Thus, they can determine — directly or via additional memory accesses — the ranges (i.e., upper and/or lower bounds) that a pointer is allowed to access and perform runtime checks on every `ld/st`. Such approaches present trade-offs in terms of *(i)* efficiency, *(ii)* runtime overheads, and *(iii)* compatibility [215]. Overall, their adoption in production systems is limited mostly due to their performance overheads (50%-150%), stemming from their application instrumentation, costly runtime checks, and extra memory accesses.

Delta Pointers. Tagged pointer approaches, such as Delta Pointers [170], provide a practical approach to ensure buffer-overflow attack prevention having low performance and memory overheads by avoiding extra memory accesses for memory safety metadata fetching and omitting the explicit runtime bound checks. More precisely, Delta Pointers employ a *pointer tagging* scheme to detect buffer overflows. Each pointer incorporates its upper bound encoded in its spare bits. Note that the altered pointer representation does not violate the language specification, promoting compatibility. The pointer tag is updated on every pointer arithmetic operation and its value indicates whether a pointer has surpassed its assigned boundary. The actual bound checking is performed at pointer dereference and requires no additional memory access to metadata. Any attempt to access an invalid pointer leads to an application crash, prohibiting data corruption and information leaks. Delta Pointers rely on a compiler instrumentation module to inject the appropriate code instructions that manage the pointer tags.

2.3 Trusted Computing

2.3.1 Trusted Execution Environments

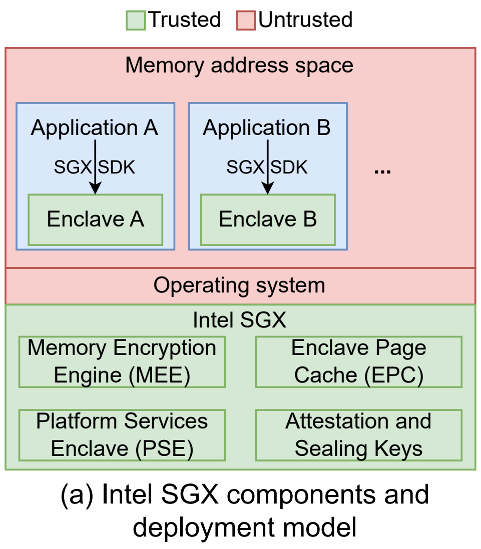
Trusted Execution Environments (TEEs) [5, 12, 15, 132, 138, 177] provide a hardware-protected volatile memory region (or *enclave*) that ensures the confidentiality and integrity properties of the enclosed running code and data residing within this isolated memory region. TEEs introduce the concept of a trusted computing base (TCB), which encompasses the code and data protected by trusted hardware. Overall, TEEs protect the executed software and runtime data against any software attacks, even from privileged code, e.g., a compromised operating system or hypervisor.

TEEs are a significant advancement over the previous state-of-the-art approaches for trusted computing, namely Homomorphic Encryption (HE) [95] and Trusted Platform Modules (TPM) [294]. TPMs have standard pre-configured functionalities by the manufacturer which ensure the integrity and confidentiality properties of the executed code. They provide similar properties for their securely stored secrets (e.g., encryption keys). However, TPMs cannot provide any further security guarantees for the used data. On the other hand, while HE allows computations on encrypted data without decryption, it lacks the capability to ensure the confidentiality and integrity of the executed code. On top of that, HE is limited as only certain computations on encrypted data are feasible [97, 208]. Unlike the former approaches, TEEs can execute arbitrary code securely and provide strong isolation for both data and code in use. Thus, they offer better security guarantees and adaptability than HE and TPMs, which come with inherent functionality limitations.

More specifically, the key security features of TEEs are:

- *Confidentiality:* TEEs encrypt the trusted, isolated memory to prevent unauthorized access or memory snooping.
- *Integrity:* TEEs prevent the tampering of the hardware-protected memory regions.
- *Data freshness:* TEEs protect against rollback attacks on their trusted memory regions, where attackers might attempt to revert the system to a stale but valid state.

TEEs [12, 138, 177] typically distinguish between untrusted and trusted ”worlds” in a system. Therefore, they mandate the separation of the applications into untrusted and TEE-protected parts and require their rewriting using specialized Software Development Kits (SDK) [162, 267] or designated APIs [296] to efficiently use the hardware-protected memory regions. However, newer TEE technologies, such as AMD-SEV(-SNP) [5], Intel TDX [132] and ARM Realms [15], have emerged, which integrate entire virtual machines (VMs) within the protection domain of the trusted hardware, offering Confidential VMs (CVMs). In particular, they offer a different layer of protection by creating a virtualized secure environment within each VM instance, which encrypts data-in-use and further ensures the isolation of sensitive data and applications. CVMs, compared to earlier TEEs, feature a larger TCB but simplify the application development process by exposing OS-based programming interfaces. This trend towards VM-



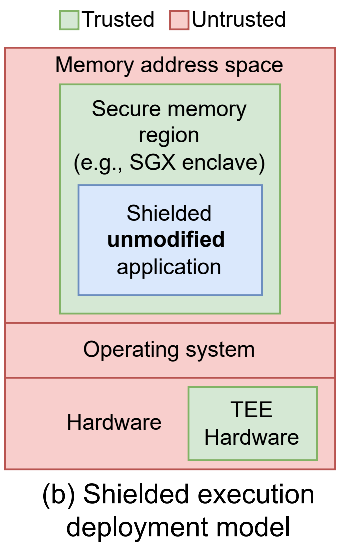

🟩Trusted 🟥Untrusted
Memory address space
Secure memory region (e.g., SGX enclave)
Shielded **unmodified** application
Operating system
Hardware
TEE Hardware
(b) Shielded execution deployment model

Figure 2.5: System stack overview for Intel SGX and shielded execution frameworks.

based TEEs has been growing over the past few years due to their enhanced ease of use and deployment.

To this end, various major CPU providers have developed TEEs, including Intel SGX [138], Intel TDX [132], AMD-SEV(-SNP) [5], Arm TrustZone [12], Arm Realms [15] and RISC-V Keystone [177]. These technologies have been adopted by significant cloud providers like Amazon Web Services (AWS) [21], Microsoft Azure [202, 203], Alibaba Cloud [44], and Google Cloud [81, 82] enabling clients to deploy security-critical applications in the cloud.

Intel SGX. Intel SGX [138] consists of a set of x86 ISA extensions for TEEs, commercially available from the Skylake generation [321]. Some of its key architectural components and its deployment model are depicted in Figure 2.5(a). It is designed to provide applications with a protected area of execution, known as an *enclave*, which ensures that sensitive data and code are isolated and protected. To make use of the enclave memory, the developers need to use a specialized SDK [267] and define the parts of the application (e.g., security-critical code and data) that need to be placed in the hardware-protected memory region. The enclave memory in SGX is mapped in the physical memory as an *Enclave Page Cache (EPC)*, where the pages are protected by an on-chip Memory Encryption Engine (MEE). Precisely, the EPC is a reserved area in the CPU's memory hierarchy where the encrypted enclave pages are stored. The EPC ensures that the pages of an enclave are kept separate from those of other enclaves and regular applications, enforcing strong isolation. The MEE is responsible for the encryption and decryption of the contents of the enclave when they are stored in the main memory. This component ensures that the data is protected from any attacks that involve direct access to the memory [84], such as cold boot attacks.

Importantly, the EPC is limited in terms of size; 128MB—256MB for SGX-v1/v2, respectively. To accommodate larger enclave sizes, SGX offers a secure paging mechanism. However, this mechanism incurs prohibitive performance overheads (up to 2000$\times$) [16, 25, 226] due to the mandatory encryption/decryption during the paging process. To alleviate this limitation, recent Intel CPUs switched from using the MEE to *Advanced Encryption Standard - XEX Tweakable Block Cipher with Ciphertext Stealing (AES-XTS)* [1] to be able to support up to 512GB of EPC [221].

Aside from the often limited EPC size, system designers have to consider the following important Intel SGX architectural aspects. SGX applications cannot execute code outside of the enclave directly (e.g., system calls), as the operating system is not trusted. To perform system calls, SGX enclave threads must exit the protected domain, namely, perform a ”world switch” and copy all associated data outside the enclave to make it available for the kernel to access. After the system call is processed, the threads must re-enter the enclave and copy the results back into the trusted environment. This process is an expensive, performance-heavy (5$\times$) operation [293]. Additionally, Intel SGX includes the Platform Services Enclave (PSE) in its software package [249]. The PSE provides additional services such as monotonic counters and trusted time, which can be used by applications within the enclave to implement their time-related logic (e.g., certificate expiration). Nonetheless, not all of the services are designed with performance and sustainability in mind. For instance, while Intel SGX offers a hardware-trusted monotonic counter, it is quite slow (60-250ms) and can wear out after some days of continuous use [194].

2.3.2 Attestation

Attestation is a security mechanism that confirms the integrity of both the hardware and software states on a machine, ensuring that the correct, expected software stack is running on the designated hardware. This process requires a trusted entity, known as the root-of-trust, e.g., the underlying TEE hardware, to verify the loaded code and generate a secure, cryptographic hash value that resembles a measurement of the loaded software. Initially, the root-of-trust calculates a measurement of all fundamental software components, including the bootloader, OS kernel, hypervisor, and preliminary software within the TEE, whichever is applicable. Subsequently, the loaded software measures additional components, such as arbitrary code of loaded applications, and reports these measurements to the root of trust. The root-of-trust then combines its initial cryptographic hash value with the new measurements, resulting in the final measurement value. This value can then be signed using a hardware key embedded in the platform during the manufacturing and later be used to prove that the software inside the TEE has not been tampered with and is running on a legitimate platform.

There exist two distinct types of attestation:

- *Local attestation* refers to the attestation process between two TEE instances on the same platform. In this scenario, one TEE instance, the challenger, can request a cryptographic report containing the final measurement of the other TEE instance's code and data, which is then signed by the processor's hardware, as described above. The challenger can use this report to confirm that the instance in question executes the expected software on the same platform. Typically, this process involves a trusted verifier entity, supported by the TEE, running on the same platform.
- *Remote attestation* refers to the attestation process between a TEE instance and a third-party entity that does not reside on the same platform. The remote entity, or challenger, can request the root of trust to sign the final measurement using the embedded hardware key. By examining this signed value, the remote process can verify the authenticity of the entire remote system stack, including the legitimacy of the hardware as well as the software state. The challenger achieves this by comparing the signed measurement, obtained from the root of trust, with an already calculated value based on the combination of the hardware of the target platform and the intended executable software. Typically, the remote attestation process involves a remote trusted entity provided by the hardware vendor [6, 102, 139].

Intel SGX Attestation. Intel SGX includes mechanisms for *attestation*, i.e., to prove the identity and integrity of an enclave to remote parties, and *sealing*, i.e., to securely store data that can only be accessed by the same enclave in future sessions. These mechanisms rely on keys derived from a hardware root-of-trust. An enclave can derive an enclave-specific SGX Report, which is forwarded to the Quoting Enclave, where an *SGX quote* is produced. The SGX quote contains the measurement of the code and data residing inside the enclave, enclave-specific attributes, and other security-relevant fields and is tied to the identity of the Quoting Enclave on this platform. This SGX quote can then be sent to a remote entity to perform the remote attestation process.

To facilitate remote attestation, Intel provides the Intel Attestation Service (IAS). The CPU manufacturer, and consequently the IAS that is managed by Intel, is considered within the trust boundaries. IAS can verify the quotes produced by a legitimate TEE and then issue a verification report upon successful validation of the TEE's quote. This verification report is then sent back to the TEE and can be provided to a remote challenger upon request. The remote challenger can then verify the IAS report to gain confidence in the authenticity of the TEE and its running software.

2.3.3 Shielded execution

TEEs are widely adopted by cloud providers to support secure applications in the cloud [21, 44, 81, 82, 103, 202, 203]. However, to attract the interest of clients, supporting legacy applications without the need to port them to use specific SDKs or APIs is crucial. To this end, *shielded execution* frameworks [16, 27, 226, 247, 272, 293, 297], built based on TEEs, aim to provide strong confidentiality and integrity guarantees for *unmodified* applications deployed on an untrusted computing infrastructure. The core system stack and deployment model of *shielded execution* is highlighted in Figure 2.5(b). Such frameworks require no application source code modifications and transparently confine the application address space inside the TEE's hardware-protected volatile memory region. Further, modern shielded execution frameworks have employed direct I/O stacks in the context of TEEs [23, 25, 293, 295] to minimize the performance implications caused by enclave transitions (world switches) and keep up with the ever-increasing performance demands in cloud computing.

In particular, the SCONE [16] framework is built based on Intel SGX. In SCONE [16], the applications are linked against a modified standard C library (SCONE libc). In this model, the interaction with the untrusted memory is performed via the system call interface. SCONE runtime provides an asynchronous system call mechanism [277] in which threads outside the enclave asynchronously execute the system calls. SCONE protects the executing application against Iago attacks [16, 35] through shields. Furthermore, it offers the option to prevent memory safety bugs in the applications running inside the SGX enclaves [172]. Lastly, SCONE provides an integration with Docker to allow for seamlessly deploying containers.

2.4 Kernel-bypass Networking

For modern disaggregated cloud systems, high-speed networking [65, 108, 152, 329] is an essential and performance-critical component to provide a holistic environment for high-performance computing and reap the benefits of the fast storage devices, i.e., persistent memory.

More specifically, high-performance networking infrastructure in the cloud is increasingly based on userspace, kernel-bypass abstractions [22, 301–306] such as DPDK [66] and RDMA [86, 155]. Unlike conventional system call-based networking, where the network stack and the I/O handling are managed by the OS kernel, kernel-bypass networking eschews the OS and alleviates any potential bottleneck in the kernel networking stack by directly interacting with the network interface card (NIC) hardware, as presented in Figure 2.6. The direct memory access (DMA) mechanism

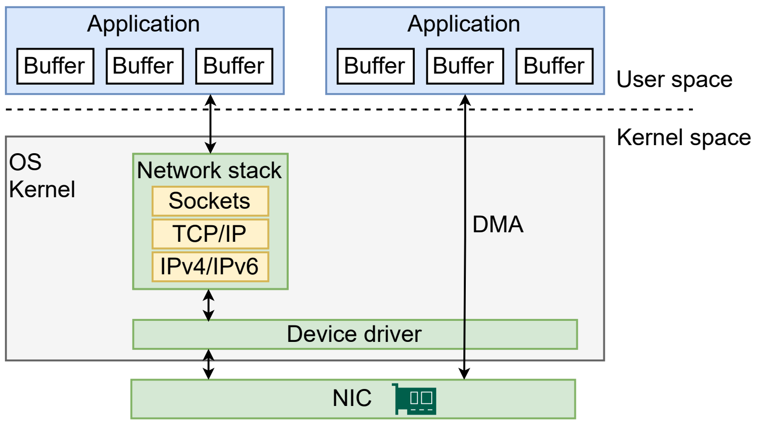


Figure 2.6: Conventional, kernel-based networking stack (left) and kernel-bypass networking approach (right).

allows for setting up a direct communication channel between the NIC and the system, enabling the NIC to read from or write to the system's main memory without CPU intervention. With DMA, an application can establish a mapping of the system memory region allocated for DMA transfers — not the NIC's internal memory or configuration space — to its virtual address space with the assistance of the operating system.

In this way, the device can be controlled directly from the userspace. Thus, the kernel-bypass approach improves both the network latency and throughput as the overheads stemming from extra data copies in and out of the kernel as well as from expensive context switches [51, 89, 145, 277, 311] are eliminated.

More specifically, one-sided RDMA [155] allows for data movement to or from a remote node without CPU intervention. It enables zero-copy networking as the data is moved directly between the application memory and the NIC buffers. The CPU on the initiator node (regardless if it is a read or a write operation) only needs to start the operation. Then, the NIC leverages DMA to transfer the data directly between the system memory of the participating remote nodes.

To simplify kernel-bypass network programming, remote procedure call (RPC) frameworks such as eRPC [153] provide a general yet performant API for asynchronous RPCs designed for high-speed networking for lossy Ethernet or lossless fabrics. Typically, such frameworks bind the device to an application and do not expose the abstractions of the transport layer (e.g., TCP/IP). They hide the complexities of managing the low-level interfaces of the transport layer (RDMA, DPDK, and RoCE) from the developer, resulting in an easier programming experience.

RDMA-based RPCs have been demonstrated by research and industry efforts [10, 67, 104, 136] to be the most efficient programming paradigm for high-performing cloud systems, especially when they incorporate fast, byte-addressable storage [37, 133, 142].

Inter-data-center communication usually relies on traditional networking protocols (e.g., Ethernet, TCP/IP) offered by telecommunication providers, while RDMA is typically used within the same data center, where the high-speed and low-latency network fabric can be tightly controlled. Thus, in such settings, security is often assumed to be inherent. In these environments, RDMA implementations often prioritize performance over extensive security features, assuming a relatively secure and controlled intra-data-center environment.

Nonetheless, as data center networks evolve, even within a single data center, the threat landscape broadens with the emergence of programmable network devices, such as SmartNICs and programmable switches. These devices, if compromised, could potentially be leveraged to leak or manipulate RDMA traffic. Addressing such threats is crucial, especially when security-sensitive or private data is being involved. Therefore, security assumptions around RDMA need to be reevaluated, and security mechanisms tailored to protect against intra-data center attacks (e.g., traffic encryption and access control at the NIC level) need to be solidified.

Chapter 3

Memory Safety for Persistent Memory

3.1 SAFEPM: A Sanitizer for Persistent Memory

In this section, we introduce SAFEPM, a memory safety mechanism that *transparently* and *comprehensively* detects both spatial and temporal memory safety violations for PM-based applications. SAFEPM's design builds on a shadow memory approach, and augments it with crash-consistent data structures and system operations to ensure memory safety even across system reboots and crashes. We implement SAFEPM based on the AddressSanitizer compiler pass and integrate it with the PM development kit (PMDK) runtime library. We evaluate SAFEPM across three dimensions: overheads, effectiveness, and crash consistency. SAFEPM overall incurs performance and memory overheads comparable with ASan while providing comprehensive memory safety, and has uncovered real-world bugs in the widely-used PMDK library.

3.1.1 Motivation

Performance-critical software systems are prominently written in low-level languages, such as C/C++. While these languages allow the programmer to explicitly control the application's memory, they can, unfortunately, lead to memory safety bugs, i.e., *illegal accesses to unintended memory regions* [28, 70, 289, 308, 332]. More specifically, memory safety violations are categorized as *spatial* and *temporal* errors. Spatial violations occur when an operation accesses a memory region outside its assigned boundaries (e.g., buffer overflows). Temporal violations are accesses to a memory object before its creation or after its deletion (e.g., dangling pointers).

This fundamental trade-off between performance and memory safety in the context of low-level unsafe languages manifests in the form of numerous dependability issues in software systems. For instance, apart from several well-known memory safety violations [94, 96, 284], three major projects: Chromium [42], Android [7] and Windows [206] report that $70-75\%$ of their discovered issues are memory safety bugs. These safety violations are widespread —three out of ten most critical software weaknesses are memory safety issues [205]. More importantly, Szekeres et al. [289] illustrate that memory safety is the root cause of security issues in software systems.

Byte-addressable persistent memory (PM), similar to volatile memory, is also susceptible to memory safety violations. In particular, PM is a non-volatile storage medium, accessible at a byte granularity via `load/store` instructions with access latencies close to DRAM [143]. PM content is directly mapped into an application's address space and is manipulated at a byte granularity through native pointers, making PM programming prone to memory safety errors, especially in the context of memory-unsafe languages (e.g., C, C++).

Over the last two decades, a range of hardware- and software-based memory safety approaches [223] have been proposed to tackle the problem of memory safety for volatile memory (§ 3.1.6). These approaches are prominently designed based on *deterministic dynamic bounds-checking*, which relies on compile-time code instrumentation and enhances the original application's memory layout with memory safety metadata, which is checked during the runtime upon each memory access.

While these memory safety approaches are extensively used in commercial software eco-systems for volatile memory-based applications, either during development [137, 198, 264] or production phases [14, 101, 280]—there exists a distinct research gap when we consider memory safety issues in the context of the emerging persistent memory technology. That is, there exists *no memory safety mechanism designed for PM-based applications written in unsafe languages*.

Importantly, the state-of-the-art memory safety approaches for volatile memory are insufficient for PM. Unlike volatile memory, which uses native volatile pointers, the PM programming model introduces a persistent pointer representation for its objects, and the persistent memory heap is handled using memory allocators designed for PM, which rely on persistent heap metadata [57, 120]. In addition, although the types of memory safety vulnerabilities on PM remain the same as those on volatile memory (e.g., buffer overflows, dangling pointers), memory safety needs to be further ensured for the recovery code paths designated to be executed after a potential crash and/or a restart.

Therefore, we need to design a memory safety mechanism that simultaneously ensures the correctness and crash consistency of both the application's data and the memory safety metadata.

To address this problem, we propose SAFEPM, a memory safety mechanism for PM-based applications. More precisely, SAFEPM provides *comprehensive memory safety* by detecting both spatial and temporal memory safety bugs. It is *transparent* to the application: while it requires recompilation of the application, no source code modifications are necessary. Lastly, SAFEPM ensures the *crash consistency* property while incurring tolerable performance and memory overheads for its debugging purposes.

At a high level, SAFEPM's design is based on a shadow memory approach (§ 2.2). It reserves a PM region (*persistent shadow memory*) where it places its memory safety metadata. This PM part stores the state — accessible or not — of each 8-byte chunk of PM. The persistent shadow memory is also mapped over a specific, precisely calculated location of the virtual address space (§ 3.1.3.1). We call this operation *overmap*. Thus, SAFEPM checks this overmapped PM region on every memory access during runtime. These checks are injected in the application through the compiler pass instrumentation of AddressSanitizer (ASan) [264]. Further, SAFEPM ensures that each PM object is surrounded by guard regions (*red zones*) that are marked as inaccessible in the persistent shadow memory. Lastly, SAFEPM incorporates a runtime library that instruments the PM management functions to reflect their modifications in the persistent shadow memory region in a crash-consistent manner.

We implement SAFEPM based on Intel's PMDK and ASan [264]. SAFEPM's persistent shadow memory adopts the same format as that of ASan, i.e., it preserves the internal structure of ASan's shadow memory and uses the same byte-to-byte mapping scheme, where every 8 bytes of application memory are represented by 1 byte of shadow memory. This shadow memory is stored as part of the persistent pool created by SAFEPM. SAFEPM keeps PMDK's memory layout intact, such that a persistent pool created by SAFEPM is a valid PMDK pool. When a persistent pool is opened, SAFEPM maps its persistent shadow memory over the relevant portion of the ASan's shadow memory. Further, SAFEPM's runtime library augments the functionality of PMDK's libpmemobj [120] PM management routines to incorporate the handling of the persistent memory safety metadata. Thus, the PM heap operations that modify the state of a PM range transparently update the corresponding part of the persistent shadow memory. These operations allow SAFEPM to keep the ASan's compiler pass and optimizations intact and use them for memory checks while supporting the programming interface and semantics of PMDK without requiring modifications to an application's source code.

We evaluate SAFEPM along three dimensions: *(i)* performance and space overheads using PMDK's benchmark framework, *pmembench* [126] and a persistent KV store, *pmemkv* [117], *(ii)* effectiveness using the *RIPE* framework [320] and *(iii)* crash consistency, which we validate for *pmembench* using the established *pmemcheck* tool [240]. Our evaluation shows that SafePM offers the same memory safety guarantees for persistent memory as ASan provides for volatile memory with reasonable performance overheads, e.g., $1.20 - 2.62\times$ slowdown for the KV store while ensuring crash consistency. Through SafePM, we have also identified two memory safety bugs in the widely-used PMDK library.

Overall, SafePM makes the following contributions:

- We present the design (§ 3.1.3) of SafePM, the first solution for comprehensive spatial and temporal memory safety for PMDK-based PM applications.
- Our design leverages a shadow memory approach transparently supporting the established SNIA NVM programming model [276], as implemented by PMDK [114], thus allowing seamless integration into existing testing workflows and toolchains.
- We prototype (§ 3.1.4) and extensively evaluate SafePM across three metrics: overheads (§ 3.1.5.2, § 3.1.5.3, § 3.1.5.6), effectiveness (§ 3.1.5.4, § 3.1.5.7) and crash consistency (§ 3.1.5.5).

3.1.2 Overview

SafePM is an efficient tool that provides comprehensive memory safety for PM applications developed with PMDK. Current state-of-the-art approaches for ensuring memory safety for PM incorporate prohibitive memory and performance overheads (e.g., memcheck [198]) or involve source code modifications, the adoption of a new library and its APIs, or even the rewriting of the application to a memory-safe language, such as Rust (e.g., Corundum [99]). Unlike those, SafePM is transparent to the application, requiring no changes to application source code or to existing PMDK-based libraries. Instead, SafePM operates at the compiler level, instrumenting the application code automatically and incurs reasonable overheads comparable to those of ASan for volatile memory.

SafePM leverages ASan to detect spatial and temporal memory safety violations within a PM pool without requiring any source code modifications. Our key insight is to benefit from the optimizations and the efficiency of the solidly engineered, well-tested, and widely-used ASan. Therefore, SafePM keeps ASan intact and adds metadata to PM pools to integrate with ASan (§ 3.1.3). In SafePM, the memory safety metadata follows the ASan's format, and the runtime checks are inserted by ASan 's compiler pass. The core difference is that SafePM keeps this metadata valid across application restarts/crashes and pool re-openings by placing it inside the persistent memory pool and updating it in a crash-consistent manner. More specifically, SafePM modifies the PM pool layout and transparently stores the persistent shadow memory as a PM object,

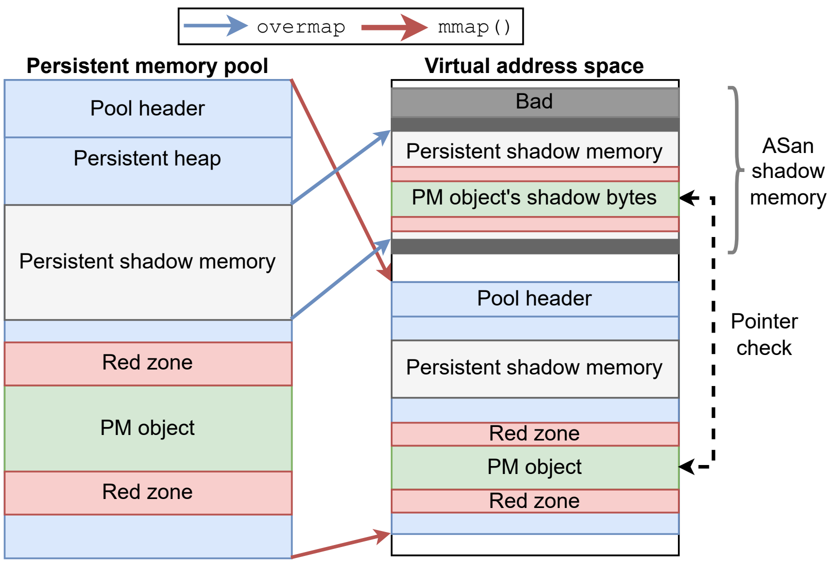


Figure 3.1: Overview of SAFEPM.

which is retrievable through the introduced shadow root object (§ 3.1.3.1). Thus, SAFEPM is compliant with PM programming concepts and is able to detect persistent memory safety violations even after unexpected shutdowns/crashes.

An overview of SAFEPM's modified PM pool design is shown in Figure 3.1. PM pools are directly mapped to the virtual address space. SAFEPM reserves a part of the PM pool heap for the *persistent shadow memory* (PSM), which maintains the memory safety metadata of the pool. The part of the pool corresponding to PSM is mapped over the ASan's shadow memory during the initialization phase of SAFEPM. We name this core operation of SAFEPM as *overmap*. Further, SAFEPM's memory allocator inserts and poisons the appropriate *red zones* in the same way ASan does for volatile memory while preserving the PM programming model and the crash consistency of the data and metadata.

Design goals. SAFEPM achieves the following goals:

- *Memory safety:* SAFEPM provides memory safety for a wide range of potential PM access violations, including both spatial (e.g., PM object-overflows) and temporal (e.g., use-after-free) in a similar manner as ASan does for volatile memory.
- *Transparency & compatibility:* We design SAFEPM based on PMDK and ASan, which allows for seamless integration in existing PM applications without code modifications.
- *High code coverage:* SAFEPM should be able to detect memory safety violations in different code paths , including recovery paths for abrupt shutdowns. This is achieved by ensuring the crash consistency for both PM (meta)data and SAFEPM's metadata

leveraging the transactional interfaces and logging mechanisms provided by PMDK.

- *Performance:* SafePM keeps the compiler pass of ASan intact and leverages its optimizations to limit the introduced overheads due to the additional memory checks, making SafePM suitable for performance-critical environments.

3.1.2.1 System Model

Fault model. SafePM aims to ensure memory safety for PMDK-based applications. It is capable of detecting and reporting both spatial (e.g., object overflows) and temporal (e.g., use-after-free) memory safety bugs in PM.

SafePM must also preserve the crash consistency property, as it targets PM-enabled applications. Crashes or unexpected system shutdowns can lead to data inconsistencies. This means that SafePM has to enforce mechanisms to ensure the recovery to a consistent state not only for the PM (meta)data but also for the memory safety-related metadata. Further, SafePM extends the scope to *also* provide memory safety guarantees for the recovery path.

Usage model. SafePM is a generic testing tool to prevent memory safety bugs in PMDK programs during the development phase. It provides the same memory safety guarantees as ASan. Further, it also offers "partial safety coverage" to manually select the code parts where the checks will be applied to limit the performance overhead.

Programming model. SafePM is based on PMDK which is entirely written in the unsafe C/C++ languages. SafePM preserves the PM programming model, its semantics as well as the PMDK APIs intact.

3.1.2.2 Design Challenges

#1: Transparency. An effective memory safety testing tool should require no source code modifications. State-of-the-art approaches, like ASan [264] and memcheck [198], already cover this need with the instrumentation of the volatile memory management functions. SafePM should provide the same level of transparency for the PM pool heap management APIs.

Approach: SafePM preserves the PM programming model and instruments the PM management functions via carefully designed wrappers (§ 3.1.4). Precisely, SafePM adapts the functions that manage the pool (create/open/close) and the PM objects (alloc/realloc/free) so that their changes are reflected in the PSM. SafePM supports PMDK's PM management APIs, requiring no source code modifications.

#2: Compatibility with ASan. ASan is one of the most prominent tools to detect memory safety violations in the volatile memory regions of an application. SafePM needs to extend ASan checks to objects residing in PM, which are managed by `libpmemobj`'s memory management functions.

Approach: During pool creation, SafePM reserves a shadow memory region inside PM pools for the PSM. This region corresponds to the respective shadow memory used by ASan. To comply with ASan's design, SafePM also augments the PMDK allocator to surround the PM objects with poisoned red zones as well as to modify the PSM following the same binary format as ASan. To preserve transparency, SafePM wraps memory allocators to skip the red zones and return pointers to the object data as a programmer would expect.

#3: Durability and crash consistency. PM applications have to ensure the durability and the crash consistency for the PM data in case of an unexpected shutdown. In SafePM, the scope of crash consistency is extended to include memory safety metadata as well. This implies that to integrate PMDK with ASan, SafePM has to ensure the durability and crash consistency of the memory safety metadata required by ASan along with the user-specific PM residing data. This is imperative to ensure that protection against memory safety vulnerabilities remains intact even after crashes and unexpected shutdowns, without the need to reconstruct the metadata from scratch.

Approach: SafePM reserves the memory safety metadata as a part of the PM pool, thus achieving its durability. Metadata durability ensures that upon an application restart or recovery after a potential crash, SafePM is aware of the (de)allocations made inside the PM pool and is able to correctly check for memory safety violations based on the persistent, valid state of the memory safety metadata.

To ensure crash consistency and, subsequently, the validity of its memory safety metadata along with PM pool's (meta)data, SafePM leverages PMDK's software transactions. Each PM object modification (alloc/realloc/free) needs to be reflected in the shadow memory representation. Since SafePM's metadata is part of the pool, its modifications are performed inside transactions along with the respective PM management operations, guaranteeing crash consistency via PMDK's logging mechanism.

#4: Coverage of recovery paths. PM applications are designed to be able to recover from abrupt crashes. This requires special code paths that restore the PM to a consistent state. SafePM has to ensure memory safety for these recovery paths.

Approach: SafePM maintains the memory safety metadata as part of the PM pool. Unlike with ASan, where the memory safety metadata is volatile and reconstructed from scratch on each application run, SafePM's memory safety metadata remains consistent and can be retrieved across reboots or failures. Thus, SafePM ensures the durability of its metadata along with its crash consistency and can enforce memory safety on PM objects even on the application's recovery code paths.

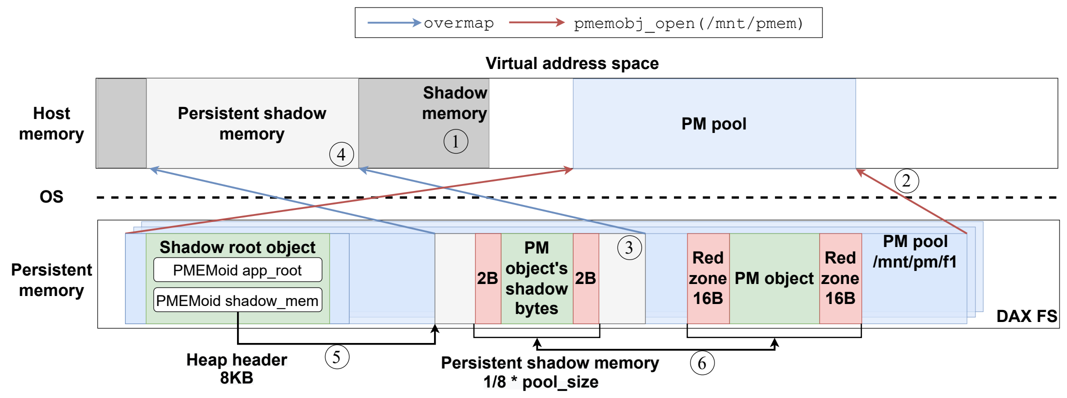


Figure 3.2: Detailed architecture of SafePM.

3.1.3 Design

Figure 3.2 illustrates an overview of SafePM's components. SafePM reserves part of the PM pool for the PSM and *overmaps* it to the location expected by ASan in the program's virtual address space. Moreover, SafePM augments the memory management operations of PMDK to surround allocated objects with poisoned *red zones* and update the PSM accordingly.

Precisely, at the startup of an application, SafePM reserves a shadow memory region ① to track the state of each memory area, similar to ASan. Then, SafePM proceeds with the creation or opening of PM pools that are mapped to the virtual address space ②. During pool creation, SafePM allocates a portion of the pool as its PSM to store the state of each byte in the PM pool ③. SafePM then maps the PSM onto its corresponding shadow memory region in the virtual address space (*overmap*), ensuring that any state updates to shadow memory are reflected in the PM file ④. A shadow root object holds a reference to the PSM, which is set in a crash-consistent so that it preserves its validity across reboots/restarts ⑤. When objects are allocated, SafePM places red zones around them in PM and marks the corresponding shadow memory regions—payload as valid and red zones as invalid ⑥. Note that this process is performed atomically. When accessing memory, SafePM checks the shadow memory to validate the state of the address: if valid, the access proceeds; if not, a memory safety violation is reported.

Further, listing 3.1 illustrates an application using SafePM, which opens a PM pool and allocates and accesses PM objects. The highlighted lines of code indicate the additional operations and checks inserted by SafePM (in blue) and ASan (in red). On line 3, the application opens an existing PM pool. SafePM transparently initializes the pool's PSM, if needed (line 4), and overmaps it on the relevant section of the ASan's shadow memory (line 5). Then, the application allocates *N* objects (line 11). Note that

```
 1  struct my_obj { int src; int dest; } // object structure
 2  ...
 3  PMEMobjpool *pop = pmemobj_open(path); // open the PM Pool
 4  init_shadow_memory(pop);
 5  overmap_pool(pop);
 6  ...
 7  PMEMoid   obj_oid[N]; // declare the object handles
 8  size_t size = sizeof(struct my_obj);
 9  for (int i=0; i<N; i++) {
10      TX_BEGIN(pop){
11          pmemobj_alloc(pop,&obj_oid[i],size+2*RZ,...);
12          snapshot_and_set_shadow_memory();
13      } TX_END
14  }
15  ...
16  int val;
17  for (int i=0; i<M; i++) {
18      sh_src = get_shadow(&D_RO(object_oid[i])->src);
19      if (*sh_src != 0 && ...)
20          error(sh_src);
21      val = D_RO(object_oid[i])->src; //load from PM
22      sh_dest = get_shadow(&D_RW(object_oid[i])->dest);
23      if (*sh_dest != 0 && ...)
24          error(sh_dest);
25      D_RW(object_oid[i])->dest = val; //store to PM
26  }
27  ...
28  pmemobj_close(pop); //close the PM pool
29  unmap_shadow_mem();
```

Listing 3.1: SAFEPM code transformation: lines in blue are injected by SAFEPM's wrappers and lines in red by ASan.

SAFEPM transparently converts each allocation to a transaction in order to ensure the crash consistency of its memory safety metadata (lines 10-13). The application then accesses the PM objects (lines 21 and 25), and ASan introduces the appropriate shadow memory checks (lines 18-20, 22-24). These checks get redirected to the overmapped PSM and leverage the memory safety metadata to ensure that the requested PM addresses are addressable. In case any of these tests fail, an error is reported (e.g., if $M > N$). Finally, the PM pool is closed, and SAFEPM unmaps the PSM (line 29).

3.1.3.1 Persistent Memory Safety Metadata

SAFEPM constructs PM data structures to store the required PM safety metadata. SAFEPM does not consume additional memory compared to ASan and reserves the same portion of an application's virtual address space as ASan does. However, it differentiates itself as the memory safety metadata is placed on PM, which has to be set and updated in a fail-safe manner so that it remains valid across application restarts. More specifically, SafePM introduces the following persistent metadata data structures: *(i)* persistent shadow memory, *(ii)* persistent red zones, and *(iii)* the shadow root object.

The persistent shadow memory. The central data structure of SafePM's design is the persistent shadow memory (PSM), which stores information about which PM pool regions are addressable. To be compatible with ASan's compiler pass, we use the same format for the PSM as the one used by ASan, which requires allocating one byte of PSM for every 8 bytes of a PM pool. Following the format of ASan for the (volatile) shadow memory, the PSM is an array of bytes, where each byte stores the number of accessible bytes for its 8 corresponding bytes, or 0 to mark them all accessible. For non-accessible 8-byte blocks, it can store why they are non-accessible, for example, that they were freed or are part of a red zone. ASan's runtime library reserves $1/8th$ of the virtual address space for shadow memory. SafePM maps the PSM over ASan's shadow memory at the corresponding location that ASan uses for the mapped pool's virtual address range. Importantly, this *overmap* operation allows SafePM to leverage ASan's shadow memory checks without modifying its runtime library. By reserving a fixed region in the lower part of the virtual address space, the corresponding shadow memory address can be easily found with a simple address translation formula, where the offset is platform and OS-dependent:

```
#define GET_SM(addr) (void *)((long long)addr >> 3 + offset)
```

SafePM needs to persist the PSM data and ensure its crash consistency. To this end, SafePM creates PSM as a persistent object during the creation of a new persistent pool. The size of the PSM is at least $1/8th$ of the pool size requested by the application. Further, SafePM initializes the PSM as inaccessible. This ensures that the application code cannot manipulate any PM regions that are not explicitly allocated by the application, including unallocated PM heap parts, PM pool's metadata, and the PSM.

The persistent red zones. Similar to ASan, SafePM places red zones around PM objects, which are 16 B in size by default. A red zone is a region of memory marked inaccessible (poisoned) in the shadow memory, which prevents user code from accessing it. This enables the checks inserted by ASan's compiler pass to detect out-of-bounds accesses. Persistent red zones are allocated on object (re)allocation. Upon object deallocation, the red zones are removed along with the object and are marked inaccessible in the PSM, providing temporal violation detection capabilities.

The red zone size constitutes a trade-off between safety (i.e., buffer overflow detection capabilities) and space efficiency (i.e., memory consumption overhead). Large red zones waste space, while small red zones might fall short in detecting non-contiguous memory violations. For instance, in case two objects are separated by a 16 B red zone, SafePM will not detect under-/overflows of more than 16 B as the problematic memory access might fall within another object's boundaries.

Shadow root object. PM pools contain a root object that is used as the reference point by the application to reach the other pool's objects. SafePM creates a shadow root object during the pool creation. It contains persistent pointers to the PSM and the user root object, as well as the size of the user root object. From `libpmemobj`'s perspective, the shadow root object is the root object of the PM pool. SafePM's wrappers hide the additional fields of the shadow root object by returning the expected `app_root` field to the application when requested by the programmer.

```
struct shadow_root {
  PMEMoid psm; //PMEMoid of the PSM
  PMEMoid app_root; //PMEMoid of the app's user root object
  uint64_t app_root_size; // size of the user root object
};
```

3.1.3.2 System Operations

In this section, we describe the operations of SafePM that manipulate the PSM and the red zones to transparently ensure both memory safety on PM and crash consistency.

PM pool creation. When a program calls the function `pmemobj_create`, SafePM's wrapper creates the PM pool, allocates and initializes the shadow root object as well as the PSM in a crash-consistent way. If the operation is torn after the pool is created but before the initialization of the shadow root object and the PSM completes, the `pmemobj_open` wrapper will recover the persistent pool using the transaction capabilities of `libpmemobj` and recreate the PSM. During the creation of the pool, the PSM is initialized so that no region of PM is user-accessible, guaranteeing that an application cannot modify the pool's metadata or access non-allocated PM regions.

```
PMEMobjpool pmemobj_create(path, size) {
    //create a pool with extra 1/8th of size for the PSM
    PMEMobjpool* pool = pmemobj_create_orig(path, size+size/8);
    //transactionally create the PSM , set to inaccessible
    PMEMoid sm_root = init_psm(pool);
    //mmap the PSM  to its designated region
    overmap_psm(sm_root);
    return pool;
}
```

Note that the PSM of a memory pool is stored alongside the data of the pool. The PSM can be located at an arbitrary position within the mapped PM pool. Thus, ASan's compiler pass won't be able to correctly map virtual addresses within the PM pool to the PSM. This would require changes to ASan's compiler pass, hampering the transparency property. As a workaround, after a PM pool is mapped to the virtual address space during `pmemobj_create`, SAFEPM overmaps the PSM region of the pool to the position determined by the virtual-address-to-shadow-address formula used by ASan. Figure 3.2 shows a schematic that visualizes this operation.

PM pool opening. `pmemobj_open` is called to open an existing PM pool . SAFEPM's wrappers check the pool's root object to determine if the shadow root object and the PSM are set up correctly. If this is the case, the creation of the PM pool was completed, and the PSM gets overmapped to the respective location in the virtual address space according to the memory location returned by PMDK's original `pmemobj_open`. Otherwise, the creation of the pool must have been torn, and the transaction for creating and initializing the shadow root object and the PSM is executed again before the overmap operation.

```
PMEMobjpool pmemobj_open(path) {
    PMEMobjpool* pool = pmemobj_open_orig(path);
    // ensure the shadow root and PSM  are set up correctly
    recover(pool);
    overmap_psm(pool);
    return pool;
}
```

Memory management operations. PMDK supports different memory management operations to (re/de)allocate memory regions within a PM pool. We classify these operations into two distinct types: transactional operations (e.g., `pmemobj_tx_alloc`) that operate within a programmer-defined transaction and non-transactional operations (e.g., `pmemobj_alloc`) that do not require a transaction, but leverage atomic operations to ensure crash consistency.

Transactional PM management operations. The transactional memory management operations are executed within a transaction and use redo/undo logs to ensure crash consistency. SAFEPM builds on this and uses the memory snapshotting capabilities of `libpmemobj` to guarantee that the changes made to the PSM during a PM operation are atomic with respect to the changes made to the PM heap state, even in case of a torn operation or the abortion of a transaction. SAFEPM snapshots the respective PSM region, thereby adding it to the undo log of the current transaction, before performing any in-place updates to the PSM to demarcate the user-accessible regions.

The *transactional allocation* takes into account the size of the red zones and adds it to the object size requested by the user. Then, the relevant region of the PSM is snapshotted, and the user-accessible region is marked as such, while the adjacent red zones are marked inaccessible. Note that the PMEMoid returned by PMDK's allocator indicates the offset at the left red zone. Therefore a simple translation is performed to return to the application the offset of the object's actual payload.

```
PMEMoid pmemobj_tx_alloc(size) {
    //allocate an object with extra space for the red zones
    PMEMoid *oid = pmemobj_tx_alloc_orig(size+2*RZ_SIZE);
    //get the corresponding address within PSM
    void *oid_psm = get_psm_address(oid);
    //add the existing PSM  region to the undo log
    snapshot(oid_psm, sm_size);
    //update the PSM  region with the correct values
    mark_addressable(oid_psm, size);
    //set the pointer returned to the start of the payload
    oid.off += RZ_SIZE;
    return oid;
}
```

The *transactional reallocation* might cause an existing object to be moved. In this case, SAFEPM marks the old location of the object as inaccessible. The corresponding PSM region to its new location is modified to reflect the new user-specified size of the object. Still, all changes to the PSM are crash consistent, thanks to the transactional support of `libpmemobj`.

```
PMEMoid pmemobj_tx_realloc(oid, new_size) {
    oid.off -= RZ_SIZE;
    //reallocate the object to the new size
    PMEMoid *new_oid = pmemobj_tx_realloc_orig(oid, new_size+2*RZ_SIZE);
    void *oid_psm = get_psm_address(oid);
    if (oid != new_oid) { //object has been moved
        //add the old corresponding PSM  region to undo log
        snapshot(oid_psm, sm_size);
        //mark the old PSM  region as freed
        mark_non_addressable(oid_psm, size);
    }
    void *new_oid_psm = get_psm_address(new_oid);
    snapshot(new_oid_psm, new_psm_size);
    mark_addressable(new_oid_psm, new_size);
    new_oid.off += RZ_SIZE;
    return new_oid;
}
```

The *transactional deallocation* uses the relevant original PMDK routine to free the specified object (`pmemobj_tx_free`). SafePM's respective wrapper marks the memory region inaccessible. Note that `libpmemobj` includes built-in protection against double-frees, but it is based on the state of the persistent heap, the modification of which is delayed until the transaction is committed. Thus, double-frees that happen within a single transaction escape detection. To detect such cases, the wrapper explicitly verifies that the region the application is attempting to free is accessible. Unlike ASan, SafePM has no explicit quarantine for freed memory regions, but based on our experience, `libpmemobj` delays reallocating a deallocated region of PM.

```
void pmemobj_tx_free(oid) {
    oid.off -= RZ_SIZE;
    //verify object's validity
    void *oid_psm = get_psm_address(oid);
    if (!is_addressable(oid_psm))
        error();
    //free the requested object
    pmemobj_tx_free_orig(oid);
    snapshot(oid_psm, sm_size);
    mark_non_addressable(oid_psm, size);
}
```

Non-transactional PM management operations. SafePM transparently replaces the non-transactional memory management operations with their transactional counterparts. This is functionally correct but forgoes the performance advantage of non-transactional operations. Unfortunately, it is inevitable because each memory management operation causes modifications to the shadow memory, which cannot be performed with a single atomic operation in conjunction with the actual PM heap metadata modification.

3.1.3.3 Additional Design Details

Crash consistency. SafePM ensures crash consistency for the memory safety metadata stored on PM: if an application crashes, both the application data and the SafePM metadata will be able to recover to a consistent state. To achieve this, we leverage the transactional interface of PMDK, which is specifically built for crash consistency [237]. In PMDK, a transaction is defined as a series of operations on PM objects that either all or none occur. If a transaction is interrupted by a crash or power failure, PMDK uses undo and redo logs to restore the PM pool to its last consistent state: the undo log reverts any changes from incomplete transactions, while the redo log applies all committed changes. In SafePM, all PM management operations, whether transactional or atomic, are executed using their transactional counterpart. A pool's PSM object is allocated, and its shadow root object is initialized in a single transaction during the creation of that pool, and the modification of the PSM happens within the transaction that modifies that state of an object. The shadow memory is modified after its state is snapshotted using the undo log, hence guaranteeing that the modifications are crash-consistent.

System recovery. If an application crashes abruptly during the execution of a transaction, SAFEPM's metadata may be left in an incorrect state. However, whenever a pool is opened, PMDK checks if there exists any valid redo or undo logs. A transaction that is interrupted before atomically validating its redo log, will apply its valid undo log to revert the PM pool's data to a consistent state. Otherwise, if a transaction is interrupted after persisting its redo log, its redo log entries will be applied. Neither case affects the correctness of the state as PSM modifications are performed in place, and its initial content is tracked in the transaction's undo log. One unique case is when an application fails after the pool was created but before the transaction that allocates and initializes the PSM, and the shadow root object persisted its redo log. To handle such cases, after a pool is opened successfully, SAFEPM checks if the pointer to the PSM object within the shadow root object is `null`. In this case, the PSM will be reinitialized, and the shadow root object will be set accordingly.

Temporal safety. Similar to ASan, SAFEPM provides probabilistic temporal safety capabilities. When a PM object is freed or moved, the corresponding shadow memory is marked as free. Any subsequent access to this region will be detected by the shadow memory checks inserted by ASan. Further, the PMDK PM allocator does not reuse freed memory regions immediately but postpones their re-allocation. This enables SAFEPM to detect violations such as use-after-free or double frees that occur before the PM region is allocated again.

Multi-threading support. PMDK transactions do not provide any level of thread safety for the PM objects and it is the programmer's responsibility to ensure the application is free of race conditions. Because different persistent objects have disjoint corresponding regions in the PSM, unless the application is racy, the modifications on the PSM will be thread-safe. Further, PMDK reserves a space within the pool that is divided into *lanes*. Lanes are thread-specific and are used to store the logs of each thread's transaction. Consequently, SAFEPM's transactional operations are also thread-safe.

Metadata protection. SAFEPM initializes the PSM and marks all the PM pool as inaccessible, including the heap metadata of PMDK. As the heap metadata region is never allocated via the `libpmemobj` API, its corresponding shadow memory is never set to accessible. Accordingly, any access to a metadata region by the application's code will be detected by SafePM, thus providing metadata protection without the need for any changes to PMDK, unlike state-of-the-art approaches [57].

Partial safety coverage. ASan is mostly used in offline testing phases due to its prohibitive instrumentation costs for every memory access. Therefore, ASan provides the option to disable the instrumentation for specific global variables and functions with the `(no_sanitize("address"))` attribute. This option deducts all ASan checks from the annotated function. It is designed for cases where the programmer trusts specific functions and wants to avoid the performance overhead.

SafePM also supports this functionality. It allows users to denote functions that will not be instrumented. For such code parts, SafePM exposes a series of 'unsafe'-prefixed wrappers, which internally call the PMDK's PM management functions without performing any PSM and red zones management. However, SafePM imposes one limitation: objects allocated with unsafe wrapper functions should only be accessed in uninstrumented functions. Accessing them in instrumented code causes SafePM to report an error, as their corresponding bytes in PSM remain marked as inaccessible.

Limitations. SafePM follows the same design for the shadow memory as ASan and relies on the ASan's compiler pass for detecting memory safety violations. Hence, it inherits the same limitations. SafePM is incapable of detecting intra-object overflows as the red zones are inserted at the object level to avoid changing the objects' memory layouts. SafePM also misses out-of-bounds access that falls within the boundaries of another object. Potential solutions to these limitations could involve using a finer-grained boundary detection (e.g., byte-level) for intra-object overflows and disallowing dereference of pointers that have been constructed via pure offset manipulation. However, these solutions would be invasive in ASan's design, resulting in high performance and memory overheads (e.g., if red zones are needed for each memory byte) or modifying the programming model, thus violating the important transparency property.

Furthermore, in SafePM, the persistent shadow memory is allocated and handled as an individual PM pool object and cannot exceed the size of 16 GB [56]. This is a limitation imposed by PMDK. Thus, since the persistent shadow memory occupies one eighth of the persistent pool, SafePM can currently support persistent pools up to 128 GB in size. To alleviate this issue, PMDK could be modified to support objects larger than 16 GB, and SafePM's PSM layout could be adapted to allow larger PM pool sizes by segmenting the shadow memory across multiple pool objects. The latter would require careful internal handling to ensure that the crash consistency of the PSM metadata is preserved across pools.

Usage in production. In SafePM, there are two sources that contribute to the higher latency of PM operations and the overall performance overhead: *(i)* ASan 's instrumentation, and the resulting PSM accesses, and *(ii)* the metadata bookkeeping of SafePM's wrappers.

Persistent pools created by PMDK are not compatible with SafePM, and vice versa. However, an application linked with SafePM can be used in production with ASan 's instrumentation disabled during compilation. The overheads of such an approach can be observed in our SafePM w/o ASan variant (§ 3.1.5). Note that, in this case, the memory safety violations are not detected. For this reason, we encourage SafePM to be used in development phases or make use of the partial safety coverage if production use is desired.

Further, if an application passes the development phase and no longer requires SafePM to be enabled, it can be linked against the vanilla PMDK for production use without any source code modifications. Unfortunately, the PM pool recreation is mandatory in this scenario. Thus, the overheads introduced both by ASan and SafePM's wrappers can be avoided entirely. This level of transparency is a key enabler that makes our approach practical.

Future extensions. SafePM builds on PMDK and ASan. However, the underlying design can be ported to other persistent memory libraries, provided they support transactional updates on PM pools. The port can be achieved by creating the respective wrappers around the memory management functions, as SafePM performs for libpmemobj [120].

The applicability of SafePM's approach to other memory safety checking tools depends on the methods that the tool uses to enforce memory safety. SafePM can be modified to support other shadow-memory based approaches [39, 250] by adopting their logic for the shadow memory handling and memory safety metadata updates. For SafePM, we choose ASan because it is widely used and integrated into several compiler toolchains [43, 76].

3.1.4 Implementation

SafePM consists of *(i)* a runtime library based on PMDK and *(ii)* the ASan's compiler pass for the instrumentation of the application code.

Runtime library. We implement the runtime library of SafePM as a fork of PMDK v1.9. In SafePM, we develop wrappers around the exposed PM management functions, which require modifications to the PSM. These wrappers augment the PMDK functions with their respective shadow memory operations while ensuring crash consistency for both the PM pool (meta)data and the memory safety metadata, as explained in § 3.1.3.2. Table 3.1 enumerates the functions that were wrapped by SafePM.

Pool management	
`pmemobj_create`	creates a PM pool
`pmemobj_open`	opens an existing memory pool
`pmemobj_close`	closes a memory pool
Memory management	
`pmemobj_tx_alloc`	transactional allocation
`pmemobj_tx_realloc`	transactional reallocation
`pmemobj_tx_free`	transactional deallocation
`pmemobj_alloc`	atomic allocation
Other operations	
`pmemobj_alloc_usable_size`	returns allocated size
`pmemobj_type_num`	returns object's type number
`pmemobj_first`	returns the first pool object
`pmemobj_next`	iterates over the objects

Table 3.1: List of PMDK API modified to support SAFEPM

To ensure *transparency* and *compatibility* with existing PMDK-based applications, SAFEPM's wrappers are named after their PMDK equivalent function. Thus, the function calls from an unmodified PMDK application, which is linked against SAFEPM, get redirected to their respective wrapped version to include the memory safety metadata management.

The PSM is created as an object within a PM pool. To be able to perform the *overmap* operation, the size of the PSM must be a multiple of the page size (4KB for x86/AMD64). `mmap` also requires that the in-file offset of the portion to be memory-mapped is a multiple of the page size. The in-pool offset of the persistent shadow memory must satisfy this condition. Furthermore, the PSM must be mapped to a starting address that is page-aligned, which requires that the persistent pool is mapped to a starting address that is aligned to eight times the page size. Finally, the entire pool needs to be padded to a multiple of eight times the page size since each shadow byte corresponds to eight application bytes. SAFEPM's wrapper for `pmemobj_create` is responsible for enforcing all of these padding and alignment constraints.

ASan prevents the application code from modifying the shadow memory by using `mprotect` to set the shadow memory's protection level to `PROT_NONE`, making it entirely inaccessible. Thanks to `mprotect`, ASan avoids allocating additional physical memory to protect the shadow memory itself. Unlike ASan, SAFEPM relies on the PSM being initialized to inaccessible to ensure these guarantees. However, since the PSM is physically allocated during pool creation, SAFEPM's approach, similar to ASan, does not add extra memory overheads.

Compiler pass. SafePM leverages ASan's compiler pass without any modifications. When an application is compiled with the `-fsanitize=address` flag enabled, the compiler runs the ASan compiler pass, as shown in Listing 3.1. ASan's compiler pass runs after all other compiler optimizations so that only memory accesses that remain after the optimizations are instrumented. In SafePM we assume that PMDK is correct and has no memory safety violations. Therefore, we do not compile the PMDK internal functions with ASan. Note that this is necessary as these functions manipulate both the PMDK metadata and the PSM.

3.1.5 Evaluation

Our evaluation is structured around three dimensions.

Space & performance overheads. We evaluate the performance (§ 3.1.5.2) and space (§ 3.1.5.3) overheads of SafePM using PMDK's micro-benchmarks as well as pmemkv [117], a persistent KV store. We further evaluate the efficiency of the partial safety coverage approach (§ 3.1.5.6).

Effectiveness. We evaluate the effectiveness of SafePM (§ 3.1.5.4) using the RIPE framework [320] to test the exploitability of a wide range of memory vulnerabilities. We also report some memory safety bugs and programming anomalies discovered during our experiments (§ 3.1.5.7).

Crash consistency. Lastly, we validate the crash-consistency property (§ 3.1.5.5) for both the application data and SafePM's metadata using the pmemcheck tool [240] provided by the PMDK's Valgrind fork. Note that, in these experiments, ASan is *disabled* due to its incompatibility with Valgrind.

3.1.5.1 Experimental Setup

Testbed. We conduct our experiments on a server machine equipped with Intel(R) Xeon(R) Gold 6212U CPU with 24 cores, 192 GB (6 channels $\times$ 32 GB/DIMM) DRAM, and 768 GB (6 channels $\times$ 128 GB/DIMM) Intel Optane DC DIMMs running Ubuntu 20.04.02 with Linux kernel version 5.4.0.

Variants. We conduct the experiments with the variants described in Table 3.2. Native refers to the application being linked against the native PMDK without ASan instrumentation, while ASan indicates that the application was compiled with `gcc`'s ASan extension and linked against native PMDK. These two variants serve as our baselines as they represent unhardened applications and applications hardened only with ASan, respectively. The ASan version indicates the inevitable overheads introduced by ASan's instrumentation. SafePM w/o ASan uses the SafePM's wrappers without compiling

Variant	Compile w/ ASan	SAFEPM wrappers
Native	No	Disabled
ASan	Yes	Disabled
SAFEPM w/o ASan	No	Enabled
SAFEPM	Yes	Enabled

Table 3.2: Benchmarking variants

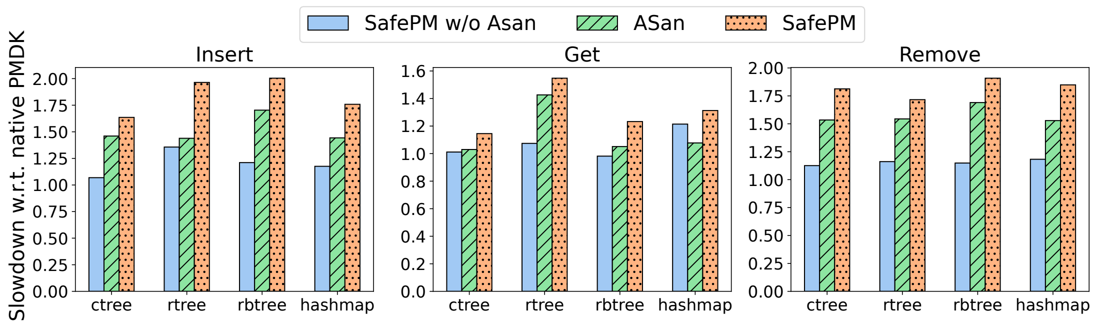


Figure 3.3: Performance overheads of persistent indices for PMDK w/ ASan, SAFEPM w/o ASan and SAFEPM versions.

the application with the ASan extension. The goal of this variant is to demonstrate the overheads incurred by our wrappers without ASan's compiler pass instrumentation. Based on this variant, we can determine the introduced overhead of SAFEPM's additional management operations to assure the crash consistency of the memory safety metadata. Finally, SAFEPM denotes our complete tool; applications are linked against our PMDK fork and are compiled with the ASan instrumentation enabled.

3.1.5.2 Performance Overheads

We evaluate the performance overheads of SAFEPM using four different persistent indices (`ctree`, `rbtree`, `rtree` and `hashmap`) and a persistent KV store (`pmemkv` [117]). We further measure the performance of SAFEPM for the atomic and transactional PM management operations (`alloc`, `realloc` and `free`), as well as creating and opening a PM pool. Lastly, we evaluate the effect of SAFEPM on the recovery process.

All experiments are conducted with a red zone size of 16 bytes. Each object is surrounded by two red zones. The reported values are the average of at least 3 runs.

Persistent indices. We evaluate the performance of SAFEPM for four persistent indices over the four variants shown in Table 3.2. We use `pmembench` [126], shipped with PMDK, and perform one million insert, get, and remove operations on each data

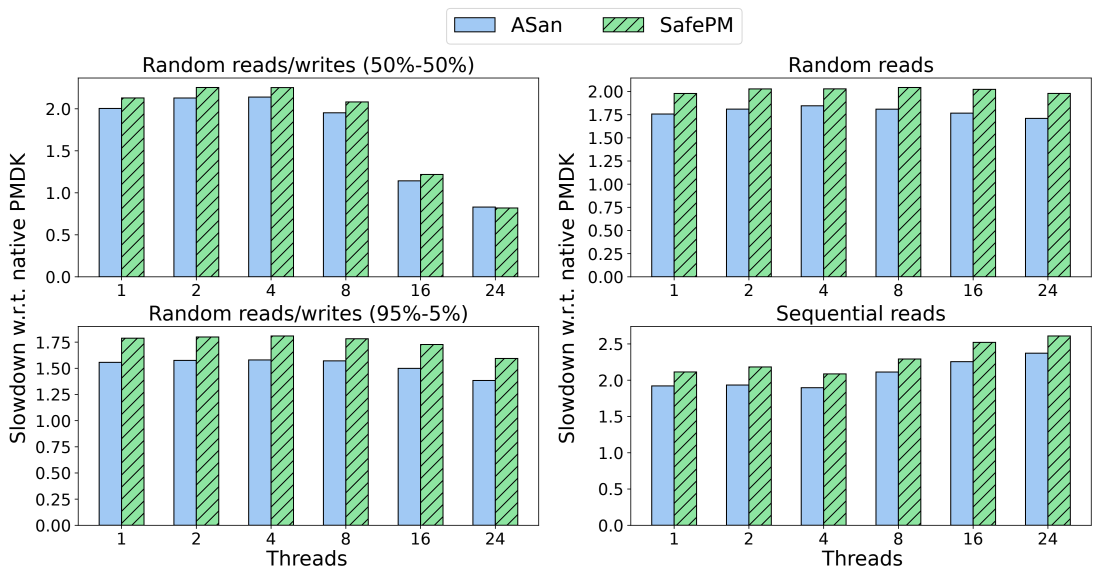


Figure 3.4: Performance overheads w.r.t. to native PMDK of `pmemkv` under different workloads and thread count.

structure. The keys are 8 bytes, and the operations choose keys at random following a uniform distribution.

Figure 3.3 illustrates the slowdown for ASan, SAFEPM w/o ASan, and SAFEPM versions normalized to the native PMDK execution. In general, SAFEPM is 1.68-2.00×, 1.16-1.50×, and 1.68-1.87× slower than the native PMDK for the insert, get and remove operations, respectively. Figure 3.3 further indicates that the usage of SAFEPM's wrappers w/o ASan does not significantly affect the performance of the indices except for the case of `rtree` insert where it incurs a 1.34× slowdown. In the other cases, the respective overhead w.r.t. PMDK remains below 20%. This demonstrates the ability of SAFEPM to achieve its performance goal by keeping its overheads very close to those of the highly optimized ASan. The only exception is the case of the `hashmap` get, where inserting red zones changes the objects' alignment, leading to additional cache line accesses.

Persistent KV store. We evaluate SAFEPM's performance using `pmemkv` [117], a persistent KV store designed for PM, with its default `cmap` backend storage engine. We use the `pmemkv-bench` [111] benchmark suite with different workloads: *(i)* update intensive (50% reads-50% writes), *(ii)* read intensive (95% reads-5%writes), *(iii)* random reads and *(iv)* sequential reads. The KV store is populated with 1M entries at the beginning of each run. Each workload consists of 10M operations where keys and values are set to 16B and 1024B, respectively.

Figure 3.4 reports the slowdown of the throughput of PMDK w/ ASan and SAFEPM w.r.t. to native PMDK while varying the number of threads. Enabling ASan with PMDK

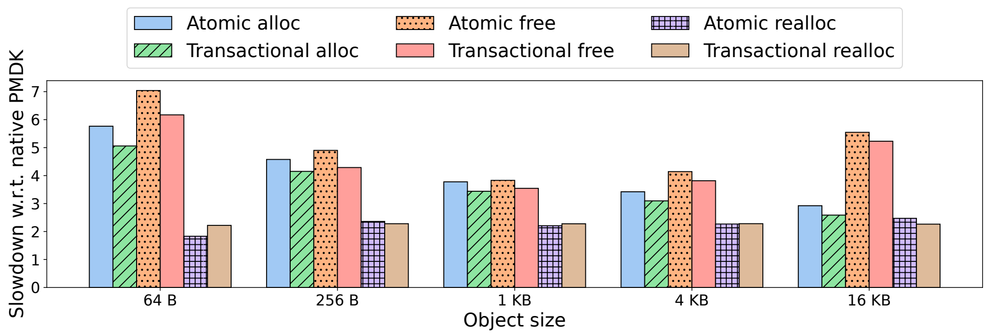


Figure 3.5: Performance overhead of SAFEPM's wrappers for selected memory operations across different object sizes.

slows down the queries' execution by 1.14-2.36× depending on the workload. The respective values for SAFEPM are 1.20-2.55×. The additional performance overhead for SAFEPM stems from the extra operations that SAFEPM needs to perform in order to ensure the crash consistency of memory safety metadata. The marginally higher overheads of SAFEPM compared to ASan further demonstrate the ability of SAFEPM to achieve its performance goal in real-life workloads. Furthermore, SAFEPM does not affect the scalability of the KV-store as its behavior is similar to PMDK and PMDK w/ ASan, even for an increasing number of threads. We can notice, though, a significant drop in the overheads beyond 8 threads in the update-intensive workloads. This is attributed to the native application suffering from increasing levels of contention while the instrumentation decreases this stress.

Atomic and transactional PM operations. We next measure the performance of the basic atomic and transactional PM management operations (`alloc`, `realloc` and `free`) for SAFEPM. We design a microbenchmark based on `pmembench` where we execute 100K operations per experiment with varying object sizes. The reported results are the average of 10 runs.

Figure 3.5 shows the throughput slowdown of SAFEPM for several PM operations normalized w.r.t. native PMDK. For object allocation, we observe that the overhead decreases for both atomic and transactional allocation as the object size grows (2.4-5.8×). The reallocation operation maintains a relatively constant overhead for all the tested data sizes (1.85-2.25×). SAFEPM incurs a higher performance overhead for the free operation (3.5-7.0×) compared to alloc and realloc for every object size. The aforementioned overheads are caused by *(i)* runtime checks introduced by ASan instrumentation and *(ii)* the added operations of SAFEPM to ensure crash consistency for PM safety metadata. Lastly, as was expected, SAFEPM poses a higher overhead for the atomic versions of the operations, as it transparently converts them into their transactional counterpart in order to atomically update the appropriate PSM region along with

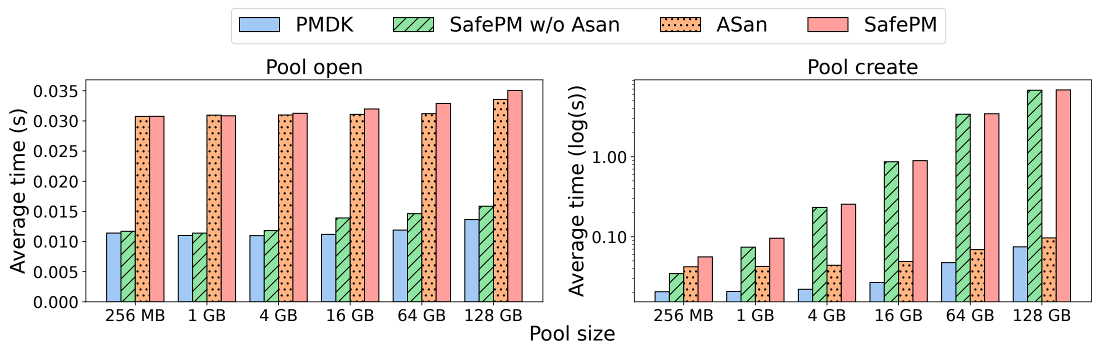


Figure 3.6: Performance overheads for creating and opening pools of various sizes.

heap state in a crash-consistent manner.

PM pool create/open. Figure 3.6 shows the average time of the PM pool *create* and *open* operations for the variants listed in Table 3.2. We created a microbenchmark using the `pmembench` framework, where we create/open PM pools of various sizes ranging from 256MB to 128GB. We observe that opening a pool with SAFEPM takes ∼30ms instead of 10ms with native PMDK, a slowdown of up to 3×. The slowdown appears to be largely caused by the introduced ASan checks because the performance of the `SafePM w/o Asan` variant is close to that of the native PMDK. Further, during pool creation, SAFEPM incurs a significant slowdown, which increases with the pool size, causing the *create* operation to take a few seconds to complete. This overhead stems from the need to overmap and initialize the PSM object, which grows with the size of the pool. It is worth noting, though, that pool creation is an one-time operation, hence, the high overhead is largely irrelevant to application performance.

Recovery time. Table 3.3 presents the average time of the recovery process with various log sizes for the variants listed in Table 3.2. The critical parameter that affects the recovery process is the size of the log that contains the entries whose application reverts the PM data to a consistent state. For this experiment, we design a microbenchmark where we create a PM pool, allocate persistent objects, each being 1 KB in size, and snapshot their content in the undo log of a transaction. The number of allocated objects in each experiment is equal to the desired length of the undo log in KBs. Thus, the size of the logs depends on the number of snapshot objects. Further, we inject a crash at this point and then reopen the pool. The time to reopen the pool includes the recovery process. We perform this experiment 100 times for each configuration.

We observe that in all variants, the recovery time gets higher as the log size increases. With ASan disabled, SAFEPM's wrappers introduce insignificant overhead (¡300 μs) in the recovery time compared to PMDK. When we enable ASan, the shadow memory checks incur an inevitable but minor overhead (approximately 10 *ms*). Over-

Variant	Log size				
	4 KB	128 KB	512 KB	2 MB	4 MB
PMDK	15.00	15.06	15.45	17.01	19.13
SAFEPM w/o ASan	14.99	15.31	15.74	17.31	19.38
ASan	25.23	25.40	25.75	27.26	29.45
SAFEPM	25.44	25.39	25.78	27.43	29.79

Table 3.3: Recovery time in milliseconds (*ms*).

Data structure	insert	remove	get
ctree	12.5%	12.5%	12.5%
rtree	14.25%	13.8%	13.8%
rbtree	12.5%	12.5%	12.5%
hashmap_tx	12.5%	12.5%	12.5%

Table 3.4: SAFEPM space overhead

all, SAFEPM does not introduce any further delays in the recovery process other than those of ASan.

3.1.5.3 Space Overhead

We measure the extra PM space that SAFEPM requires. The space overhead is comprised of *(i)* the persistent shadow memory and *(ii)* the object red zones. This section ignores the shadow root object, as it represents a small, fixed overhead independent of the pool size or allocated objects. Note that SAFEPM uses the same virtual address ranges reserved by ASan (§ 3.1.3.1). We conduct experiments on the same four persistent indices performing `insert`, `get` and `remove` operations, as discussed in § 3.1.5.2. We report the peak space overhead when applications are linked against SAFEPM.

Table 3.4 summarizes the PM space overheads of SAFEPM expressed in percentage of the total pool size. The persistent shadow memory always occupies one eighth of the pool, which corresponds to an overhead of 12.5%. For the `ctree`, `rbtree` and `hashmap_tx`, we observe that this is the only considerable space overhead as the persistent object red zones occupy space which is wasted to padding by the native PMDK allocator. For the `rtree` index, the object red zones increase persistent memory usage, leading to slightly higher space overheads. However, even in this case, the main contributor to the increase in PM space usage is the persistent shadow memory.

RIPE variant	Always	Sometimes	Never
Intact	306	14	1014
ASan w/ system heap	27	1	1306
ASan w/ PM pool heap	119	12	1203
SAFEPM	27	1	1306
memcheck	62	0	1272

Table 3.5: Number of RIPE attacks that always, sometimes, or never succeed with different protection mechanisms.

3.1.5.4 Effectiveness

We evaluate the effectiveness of SAFEPM using the RIPE framework [320], a comprehensive suite of memory vulnerability exploits. We modified the 64-bit port of the RIPE benchmark [255] to compare the effectiveness of the following variants: *(i)* Intact, where the victim application is unmodified, *(ii)* ASan w/ system heap, where the application is compiled with ASan, *(iii)* ASan, where the application uses the persistent heap and is compiled with ASan, which protects only the volatile heap and not the PM heap, *(iv)* SAFEPM, which extends ASan's memory safety to the PM heap, and *(v)* memcheck [238], the current state-of-the-art for detecting memory violations in persistent memory. All variants use gcc 9.3.0 and are compiled with the default GCC stack protections enabled.

The RIPE benchmark performs each exploit several times, *3* by default. If an exploit succeeds in all attempts, in some trials, or in none of the runs, it is marked as *always*, *sometimes*, or *never*, respectively. Note that, we run the RIPE benchmark suite several times to make sure the results are stable.

Table 3.5 reports the number of exploits that either *always*, *sometimes*, or *never* succeed. We observe that when the victim application uses the volatile (system) heap, ASan is able to prevent most attacks. When the victim application is modified to use the PM heap, the number of exploitable vulnerabilities increases, as the layout of the persistent heap is not available to ASan. However, linking the application against SAFEPM restores ASan's protection capabilities even for memory violations occurring in the persistent heap, reducing the number of exploitable vulnerabilities back to the levels observed with the system heap. In other words, SAFEPM achieves memory safety effectiveness for the PM heap equivalent to that achieved by ASan for the volatile (system) heap. Finally, we observe that SAFEPM is able to detect and prevent a higher number of memory vulnerabilities compared to the state-of-the-art memcheck [238].

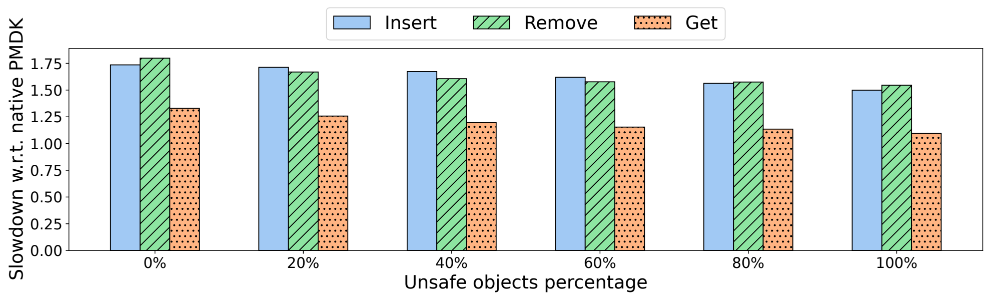


Figure 3.7: Performance overheads of persistent hashmap index with varying percentage of unsafe PM objects.

3.1.5.5 Crash Consistency

We validate the crash-consistency property for both the application data and SafePM metadata using existing tools, `pmemcheck` [240] and `memcheck` [220]. To perform our experiments, we disable ASan as it is incompatible with Valgrind. As SafePM wraps PMDK routines instead of modifying them or the pool layout, these tools work without modifications. We run the persistent indices and PM operations benchmarks described in § 3.1.5.2 with `pmemcheck` and `memcheck` enabled. The number of operations for each index is limited to 10000 to keep the runtime reasonable despite the slowdown caused by Valgrind. We observe that for the tested indices, neither `pmemcheck` nor `memcheck` report any error. For the PM operations benchmark, `pmemcheck` again reports no error, while `memcheck` does not report any error beyond the ones also reported for the case of unmodified PMDK.

3.1.5.6 Partial Safety Coverage

We evaluate the efficiency of our proposed partial safety coverage approach. In this experiment, we use the persistent `hashmap` in a similar fashion to 3.1.5.2. We vary the proportion of operations that are performed using memory-safe (instrumented) and memory-safe (uninstrumented) objects. Figure 3.7 illustrates the performance slowdown for each operation as the proportion of used unsafe objects increases, normalized w.r.t. the native PMDK execution. We observe that for all three operations, the relative overhead decreases as more objects are excluded from the ASan instrumentation and runtime checks. However, there is still an inevitable overhead that stems from ASan intercepting the volatile heap management functions, which are used by PMDK internally. Note that with the get operation, there is no overhead as there are no intercepted `malloc/free` calls.

3.1.5.7 Discovered Bugs and Anomalies

Our experiments led to discovering and reporting the following bugs [235, 236]: *(i)* in the btree example of PMDK version 1.9.2, a call to *memmove* on line 378 of *btree_map.c* causes an off-by-one overflow on PM residing data objects and *(ii)* in the transactional operations benchmark, shipped as part of `pmembench`, a configuration file lacks the configuration setting `nestings`. This causes the transaction to not be aborted, which triggers invalid frees at line 295 in *pmemobj_tx.cpp*, that is detected by SAFEPM, when the benchmark attempts the cleanup.

3.1.6 Related work

Persistent-memory based systems. Several well-known, high-performant data management systems, such as RocksDB [256] and Redis [253], have already been adapted to incorporate persistent memory in their system stack [117, 199, 253, 256]. Apart from that, there exist proposed filesystems specially designed to benefit from PM as a storage medium [46, 157, 312, 325]. Additionally, accessing PM remotely is an active area of research in order to enable PM usage in distributed settings [116, 133, 152, 189, 273, 329]. While these systems mainly target to ensure crash consistency and high performance with the use of innovative PM technology, SAFEPM focuses on the important aspect of memory safety in PM programming.

SW-based memory safety. Protecting against DRAM memory safety bugs with software-based approaches has been the target of several works [3, 170, 172, 264]. They leverage different techniques, such as compiler pass instrumentation accompanied by runtime libraries and compact representation of upper and lower pointer bounds, needed to perform the appropriate runtime checks. They aim to minimize performance and memory overheads while maintaining compatibility and efficiency. Another alternative for ensuring memory safety is to use memory-safe languages. Corundum [99] is a generic library for PM management written in Rust, which statically enforces language-based memory safety for PM. SAFEPM, contrary to Corundum, does not require programmers to use specific libraries or languages but targets applications developed using PMDK, the de facto library for PM while requiring no source code modifications. The `memcheck` tool [238] uses Valgrind and instrumentation built into PMDK to achieve memory safety similar to SAFEPM. Unlike ASan, it does not require compiler support, as it uses runtime translation. Further, unlike SAFEPM, it has no PM overhead. The trade-offs are that it incurs a much larger performance overhead, and its spatial violation detection capabilities are not as precise as SAFEPM's.

HW-based memory safety. There exists a large body of work that enforces memory safety for volatile memory using hardware extensions [14, 69, 223, 224, 322]. Lowfat pointers [69] enforce spatial safety by associating the pointer with its bounds. It contains gate-level implementations of the logic for updating and validating the compact fat pointers. Cheri [322] ensures memory safety bug detection with the support of hardware capabilities. Intel MPX [223, 224] provides ISA extensions of Intel x86-64 architecture for memory protection. Arm MTE [14] is an ARM extension that enables hardware-assisted memory tagging to detect both temporal and spatial memory safety bugs. Besides being designed for volatile memory only, these solutions require specialized hardware, whereas SAFEPM can be deployed in commodity servers to ensure memory safety for PM.

PM debuggers, allocators and libraries. Several frameworks have been developed to test the correctness of PM software [62, 75, 83, 185–187, 217, 240], an orthogonal problem to memory safety. Many projects strive to manage persistent memory efficiently while also ensuring crash consistency [45, 57, 120, 313]. Mnemosyne [313] exposes a simple interface for PM programming with respect to crash consistency and persistence. NVHeaps [45] is a lightweight, high-performance persistent object system with transaction support and persistency semantics. Poseidon [57] is another allocator designed for PM that relies on Intel MPK [228] to avoid the corruption of the persistent metadata by memory bugs.

3.1.7 Summary

In this section, we present SAFEPM, a framework that ensures the memory safety of PMDK-based PM applications by detecting spatial and temporal memory safety violations. To this end, SAFEPM leverages the compiler pass and reporting mechanism of the popular ASan. During the creation of a memory pool, SAFEPM creates a persistent shadow memory object that is mapped when the pool is opened to the corresponding location in ASan's shadow memory. SAFEPM manipulates the persistent shadow memory along with the persistent heap in a transparent and crash-consistent manner. Consequently, any PMDK-based application can make use of SAFEPM without source code modifications to test for memory violations at runtime, including the recovery process. Our extensive evaluation shows that SAFEPM provides the same level of memory safety guarantees for PM applications as ASan provides for volatile memory at a cost of marginal overheads.

Software artifact. SAFEPM is publicly available along with its experimental setup.

3.2 SPP: Safe Persistent Pointers for Memory Safety

In this section, we present Safe Persistent Pointers (SPP), a practical memory safety approach against buffer overflows for PM applications. Our goal with SPP is to provide a memory safety solution that overcomes the performance limitations that SafePM (§ 3.1) introduces, minimizes the PM space overheads and aims to be applicable in production deployments.

In a nutshell, SPP augments persistent pointers with memory safety properties. SPP is based on a simple combination of tagged pointers, efficient persistent memory layout, and transactional updates to the memory safety metadata for ensuring crash consistency. SPP's efficient pointer representation does not require additional memory lookup operations at run-time while incurring minimal space overheads for storing the memory safety metadata.

We implement SPP based on the LLVM compiler infrastructure accompanied by a runtime library and an adapted version of the PM development kit (PMDK). Our evaluation demonstrates that SPP incurs low runtime and space overheads while preserving the crash-consistency property and maintaining the PMDK API intact, i.e., requiring no source code modifications.

3.2.1 Motivation

Low-level unsafe languages, such as C/C++, provide developers with control over the system's memory. While this is crucial performance-wise [232], it can lead to potentially harmful memory safety bugs [28, 70, 289, 308, 332]. These bugs are broadly separated into two categories: *spatial*, e.g., buffer overflows, and *temporal*, e.g., dangling pointers.

Memory safety bugs cause many critical security and reliability issues [94, 96, 205, 284]. The severity of memory safety violations is also confirmed by the reports of major software projects, such as Windows [206], Android [7] and Chromium [42], where $70-75\%$ of the detected issues stem from memory safety bugs. According to Szekeres et al. [289], the majority of security attacks in software systems occur through exploiting memory safety vulnerabilities.

Designing efficient approaches to enforce memory safety is an active area of research for *the volatile main memory*, including software and hardware-based memory safety solutions (§ 3.2.6). At a high level, these approaches implement *deterministic dynamic bounds checking* [211, 289], which utilizes runtime metadata (bounds information) [170], rather than relying on probabilistic heuristics [28, 190]. These approaches instrument the code during compilation and inject run-time metadata management into an application that allows deterministic run-time checks for validating memory accesses.

Unfortunately, existing memory safety approaches are *restricted to the volatile main memory devices and are inadequate for byte-addressable persistent memory (PM) devices*. In particular, the emergence of the Compute Express Link (CXL) technology [47] is enabling byte-addressable PM storage devices [90, 271]. These PM devices are either attached to the memory bus [219] or the PCIe bus [47] and, with the use of NICs, can *also* be accessed over the network (e.g., via RDMA) [133]. PM applications memory map (`mmap()`) these devices directly to their address space. Using pointers, their mapped content is accessed at a byte granularity via the `ld/st` interface.

However, the PM pointer representation is persistent, i.e., its offset and the associated PM object are durable. Therefore, addressing memory safety issues for PM is challenging, especially due to the idiosyncrasies of the PM programming model [239]. More specifically, PM applications rely on persistent pointers [243] and use specialized, crash-consistent memory allocators [57, 120]. This entails two challenges: (a) how do we preserve crash consistency for memory safety metadata?, and (b) how do we ensure memory safety on the recovery paths after a system crash or reboot? Unfortunately, current memory safety mechanisms for PM, with the most prominent being SafePM [30], our previously proposed shadow memory-based memory safety solution (§ 3.1), are deemed as *impractical* since they either require adopting a new programming model/language [99] or are restricted to the offline testing phase [30, 240] due to prohibitive performance costs.

To this end, we propose Safe Persistent Pointers (SPP), a practical memory safety approach for applications accessing byte-addressable PM storage devices via PM pointers. SPP provides *PM buffer overflow protection*. Its design is based on DeltaPointers [170], a memory safety approach for volatile memory. SPP essentially extends DeltaPointers to PM. SPP is built on the prevalent PM programming model and employs *tagged pointers*, as well as an *efficient PM layout*, in combination with transactional updates to the memory safety metadata.

Our SPP prototype consists of an adapted PMDK [239] version, the state-of-the-art PM programming toolchain, and an instrumentation using LLVM [175]. The evaluation of SPP is structured around three dimensions: performance and space overhead, effectiveness, and crash consistency. We measure the performance and space overheads of SPP using PMDK micro-benchmarks and a persistent KV store [117]. We evaluate the effectiveness of SPP with the RIPE framework [320] that contains a set of memory safety exploits. Lastly, we validate the crash consistency of SPP's metadata using the `pmemcheck` [240] tool. SPP incurs low performance overheads and requires no source

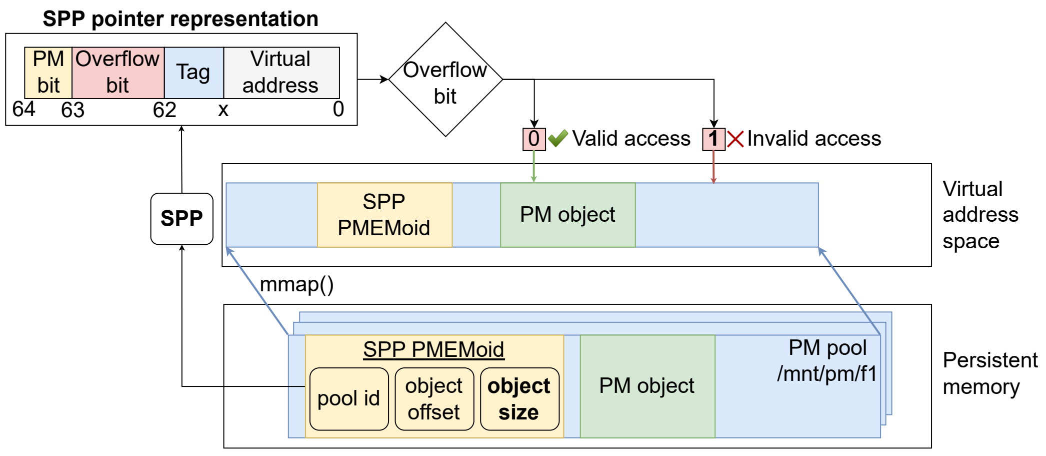


Figure 3.8: *The* SPP *PM pointer representation (*SPP *PMEMoid) is used to derive the tagged pointer to the PM object (*SPP *pointer representation). On a PM access, the PM bit and the pointer tag get masked while the overflow bit is preserved. If the access is valid (in green), the overflow bit is 0, and the access succeeds. In the case of a PM buffer overflow (in red), the overflow bit is 1, which makes the address invalid.*

code modifications while preserving the crash consistency property.
Altogether, SPP makes the following contributions:

- SPP introduces *safe persistent pointers* (§ 3.2.3), a spatial memory safety solution against PM buffer overflows for PMDK applications. They are the first tagged pointer scheme specifically designed for PM. They consist of an enhanced, durable representation of PMDK's persistent pointers and a native-pointer tagging scheme.

- SPP offers a configurable pointer encoding scheme, inspired by DeltaPointers [170], to fit the PM management requirements of every PM application. The number of bits in the pointer tag is adjustable and and can be easily tuned without breaking the compatibility with pre-compiled, uninstrumented libraries.

- We implement the SPP prototype and design our compiler optimizations based on LLVM (§ 3.2.4). Our evaluation (§ 3.2.5) indicates that SPP is a practical approach that prevents PM buffer overflow exploits while incurring low performance and negligible space overheads.

3.2.2 Overview

SPP is a system that provides spatial memory safety for PMDK-based applications. It requires no source code modifications and introduces minimal performance and space overheads. Thus, SPP is a practical PM safety solution, in contrast to the state-of-the-

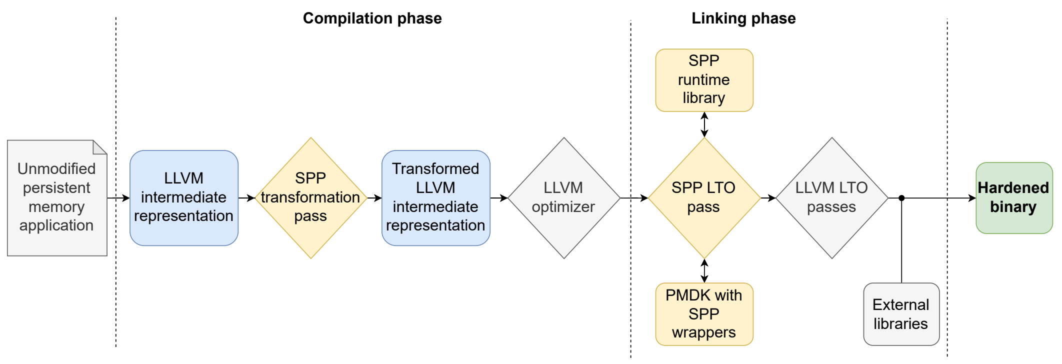


Figure 3.9: SPP overview (yellow colored boxes denote the SPP components): *The unmodified PM application is converted to its LLVM IR, where the* SPP *transformation pass transforms the runtime function calls for the pointer tag management. At the link phase,* SPP *applies its optimizations via its LTO pass, and the transformed application is linked against the* SPP *runtime library, adapted PMDK version and external libraries to produce the final binary.*

art approaches that incur significant space and runtime overheads [30, 198] or demand the usage of specific memory safe languages [99].

Figure 3.8 captures the SPP PM pointer representation (SPP PMEMoid), the tagged pointer structure and how SPP handles PM accesses. Precisely, SPP enhances the SPP PMEMoid with a field containing the object size. This persistent data structure is used to generate the tagged pointer [170] to a PM object. Importantly, SPP sets and updates the SPP PMEMoid in a crash-consistent manner either by wrapping its content inside transactions or through atomic operations. In the case of a PM access, SPP preserves only the virtual address and the overflow bit of the tagged pointer. If the access lies within the bounds of the PM object, the overflow bit is clear, and the access proceeds normally using the virtual address. However, if the pointer is beyond the object's boundaries, the overflow bit is set through the SPP pointer tag operations, rendering the address invalid. Thus, the upcoming access triggers a fault.

An overview of SPP's workflow is shown in Figure 3.9. An unmodified PM application is initially instrumented with SPP's transformation pass that inserts the PM pointer tag operations and the runtime checks. The instrumented code is then linked against SPP's runtime library and the modified PMDK that includes the enhanced PM pointer representation and the adapted PM management functions. Note that the programming model and the APIs of PMDK remain intact. During the linking process, SPP's link time optimization (LTO) pass scans the application for external function calls and masks away the tag from the PM pointers passed as arguments to preserve compatibility. Thus, SPP can be seamlessly integrated in existing workflows and deployments, providing complete control of the memory safety-critical code parts.

3.2.2.1 System Model

Fault model. SPP protects against spatial memory safety bugs on PM. It detects PM buffer overflows in PMDK-based applications while preserving crash consistency for both application data and metadata. SPP correctly reconstructs tagged pointers across crashes and provides complete code coverage, including the application's recovery code paths.

Usage model. SPP aims to be integrated into production deployments of PMDK-based systems. It can be tuned to fit multiple use cases. SPP also provides complete control to the developers to define the memory safety-critical code files to further reduce the instrumentation and run-time overheads.

Programming model. SPP supports the native PM programming model and the PMDK APIs. It provides spatial memory safety against PM buffer overflows for PMDK applications.

3.2.2.2 Design Goals and Key Ideas

#1: Transparency. For practical memory safety, SPP should transparently provide memory safety using the native PMDK API, similarly to prior approaches [30, 198, 240]. This is essential to ease the integration into existing toolchains and promote applicability.
Approach: SPP adapts the PMDK functions for PM object management and transactional logging to consider the additional *size* field of the PM pointer representation without altering the APIs (§3.2.4). It further adapts PMDK to construct PM pointers with the SPP's encoding transparently.

#2 Performance and compatibility. SPP aims to be deployed in performance critical environments. To this end, SPP must *(i)* keep runtime and storage overheads at the bare minimum levels while offering high code coverage, and *(ii)* be compatible with existing, uninstrumented, external libraries.
Approach: To minimize the performance and storage overheads, SPP includes optimizations (e.g., pointer tracking) and limits its metadata to the *size* field (8 B), added to the *PMEMoid*. Further, to preserve compatibility, SPP identifies the external library calls and removes the tag from their pointer arguments. Thus, linking against shared libraries is supported without recompilation. Note that SPP cannot provide any memory safety for the code paths of the external functions.

#3: Heterogeneous memory systems. Modern applications are designed to operate on heterogeneous systems that combine PM and volatile memory [117, 193, 253, 256]. A program accesses both PM and volatile memory via native 64-bit pointers. Therefore, SPP should distinguish between the instrumented PM pointers and the uninstrumented volatile memory pointers.
Approach: To identify pointers to PM, SPP *sets* their most significant bit. In that way, SPP tags and instruments exclusively the PM pointers. Additionally, SPP preserves the volatile memory management of an application intact, since in current systems, pointers utilize only 48-57 bits [245].

#4: Crash consistency. PM applications are designed to recover from crashes and maintain the PM data consistent. This process requires designated recovery code paths, which inevitably include PM accesses. Therefore, SPP should be able to reconstruct tagged PM pointers correctly to provide memory safety across restarts and cover the recovery paths.
Approach: SPP enhances the PM pointer representation with a field holding the *size* of the PM object. This representation is updated in the adapted PM management operations using atomic operations or PMDK software transactions. Thus, SPP is able to recreate the PM pointer tags across reboots/crashes since the PM object size information is durable and valid.

3.2.3 Design

SPP enforces a *tagged pointer* (§ 3.2.3.1) approach to detect PM buffer overflows in PMDK applications. SPP consists of *(i)* an adapted PMDK version (§ 3.2.3.2), *(ii)* static analysis compiler passes (§ 3.2.3.3), and *(iii)* a runtime library (§ 3.2.3.4).

3.2.3.1 SPP Pointer Representation

SPP introduces the first tagged pointer scheme for PM. SPP encodes memory safety metadata in the upper bits of each PM pointer [245], as performed in prior tagged pointer approaches [170, 172, 216, 326]. More specifically, a native 64-bit PM pointer is split into four distinct parts (Figure 3.8). Its most significant bit (MSB) is set to 1 to indicate that it points to a PM address. The following bits contain an *overflow bit* and the *tag*, which has a configurable size. The lower bits maintain the actual virtual address of the pointer in the mapped PM file. SPP's pointer *tag* specifies the current distance from the upper bound of the PM object. SPP initializes it as the negated allocated object size, similarly to Delta pointers [170], and updates it on pointer arithmetic operations.

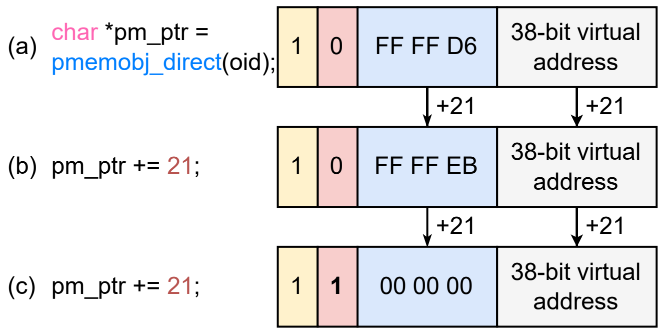


Figure 3.10: **SPP pointer management:** *On each pointer arithmetic operation, the respective action is applied to the tag of the pointer (b). When a pointer surpasses the object's upper bound, the overflow bit gets implicitly set (c).*

Figure 3.10 presents an example of an SPP pointer with 24 tag bits for a 42 B object. Initially, the `pmemobj_direct` function receives the enhanced *PMEMoid* of the object and returns the tagged pointer (Figure 3a). On every pointer arithmetic operation, the virtual address modification is also applied to the tag (Figure 3b). Once the pointer surpasses its upper bound, the overflow bit gets set as shown in Figure 3c. Thus, on an out-of-bounds access, an error is triggered since the pointer is implicitly invalidated due to the overflow bit. This means that SPP requires no explicit, actively-performed runtime bound checks. However, if subsequent pointer arithmetic operations bring the pointer back within its assigned boundaries, the overflow bit gets unset, and the pointer becomes valid again.

SPP's tag encoding is designed to detect PM buffer overflows while aiming to reduce the performance and space overheads. To incorporate protection against additional memory safety bug types, different encoding schemes that maintain the location of memory safety metadata (e.g., lower bound) [172, 326] or fat-pointer approaches [322] can be adapted for PM. In every case, the system needs to consider the crash consistency of the additional metadata.

Further, if the usable pointer bits were not limited, SPP could include a second part in the tag for the lower bound. However, this approach would significantly limit the buffer size, making the approach non-practical. It would also require further manipulation of the tag on pointer operations, which would introduce additional overheads.

3.2.3.2 PMDK Modifications

SPP adapts PMDK to correctly construct the pointer tags even across restarts or crashes. SPP enhances the PM pointer representation (*PMEMoid*) with an additional field maintaining the size of the PM object, incurring minimal space overheads. This choice also improves the access locality and reduces cache pollution, as SPP does not need to search in disjoint memory areas to fetch the metadata.

```
struct PMEMoid {
 uint64_t pool_uuid_lo; // pool id of the pool
 uint64_t off; // offset of the PM object
 uint64_t size; // size of the PM object
};
```

To ensure fault tolerance, PMDK performs the PM object allocation and management either with atomic operations or software transactions. A PMEMoid is considered valid only after its offset field is set. Therefore, SPP adapts the PM management functions (e.g., alloc, realloc) to set the size field, given as an argument to each PM management function call, right before the offset field, thus preserving the semantics of PMDK. In a similar fashion, when a PMEMoid is modified inside a PMDK transaction using the dedicated PMDK API, SPP's size field is logged to preserve crash consistency.

PMDK uses PMEMoids to locate objects across the runs of an application. It exposes the `pmemobj_direct` function that converts a PMEMoid to a 64-bit pointer to the PM object. SPP modifies this function to consider the size field of PMEMoid and return a tagged pointer (§ 3.2.3.1). The additional operations to construct the PM pointer are presented below:

```
#define ADDRESS_BITS (PTR_SIZE - TAG - OVERFLOW - PM_BIT)
#define PM_PTR_BIT ((uint64_t)1 << (PTR_SIZE - 1))
#define OVERFLOW_BIT (~((uint64_t)1 << (PTR_SIZE - 2)))
...
void* pmemobj_direct(PMEMoid oid) {
 //calculate untagged PM pointer
 ...
 //Take the two's complement of the size
 uint64_t tag = (~oid.size + 1) << ADDRESS_BITS;
 return (void*)(pm_ptr | tag & OVERFLOW_BIT | PM_PTR_BIT);
}
```

Despite these changes, SPP does not alter the semantics of the PM programming model. It leaves both the atomic and the transactional APIs [242] intact, supports the type-safety macros [298], and provides multi-threading support with the same thread-safety guarantees with PMDK. Note that SPP's approach is not bound to PMDK but can be adapted for PM programming frameworks following similar principles.

C++ support. PMDK exposes a C API. However, the imposed limitations by the C semantics led Intel to develop C++ bindings in `libpmemobj-cpp` [129]. This library enriches `libpmemobj` with C++ features such as containers and smart pointers. To

```
 1 PMEMoid obj_id;
 2 pmemobj_alloc(pool, &obj_oid, size, ...);//alloc a PM Object
 3 void* pm_ptr = pmemobj_direct(obj_id);//get tagged ptr
 4 pm_ptr += 42;//apply ptr arithmetics
 5 __spp_updatetag(pm_ptr, /*off*/ 42);//update the tag
 6 ...
 7 __spp_checkbound(pm_ptr, sizeof(int));//impl. bounds check
 8 int x = (int)*pm_ptr;
 9 ...
10 clean_pm_ptr = __spp_cleantag(pm_ptr);//tag masking
11 uint64_t ptrtoint = (uintptr_t)clean_pm_ptr;
12 ...
13 internal_foo(pm_ptr);//internal function call
14 clean_pm_ptr = __spp_cleantag(pm_ptr);//tag masking
15 external_foo(clean_pm_ptr);//external function call
16 ...
17 __wrap_memcpy(src_pm_ptr, dst_pm_ptr, 42);//memcpy call
18 __wrap_strcpy(src_pm_str, dst_pm_str);//strcpy call
```

Listing 3.2: **SPP code transformation**: *modifications are highlighted in blue.*

provide complete support for applications developed in C++, SPP adapts the base class for PM pointers to transparently use the modified `pmemobj_direct` function and consider the additional *size* field of the PMEMoid.

3.2.3.3 Compiler Passes

In this subsection, we explain the design and purpose of SPP's compiler passes that are presented in Figure 3.9.

Transformation pass. The transformation pass of SPP instruments the target application by injecting the appropriate function calls to update the PM pointer tag, propagate its value and perform its masking. It is placed before the LLVM optimizer [283] in the compiler pipeline (see Figure 3.9) and identifies the instructions that involve pointer arithmetic operations and updates the tag accordingly (Listing 3.2 Line 5). It further cleans up the tag prior to *ld/st* instructions to perform the bound check implicitly on the upcoming PM access (Listing 3.2 Lines 7-8).

However, in LLVM intermediate representation (IR) there is no distinction between the pointers to volatile memory and those to PM. SPP addresses this by performing static analysis on the produced IR: it tracks the pointer origins and skips inserting runtime checks to operations for pointers that are statically identified to point to volatile memory. Thus, SPP can remove the instrumentation for pointers to volatile memory.

Similarly, for pointers that are guaranteed to point to PM, SPP can directly perform the tag cleaning without checking the PM bit. For pointers whose type cannot be deduced by SPP on the compilation, SPP preserves their instrumentation, and the runtime operations are performed based on their PM bit.

Additionally, pointers can be converted to integers and be used as operands in mathematical or comparison operations. The insertion of the *tag* can affect the correctness of these calculations. Therefore, SPP masks the tag of the pointer prior to the pointer-to-integer conversion (Listing 3.2 Line 10).

LTO pass. SPP's link-time-optimization (LTO) pass ensures compatibility with non-instrumented shared libraries [216]. It precedes almost all LTO passes [282] in the LTO optimization pipeline (see Figure 3.9). It scans through the application code and locates the external function calls with pointer arguments. Right before these calls, the LTO pass masks the pointer tag and passes the untagged pointers to the external function (Listing 3.2 Lines 14-15).

Further, SPP interposes the memory management and string functions (e.g., memcpy, strcpy) with wrapper functions (Listing 3.2 Lines 17-18). The wrappers verify the validity of the accessed address ranges based on the function parameters and perform the respective operation if no violation occurs.

3.2.3.4 Runtime Library

SPP's runtime library contains the implementation of the functions that are injected through the compiler instrumentation. These functions operate on SPP pointers to clean and update the tag after they verify that the pointer points to PM.

Precisely, the `__spp_cleantag` function returns the PM pointer after masking out its tag and PM bit as shown below:

```
/* PTR_BITS denote the bits for the virtual address */
#define PTR_MASK (1ULL << 62) | ((1ULL << PTR_BITS) - 1)
...
void* __spp_cleantag(void *ptr) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    /* keep the overflow and the virtual address bits */
    return ptr & (PTR_MASK);
}
```

Thus, the application gets the actual virtual address, which can be normally accessed through *ld/st* instructions. The overflow bit is preserved so that any subsequent memory access through an overflown pointer is detected.

Further, the `__spp_updatetag` function is invoked when a tag needs to be updated due to a pointer arithmetic operation. The provided offset to this function is determined via the static analysis compiler pass. The tag of the given pointer is extracted and incremented by the offset value. If a PM pointer overflows, this operation implicitly sets the overflow bit. After the tag update, the new value is merged into the PM pointer, which is returned back to the application, as presented here:

```
void* __spp_updatetag(void *ptr, int64_t off) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    /* extract and update the tag */
    int64_t tag = (int64_t)__spp_extract_tag(ptr);
    tag = tag + off;
    /* return the updated tagged pointer */
    return (void*)__spp_insert_tag(ptr, tag);
}
```

Additionally, the runtime library implements the `__spp_checkbound` function, shown below. It is called prior to every PM access. It updates the tag based on the size of the dereferenced pointer type since this indicates the upper bound of the memory access. After the update, the PM pointer gets masked and is returned to the application to perform the intended access. If the overflow bit is set, this access will trigger a segmentation fault or a bus error.

```
void* __spp_checkbound(void *ptr, size_t deref_size) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    void* upd_ptr = __spp_updatetag(ptr, deref_size - 1);
    return __spp_cleantag(upd_ptr);
}
```

Lastly, the SPP runtime library includes the wrapper functions for memory intrinsic (e.g., memcpy, memmove) and string management functions (e.g., strcpy, strcmp). These functions perform memory accesses to specified address ranges. Therefore, SPP calculates the maximum addresses they intend to access for each PM pointer argument and updates the tag(s) before the actual function call. If any of the addresses lies outside the defined PM objects' boundaries, the respective pointer's overflow bit is set. Then, SPP masks out the tags and the PM bit and executes the built-in function. This execution will raise an error if any masked pointer is invalid due to the overflow bit, preserving SPP's memory safety properties.

3.2.3.5 Optimizations

SPP aims to provide spatial memory safety for PM with low overheads. To this end, the instrumentation includes optimizations to reduce the SPP function calls (e.g., for pointers to volatile memory) and merge or omit instrumentation steps whenever possible (e.g., constant pointer increments in a loop).

Pointer tracking. SPP's compiler passes perform pointer origin tracking and divide the pointers into three categories based on the memory type they point to, namely, *volatile*, *persistent*, and *unknown*. The category of each pointer is decided based on the API that generates it. More specifically, in SPP, we refer to the pointers returned by the traditional volatile memory management APIs (e.g., malloc, realloc, new) as *volatile*. Pointers referring to C++ Vtables and error handling are also known to be *volatile*. Equivalently, pointers that were constructed by specific PMDK functions (e.g., pmemobj_direct) are characterized as *persistent*. The remaining pointers (e.g., pointers loaded from memory) are considered *unknown*. SPP also tracks the derived pointers (e.g., via pointer arithmetics) and adds them in the category of their predecessor, if specified.

SPP's LTO pass proceeds one step further and analyzes the function pointer arguments. It scans the calling sites of each function and records the type of the pointer arguments passed by the caller. With this method, SPP can determine the category of a function pointer argument, provided that *all* the callers use pointers falling into a single category.

The benefit from the pointer classification is twofold. First, SPP can omit the instrumentation for the *volatile* pointers. avoiding multiple redundant function calls injection. Second, SPP can skip the PM bit check in its hook functions when operating on known *persistent* pointers. For pointers whose category cannot be determined (*unknown*), SPP keeps the instrumentation, including the runtime pointer type checks.

Currently, the pointer tracking is designed for PMDK APIs. However, it can be adapted and incorporated into different PM frameworks that can benefit from or require the characterization of whether a pointer points to PM or volatile memory.

Bound checks preemption. SPP leverages the static analysis to identify basic code blocks and simple loops that include consecutive updates on the same pointer with constant offsets or following a known pattern during compile time. An example of the bound check preemption is shown below. In this case, SPP's transformation pass calculates the maximum pointer offset and performs a single tag update followed by a dummy memory access on the updated pointer (blue highlight). This memory access acts as a bound check. It silently verifies the validity of the upcoming memory accesses related to this pointer, and, thus, SPP can omit the associated tag updates and bound checks in the specified code block (red highlight).

```
void* pm_ptr = pmemobj_direct(obj_id); // get tagged ptr
__spp_updatetag(pm_ptr, /*total_off*/ 16);
__spp_checkbound(pm_ptr, sizeof(int));
pm_ptr += 8;
~~__spp_updatetag(pm_ptr, /*current_off*/ 8);~~
~~__spp_checkbound(pm_ptr, sizeof(int));~~
int x = (int)*pm_ptr;
pm_ptr += 8;
~~__spp_updatetag(pm_ptr, /*current_off*/ 8);~~
~~__spp_checkbound(pm_ptr, sizeof(int));~~
int y = (int)*pm_ptr;
```

3.2.3.6 Additional Design Details

Crash consistency. SPP preserves the crash consistency property for the PM data. SPP adapts PMDK internally so that the added *PMEMoid* field is set and updated in a fail-safe manner when the application uses the dedicated PMDK APIs.

Precisely, in PM allocations, SPP atomically sets the *size* field of the *PMEMoid* before PMDK validates the object allocation by assigning it with its *offset*. This is achieved through writing the *size* object in the redo log, which ensures that the setting of this field precedes the setting of the *offset*.

For the case of reallocation of a PM object, the entire *PMEMoid* structure is captured in a log. Since the amount of logged bytes is determined by the size of *PMEMoid* object, SPP does not have to interfere with this operation, further than simply setting the new *size* of the reallocated object.

Lastly, for the general case that a PM object containing a *PMEMoid* needs to be snapshotted in a transaction, the additional 8 B, that SPP introduces, are implicitly added in the transactional undo log. This is achieved with the help of the type system that accounts for these bytes when calculating the *PMEMoid* size, e.g., with the `sizeof()` function.

However, if a *PMEMoid* is updated manually, it must be wrapped in a transaction and be snapshotted in the undo log by the developer. Thus, in case of an unexpected crash, the recovery process of PMDK will restore the logged value.

Address space layout. Reserving a part of the pointer for the PM bit and the tag reduces the number of available bits leading to virtual address space limitation. This limitation only affects the regions where a PM pool is mapped. Therefore, we configure our PMDK version to map the PM pools in the lower part of the virtual address space. The exact address space limit depends on the configurable tag size so that every PM object can be addressed using $(64 - tag\_bits - 2)$ bits. Volatile memory management can utilize 63 out of the 64 bits (excluding the PM bit), which are sufficient for current systems [245]. In this case, the address space layout randomization (ASLR) is disabled for PM mappings. While ASLR has a broader memory safety spectrum than SPP (e.g., use-after-free), its guarantees are probabilistic. Instead, SPP offers deterministic PM buffer overflow protection. Overall, a more sustainable future solution would be to use fat-pointers (e.g., 128 bits) where the first 64 bits contain the safety metadata. This approach comes with higher performance overheads as it requires additional memory accesses for metadata fetching.

3.2.3.7 Limitations

SPP detects PM buffer overflows in PMDK applications, provided that PM is managed with the PMDK APIs. SPP comes with inherent limitations due to *(i)* limited pointer bits, *(ii)* potential arbitrary pointer operations and *(iii)* the requirement for compatibility with pre-compiled shared libraries.

PM object & PM pool size. In SPP design, we face the limitations imposed by the 64-bit native pointers. The PM and overflow bits decrease the number of available bits to 62, which should enclose both the tag and the virtual address of the PM objects. The number of tag bits limits the PM object size, while the virtual address space bits limit the maximum size of a PM pool. To deal with this limitation, a redesign to store the memory safety metadata in disjoint memory locations during runtime can be considered, but this inevitably would increase the performance costs due to frequent lookups, additional memory loads, and cache pollution.

Targeting for minimal overheads, in our approach, we configure our PMDK version to set the maximum PM object size to *1<<tag_bits* bits and the maximum PM pool size to *1<<62-tag_bits* bits. However, SPP allows for a configurable amount of *tag_bits*, which can be tuned by the developers. Notably, regarding the PM space overheads, SPP's tag-based approach is immune to the object size and only depends on the amount of PM pointers that an application is using. Importantly, many PM applications often deal with small to medium-sized objects. To highlight this, we apply SPP on sample applications shipped with PMDK and on a key-value store [117]. We observe that SPP provides complete coverage for these applications with minimal PM space overheads on average.

In summary, the use of tag bits in SPP presents certain tradeoffs. SPP is designed for scenarios where the benefits of fast runtime bounds checking outweigh the minimal PM overhead and the PM object size limitations introduced by the number of tag bits. It is particularly effective for applications dealing with sensitive data or those requiring high reliability, where the cost of potential memory safety violations far exceeds the PM space and object allocation size limitations. However, for extremely memory-constrained environments or applications requiring more flexibility in the PM pool and PM object size, SPP's approach may not be ideal.

Pointer operations. Typically, the pointer subtraction is performed after converting the pointers to integers based on the LLVM standard. Such operations on tagged pointers can lead to incorrect results. Therefore, SPP masks the pointer before the subtraction to provide the expected operation outcome.

Similarly, in cases of pointer comparisons, SPP also masks the pointers to ensure correctness. This process does not affect SPP's guarantees since the converted values are only used for the comparison and are never dereferenced.

Additionally, when an application performs a pointer to integer operation, SPP preserves only the virtual address bits. Thus, the application receives the expected value. However, when an integer is converted to a pointer, SPP cannot provide its memory safety guarantees since the integer does not contain a tag, even if it was derived from a previously tagged pointer. The latter case could be addressed by tracking the origin and type of such pointers (e.g., with the use-def chain of LLVM [300]) and maintaining the tag in a data structure to restore it in an upcoming integer-to-pointer conversion.

In general, arbitrary pointer operations can result in an out-of-bounds pointer by an offset that resets the overflow bit, hindering SPP from reporting the memory safety violation. A typical example is when the offset of a pointer exceeds the representation range of the address bits. Currently, this case is not handled explicitly, but SPP can be enhanced to either emit an error or manually set the overflow bit. The former would be a better approach to prevent any further pointer misuse since such actions mostly originate from malicious activities.

Lastly, SPP does not protect against arbitrarily generated pointers that might end up landing on PM, as it can be neither predicted nor prevented. Similar limitations apply in most memory safety approaches.

Shared libraries. SPP masks the pointers passed to shared libraries to preserve correct functionality. For the pointers returned from shared libraries, SPP cannot provide any memory safety guarantees as they are not guaranteed to be tagged, and the way they originated within the shared library is unknown. Therefore, SPP cannot assign them with a tag. SPP provides memory safety guarantees for PM pointers generated and preserved in the compilation units it has access to.

Additionally, SPP is not currently able to identify whether a shared library is instrumented because SPP treats each compile module on its own. Therefore, the functions of shared libraries that an instrumented application is linked against are seen as external. SPP precedes calls to these functions with a tag-cleaning operation for their pointer arguments. Thus, despite the libraries being potentially instrumented, the tag is not propagated to them by the application, and memory safety guarantees cannot be ensured in such cases.

3.2.4 Implementation

SPP is built based on PMDK v.1.9 and LLVM v.12.0. It consists of *(i)* a static analysis transformation pass, *(ii)* a link-time-optimization pass, and *(iii)* a runtime library.

3.2.4.1 Compiler Support

Transformation pass. The SPP static analysis transformation pass scans the application and inserts the appropriate SPP runtime function calls. The pass operates on the LLVM Intermediate Representation (IR) of every translation unit and is placed before the LLVM optimizer in the compiler toolchain [283].

Initially, the SPP transformation pass tracks the pointer variables of the IR. Global pointers and pointers to volatile heap-allocated objects are *volatile*, while pointers derived through the `pmemobj_direct` function are *persistent*. The rest are classified as *unknown*. Following, each instruction of the module is examined. Based on the instruction type, the SPP's transformation determines where to insert the appropriate *callsites* to SPP's functions in the target module's IR.

More precisely, when the pass locates a pointer arithmetic operation, or a *GetElementPtrInst* (`GEP`) in LLVM terminology, it calculates the offset of the operation and inserts a call to the `__spp_updatetag` to update the tag after the `GEP`, as shown in Figure 3.10. Similarly, on *ld/st* instructions, SPP updates the pointer tag based on the pointer type size and masks the pointer for the actual dereference by inserting a call to `__spp_checkbound`, as described in § 3.2.3.4. Further, calls to `__spp_cleantag` are injected before the pointer-to-integer conversions (*PtrToIntInst*) to preserve correct code behavior. Lastly, SPP's transformation pass masks the pointer for function arguments passed by value (*Attr*::*ByVal*) since they implicitly perform an object copy.

LTO pass. The link-time-optimization (LTO) pass of SPP performs an analysis and instrumentation of the whole program during the linking phase for further optimizations. In our implementation, we use the *gold* linker [79, 188].

We place our pass before the LLVM inliner in the pipeline [282]. SPP compiles its runtime functions into object files. These files are linked against the target application's IR. In this way, SPP allows LLVM to apply its effective optimizations and perform the inlining of SPP's functions whenever possible. SPP hints the compiler to `always_inline` its functions and prevents them being optimized out with the `used` attribute.

Further, the LTO pass performs a more exhaustive pointer tracking since it iterates over all the compile units of the application. Thus, SPP classifies further *volatile* and *persistent* pointer arguments by examining each function's calling sites. Using this information, it omits the pointer type check in the hook functions for the identified *persistent* pointers. Apart from that, this tracking allows SPP to prune injected calls when they have a *volatile* pointer as an argument whose category could not be determined via the transformation pass.

3.2.4.2 Runtime Library

PMDK wrappers. SPP includes wrappers for the core PMDK operations (i.e., PM heap management). The exposed PMDK API remains intact. The only deviation is that the environment variable `PMEM_MMAP_HINT` is set to 0 so that the PM pool is mapped to the lower part of the virtual address space.

SPP's wrappers are responsible for setting and updating the introduced 64-bit *size* field of a *PMEMoid* without violating the crash consistency property. Both the atomic and the transactional operations that affect the object's size (e.g., alloc, realloc) are considered, covering all the PM heap allocations. Precisely, for the persistence of the pointers, SPP provides the same atomicity guarantees with PMDK. In case of an object (re)allocation outside a PMDK transaction, SPP leverages the PMDK redo logging and performs an atomic operation that validates the (re)allocation after setting the *size* field in the *PMEMoid*. When the object management is performed within a PMDK transaction, SPP intercepts the functions that perform the snapshotting to ensure that the additional *size* field is included in the undo log so that it can be restored in case of a crash during the transaction. SPP also performs bounds checks to prevent overflows on the snapshotted objects that could lead to information leakage through the transaction logs.

Lastly, SPP adapts the `pmemobj_direct()` function to construct a tagged pointer from a *PMEMoid* (§ 3.2.3.2). It leverages the *size* field of the *PMEMoid* to create the tag and returns the tagged pointer to the caller.

Hook functions. SPP's runtime library contains a set of hook functions that are injected in the LLVM IR of the instrumented code. These functions update and mask the tag of PM pointers, as explained in § 3.2.3.4. Apart from the described hook functions, SPP implements equivalent functions with a `_direct` suffix that omits the pointer type check. They are only used when a pointer is determined to point to PM.

SPP handles separately the memory management functions, e.g., `memcpy`, `memset` and `memmove`. For each pointer operand, SPP injects a call to its `__spp_memintr_check` function. This function updates the tag based on the maximum address that the function operates on and masks the pointer. Then, the masked pointer is passed to the original memory management function. If the tag update sets the overflow bit, a fault will be triggered due to the invalid pointer during the function execution, showcasing the overflow.

Similarly, SPP interposes the common string manipulation functions (e.g., `strcpy`, `strcat`) at link time. The runtime library includes wrapper functions that perform the tag update and tag masking based on the arguments of each string function and then call the original function.

Lastly, SPP ensures *compatibility* between instrumented code and pre-compiled, uninstrumented libraries via its `__spp_cleantag_external` function. It is injected before the calls to external functions and removes the tag and the PM bit from the pointer arguments promoting interoperability. However, SPP has a caveat; a tagged PM pointer can be mistakenly passed to an external function as part of a struct since SPP currently does not perform any intra-object analysis.

3.2.4.3 Optimizations

Pointer tracking. SPP compiler passes perform pointer tracking to differentiate between *volatile* and *persistent* pointers. This is to avoid redundant attempts to perform bounds checking and tag cleaning on tag-free volatile pointers.They iterate over each translation unit in the LLVM IR and categorize each pointer based on how it is derived. More specifically, if a pointer is obtained via the `pmemobj_direct` of PMDK (or its equivalent `get()` function in C++), it is considered *persistent*. Similarly, pointers created via volatile allocation functions (e.g., `malloc`), pointers to C++ VTables (e.g., `vfn` or `vtable` prefixed), or pointers used by common functions that are known to point to the volatile heap (e.g., `pthread_create`) are classified as *volatile*. Further, pointers returned by external functions are accounted as *volatile* to avoid the instrumentation since they are not tagged. These pointer categories are also propagated via the `GEP` and `BitCast` LLVM instructions. The remaining pointers are characterized as *unknown*. This classification enables SPP to remove useless function calls that are injected for *volatile* pointers and, equivalently, to omit the pointer type check for the *persistent* ones by using the `_direct` suffixed version of the hook functions. For the pointers with *unknown* type, SPP preserves the instrumentation and checks the PM bit to determine their type and perform the appropriate action.

Bound checks preemption. During development, we observed that many pointers are consecutively updated and dereferenced in a single LLVM *BasicBlock*. Therefore, instead of performing a costly tag update and masking for each `GEP` and `ld/st`, SPP calculates the maximum offset that is added to the pointer and updates the tag only once before the first GEP. Then, it places a dummy `ld` to implicitly perform the bound check and replaces the uses of the pointer with the masked one, which gets exempt from further instrumentation. The injected `ld` is tagged *v*olatile to avoid being optimized out by the compiler.

To further reduce the SPP's overheads, SPP hoists bound checks out of loops whenever possible. SPP checks every monotonic loop for existing *loop-invariant* expressions referring to pointers using LLVM's *scalar evolution*. If a pointer can be hoisted, SPP calculates its max offset that is used for dereference in the loop and places a tag update and a dummy `ld` in the loop *pre-header*. In this optimization, the `ld` is also tagged *v*olatile. Thus, SPP performs the pointer instrumentation only once rather than at every loop iteration.

However, due to the bound check preemption, SPP might indicate a false code location for an overflow, pointing to the injected dummy `ld`. To address this issue, bound check preemption optimizations can be optional, and the programmer can choose to enable them depending on the target environment.

3.2.5 Evaluation

We evaluate SPP on the following three aspects:

- **Performance & space overheads:** We measure the performance (§ 3.2.5.2) and space (§ 3.2.5.3) overheads of SPP using PMDK's microbenchmarks, a persistent key-value (KV) store, designed and optimized for PM, namely pmemkv [117] and a port of Phoenix 2.0 [251] benchmark suite to PM.
- **Effectiveness:** We evaluate the capability of SPP in detecting PM buffer overflows (§ 3.2.5.4). We use the RIPE framework [320], where we focus on buffer overflow exploits. We also detect memory safety bugs in the PMDK examples.
- **Crash consistency:** We verify the crash-consistency property (§ 3.2.5.5) with Valgrind's `pmemcheck` tool [240].

3.2.5.1 Experimental Setup

Testbed. We conduct our experiments on a two-socket server machine equipped with Intel(R) Xeon(R) Gold 6326 CPU (16 cores), 64 GB (4 channels $\times$ 16 GB/DIMM) DRAM and 1 TB (4 channels $\times$ 256 GB/DIMM) Intel Optane DC DIMMs per socket. PM is configured in App-Direct mode [127]. The machine is running NixOS 22.05 with kernel version 5.15.49.

Variant	Description
PMDK [239]	PM application using unmodified PMDK
SafePM [30]	PM application instrumented with SafePM
SPP	PM application instrumented with SPP

Table 3.6: Benchmarking variants

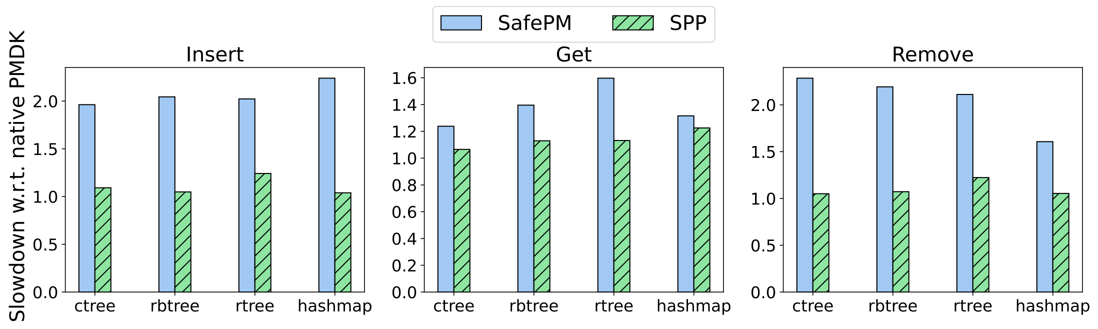


Figure 3.11: Performance overheads (throughput) of persistent indices for SPP and SafePM w.r.t. the native PMDK execution.

Variants. We perform our experiments with the variants of Table 3.6. As our baselines, we consider the application compiled with *(i)* native PMDK, and *(ii)* SafePM sanitizer enabled. We set the optimization level to O2 and use 26 *tag* bits. The results present the average of 3 runs unless otherwise specified.

3.2.5.2 Performance Overheads

Persistent indices. To measure the performance overhead, we use `pmembench` [126]. For each variant of Table 3.6, we execute experiments on the PM indices of PMDK, namely *ctree*, *rbtree*, *rtree* and *hashmap*, with a single query type (*insert*, *get*, *remove*) per run. Each workload consists of one million queries. The keys are 8 B and follow a uniform distribution.

Figure 3.11 reports the normalized performance overhead for SafePM and SPP having the native PMDK as a baseline. Overall, SPP achieves 9.25%, 13.75%, and 10.5% lower average throughput compared to PMDK for each query type. The respective values for SafePM are 101%, 37.75%, and 101.75%. The large overhead difference comes from SPP's compiler optimizations and tagged pointer utilization – SPP's LLVM pass removes redundant runtime checks on volatile pointers and unlike SafePM, SPP does not access remote memory regions for bounds information at every `ld/st`. Further, `pmembench` has limited external function calls. This allows SPP to perform better pointer tracing and reduce the tag-cleaning operations for external functions. Additionally, compared to DRAM memory safety approaches, SPP introduces lower rel-

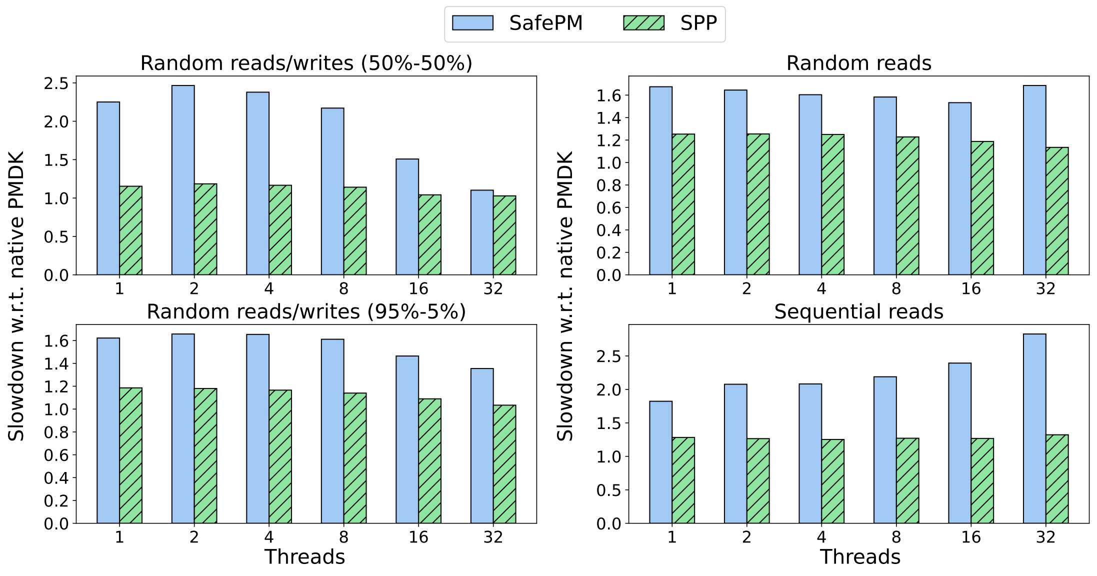


Figure 3.12: Performance overheads (throughput) of SPP and SAFEPM w.r.t. the native PMDK execution for `pmemkv`.

ative overheads since the performance impact of tag updating and cleaning operations in SPP is proportionally lower due to the slower PM access. For certain experiments, SPP approaches the native PMDK performance having an overhead of around 6%. This indicates the practicality of SPP.

Persistent KV store. We measure the performance impact of SPP on *pmemkv* [117] using its non-experimental, concurrent, persistent engine [241]. For our benchmarking, we use `pmemkv-bench` [111], which is based on the *db_bench*. We consider four workload types: *(i)* update intensive (50%R-50%W), *(ii)* read intensive (95% R-5%W), *(iii)* random reads and *(iv)* sequential reads. We perform 10M operations for each workload. The key size is set to 16 B and the value size to 1024 B. Prior to each run, we insert 1M keys into the KV store.

Figure 3.12 illustrates the performance overhead of SPP and SAFEPM normalized to the PMDK. SPP causes an average 18.3% throughput decrease across the workloads while the respective value for SAFEPM is 84.4%. The overhead of SPP mostly stems from redundant checks for volatile pointers that SPP cannot identify at compile time. Regarding the scalability, we observe that SPP follows a similar pattern to the PMDK, indicating the minimal effect of SPP on the parallel execution.

Phoenix benchmark suite. We evaluate the performance impact of SPP in CPU intensive scenarios. We port all 7 applications of the Phoenix benchmark [251] to use PM objects via the PMDK API. For each application, we use 8 threads and the largest provided dataset as input. To accommodate larger allocation sizes (e.g., the input files),

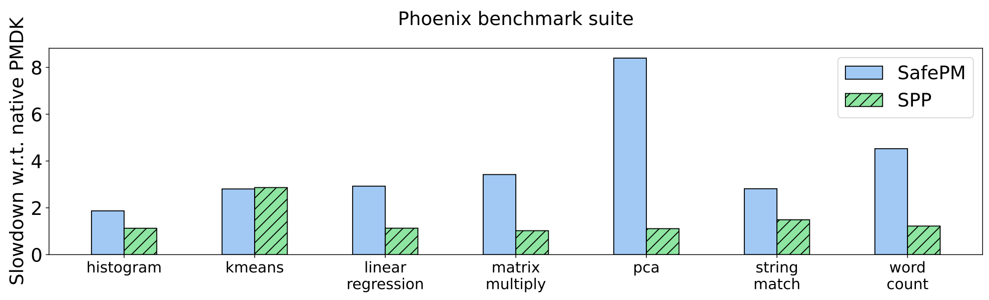


Figure 3.13: Performance overheads of SPP and SAFEPM w.r.t. the native PMDK execution for the `Phoenix` benchmark suite.

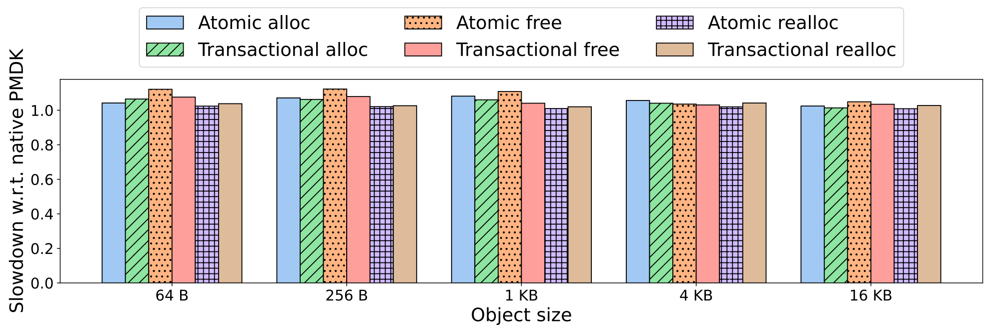


Figure 3.14: Performance overhead of SPP for PM management operations.

we set the tag bits to 31 for SPP. The presented results are the average of 20 runs.

Figure 3.13 presents the slowdown of SPP and SAFEPM having PMDK as the baseline. SPP causes a slowdown of 2-23% for Phoenix benchmark applications, except for the *kmeans* where it incurs 180%. The respective values for SAFEPM range from 83% to 750%. The significantly reduced overheads of SPP can be reasoned by the effective pointer tracking since SPP has all the source code of the applications available for its analysis. The unique case of the *kmeans* benchmark can be justified as this application iterates constantly over its working set, leading to a higher performance impact of the SPP's instrumentation in its execution. Note that the Phoenix port is not optimized for PM. It uses plain memory intrinsic functions (e.g., `memcpy`), which do not allow SPP to avoid some pointer type checks. This implies that in a more sophisticated, PM-oriented port, the overheads of SPP can be further diminished.

Atomic and transactional PM operations. We evaluate the effect of SPP on PMDK's atomic and transactional PM management functions. We use `pmembench` [126], where we configure each experiment to perform 100K operations while varying the object size. We present the average of 10 runs.

Figure 3.14 reports the normalized slowdown of SPP for each PM management operation. For almost all the operations, the performance of SPP is close to the PMDK for

	Snapshotted *PMEMoids*				
Variant	**100**	**1000**	**10K**	**100K**	**1M**
PMDK	17.62	17.78	18.82	28.52	119.77
SPP	17.77	17.86	18.87	28.66	120.00

Table 3.7: Recovery time in milliseconds (*ms*).

Data	**Insert**		**Get**	
structure	**(MB)**	**(%)**	**(MB)**	**(%)**
ctree	0	0%	0	0%
rtree	2127	39.7%	2127	39.7%
rbtree	0	0%	0	0%
hashmap_tx	5	0.43%	5	0.43%

Table 3.8: SPP space overhead

the various object sizes (1-8% slowdown). It can be justified, as this microbenchmark only allocates PM objects and performs no PM access after the respective operation. Therefore, the overhead comes from redundant checks for the pointer type that SPP's static analysis cannot optimize away. The only operation with high overheads(7-17%) is the *atomic free*. This is due to SPP's required runtime checks, compared to a single atomic operation that PMDK requires to *free* the PM object.

Recovery time. We measure the recovery time of an application and compare SPP with PMDK. We develop a microbenchmark that allocates PM objects in a pool. We present a worst-case scenario for SPP where an application snapshot exclusively *PMEMoid*s in a transaction, resulting in larger logs for SPP. The number of objects per experiment is shown in Table 3.7. After the snapshotting, we inject a crash and trigger a recovery. Our results indicate an average of 100 runs.

The slightly increased recovery time is caused mainly by the need for restoration of the additional *size* field of the *PMEMoid*. Note that SPP does not interfere with the internal recovery process of PMDK. However, user-defined recovery functions could pose higher overhead since they are also subject to SPP's instrumentation as part of the application, where SPP must provide its spatial memory safety guarantees.

3.2.5.3 Space Overhead

We measure the space overhead introduced by SPP compared to the native PMDK. SPP's space overhead is caused by the *size* field, added to the *PMEMoid* of PMDK. We reuse the four PM indices with the *insert* and *get* workloads and 1M keys, as explained in § 3.2.5.2. The reported values indicate the space overhead after the execution of the

RIPE variant	Successful	Prevented
Volatile heap	83	140
PM pool heap	83	140
SafePM	6	217
SPP	4	219
memcheck	20	203

Table 3.9: RIPE attacks using different protection mechanisms.

application.

The PM space overheads of SPP are presented in Table 3.8. We conclude that SPP wastes minimal PM space to store its memory safety metadata (0-0.43%) for all persistent indices except the rtree. In the extreme case of rtree, SPP consumes 39.7% more PM space compared to PMDK. It occurs as each rtree node contains 256 *PMEMoids* and SPP's space overhead is proportional to the number of *PMEMoids* an application stores in PM. Note that this is not a common pattern in PM applications based on our observations. A future design can completely eliminate the PM space overhead if the object size gets encoded along with the object offset in the *PMEMoid* structure, thus requiring no extra PM space. This feature is not included in the current SPP version due to time constraints.

3.2.5.4 Effectiveness

In this experiment, we examine the effectiveness of SPP. We use the RIPE benchmark framework [320]. It contains a set of different memory vulnerability exploits. We focus on buffer overflows. We use the 64-bit version of RIPE [255] that allocates objects on PM via PMDK [30]. We consider the following variants: *(i)* Volatile heap, where RIPE uses volatile memory, *(ii)* PM pool heap, where it uses the PM heap, *(iii)* SafePM, where it uses the PM heap with SAFEPM sanitizer enabled, *(iv)* SPP, where the application uses the PM heap and is instrumented with SPP and *(v)* memcheck [238], a Valgrind tool for memory bugs in PM. Each memory exploit is executed 3 times in each run. We perform the RIPE experiments several times to ascertain the stability of our reported results.

Table 3.9 shows the exploits that are successful or prevented throughout our runs. We observe that porting RIPE to use PM preserves the number of potential buffer overflow exploits (83). Out of these attacks, SPP is able to prevent 79 while SAFEPM detected 77. The memcheck [238] identified 63 attacks. We further examine the non-detected attacks by SPP and realize that the constructed PM buffer is only directly accessed in bounds. Overall, SPP is capable of detecting almost every PM buffer overflow with a notably lower performance overhead compared to its state-of-the-art counterparts.

Reproducing bugs. To further verify the effectiveness of SPP, we reproduce and detect a reported PM buffer overflow bug in PMDK's btree index [236]. More specifically, on line 378 of *btree_map.c* file, the *memmove* call leads to a buffer overflow on a PM data object, which SPP is able to identify and report.

Additionally, we test various examples shipped with PMDK [131]. We apply SPP on implementations of an array, a queue, a FIFO list, a solution of Buffon's Needle problem, a program for the π calculation, and a slab allocator. Using SPP, we identify three PM buffer overflows in the array example. Precisely, when an array *realloc* is requested, its return value is not checked. In case of a failed reallocation to a larger size, the application attempts to fill the newly supposedly allocated array, which results in an overflow, as the original array is not resized. This bug occurs in lines 215, 235, and 257 of the array example [130]. The remaining examples do not report any error throughout their execution with arbitrary inputs.

Further, we identify an off-by-one buffer overflow in the `string_match` benchmark of Phoenix, when the `read` function is used for the input file. This bug is detected when we execute the ported version with SPP. It occurs when trying to access a character beyond the input buffer [169]. We verify our finding using ASan [264] on the volatile memory version. We report the bug [233] and its respective fix [234].

On top of that, we develop sample examples with various kinds of PM buffer overflows (e.g., overflows during snapshotting, built-in memory functions overflows, etc.) for testing. SPP identifies and reports all of the above cases.

3.2.5.5 Crash Consistency

We verify that SPP preserves the crash consistency for the PM data despite the addition of the *size* field in the PM pointer. We use `pmemcheck` [240] and `memcheck` [220]. `pmemcheck` is a Valgrind plugin that allows for exploring and verifying the data consistency in PM applications [30, 128, 144, 185]. Its output is passed to `pmreorder` [244] to explore the state space. We perform the same experiments as in § 3.2.5.2. Due to Valgrind's overheads, we set the number of operations to 10000 to shorten the execution time.

`pmemcheck` and `pmreorder` do not report any error. `memcheck` also has an empty error log for the persistent indices. For the PM operations, `memcheck` provides the same error output with the case of unmodified PMDK which does not indicate any crash consistency violation.

3.2.6 Related work

Persistent memory systems. There exists a large body of work on PM filesystems that aims to reap the performance benefits of persistent memory as a storage medium [46, 157, 312, 325, 341]. Further, persistent memory has already been incorporated in the design of many high-performant data management systems [117, 165, 199, 253, 256]. On top of that, several proposed distributed systems leverage the capability of accessing PM remotely via RDMA to improve their efficiency [133, 152, 189, 273, 329, 338]. Unlike these systems that focus on high performance and correctness, SPP efficiently tackles the problem of memory safety on PM.

Further, recent works [29, 193, 317] leverage accelerators (e.g., FPGAs) where they offload PM operations (e.g., cache line flushes, logging). Such systems can improve the PMDK performance. We expect that the performance boost will be similar for SPP, as it uses PMDK underneath for these operations, does not hamper the cache locality, and has a minimal contribution to the amount of logged data.

SW memory safety approaches. Various software-based approaches have been proposed to deal with the memory safety bugs for *volatile* memory [3, 170, 172, 181, 182, 212, 216, 264, 331]. The main target of these approaches is to preserve compatibility and high efficiency while incurring low performance and memory overheads. They apply different techniques such as pointer tagging [170] or the shadow-memory concept [264] and are often accompanied by compiler instrumentation [212, 331] and runtime libraries. Differently from such approaches, SPP focuses on memory safety for PM and achieves low runtime overheads for PM applications due to its conservative, yet effective, distinction of pointer types through the introduced PM bit as well as its selective instrumentation of PM pointers that eliminates redundant runtime function calls for volatile memory pointers. However, for complete memory safety, SPP can be combined with other SW-based memory safety approaches targeting volatile memory since it practically only affects the PM pointer representation. Note that at the cost of additional performance overhead, SPP could be generalized and include instrumentation and checks for volatile memory pointers, similarly to prior work [170].

Targeting memory safety for PM, our SafePM [30] and the valgrind-based `memcheck` [238] tool have been proposed. As a reminder, SafePM is built on Google's AddressSanitizer [264] while `memcheck` leverages Valgrind and PMDK's internal code annotations to provide memory safety. Both these approaches incur considerable performance and space overheads and are destined for debugging purposes. On the contrary, SPP intends to be an efficient, low-overhead memory safety solution.

Lastly, Corundum [99] is a PM management library in Rust that enforces language-based memory safety. In contrast, SPP can be deployed in existing PMDK applications without any source code modifications and does not require re-developing applications with certain libraries or languages.

HW memory safety approaches. Striving for lower performance overheads, several works introduce HW extensions to provide memory safety for volatile memory [14, 59, 69, 77, 209, 210, 223, 224, 299, 322, 334]. Cheri [322] employs hardware capabilities while lowfat pointers [69] offer spatial memory safety using compact fat pointers that contain the encoded object bounds. They propose a hardware-level implementation for faster decoding and pointer validation. Intel MPX [223, 224] and Arm MTE [14] provide ISA extensions to prevent memory safety bugs for Intel x86-64 and Arm architecture respectively. Hardbound [59], SafeProc [77] and WatchdogLite [210] propose further ISA extensions that work collaboratively with a compiler instrumentation aiming towards better performance. In contrast to these approaches that rely on specialized HW or require ISA modifications and are dedicated to volatile memory safety, SPP can be used in commodity HW and detect memory safety bugs on PM.

PM allocators and libraries. PM allocation and management is an active area of research [34, 45, 57, 120, 227, 313]. Mnemosyne [313], NVHeaps [45] and PMDK [120] provide APIs for PM management and implement transactions to guarantee crash consistency. They distinguish between pointers to volatile memory and PM to avoid ephemeral or stale references being reused across restarts if the developer uses their APIs correctly. Notably, NVHeaps [45] and Mnemosyne [313] are evaluated using simulated PM while PMDK is optimized to leverage the HW features of actual PM devices. These approaches, unlike SPP, do not provide any memory safety but only ways to manage PM in a correct manner, which SPP also performs as it is based on PMDK by design.

Poseidon [57] is a PM allocator that prevents metadata corruption using Intel MPK [228]. GPM [227] exposes an API to access PM directly from the GPU with respect to crash consistency and persistence. However, ensuring crash consistency is a non-trivial task. Therefore, multiple frameworks have been proposed to verify, or potentially ensure, this property for PM applications [29, 62, 75, 83, 151, 185–187, 217].

3.2.7 Summary

In this section, we present SPP, the first tagged pointer-based mechanism designed to provide practical memory safety for PM applications. The PM pointer tag indicates the pointer distance from the end of the PM object and gets implicitly invalidated when it surpasses this boundary. SPP consists of a compiler instrumentation based on LLVM, a runtime library, and an adapted PMDK version. It enhances the persistent pointer representation of PMDK with memory safety metadata, which is set and updated in a crash-consistent manner. Its runtime functions ensure the correct tagged pointer management and provide compatibility with pre-compiled external libraries. SPP maintains the PMDK API intact. Consequently, SPP requires no source code modifications and can be seamlessly integrated into existing PM software. Our thorough evaluation shows that SPP effectively detects PM buffer overflows with low-performance costs and negligible space overheads.

Software artifact. SPP is publicly available along with its experimental setup.

Chapter 4

Security for Persistent Memory

4.1 ANCHOR: A Library for Building Secure Persistent Memory Systems

This section highlights the design of ANCHOR, a library for building secure PM systems. ANCHOR provides strong hardware-assisted security properties while ensuring crash consistency. ANCHOR exposes APIs for secure data management within the realms of the established PM programming model, targeting byte-addressable storage devices. ANCHOR leverages trusted execution environments (TEE) and extends their security properties on PM. While TEE's protected memory region provides a strong foundation for building secure systems, the key challenge is that *TEEs are fundamentally incompatible with PM and kernel-bypass networking approaches—in particular, TEEs are neither designed to protect untrusted non-volatile PM, nor the protected region can be accessed via an untrusted DMA connection.*

To overcome this challenge, we design a PM engine that ensures strong security properties for the PM data, using confidential and authenticated PM data structures while preserving crash consistency through a secure logging protocol. We further extend the PM engine to provide remote PM data operations via a secure network stack and a formally verified remote attestation protocol to form an end-to-end system. Our evaluation shows that ANCHOR incurs tolerable overheads while providing strong security properties.

4.1.1 Motivation

Cloud storage and networking infrastructure is going through a dramatic shift to favor the design of modern disaggregated data management systems [67, 98, 152, 156, 323], especially with the recent introduction of the Compute Express Link (CXL) technology [47]. On the storage front, byte-addressable Persistent Memory (PM) aims to bridge the gap between volatile main memory and SSDs [106, 176, 201], providing opportunities for high-volume pools of low-latency non-volatile memory. Similarly, on the networking front, kernel-bypass I/O based on RDMA [86, 155] or DPDK [66] offers superior throughput and low latency [36, 145, 153], and is necessary to efficiently use byte-addressable storage in disaggregated system setups [152, 258, 338], such as for high-performance databases [270, 315], and real-time analytics systems [278]. To leverage these hardware advancements, the research community is actively working on combining PM with kernel-bypass networking to build high-performance storage systems [37, 189, 268, 273].

While the current research is primarily focusing on performance and crash consistency aspects, it is also imperative to address the security threats of these systems when hosted in untrusted cloud environments. In the virtualized cloud infrastructure, where the underlying storage, network, and computing stacks are owned and operated by an untrusted third-party provider, an adversary, such as a malicious system administrator or co-located tenants, can potentially compromise the security properties of both persistent data and storage operations [261, 262]. Prior work has shown that software bugs, configuration errors, and security vulnerabilities pose a real threat to storage systems [52, 72, 80, 171, 262].

In the context of PM-based systems, attackers can tamper with the persistent state and data operations, violating the *confidentiality* and *integrity* properties. They can arbitrarily rollback the PM data into a stale but valid state violating the *freshness* property. Further, PM *crash consistency* mechanisms constitute an added vulnerability vector, where the logs are also susceptible to these security violations. Moreover, they can manipulate the untrusted network, thus being able to remotely compromise data management operations.

To target these threats, our work focuses on: *How can we design a secure PM system for untrusted cloud environments while preserving performance and crash consistency within the realms of the established programming model for byte-addressable storage?*

A plausible direction would be to use Trusted Execution Environments (TEEs) to base a secure PM library. Indeed, it seems promising because TEEs provide a secure memory area where the enclosed code and data are protected by the CPU against all system layers, including the OS/hypervisor [202]. Based on this promise, TEEs are now available in all major commodity CPUs [5, 12, 15, 132, 138, 177], and are offered by major cloud providers [44, 82, 103, 203].

Unfortunately, in our context, TEEs are fundamentally incompatible with both PM and RDMA, as the direct mapping of PM files and RDMA buffers to protected memory is not allowed. In particular, TEEs are primarily designed to protect stateless (volatile) memory regions, and their security properties are not extended on the untrusted PM device, where data remains durable across system reboots/shutdowns. Moreover, TEEs prohibit access to the protected memory region via an untrusted DMA connection. Consequently, TEEs cannot be used out-of-the-box for designing an end-to-end secure PM system.

More specifically, we address the following challenges.

Firstly, the security properties of TEEs do not extend to the untrusted PM storage as TEEs are not designed to protect data at rest. To extend the trust of the TEE to the untrusted PM and preserve the security properties across system reboots/crashes, we design secure data structures that ensure confidentiality, integrity, and freshness of the data residing in PM and their associated operations.

Secondly, while crash consistency is already a major issue for PM systems due to the non-atomic and out-of-order architectural interface between the CPU cache and PM, it is exacerbated in our setting as we need to ensure the consistency of both the data and the security metadata. To this end, we design a "secure crash consistency" mechanism based on secure logging that provides the desired atomicity guarantees.

Thirdly, conventional approaches for network I/O (e.g., kernel-sockets) incur great overheads [51, 145]—especially in the context of TEEs due to switches between the trusted and untrusted world [23, 295]. While direct I/O vastly optimizes network operations, it is incompatible with TEEs as untrusted DMA operations are prohibited in the protected memory [23]. On top of that, ensuring security and crash consistency when accessing PM via RDMA is another major challenge [152]. To address these issues, we design a secure network stack by adapting direct I/O to the contexts of TEEs and PM.

To overcome these challenges, we present ANCHOR, a library for building PM-based applications that provides strong security properties — confidentiality, integrity, authenticity, and freshness. Further, it ensures crash consistency and performance within the realms of the established PM programming model [114]. ANCHOR achieves these design properties by co-designing an end-to-end system leveraging three hardware technologies: high-performance PM storage, hardware-assisted TEEs, and kernel-bypass networking. ANCHOR is designed to target generic PM applications developed with PMDK, requiring minimal adjustments to the APIs used to interact with PM and the network.

Overall, ANCHOR makes the following contributions:

- **Secure data management APIs (§ 4.1.3):** We expose generic APIs for secure data management within the realms of the established PM programming model [114], applicable on byte-addressable storage mediums with similar architectural properties. Our APIs extend the Persistent Memory Development Kit (PMDK) to support secure PM management, transactions, remote attestation, and networking for remote operations. These APIs can be used to develop trusted applications in a single-node setup or even distributed systems.

- **System architecture (§ 4.1.4):** We propose a system architecture where we provide a secure PM management engine that encapsulates confidentiality-preserving and authenticated data structures. It further ensures data integrity at an object level to be able to detect PM data tampering. Our engine extends the trust of TEEs to the data on untrusted PM, where we judiciously partition our data structures between the trusted enclave, the untrusted host memory, and the untrusted PM. Further, ANCHOR's design includes an asynchronous trusted counter interface to guarantee freshness while preserving crash consistency. Lastly, we extend the scope of our PM engine to enable remote operations by designing a TEE-compatible network stack for PM based on kernel-bypass networking, whose authenticity can be verified through our formally proven remote attestation protocol (§ 4.1.6.2).
- **System operations (§ 4.1.5):** We present ANCHOR's operations for building secure PM applications. We highlight the workflow of read and write operations and describe our secure bootstrap and recovery process, based on our formally proven secure logging protocol (§ 4.1.6.2), for ensuring crash consistency and data freshness.

Based on these contributions, we implement a prototype leveraging Intel SGX [138] and integrate with our PM engine based on PMDK [114] and secure network stack based on eRPC [153], a direct I/O networking library. We evaluate ANCHOR with the YCSB benchmark suite [48, 330]. Our evaluation, primarily based on KV stores, which are fundamental building blocks for many cloud-based applications, shows that ANCHOR incurs acceptable overheads considering its strong security properties.

4.1.2 System Model

Threat model. ANCHOR extends the standard SGX threat model [27], as we need to protect the untrusted storage (PM) and network. We aim to protect against an active adversary [64] that can gain full control of the entire system software stack (including the OS/hypervisor) and perform physical attacks (e.g., memory probes). For PM, we strive to guarantee rollback and forking attack resilience where adversaries can arbitrarily restart the system from a stale state or fork system instances. Moreover, we assume that adversaries can control the network stack and tamper with network traffic. However, we do not consider side-channel attacks, denial of service attacks, or memory access pattern attacks [87, 168, 183, 207, 307, 309, 310, 327].

Fault model. ANCHOR mandates crash consistency [49, 174, 269], which implies that data and metadata stored in PM can be recovered to a consistent state after a crash. ANCHOR also requires a protection mechanism against rollback attacks to guarantee data freshness. Additionally, ANCHOR needs to extend these properties to the associated security metadata and the required logs for the case of recovery. Likewise, crash consistency needs to be ensured for untrusted remote PM network operations, where partial writes on PM can lead to an inconsistent state [100, 108, 152].

Programming model. ANCHOR offers a transactional programming model based on PMDK [114]. To maintain consistent object references across reboots, PMDK relies on *persistent pointers*. They are based on a 16 B fat-pointer structure, called *PMEMoid*, storing the *pool_id* and an offset relative to the start of the pool. PMDK provides a function to convert this structure to a native pointer. Additionally, PMDK offers transactional APIs [115] with strict durability, consistency, and atomicity semantics in the *libpmemobj* library. Transactions are realized with the use of *redo* and *undo* logs.

Despite Intel announcing the discontinuation of Intel Optane DC Persistent Memory [279], byte-addressable storage devices are expected to follow a similar programming model, though they may differ in characteristics. Especially the current emergence of the Compute Express Link (CXL) technology [47] enables the development of an ecosystem with non-volatile byte-addressable storage solutions [53, 73, 167], e.g., CXL-capable SSDs [260, 271] or SSDs exposing memory semantics [91]. On top of that, CXL will be combined with Confidential Computing [54, 107, 148] to build end-to-end secure systems. In such setups, ANCHOR can be used to securely manage byte-addressable storage devices due to their compatibility with the existing PM concepts and libraries [53].

4.1.3 Overview

4.1.3.1 System Overview

ANCHOR offers a PM library with the following properties:

- *Security*: ANCHOR ensures the confidentiality, integrity, authenticity, and freshness of the data and storage operations.
- *Crash consistency*: ANCHOR offers a secure crash consistency mechanism for local and remote operations, where it maintains a consistent and secure state in case of failures.
- *Programmability*: ANCHOR offers a transactional programming model and associated secure data management APIs, similar to the established PM programming

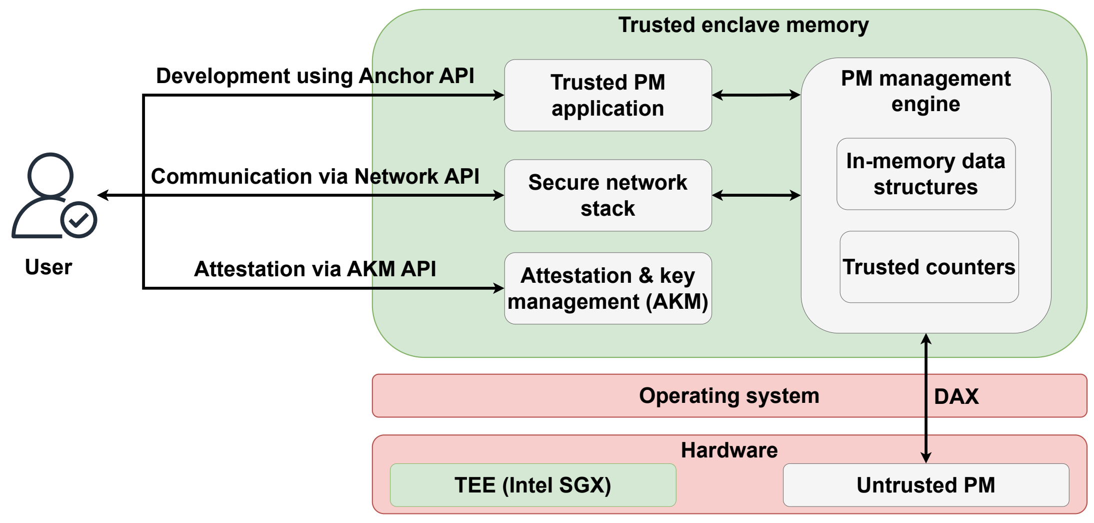


Figure 4.1: System overview *(green regions are trusted and red regions are untrusted)*

model [276], as implemented by PMDK [114].

The key insight of ANCHOR is to maintain confidential and authenticated data structures on PM that are manipulated inside the enclave to extend the TEEs' security properties to PM and networking. Figure 4.1 shows the architecture of ANCHOR. ANCHOR's design adopts the principle of a small trusted computing base (TCB), where it partitions the system into the trusted enclave and the untrusted host memory & PM. In particular, the core control logic of ANCHOR (green regions) resides in the protected enclave, but the actual user data (red regions) resides in the untrusted PM.

The details of ANCHOR's core component, the PM management engine (§ 4.1.4.2) are shown in Figure 4.2. It primarily interfaces with the untrusted PM via DAX and provides a secure memory allocation mechanism and transactional programming model. The PM engine ensures crash consistency and security for the untrusted PM. It further provides freshness guarantees for the PM objects with ANCHOR's trusted counters (§ 4.1.4.5). We also design in-memory data structures (§ 4.1.4.3) consisting of an index and an object cache optimizing the read path. ANCHOR also exposes remote access to PM through a secure network stack (§ 4.1.4.4). Lastly, we offer a remote attestation and key management (AKM) service for clients to ensure trustworthiness and authenticity (§ 4.1.4.6).

At a high level, for read operations, ANCHOR checks the integrity of the object and decrypts it before returning. For write operations, ANCHOR fetches and decrypts the object inside the enclave, updates its content in the protected buffer, and recalculates its integrity signature. For remote accesses, clients communicate through a TLS channel with the ANCHOR controller, which contains an AKM service. AKM instructs the

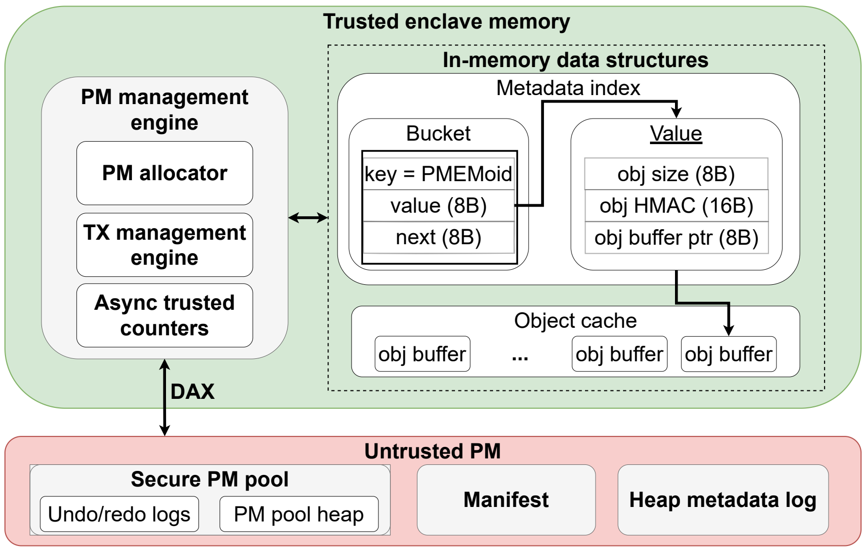


Figure 4.2: Detailed presentation of ANCHOR's PM management engine

enclave to generate a signed measure of its identity, whose authenticity is verified by a trusted third-party entity [102]. After a successful attestation, the client provides its encryption keys and can then access the ANCHOR library and execute queries via the secure network stack.

During bootstrap, ANCHOR scans the manifest to fetch all the object metadata and signatures into the enclave. We use the signatures to prove the integrity of the PM objects. Afterward, the recovery mechanism restores the most recent consistent and secure state of PM based on valid logs. ANCHOR ensures the freshness of all logs by checking the counter values of the entries along with the latest secure trusted counter value (§ 4.1.5.2).

4.1.3.2 ANCHOR System APIs

ANCHOR exposes secure data management APIs (shown in Table 4.1) by adopting and extending the well-established APIs of PMDK [114]. ANCHOR is an embedded library that can also be used to build secure server-side and distributed system applications. Any existing PMDK-based application can be adapted to use the secure ANCHOR API. The shielded execution framework further eases the deployment, as no source code modifications are required.

Pool management APIs. ANCHOR's API provides, similar to the native API, three functions (*create, open, close*) to create, open, and close a secure PM pool. These functions take the paths of the PM-resident log files as extra arguments and perform the setup of the provided storage encryption key.

Pool management APIs	
secure_pool_create()	Creates a secure pool in PM.
secure_pool_open()	Opens a secure pool (if exists).
secure_pool_close()	Closes a secure pool (if exists).
Transactions APIs	
secure_obj_tx_alloc()	Allocates an object (as part of a transaction).
secure_obj_tx_free()	Frees an object (as part of a transaction).
secure_obj_tx_add_range()	Takes a snapshot of an object.
Object management APIs	
secure_obj_root()	Gets or creates the root object.
secure_obj_direct()	Gets a PM object in an enclave buffer.
Secure network APIs	
prepare_req()	Prepares a request to be sent.
enqueue_req()	Submits a message for transmission.
enqueue_resp()	Submits a response to a request.
send()	Sends enqueued messages.
recv()	Receives incoming messages.
register_req_handler()	Registers a request handler.
Attestation API	
attest(measurement)	Attests based on the measurement.

Table 4.1: ANCHOR's secure data management APIs

Transactions API. ANCHOR implements a secure API for transactions that allows arbitrary data sizes to be written to PM with strict security, durability, consistency, and atomicity semantics. Both PMDK and ANCHOR do not provide thread safety for concurrent accesses to PM objects. Developers must employ their own locking mechanisms. For the (de)allocation and objects' snapshots, ANCHOR provides three functions, *secure_obj_tx_alloc*, *secure_obj_tx_free* and *secure_obj_tx_add_range*, that realize transactions through a *redo* log which stores the metadata updates and an *undo* log keeping the initial state of the transaction's write set for the case of a crash.

The allocation function *secure_obj_tx_alloc* returns an object id (*PMEMoid*). Upon updates, similarly to PMDK, users have to explicitly snapshot the modified PM objects in the undo log. Snapshots ensure that the modified object is also added to the ongoing transaction write set (with *secure_obj_tx_add_range*). Afterward, the application can manipulate the object's buffer inside a transaction, and the changes will be persisted during the commit phase.

Object management APIs. ANCHOR's API for PM-object management offers security while preserving similar semantics with PMDK. PMDK, and consequently ANCHOR, imposes one requirement to avoid PM leakage: all objects are reachable through some path from a root object. ANCHOR exports the *secure_obj_root* function that creates/gets the root object. *secure_obj_direct* function accepts a PMEMoid as an argument and returns a pointer to a secure volatile buffer with the decrypted data.

Secure network APIs. To form an end-to-end setup, ANCHOR integrates userspace networking technologies (e.g., RDMA) with PM and SGX that enable secure remote access to PM via an established RPC API [153]. Our library offers asynchronous network operations—we provide three core functions, *prepare_req*, *enqueue_req*, and *enqueue_resp*, for requests and responses. These functions do not send the message over the network, but users need to execute the *send* and *recv* functions to burst and drain messages from the transmission and reception network queues. For each request, the remote application executes a request handler that is registered on initialization (*register_req_handler*). ANCHOR further encrypts the network messages, whose integrity can be verified, and incorporates counter values in the message headers to ensure freshness.

Attestation API. ANCHOR provides an attestation API that allows applications to verify their trustworthiness to remote clients. Particularly, we provide the *attest* function that takes as arguments the IP of a trusted third-party service, Intel Attestation Service (IAS) [102] IP for ANCHOR, and a generated enclave measurement of the code. Then, this service verifies that both the enclave signer and measurement are in the expected state and replies accordingly.

4.1.3.3 Design Challenges and Key Ideas

#1 Untrusted persistent memory.

Problem: TEEs are designed to protect only the volatile enclave memory — PM regions, that are directly mapped to into the application's address space, are not subject to the memory verification procedures that TEEs provide. Thus, the security properties do not naturally extend to the untrusted PM, where we need to ensure the security for stateful operations across system reboots or crashes. Additionally, while applications in TEEs can read and write data to and from conventional block devices, they often employ prohibitively expensive, in the context of TEEs, I/O mechanisms (e.g., read/write syscalls) and provide crash-consistency and data persistence in larger granularities (e.g., 4K blocks) compared to PM.

Approach: ANCHOR offers security beyond the protected enclave and extends the trust of TEEs to the untrusted PM. We design data structures that ensure confidentiality, integrity and freshness. ANCHOR achieves this by *(i)* encrypting and persisting data and metadata on PM on an arbitrarily-sized object-level granularity, and *(ii)* extending the

PMDK's metadata structure's layout with an append-only log, the *manifest*, for security metadata. ANCHOR's manifest maintains the hash values of all PM objects. Further, it ensures rollback protection across restarts by assigning a deterministic unique counter value to each entry. In particular, if the PM data (e.g., objects, manifest) has been tampered with, it will either be detected at runtime, during the integrity checks, or at the upcoming boot phase if any manifest entry integrity check fails or the counter does not reach the expected latest trusted value. Thus, ANCHOR effectively extends TEE properties to PM. Moreover, ANCHOR realizes all operations as transactions. Uncommitted updates are buffered in memory; they are persisted during the commit phase. Our approach combines security with performance by the following key insight: any update should be made persistent as long as ANCHOR can ensure its confidentiality, integrity, and freshness. Therefore, ANCHOR defers writes to PM until their freshness property is secured.

#2 Secure crash consistency.

Problem: PM guarantees atomicity only for aligned 8-byte stores. While libmemobj [115] implements software transactions for atomic writes of arbitrary data sizes, ANCHOR needs to keep its security metadata crash consistent. In particular, we need to ensure crash consistency and security for all data and metadata. We refer to this property as *secure crash consistency*: any non-trusted PM content will be discarded in favor of the latest trusted and correct content.

Approach: ANCHOR offers secure crash consistency by extending the transaction logic and providing a secure logging protocol. Firstly, a transaction needs to snapshot the latest secure state of a modified object to be able to revert it if needed. Secondly, ANCHOR needs to ensure the freshness, integrity, and confidentiality of the logs that reside in the untrusted PM. This is achieved by encrypting the payload of the log entries and enhancing them with security metadata (i.e., trusted counters, integrity signatures). We design our secure crash consistency mechanism with respect to freshness based on asynchronous counters, originally proposed in Speicher [25]. To prohibit attackers from arbitrarily deleting redo/undo logs or replacing them with obsolete yet correct logs, our transactions log their start, commit, and end to the manifest. Lastly, since we only commit stable transactions, viz. transactions that own rollback-protected logs, ANCHOR can replay the secure logs and bring the PM to the correct trusted state across reboots/restarts. At recovery, ANCHOR will roll back any aborted transaction or redo any marked-as-committed transaction that got interrupted.

#3 Fast network I/O.

Problem: PM's low access latency shifts the bottleneck from storage to network I/O. Traditional enclave I/O issues such as enclave transitions and asynchronous syscalls

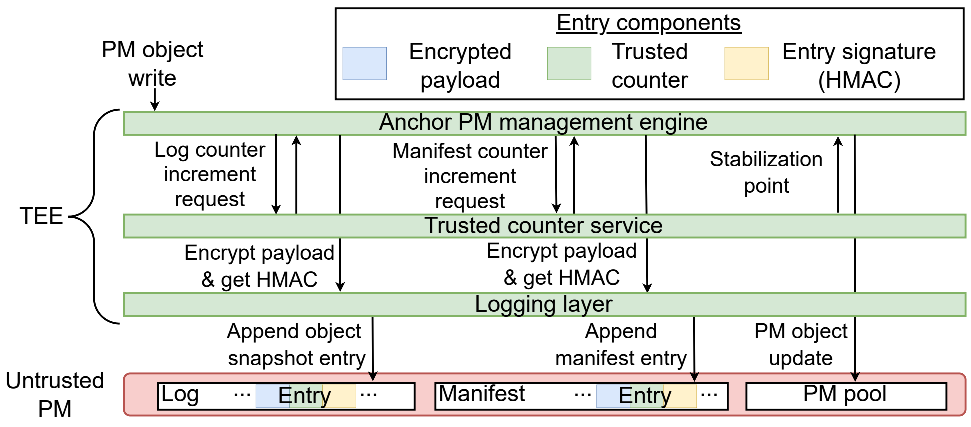


Figure 4.3: ANCHOR's logging protocol & log structure

execution [16] further increase the latency compared to conventional approaches (e.g., sockets) that are already the bottleneck in networking systems [51, 89, 145]. While direct I/O networking solutions such as RDMA are prominently deployed in data centers to overcome the I/O bottlenecks [152], they are not directly applicable to ANCHOR due to two core challenges: *(i)* RDMA buffers cannot be allocated inside the enclave memory, as this would violate the security guarantees of SGX [23] and *(ii)* RDMA operations might lead to inconsistent PM state.

Approach: ANCHOR overcomes these limitations by integrating userspace networking [153], PM, and TEEs to optimize the throughput and provide remote access to PM. In particular, ANCHOR designs a secure network stack that preserves the crash consistency property for remote PM operations, overcomes the I/O bottlenecks of TEEs, and is compatible with deployments of RDMA technology in the cloud. Additionally, it introduces a secure message format to ensure the security properties of the network traffic. ANCHOR further tackles the challenge that untrusted resources/memory cannot be mapped into the enclave. Our network stack is placed inside the enclave but maps the DMA and message buffers into the untrusted host memory, which is accessible by the enclave. This design optimizes the limited EPC memory usage. ANCHOR overcomes the second challenge by executing remote queries as transactions. We rely on ANCHOR's crash consistency mechanism to ensure crash-consistent remote operations. Note that ANCHOR currently supports transactional PM updates on a single server node.

4.1.4 Design and Implementation

4.1.4.1 Persistent Data Structures

ANCHOR stores data on the untrusted PM using three persistent structures, which we explain first.

Secure PM pool. The secure PM pool is where the actual data resides. Identically to PMDK's pool structure, it is composed of: *(i)* a pool header, *(ii)* an area for transactional logs, *(iii)* heap metadata, and *(iv)* the persistent heap, where the objects are stored. The header contains metadata of the pool (e.g., size) and heap metadata that is used for managing (de)allocation in the persistent heap.

Manifest. Manifest is an append-only persistent secure log that keeps the security metadata of all objects in a pool. For each object update, a new entry is appended to the manifest. Each entry contains an encrypted payload, a trusted counter, and a cryptographic hash over both (Figure 4.3). With this format, ANCHOR is able to argue about Manifest's confidentiality, integrity, and freshness on every startup. The payload consists of the object's hash, its PMEMoid, and its size. Manifest entries allow ANCHOR to ensure the integrity (with the signature) and the freshness (with the counter) of all objects.

Secure undo/redo logs. Undo/redo logs ensure crash consistency and atomicity of data operations. An undo log entry stores an object snapshot before it is modified, while redo logs track pool/heap metadata modifications. ANCHOR secures its logs in a similar fashion as the manifest. For each log, we create a unique trusted counter. Each undo log entry consists of the encrypted payload, a trusted counter value, and a hash over both (Figure 4.3), similarly to the Manifest. The redo log entries do not require a hash as they are stored in bulk, and a hash over the whole redo log is placed in its header along with the total log size.

4.1.4.2 PM Management Engine

The PM management engine consists of the PM allocator and the transactions management engine. It ensures the crash consistency and the security properties of the persistent data structures. The PM management engine stores the PM data encrypted, guaranteeing confidentiality. Additionally, for PM data encryption, it uses the *A*ES-GCM-128 algorithm of OpenSSL [225] that directly provides cryptographic signatures, which can be used for integrity checks. The encryption library is entirely placed inside the enclave.

PM allocator. ANCHOR's PM allocator offers secure, transparent, and dynamic PM memory management. The allocator manages the secure pool's heap to (de)allocate PM objects. It relies on redo logs to avoid metadata corruption. ANCHOR logs heap metadata modifications that reflect the status change of a block (occupied/free). The allocator frequently accesses the heap metadata. Therefore, we maintain their core part (e.g., PM block headers) in the enclave memory during runtime. Additionally, the allocator's volatile data structures (e.g., buckets) remain intact and reside in the protected memory.

TX management engine. The TX management engine implements transactions that, in turn, ensure security, data atomicity, and crash consistency for the modified objects. Particularly, we offer ACD (**A**tomicity, Crash **C**onsistency, and **D**urability) semantics for transactions; however, similar to PMDK, we do not offer any isolation guarantees. ANCHOR ensures these properties by tracking the modifications on the pool/heap metadata and snapshotting the modified objects in its PM logs using a secure logging protocol. In contrast to the native PMDK, ANCHOR needs to further consider rollback protection as part of the crash consistency mechanism. Towards this direction, we keep the modified objects of an uncommitted transaction in the enclave buffers which are only flushed to PM at the commit phase. This practice is mandatory, as ANCHOR has to ensure that the log entries of the snapshotted objects are persisted and rollback-protected through their counter so that, in case of a crash, the previous state of the objects can be securely restored. These objects are tracked with the use of a transaction-local list holding their offsets (§ 4.1.5.1). In this way, if the logs are detected to be unstable during recovery, we are sure that the interrupted transaction never performed actual PM updates, as they are only applied after the stabilization of the respective logs. To support *concurrent transactions*, similarly to PMDK, ANCHOR reserves a space in the secure PM pool that is split into lanes. Each lane is assigned to a distinct thread to store its respective transaction logs.

4.1.4.3 In-memory Data Structures

To accelerate the operations of ANCHOR, we maintain security metadata and an object cache in the enclave memory.

Metadata index. ANCHOR logs the objects' integrity signatures along with the trusted counter in the manifest. Consequently, an object's access would require *(i)* to prove the manifest's freshness and integrity and *(ii)* iterate the entire manifest to locate the most recent entry for that object. We opt for optimizing the data path by introducing an in-memory hashmap index, that maintains only the necessary metadata, aiming for better EPC utilization. Precisely, the index stores all integrity signatures (16B) and the object sizes (8B) having as a key the object's PMEMoid (16B). As a result, object reads bypass the manifest (§ 4.1.5.1). The index is trusted at run-time since it resides in the enclave; we populate the metadata index entries during a successfully attested bootstrap (§ 4.1.5.2).

Object cache. To further accelerate the read path, we expand the scope of the metadata index to an object cache that buffers recently accessed objects in the enclave. This

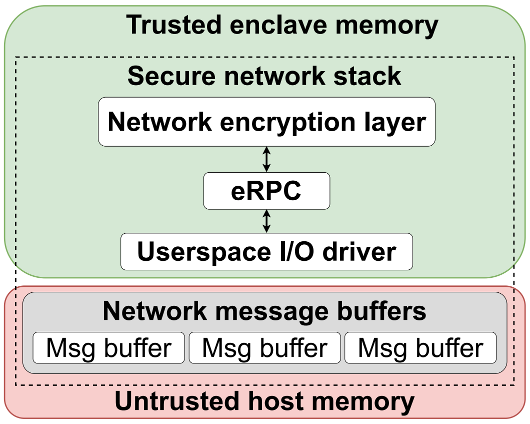


Figure 4.4: Overview of ANCHOR's network stack

eliminates the decryption calls and access to the PM. A reference of each buffered object is stored in its respective entry in the metadata index. Additionally, to eliminate repeated volatile object buffer allocations and control the EPC usage, ANCHOR enforces an epoch-based data caching mechanism [179]. Each metadata index entry is assigned an incremental value (*epoch*) when it is accessed. We define a configurable memory limit that the object cache can occupy. When it is reached, a background thread reclaims the memory of cached objects after making sure, based on their epoch, that they do not belong to an ongoing transaction.

4.1.4.4 Network Stack

Since networking is an essential component of disaggregated cloud systems, ANCHOR includes a network stack that integrates kernel-bypass networking with TEEs to securely access and manage PM data. Our design is optimized for performance; rather than adopting the costly kernel-based networking, ANCHOR bypasses the kernel [66, 153] and avoids performance-expensive enclave switches.

In particular, ANCHOR exposes asynchronous, secure RPCs based on two-sided RDMA. ANCHOR RPCs involve the CPU in order to verify the integrity, authenticity, and freshness of the network traffic. The network stack code (e.g., RPC-library, drivers) resides in the enclave while the network data (e.g., messages' buffers, NIC queues) is in the untrusted host memory, as shown in Figure 4.4. ANCHOR stores the messages encrypted in DMA-capable buffers in the untrusted host memory satisfying two requirements (§ 4.1.3.3, #3): *(i)* DMA-ed memory cannot be inside the enclave and *(ii)* EPC usage is optimized. We integrate eRPC [153], with DPDK [66] as a transport layer, along with the userspace drivers into SCONE to shield the execution of the network operations. Our network stack reserves unprotected—accessible by the NIC—2 MiB hugepages for the DMA-ed memory. To achieve that, we extend the eRPC allocator to open shared memory files in the *hugetlbfs* virtual filesystem and pass the file descriptors to the *mmap*.

ANCHOR's network library further extends the trust to the untrusted network through a secure messaging layer. Precisely, we construct a secure message format that is comprised of three parts: *(i)* the encrypted payload, *(ii)* the initialization vector (IV), and *(iii)* a hash value. For the encrytpion of the messages, ANCHOR uses the *A*ES-GCM-128 algorithm of OpenSSL [225]. In the payload, ANCHOR reserves the first 8 B for a sequence number. The sequence number is unique for each client and is deterministically increased for each operation, exposing *at-most once* execution semantics and guaranteeing message freshness. In this way, our network stack protects against replay attacks on the network.

On the client side, ANCHOR maintains a queue for the message transmission. The queue size can be tuned depending on the system's requirements to optimize the network latency and throughput or maintain a balance between them. ANCHOR's network stack also stores the sequence number of each pending request in the queue, which is then used to check whether the sequence number of a response matches the one from the request.

On the server side, ANCHOR processes clients' requests. While ANCHOR's network library considers lossy networks and a malicious attacker that can tamper with the network traffic, ANCHOR provides reliability based on the message sequence numbers. Precisely, ANCHOR's server accepts sequence numbers in a fixed, configurable range based on the previously received sequence numbers of the client requests. While we do not provide ordering for the packets inside the range, ANCHOR can still detect missing packets. For total ordering, this range can be configured to be 1 at the cost of performance. On top of that, ANCHOR's network stack verifies the uniqueness of each sequence number. After this verification, the server processes the request and sends a response to the client. Note that the PM security and crash consistency properties are ensured via the server's PM management engine.

4.1.4.5 Asynchronous Trusted Counters

ANCHOR uses trusted counters to ensure rollback resilience for the PM logs. We design ANCHOR based on an asynchronous monotonic counter (AMC) interface, originally proposed by Speicher [25], that allows fast increments while overcoming the limitations of SGX counters. ANCHOR creates one asynchronous counter for each log and persists their state in a file. To protect this file from rollback attacks, the AMC uses a hardware-trusted monotonic counter—in our case, Intel SGX monotonic counter [266]. While

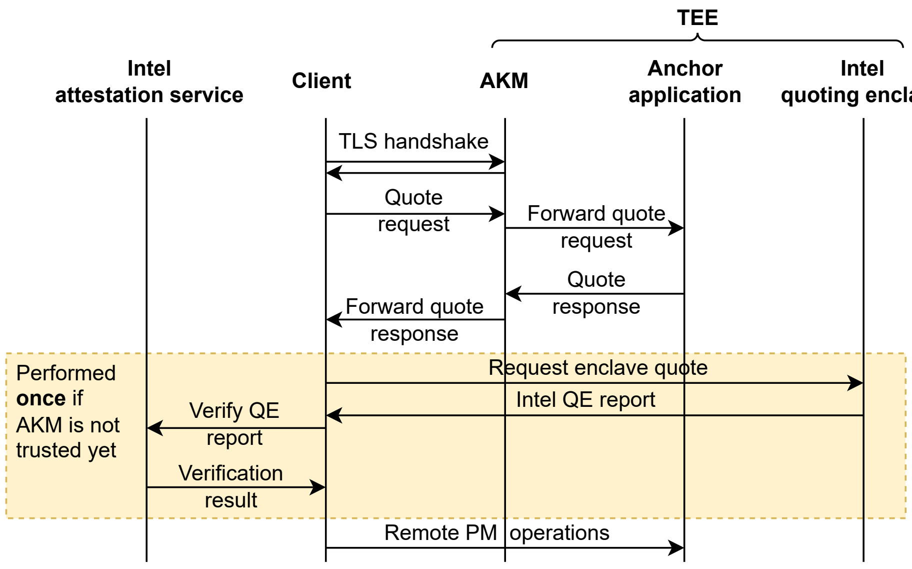


Figure 4.5: ANCHOR's attestation protocol

the asynchronous counter offers fast increments, the freshness can only be ensured when the counters' values are secured in the file along with the SGX counter. The time when an asynchronous counter value is written to the file with the SGX counter is called *stabilization point* and occurs when the SGX counter is successfully increased after an increment request ($60 - 250\,\mathrm{ms}$). ANCHOR's recovery mechanism only trusts entries with stable counter values (§ 4.1.5.2).

Although ANCHOR is not bound to a specific trusted counter, we build ANCHOR using our AMC in a single-node setting. While ANCHOR optimizes throughput by batching operations before an SGX counter increment; unfortunately, the counter stabilization delays incur an inevitably increased latency to provide rollback protection. System designers might want to adapt ANCHOR to leverage lower-latency (remote) trusted counters (e.g., ROTE [194]) that implement a trusted counter as a service in distributed settings [78] to reduce the stabilization time and ensure longevity.

4.1.4.6 Attestation and Key Management (AKM)

Remote clients need to establish trust with ANCHOR's applications. Further, ANCHOR needs to securely distribute keys and configuration to clients. ANCHOR's AKM provides these services by extending Intel Attestation service [102] and integrates a key management system, which provisions clients with keys (e.g., for communication).

Attestation protocol. Figure 4.5 demonstrates ANCHOR's attestation protocol. More precisely, clients connect to the AKM service via a secure TLS channel. Following, they

Algorithm 1: Read operation

```
Input  : Persistent object id
Output: Persistent object data buffer
read(oid)
begin
    /* object entry lookup in the metadata index */
    epc_entry ← index_lookup(oid);
    if epc_entry == NULL then
        //object not found in EPC index
        return object_not_found;
    else if epc_entry.cached_obj ≠ NULL then
        //object already in EPC cache
        return epc_entry.cached_obj;
    else
        //fetch and decrypt the object data
        obj_buffer ← decrypt_and_verify(oid, epc_entry.hash);
        /* update the in-memory index entry with the buffer */
        epc_entry.cached_obj ← obj_buffer;
        return obj_buffer;
    end
end
/* object entry lookup in the EPC metadata index */
index_lookup(oid)
begin
    key ← hash_func(oid);
    epc_entry ← hashmap_lookup(key);
    return epc_entry;
end
```

request to attest the ANCHOR application. If the AKM service is not trusted yet by the client, the Intel attestation process is invoked to establish trust between the client and the AKM service. Then, the AKM service, the *verifier*, attests the ANCHOR application by requesting a quote. The enclave requests a report from SGX hardware and transmits it to the Intel Quoting Enclave (QE), which verifies, signs, and sends back the report. The ANCHOR application forwards it to the verifier. This quote can be verified using the Intel verification service [140]. After a successful attestation, AKM generates ANCHOR's application keys and distributes them to the client for secure network communication. Note that ANCHOR currently lacks explicit access control features but can be enhanced to include them by incorporating key separation mechanisms.

4.1.5 System operations

4.1.5.1 Transactions

Read path. Read requests involve two steps (Algorithm 1): *(i)* locate the object in the PM or the object cache and *(ii)* verify its security properties. ANCHOR first looks up the object in the in-memory object cache. If the object is in the cache, ANCHOR does not need to perform any additional step; the object cache is already secured by the

Algorithm 2: Write operation

```
Input  : Persistent object id & New object data
Output: Sucess/Failure & Stabilisation time
write(oid, new_obj_data)
begin
    //snapshot the object and add it to TX write set
    snapshot(object_id);
    obj_buffer ← read(oid);
    //update the object data
    obj_buffer ← store(new_obj_data);
    //defer PM writes on commit
    return success;
end
alloc(size)
begin
    mem_block ← find_block(size); //find the appropriate memory block
    obj_oid ← extract_oid_from_block(mem_block);
    //add the redo log entry for the occupied block
    add_redo_entry();
    return obj_oid; //return the new object id
end
On commit:
begin
    persist_redo_log();
    append_manifest(modified_object_entries);
    append_manifest(TX_COMMIT);
    //stabilisation point
    foreach object_id ∈ write_set do
        update_epc_index(hash(obj_buffer));//update the new object hash
        store(encrypt(obj_buffer)); //store new object data in PM
    end
    apply_redo_log();
    append_manifest(TX_FINISH);
end
```

enclave. Otherwise, ANCHOR fetches the PM object inside the enclave, checks if the object's calculated signature matches the protected signature in the metadata index, and decrypts it. Note that all security metadata is populated in the index during system bootstrap (§ 4.1.5.2) and has its integrity and freshness proved.

Commit protocol. ANCHOR implements a secure commit protocol to ensure crash consistency and rollback protection. ANCHOR first ensures that the logs are persistent and rollback protected, and then, it updates the PM content. ANCHOR's logging process is demonstrated in Figure 4.3. Precisely, the object snapshots are added to the undo log during a transaction. On commit, ANCHOR persists the heap metadata updates to the redo log. It further appends the objects' new signatures to the manifest and a mark-as-committed entry for the transaction. Note that the log and manifest entries are stored encrypted, and each of them contains a unique trusted counter value. Our protocol defers PM updates until all logs are stable. Then, ANCHOR updates the PM as its recovery mechanism can ensure crash consistency based on the secure logs.

Write path. Since new object creations and existing object updates modify the security metadata (signatures, etc.), ANCHOR realizes all write operations (Algorithm 2) as transactions to guarantee security and crash consistency. Users, similarly to the PMDK, need to explicitly take snapshots of a transaction's write set which are persisted to the secure undo log. Each undo log entry receives its own unique trusted counter value, as shown in Figure 4.3. After the snapshot, ANCHOR searches for the object based on its PMEMoid, equivalently to a read operation. ANCHOR updates the object in the object cache, but it does not modify it in place in the PM. This is our core difference with PMDK; we keep uncommitted updates in the enclave buffers that are written to PM through ANCHOR's secure commit protocol, after the stabilization of the log entries.

Memory operations. ANCHOR relies on the PM allocator of PMDK. PM operations result in modifications to heap metadata; consequently, ANCHOR realizes them as writes. Memory operations should also be crash-consistent to avoid memory corruption and leakage. In contrast to write operations, alloc/free operations do not require PM data snapshots. However, memory operations pass through the same secure commit protocol. All the heap metadata updates are only applied based on the redo log entries at the commit phase.

4.1.5.2 System Bootstrap and Recovery

System bootstrap and recovery (Algorithm 3) bring ANCHOR to a consistent state. After the attestation via the AKM service, ANCHOR reads the logs (manifest, undo/redo logs) and restores information about the objects' signatures and interrupted transactions. The goal of this process is to *(i)* verify the security properties of the logs, *(ii)* retrieve the signatures for integrity checks, and *(iii)* commit/rollback any uncompleted transactions to restore PM data consistency.

In ANCHOR, logs are scanned sequentially. Each log entry is integrity checked by a hash, and its freshness is ensured by the log's trusted counter. The counter is incremented deterministically. ANCHOR uses its value to check if all entries are present. Thus, in case of a rollback attack on the manifest or the transaction logs, ANCHOR is able to detect if entries are missing. Entries with counter values higher than the stored stable, trusted counter are ignored. On a successful log verification, ANCHOR is assured that all the entries are valid and originated from an authentic ANCHOR instance.

```
Algorithm 3: System recovery and bootstrap
Input  : Persistent memory pool & Manifest
Output: Consistent PM pool & In memory metadata index
pool_open(path, Manifest)
verify_log(Manifest, manifest_type);
recover(pool_lanes); //trigger the recovery process
//verify pool header
pool_header ← read(pool_header_oid);
//verify PM heap headers
heap_headers ← read(heap_headers_oids);
return success;
recover(pool_lanes)
foreach lane ∈ lanes_with_unfinished_tx do
    redo_entry_list ← verifyLog(redo_log, redo_type);
    if redo_in_progress then
        //apply verified EPC-residing redo log entries
        apply(redo_entry_list);
    else
        undo_entry_list ← verifyLog(undo_log, undo_type);
        //apply verified EPC-residing undo log entries
        apply(undo_entry_list);
    end
end
return success
verifyLog(log, log_type)
/* verify the log entries and fetch the content in EPC */
begin
    counter ← log.firstCounter;
    // if it's a redo/undo log keep the pending updates in a list
    if log_type ≠ manifest_log then  entry_list ← init_entry_list() ;
    foreach entry ∈ log do
        entry ← decrypt_and_verify(entry);
        if counter ≠ entry.counter then
            return Counter does not match;
        end
        if counter > entry.counter then  break ;
        if log_type ≠ manifest_log then
            entry_list.add(entry);
        else
            if entry == tx_commit_entry then
                // mark the transaction as commited but non-finished till the
                 tx_finish_entry is read
                mark_tx_commited();
                mark_tx_lane_unfinished();
            else if entry == tx_finish_entry then
                // mark the transaction as finished
                mark_tx_lane_finished();
            end
        end
        inc(counter);
    end
    if counter ≠ log.trustedCounter then  return Counter does not match; ;
    if log_type ≠ manifest_log then  return entry_list; ;
    return success;
end
```

ANCHOR first scans the manifest log and populates all objects' signatures in the metadata index. During this process, ANCHOR retrieves information about interrupted transactions. Transactions that were not committed are ignored since they have not modified PM data. However, there might be transactions that are marked as committed in the manifest, but the commit protocol is not completed. ANCHOR examines whether the transaction was stopped during the redo log application. In this case, ANCHOR applies the redo log since all the PM objects were successfully persisted. Otherwise, the undo log is replayed. Lastly, after the integrity checks on the pool header, ANCHOR opens the pool. Note that ANCHOR constructs its metadata index with the latest signatures for its objects through its bootstrap process. In that way, if an object is rolled back to a previous valid state, any upcoming operation on it will report this violation.

4.1.5.3 Manifest Compaction

When the manifest reaches a configurable threshold, ANCHOR activates a compaction mechanism that copies the latest security metadata to a new manifest. Since ANCHOR keeps all objects' metadata in memory, the compaction requires copying the content of the in-memory index to the new manifest. To this end, we design a *background* compaction mechanism.

While compaction is in progress, ANCHOR needs to ensure recoverability. Therefore, during compaction, ANCHOR keeps updating the old manifest while constructing the new one. We use a separate trusted counter for the new manifest to preserve the deterministic increment for both manifests. Note that the new manifest is written in the background by a dedicated thread without implications on other system operations. If the system crashes during a compaction, the application will recover as the old manifest still contains all the latest entries.

4.1.5.4 Network Operations

During the initialization, the communication participants register the callback functions for the supported operations. To send a message (Algorithm 4), ANCHOR initially gets a pre-allocated buffer and constructs the message header (§ 4.1.4.4) and its payload. ANCHOR encrypts the message and places it in a buffer residing in the untrusted host memory (i.e., outside the SGX-protected memory region) with its authentication tag. Then, the message is ready to be sent. This allows ANCHOR's network stack to handle fast transmission directly from the untrusted buffer. An ANCHOR server polls for new connections and incoming requests. Upon the arrival of a request (Algorithm 5), the receiver decrypts the message and verifies its integrity and the sequence number. The content of the message is stored in trusted enclave buffers. The receiver then executes

Algorithm 4: Network send operation

```
Input  : ID of the desired operation, Callback for arrival of the response, Message content
         (arguments and their length)
Output: Message with ciphertext buffer in untrusted memory
create_message(operation_id, callback)
begin
    //Fill in sequence number and operation ID and
    //add callback for arrival of the response
    Message message(current_seq, operation_id, callback);
    //Get a pre-allocated message buffer
    message.buf ← buffer_by_seq(current_seq);
    //The response sequence number is current_seq + 1
    //That is why we increment it by 2
    current_seq ← current_seq + 2;
    //Current ciphertext position inside the buffer
    message.ciphertext_pos ← message.buf;
    //Add Initialization Vector for encryption
    message.ciphertext_pos ← generate_IV();
    //Encrypt the header (sequence number + ID)
    ciphertext_pos ← encrypt(message.header);
    return message;
end
add_arg(message, arg_len, arg)
begin
    //The argument and its length are encrypted (arg_len + 4 B)
    if $arg\_len + 4 > remaining\_size(message)$ then
        return False; //buffer not big enough
    message.ciphertext_pos ← encrypt(arg, arg_len);
    return True;
end
enqueue_req(message)
begin
    //Write the Authentication tag
    message.ciphertext_pos_tag ← write_tag(arg);
    //Enqueue the request in eRPC
    eRPC_enqueue_request(message.buf);
end
```

the registered callback for the message type, and returns a response, with the previously explained process, to the sender. eRPC is responsible for the UDP headers, while DPDK constructs transport layer headers.

4.1.6 Security analysis

ANCHOR extends the standard SGX threat model, as described in § 4.1.2, i.e., TEE correctly implements the secure enclave abstraction. To ensure the security principles of ANCHOR, we have to *(i)* make sure that the ANCHOR code running inside the enclave is *memory safe* [289] while preserving crash consistency, and *(ii)* prove the security properties of ANCHOR's protocols that are implemented beyond the SGX trust boundaries. To this end, we leverage dynamic analysis tools for security analysis, i.e., AddressSani-

Algorithm 5: Network receive operation

```
Input  : Buffer with encrypted message
Output: Decrypted message object
decrypt_message(buf)
begin
    //Message must contain an IV, a header and a MAC
    assert(buf.len >= min_msg_len);
    Message message(buf);
    ciphertext_pos ← buf + iv_size;
    message.header ← decrypt(ciphertext_pos, sizeof(message.header));
    while remaining_size(message) > 0 do
        //There must be space for the argument length
        if remaining_size(message) < 4 then return error ;
        arg_len ← decrypt(message.ciphertext_pos, 4);
        if arg_len > remaining_size(message) then return error ;
        arg ← decrypt(message.ciphertext_pos, arg_len);
        append(message.args, pair(arg_len, arg));
    end
    //Check the authentication tag
    if ¬ verify_decryption(message.ciphertext_pos) then return error ;
    //Check the sequence number
    if message.header.seq - expected_seq < 0 then
        assert(is_fresh(message.header.seq));
    else
        assert(seq_threshold > expected_sec - message.header.seq);
    end
    return message;
end
```

tizer [264] and Valgrind [240], to verify the memory safety of ANCHOR's enclave code and ANCHOR's crash consistency property (§ 4.1.6.1).

Importantly, we further formally prove the security principles of ANCHOR's remote attestation and secure logging protocol using the Tamarin prover [196] (§ 4.1.6.2). For our proofs, we rely on SGX to ensure the integrity and confidentiality of the enclaves. Additionally, we require that the proper software attestation of the enclaves guarantees authenticity. In particular, this means that we assume that the SGX Quoting Enclave works correctly. Note that the models of our protocols allowed Tamarin to efficiently exhaust the search space and terminate without the need for oracles.

4.1.6.1 Dynamic Analysis for Security Issues

Memory safety using AddressSanitizer. Memory safety bugs and memory leaks are common causes of security vulnerabilities. Therefore, we need to verify that ANCHOR does not include such memory errors. To this end, we compile the native ANCHOR with AddressSanitizer (ASan) [264], a state-of-the-art tool for detecting memory safety issues. We conduct experiments to pinpoint memory safety bugs in ANCHOR. We use a set of persistent indices shipped with PMDK. During the execution of our experiments, ASan reports neither spatial (e.g., buffer over-/underflows) nor temporal (e.g., use-after-free) memory safety violations. Additionally, ASan does not detect *any memory leaks*. Thus, we verify that ANCHOR's components do not expose any security vulnerabilities through memory safety bugs or memory leaks.

Crash consistency using Valgrind's pmemcheck. We use the Valgrind-based pmemcheck [240] and pmreorder [244] to verify ANCHOR's crash consistency property. First, we conduct experiments with the native ANCHOR using workloads of 10000 operations with the persistent indices of PMDK, to keep the runtime reasonable due to Valgrind's instrumentation. Throughout our experiments, pmemcheck did not report any issues. Additionally, we port one PMDK recovery test [222] to ANCHOR. All the test cases [252] passed without indicating any crash consistency violation. Lastly, we adapt a pmreorder test of PMDK [26] and the core pmreorder example of the PM book [263] to ANCHOR's API. Our tests did not report any reordering or consistency issue, which, along with the object recovery test, our recovery microbenchmark, and the pmemcheck tests, highlight that ANCHOR preserves the crash consistency property.

4.1.6.2 Formal Verification of Security Protocols

Remote attestation protocol. We model ANCHOR's attestation protocol (Figure 4.5), described in § 4.1.4.6, using Tamarin [196]. In our model, all messages are handled as atomic and we consider that the cryptographic functions are perfect without side effects. Further, we build on the formally proven TLS handshake [290] to establish an authentic session between agents that includes a secret symmetric key for further communication. Lastly, IAS approves only quotation engine reports running on genuine Intel SGX hardware.

In our model, the protocol states are modeled as a multiset. The state transitions are represented as multiset rewriting rules. Our model is checked for correctness through a set of control lemmas. They ensure that certain valid states are reachable. Our model is used to prove ANCHOR's desired security properties. Precisely, an attestation lemma holds if and only if: once a client trusts an ANCHOR application, this application is in a valid, expected state.

Tamarin verifies the specified lemmas, by *(i)* finding at least one valid trace (series of state transitions) for the required states and *(ii)* showing that there exists no trace leading to invalid states. In our model, Tamarin found at least one trace for every control lemma and proved that there is no trace to any state where our lemmas are violated. Thus, our attestation lemmas hold for our model.

Secure logging protocol. ANCHOR's secure logging protocol is modeled in Tamarin based on Figure 4.3. ANCHOR's logs share a unified format. Each entry contains an encrypted payload, a trusted counter value, and an integrity signature.

The confidentiality and integrity of the entries is ensured through the encryption and the cryptographic hash that is checked on the decryption of the entries. Our model is used to prove that if a log is successfully verified during bootstrap (§ 4.1.5.2), *(i)* there is no stable entry missing, and *(ii)* all entries are valid and from a genuine source. Note that the version of Tamarin that we used for the proofs does not contain direct support for counters. We worked around this limitation by modeling our counters using temporal variables associated with the action facts. Precisely, our proof uses the timestamp of the action facts to model the counter values since ANCHOR's trusted counters are unique and in a sequential, increasing order.

Tamarin identified at least one trace for our aforementioned lemmas and indicated that there is no set of transitions leading to a violating state. In this way, it formally proves the security principles of our secure logging protocol.

4.1.7 Evaluation

4.1.7.1 Experimental Setup

We conduct our experiments on a server machine with SGXv.1, equipped with Intel(R) Core(TM) i9-9900K CPU with 8 cores (16 threads), 64 GiB dual-channel memory and 32 KiB (L1D, L1I), 256 KiB (L2) and 16 MiB (L3) caches. At the time of this research, commercially available hardware did not support simultaneous SGX and PM [134, 135], necessitating emulation of PM with DRAM. Consistent with prior studies [150, 159, 214, 313], we inject write latency on cache line flushes. For the network experiments, the nodes are equipped with an Intel Corporation Ethernet Controller XL710 network card and are connected over a 40GbE QSFP+ network switch.

Although newer servers provide more compute power and larger enclave memory capacity, the foundational characteristics of memory and network interactions remain consistent: the latency and durability overheads of PM, enclave memory constraints (e.g., EPC paging), and network bandwidth bottlenecks remain relevant.

For our evaluation, we use two classes of workloads: *(i)* 5 well-known persistent indices (`ctree`, `btree`, `rbtree`, `rtree` and `hashmap`) to showcase how ANCHOR performs in real-life workloads and *(ii)* microbenchmarks to perform a sensitivity study on different operations. We benchmark the indices using different YCSB workloads (Zipfian distribution) [48, 330] with varying R/W ratios. For our client-server evaluation, we use iPerf [141] and YCSB.

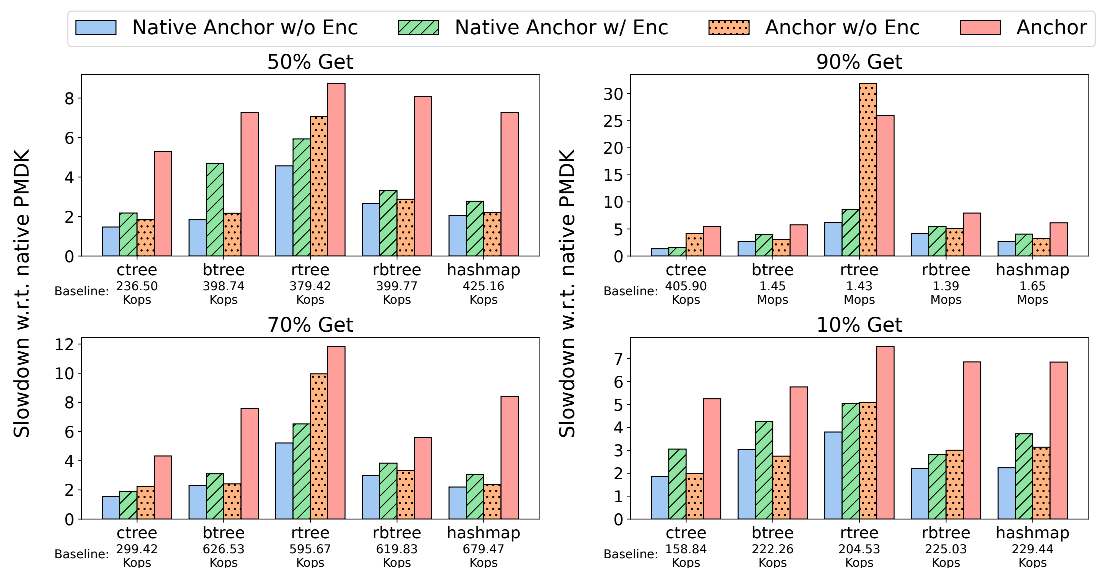


Figure 4.6: Performance of PM indices under YCSB workloads for secure, native, w/ and w/o encryption ANCHOR versions.

4.1.7.2 Persistent Indices

We evaluate the performance of ANCHOR for five different PM indices (ctree, btree, rbtree, rtree, and hashmap) under four YCSB workloads (10 % Get, 50 % Get, 70 % Get and 90 % Get). We compare the performance of all five indices over five competitive baselines: *(i)* ANCHOR, *(ii)* ANCHOR w/o Encryption, *(iii)* ANCHOR running outside SCONE (Native ANCHOR) *(iv)* Native ANCHOR w/o Encryption, and *(v)* Native PMDK. Our experiments seek to quantify two inevitable overheads: *(i)* the overheads to ensure confidentiality through the comparison between the versions with and without the encryption layer and *(ii)* the overheads of the TEE (e.g., due to limited enclave memory) by comparing the versions that run natively with those running inside SCONE. We use 10 M operations on 100 k keys grouped in transactions and fixed key-value sizes equal to 8 B and 512 B, respectively. Note that these experiments are conducted on a single node, with the workload directly provided to the PM indices without network interference.

Figure 4.6 illustrates the average slowdown for the four ANCHOR's versions normalized to the native PMDK. In general, ANCHOR's throughput is 4.33-8.40× lower for every data structure except rtree, whose slowdown ranges from 7.54× to 25.96×. To better understand the results and the overhead sources, we collected statistics for the native ANCHOR, as taking precise timestamps inside SCONE does not give reliable results due to enclave exits, shown in Figure 4.7. We further observe that the application integration into SCONE leads to a slowdown of 1.45-3.47×, depending on the workload.

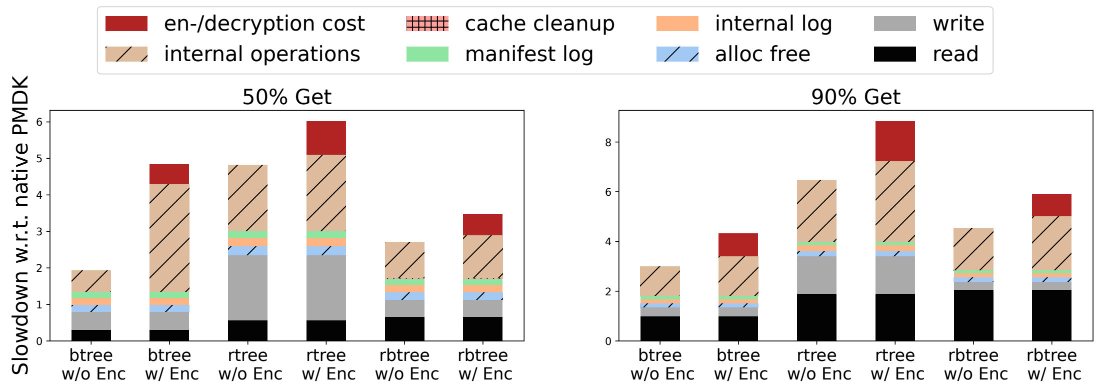


Figure 4.7: Overhead breakdown in native ANCHOR version for PM indices under varying YCSB workloads.

The data en-/decryption also contributes significantly to the total overhead, especially inside SCONE, leading to up to 3.54× slowdown compared with the respective version without encryption. An exception is the rtree inside SCONE with the read-intensive workload. The introduction of the encryption layer leads to a lower overhead due to the slower pace of data fetching, which reduces the EPC pressure and decreases the frequency of the cache cleanup.

Moreover, we note that the average overhead slightly increases when the read ratio is increased. In the 90 % Get workload, the throughput is 5.49-25.96× lower than PMDK, while in the case of 50 % Get the respective values are 5.28-8.75×. Figure 4.7 confirms that the read operations contribute significantly to the overhead compared to write operations in such cases. The number of reads is much higher than the number of writes, as even put operations require a traversal of the index to locate the update/insert positions. Thus, we can account for this behavior to the faster pace that ANCHOR fetches PM data into their volatile buffers, causing higher EPC pressure and more frequent cleanups of the object cache.

Finally, the higher overhead (7.54-25.96×) of rtree stems from the size of its nodes (4 KiB). While PMDK only requires a partial direct read/write to a node, ANCHOR needs to fetch it entirely. This copying, compared with the PMDK's direct read/write along with the increased EPC usage and number of cleanups, results in significantly higher overheads for rtree when running in SCONE. The EPC paging effect is highlighted through Table 4.2, which shows considerably lower overheads for rtree with smaller memory footprint.

Overall, the overheads of ANCHOR mostly stem from the expensive EPC paging. However, upcoming trusted computing paradigms such as confidential VMs (e.g., Intel TDX [107]) will eliminate the limited EPC issue, thus leading to reduced overheads.

Version	50% Get	70% Get	90% Get
ANCHOR w/o Enc — 10k keys	3.4×	3.6×	4.2×
ANCHOR — 10k keys	4.8×	5.3×	7.0×

Table 4.2: rtree overhead for 10k keys

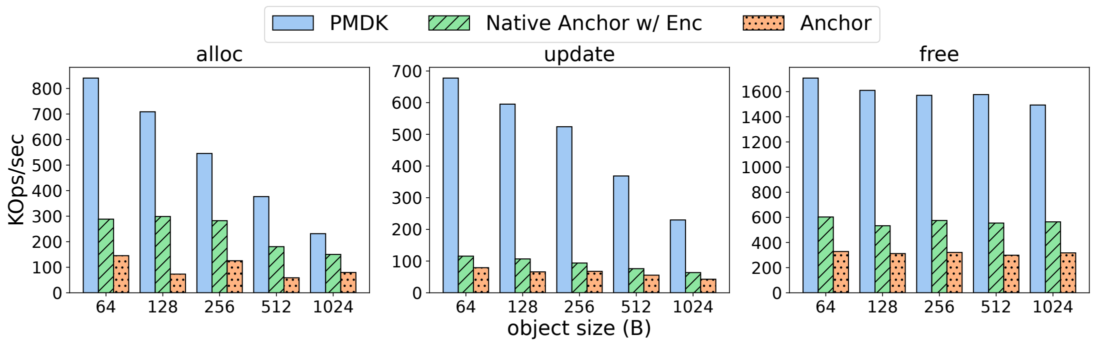


Figure 4.8: Performance evaluation for TX memory operations for secure & native ANCHOR versions w/ encryption.

4.1.7.3 Operation Performance

We evaluate ANCHOR using a microbenchmark based on *pmembench* [126] to assess the performance of three operations supported by ANCHOR, namely alloc & init, update, and free. We perform a series of transactions where each transaction consists of multiple operations on different objects selected with uniform access pattern. We vary the size of the objects to examine its impact. For update and free operations, we pre-allocate the PM objects. We compare both *(i)* ANCHOR and *(ii)* Native ANCHOR against PMDK.

Figure 4.8 shows the throughput of the memory management operations. In the case of allocation, ANCHOR is 1.9-4.1× and 2.9-9.7× slower compared to native ANCHOR and PMDK, respectively. For object deallocation, the respective slowdowns are 1.7-1.9× compared to the native version and 4.7-5.3× compared to PMDK. For both alloc and free, we observe that ANCHOR's behavior is similar to PMDK's with increasing object sizes. This is expected as de-/allocations only perform metadata modifications that are not affected by the object size. In an update operation, ANCHOR needs to perform an extra copy of the PM data inside the enclave in case of a cache miss. This leads to a smaller relative overhead (9.0× down to 5.4×) with the increasing size, as objects reside in the cache and are updated in place before the TX commit phase, avoiding multiple costly copies.

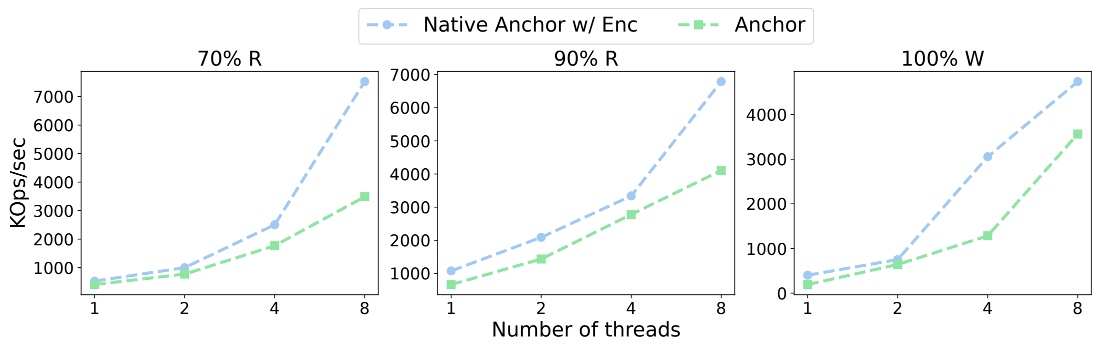


Figure 4.9: R/W performance evaluation of secure and native ANCHOR versions w/ encryption with varying number of threads.

4.1.7.4 Scalability

We next evaluate the scalability with increasing number of threads from 1 to 8, the maximum number of cores in our server. Each thread maintains its own object set. We set the object size to 256 B and perform read and write operations.

Figure 4.9 shows ANCHOR's scalability. The lower scalability rate for ANCHOR is mainly caused by the frequent updates and look-ups to locate each object's metadata in the metadata index. It inevitably increases the cost of each operation due to the mandatory lock usage, which is expensive in SCONE.

4.1.7.5 Effectiveness of Optimizations

We evaluate the effectiveness of our caching optimization using the ctree, btree, and rbtree indices. We use two YCSB workloads: one update- (50 % Get) and one read-intensive (90 % Get). We use 10 M operations and key-value sizes equal to 8 B and 512 B, respectively.

Figure 4.10 reports the performance improvement of our object cache. We observe a performance boost in most scenarios for our data structures. For the update-intensive workload (50 % Get), ANCHOR has up to 1.49× speedup compared to the non-optimized version. The performance gain becomes more obvious in the read-dominated workload (90 % Get) where ANCHOR's throughput improves up to 3.94×. For the case of btree with the write-intensive workload, we observe a small performance loss due to EPC paging effects. Overall, our technique reduces the cost of the read operation since it decreases the number of decryptions, as the content of an object can be directly found in the volatile, protected object cache. Table 4.3 shows the cache hit ratio for the PM indices averaged across the three different workloads shown in Figure 4.6. We notice that all workloads achieve more than 80 % hit rate, confirming the usefulness of this optimization.

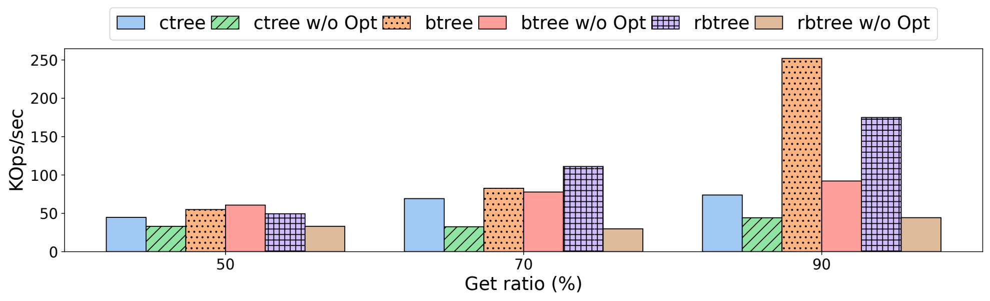


Figure 4.10: Effectiveness of optimizations w/ YCSB.

Index :	ctree	btree	rbtree	rtree	hashmap
Object reads (M)	203.29	87.9	215.77	99.72	60.02
Hit ratio (%)	92.34	89.48	95.01	82.11	84.23

Table 4.3: Average cache hit ratio

	PMDK	ANCHOR	
Operation	Undo/redo log (B)	Undo/redo log (B)	manifest (B)
tx_alloc	1.66	3.68	77.41
tx_update	1049.32	1082.88	70.72
tx_free	1.66	3.68	71.63

Table 4.4: Write amplification normalized per object

4.1.7.6 Persistent Memory Write Amplification

We compute the PM write amplification as the total bytes of the manifest and the extra bytes written to the logs in three basic memory management operations, namely alloc, update, and free. We performed a series of transactions on discrete 1024 B objects.

Table 4.4 lists the number of bytes PMDK, and ANCHOR persist on average per object and operation to the logs. ANCHOR persists 2.2× more data on average in the secure logs for alloc and free operations. Both alloc and free involve only the redo log, whose entries for PMDK are 16 B, and thus, even the small additions of the trusted counter value and size to ensure the integrity and freshness properties double their size. Note that as allocations and frees are performed via bitmap updates, they can be merged. Thus, multiple allocations/frees in a transaction can be recorded in a single redo log entry. This factor is less remarkable in the update case where the object is snapshotted. More specifically, the increase in the required bytes for the secure undo log of ANCHOR is 3.2 % on average and roots from the required metadata in the log entries. For each operation, ANCHOR inevitably appends manifest entries at the commit phase to keep track of the updated integrity signatures. Along with the user objects'

Manifest size (MiB)	96	138	224	266
Recovery time (s)	2.60	3.02	4.17	5.16

Table 4.5: Boot-up time with varying manifest size

Manifest size (MiB)	138		224	
Log size (MiB)	0.98	4.88	0.98	4.88
Recovery time (s)	3.02	3.09	4.11	4.12

Table 4.6: Recovery time with varying manifest and log size

entries, metadata modifications need to be tracked in the manifest, as well as entries indicating the transactions' phase.

4.1.7.7 Recovery Overheads

We measure ANCHOR's recovery time, which is affected by two factors: *(i)* populating the metadata index from the manifest (500k objects) and *(ii)* applying undo/redo logs. We assess the impact of the manifest and undo/redo log sizes.

Table 4.5 and Table 4.6 show the required time to open a pool, scan the manifest, reconstruct the metadata index, and perform the recovery if needed. Table 4.5 highlights the effect of the manifest. As the manifest size grows, the same applies to the boot-up time. It is expected since each entry must be verified, decrypted, and checked against the expected counter value. In Table 4.6, we observe that the size of the logged objects has a negligible impact on the recovery time. Even with a relatively large (5 MiB) log, the recovery time is barely increasing. The reason is that the manifest scan, verification, and metadata index restoration are dominating the boot-up.

4.1.7.8 ANCHOR's Network Stack

Next, we evaluate the performance of ANCHOR's Network Stack (NS). We further measure the performance of ANCHOR in a client-server model using a persistent hashmap index. The server accepts multiple client requests before a TX commit to limit the effect of the waiting time for stabilization. We use 6 different setups: *(i)* iperf [141], *(ii)* ANCHOR-NS outside SCONE w/o Encryption, *(iii)* ANCHOR-NS outside SCONE w/ Encryption, *(iv)* iperf in SCONE, *(v)* ANCHOR-NS w/o Encryption and *(vi)* ANCHOR-NS varying the data size per request. To simulate the behavior of iperf in our implementation, we send requests with a payload of the given data size. At the server, we count the number of arriving requests in a certain time span.

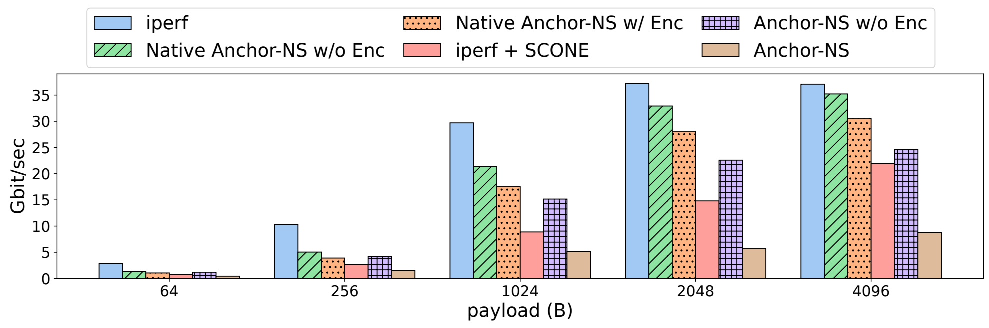


Figure 4.11: Network stack throughput with iperf.

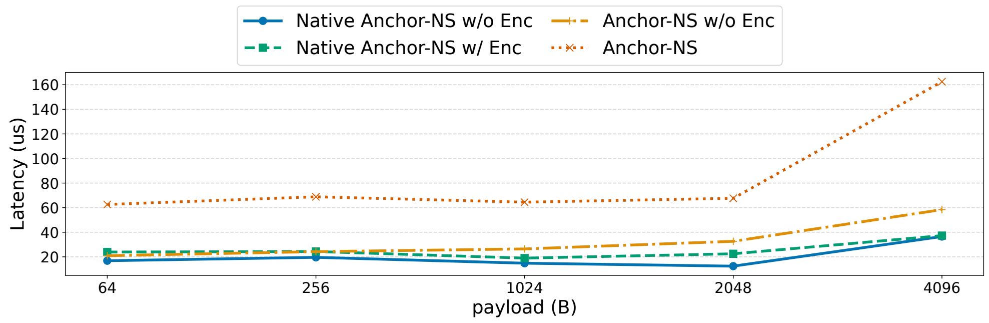


Figure 4.12: Network stack latency.

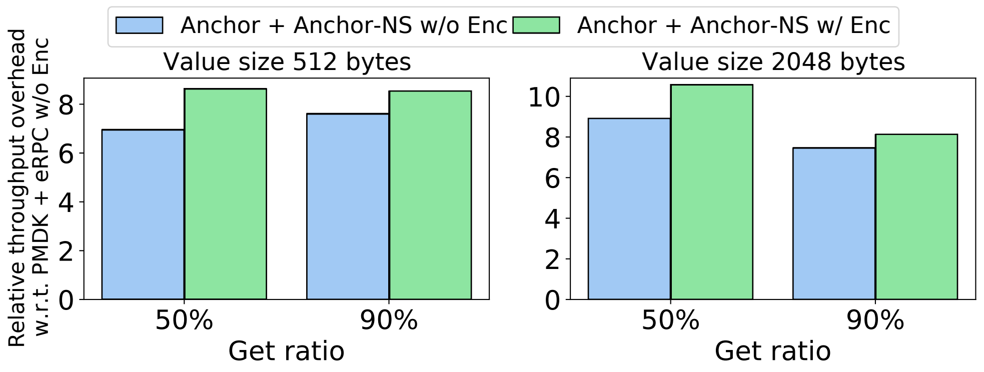


Figure 4.13: Evaluation of ANCHOR w/ NS for KV store under YCSB workloads (50 % Get, 90 % Get) with different value sizes.

Figure 4.11 and 4.12 show ANCHOR's NS throughput and latency. We notice that the encryption overhead depends on the payload size. The overhead is higher for smaller payloads. The same applies to the slowdown caused by SCONE. It is explained as each encryption induces a constant overhead, i.e., for the encryption of a message header and writing of the authentication tag. In the native case, iperf outperforms eRPC. However, the sockets approach is slower inside the enclave due to syscalls which justifies our choice for eRPC.

Further, Figure 4.13 demonstrates ANCHOR's performance to manage a persistent index remotely. The index throughput is 6.96×-10.58× lower when accessed via our ANCHOR-NS, compared with native PMDK using eRPC. The encryption layer does not significantly impact the performance (1.09×-1.24×) as its overhead is absorbed by the network latency and the mandatory stabilization-point waits of the KV management system. The higher overheads in the write-dominant workload can be explained as put operations having a bigger latency compared to the get operations, which can lead to higher congestion at the network queues. Remotely accessed ANCHOR shows similar behavior regarding the get/put proportion with the one observed in ($ 4.1.7.2).

4.1.8 Related Work

Persistent memory. PM systems are actively researched across several dimensions, such as filesytems [38, 149, 158, 325], KV-stores [37, 150, 178, 324, 339], crash consistency & reliability [41, 217, 254, 336, 337, 340] and testing tools [30, 185–187]. In contrast to the prior work, our focus in on building a secure PM library. Securing durable PM against data permanence attacks has been the goal of several works [18, 19, 40, 184, 328, 333, 342, 343]. Unlike these systems, ANCHOR ensures data freshness, does not require an explicit distinction between volatile and PM data, and can easily be adapted to work on existing platforms due to its intuitive programming model.

Secure storage systems. Building secure databases and storage systems in the cloud is crucial to avoid undesirable access to sensitive data. Several works pursue this goal for single-node [2, 25, 63, 163, 164, 200, 248, 288, 319] and distributed settings [23, 191, 246, 318] based on different TEEs-compatible designs. However, all these systems target traditional storage stacks with volatile memory and block-based persistent storage, while ANCHOR focuses on PM, which introduces its own, novel programming model.

Secure I/O stack. There exist several solutions for data transmission from TEEs over untrusted networks. Kernel-based approaches suffer from high overheads of world switches due to system calls [277]. Asynchronous system calls [16, 226] alleviate these overheads, but still require copying data to and from the trusted environments. Further works [200, 257, 274] target security challenges of one-sided RDMA. ANCHOR targets similar challenges for two-sided RDMA (RPCs) which has been shown to be the most effective for the design of storage systems [153–156].

On the storage front, while the modern shielded execution frameworks have employed direct I/O in the context of TEEs [23, 25, 200, 293, 295], they are incompatible with PM, which mandates remote crash consistency. ANCHOR builds on the direct I/O mechanism but ensures crash consistency for data written to remote PM devices.

Remote PM access. Recent research efforts aim to expand the RDMA interfaces for the PM to include durability semantics; performance, and crash consistency for remote operations [98, 100, 152, 160, 329]. ANCHOR's secure network stack for PM is based on these advancements, where it adopts kernel-bypass networking to achieve performance by avoiding the prohibitive overheads of system calls and ensures crash consistency for remote PM accesses [274, 291].

4.1.9 Summary

In this section, we present ANCHOR, a secure persistent memory library. ANCHOR allows for building secure PM data management systems by offering a programming model and APIs similar to the established PM programming model and PMDK APIs, while preserving crash consistency through its formally verified secure logging protocol. To achieve this, ANCHOR combines three non-trivially compatible, recent hardware advancements: TEEs, PM, and kernel-bypass networking. ANCHOR leverages the TEE provided by Intel SGX and designs a PM management engine that builds on PMDK, enhanced with confidential and authenticated data structures. It further integrates a secure kernel-bypass network stack based on eRPC and a formally proven remote attestation protocol for trust establishment. Our evaluation using the YCSB workloads over PM indices shows that ANCHOR incurs reasonable overheads.

Software artifact. ANCHORis publicly available along with its formal proofs.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The emergence of Compute Express Link (CXL) technology revolutionizes the cloud computing infrastructures towards disaggregated architectures, where byte-addressable storage devices (e.g., PM [73, 113, 167, 201], SSDs with memory semantics [91], CXL SSDs [90, 260]) have the potential to become prominent for data storage. However, despite its promising performance benefits, the PM also adds an additional attack vector in the cloud. Malicious attackers can exploit potential memory safety bugs in PM applications or perform attacks, both physical and in software, trying to compromise the PM data, raising severe security and privacy concerns for its adoption. As a countermeasure to this challenging problem, this thesis explores the design space of how to build dependable — safe, reliable, and secure — persistent memory systems for the next generation of untrusted cloud environments.

First, we design SafePM, a framework that ensures memory safety in PM-based applications by detecting both spatial and temporal memory safety violations. SafePM utilizes the compiler instrumentation and reporting mechanisms of the widely-used, battle-tested AddressSanitizer (ASan), enabling developers to detect memory safety violations without additional complexity since SafePM fits naturally into existing development and deployment workflows, making it accessible and practical for cloud users. In terms of its functionality, when a PM pool is created, SafePM creates a persistent shadow memory region, which is mapped to the corresponding location in ASan's shadow memory. SafePM manages the persistent shadow memory alongside the persistent heap transparently to the application and preserves the crash consistency for the PM data and memory safety metadata. As a result, any PM-based application can use SafePM to test for memory violations at runtime, including during the recovery process, without needing to modify their source code. Our comprehensive evaluation demonstrates that SafePM provides the same level of memory safety for PM applications as ASan does for volatile memory while incurring reasonable overheads.

Secondly, we want to expand our memory safety solutions beyond debugging environments. To this end, we propose SPP, the first tagged-pointer-based solution for PM to offer practical memory safety. SPP comprises compiler instrumentation based on LLVM, a runtime library, and a modified version of PMDK. It augments the PM pointer representation with memory safety metadata that is set and updated in a crash-consistent manner. SPP leverages the spare bits in the pointers to place a tag and renders the PM pointers invalid when they exceed their assigned boundaries, thus lowering the cost of bounds checking. Its runtime functions ensure the correct management of tagged pointers and compatibility with pre-compiled external libraries. Additionally, SPP preserves the PMDK API, requiring no source code modifications, and can be seamlessly integrated into existing PM software. Our extensive evaluation highlights that SPP effectively detects PM buffer overflows with minimal performance impact and negligible space overhead, constituting it a good candidate for production deployments.

Lastly, to seal our vision for building an end-to-end dependable PM architecture, we introduce Anchor. Anchor is a PM library designed for building secure PM data management systems. To achieve this, Anchor integrates three recent hardware advancements that are inherently incompatible: Trusted Execution Environments (TEEs), Persistent Memory (PM), and kernel-bypass networking. Anchor utilizes Intel SGX's TEE and builds a PM management engine based on PMDK, enhanced with confidential and authenticated durable data structures. Additionally, Anchor offers a programming model similar to the established PM standards and ensures crash consistency for the PM residing data through its formally verified secure logging protocol. It also incorporates a secure kernel-bypass network stack using eRPC and a formally proven remote attestation protocol for trust establishment. Our evaluation demonstrates that Anchor incurs reasonable overheads, considering its strong security properties.

All these projects contribute to the design of our end goal, an end-to-end dependable PM architecture for untrusted cloud environments. A system designer can combine these projects, or parts thereof, to build their system depending on their specific safety and security requirements, as well as targeted deployment environments.

5.2 Future Work

This thesis explored the design space of dependable persistent memory architectures destined for untrusted third-party cloud infrastructures. Despite the discontinuation of the Intel Optane product line [58, 279], the ongoing shift towards disaggregated and heterogeneous system setups brings about new challenges. These systems are equipped with various new Trusted Execution Environment (TEE) technologies and have access to large distributed memory and storage pools, which can reside either within the same data center or be accessed remotely through the network. Consequently, building trustworthy and reliable data management systems becomes increasingly complex. Inspired by our work, we discuss new opportunities for impactful research in the domains of memory safety and security for systems incorporating byte-addressable storage in this section.

Memory safety. Memory safety still remains a critical issue [205], which justifies it being a very active area of research. While memory-safe languages, such as Rust [195], are on the rise, the existing legacy codebases consisting of billions of lines of C and C++ necessitate the employment of memory safety mechanisms. On top of that, the adoption of the Compute Express Link (CXL) [47] technology is anticipated to facilitate new programming models for byte-addressable storage devices [17], which will also mandate crash consistency. Therefore, it is important to rethink memory safety in such heterogeneous systems, where memory access patterns deviate from the standardized norms – (persistent) memory pools can be accessed by various devices using cache-coherent interconnects and Direct Memory Access (DMA). Consequently, approaches like SafePM and SPP can be a solid foundation but may require adaptation to the new programming standards to ensure memory safety in CPU-less byte-addressable storage nodes accessible through a CXL-capable networking infrastructure.

Additionally, the memory safety community is characterized by a continuous effort to minimize the overheads incurred by the proposed solutions. On the one hand, probabilistic or sampling-based approaches (e.g., GWP-ASan [265]) offer a trade-off between safety and performance, primarily aiming toward production environments. Enforcing such a solution in our proposed systems (e.g., in SafePM) is a viable direction to be well-prepared for the widespread adoption of byte-addressable storage. On the other hand, using hardware-assisted memory safety solutions, such as CHERI [322] and ARM's Memory Tagging Extension (MTE) [14], is gaining traction. Towards this direction, porting SPP to leverage MTE or CHERI, considering their unique pointer characteristics, could lead to a high-performance memory safety solution for durable byte-addressable devices. Importantly, all these approaches and extensions must ensure the durability of data structures and preserve crash consistency property for both data and memory safety metadata. To this end, integrating memory safety techniques with seamless hardware-assisted persistency [29] is worth exploring in the future as it can greatly assist towards this goal.

Secure end-to-end data management systems. Disaggregated, heterogeneous computing setups define the next generation of cloud systems, and Compute Express Link (CXL) [47] is pitched to be their backbone. Precisely, the upcoming PCIe 6.0 specification, the layer upon which CXL is built, introduces new security features [230, 231]. These features, among others, include the Component Measurement and Authentication (CMA) and the Integrity and Data Encryption (IDE). Together, they aim to simplify the processes of device authentication and secure data transmission. Such protocols can be leveraged to build secure systems that incorporate PCI-attached byte-addressable storage devices [90, 260, 271] and enforce their principles directly on the wire level.

Further, in our context, the integration of accelerators, such as FPGAs, into heterogeneous cloud environments offers additional computing resources that can significantly reduce the performance overheads caused by the mandatory operations to ensure security or crash consistency. For instance, in ANCHOR, offloading functionalities like transactional logging or encryption algorithms to these accelerators can result in higher system throughput and improved resource utilization. This is particularly beneficial in multi-tenant environments, where efficient resource allocation in conjunction with the maximization of performance is a high priority.

Apart from that, TEE technologies evolve and new ones emerge [5, 13, 132], such as the Confidential Virtual Machines (CVMs) that allow for running a full-fledged virtual machine inside a hardware-based TEE. CVMs open up new ways for designing secure data management systems, where the size of the secure memory region is no longer a limiting factor. In this way, the system designer can focus on providing a high-performance, secure software stack rather than minimizing the TCB. Precisely, in ANCHOR, we could rethink the design of its in-memory data structures and re-evaluate and optimize the workflow of its operations to optimize them for CVMs that access byte-addressable storage. However, undoubtedly, a larger TCB increases the potential attack surface. Therefore, it is essential for a secure system to be able to sanitize its inputs from the untrusted world, especially from untrusted devices (e.g., PM). To this end, protocols such as Security Protocol and Data Model (SPDM) [281] and TEE Device Interface Security Protocol (TDISP) [292] can be utilized in the system design to ensure a trusted and authenticated I/O.

On the networking frontier, the development of programmable network cards (SmartNICs) is a promising hardware advancement [20, 33]. For instance, ANCHOR's network stack can offload part of its operations to the SmartNIC (e.g., packet encryption), thereby enhancing performance and reducing CPU load. Additionally, in a scenario where a SmartNIC is attested and the message verification mechanism is offloaded to its computational unit, it can enable the use of one-sided Remote Direct Memory Access (RDMA). Specifically for byte-addressable storage, support for atomic operations on encrypted data over the network would allow efficient implementation of data structures and algorithms. On top of that, hardware-assisted crash consistency mechanisms enforced by the NIC hardware could resolve the major challenge of crash consistency and further reduce the software overhead for ensuring data durability in one-sided operations. Such enhancements could allow for using one-sided RDMA to build a secure network stack for byte-addressable storage and significantly improve performance and security by relieving CPU stress without the danger of violating the network security properties.

Lastly, while our proposed systems provide software-based safety and security properties, it's crucial to acknowledge their limitations in the face of hardware attacks. Physical access to PM modules could potentially subvert their guarantees. For instance, cold boot attacks or direct probing of memory chips could bypass some of the protection mechanisms of SAFEPM and SPP. Additionally, techniques like bus snooping or DMA-based attacks could expose sensitive data during transmission between the CPU and PM. ANCHOR's use of Intel SGX provides protection against physical attacks to some extent, but side-channel attacks exploiting hardware vulnerabilities, such as speculative execution or power analysis, could still pose a threat. To address these concerns, a multi-layered approach is necessary. This could include physical security measures for data centers, embedded hardware-based encryption for PM modules, and continuous monitoring for unusual access patterns. For example, software-based protections, such as obfuscation or intelligent padding and alignment, can partially mitigate side-channel attacks. On top of that, hardware-based protections can also be employed (e.g., hardware isolation, randomization via noise insertion) to alleviate this issue. Overall, future work should focus on combining these hardware-level protections with our software-based solutions to create a robust and comprehensive security framework.

CXL for byte-addressable storage systems. While the present work focuses on PM devices directly attached to the memory bus, the CXL technology introduces a new paradigm, enabling disaggregated architectures with byte-addressable storage over PCIe. Byte-addressable storage over CXL differs fundamentally from PM directly attached to the memory bus in its consistency, coherency, and persistency semantics. Unlike PM, PCIe/CXL interconnects may reorder messages, necessitating new CPU instructions or memory barriers to guarantee write ordering and completion.

Furthermore, the architectural complexity of CXL results in challenges in crash consistency. Current remote PM consistency models, like reading after writing to ensure persistency, are insufficient under CXL due to potential reordering at the interconnect level. Thus, existing systems would require adaptation to support CXL's flushing and coherence protocols.

In addressing these challenges, future work should focus on developing abstractions and programming frameworks to integrate byte-addressable storage with the CXL protocols. There needs to be a formalization of how to ensure the persistence and crash consistency properties. This would allow existing applications to harness the benefits of CXL environments without major rewrites. In this way, assuming that byte-addressable storage is meant to be accessed via a pointer interface (`ld/st`) from the application's perspective, systems like SAFEPM and SPP can be adapted to leverage the new programming frameworks, APIs and crash consistency mechanisms (e.g., a different form of logging) to provide their guarantees without major changes due to their simplistic design. It is important to note that specifically for SPP, the storing of metadata in disjoint regions might be inevitable to accommodate larger pools and objects, leading to increased performance overheads. Regarding ANCHOR, ensuring a correct crash consistency mechanism would require adaptations to handle CXL's message reordering. Enhancing the secure logging protocol internal operations (including the Manifest) to account for this factor is essential for durability, crash consistency, and rollback resilience. Further, providing confidentiality, integrity, and freshness guarantees for data and operations across disaggregated nodes would demand synchronization considerations and modifications in the internal primitives that ANCHOR uses to achieve data persistence, as the currently-used cache flushing and non-temporal stores are not effective for remote byte-addressable storage.

Overall, the concept of byte-addressable storage over CXL is still immature — Samsung recently proposed a battery-backed CXL device [260] to bridge gaps in persistency guarantees. Additionally, the emerging CXL SSDs [90, 271] propose memory-mapped, persistent storage over CXL, raising opportunities for following a similar direct memory access programming model. Extending our systems to such hardware would require re-engineering to use the proper APIs or low-level primitives to align with the unique memory-mapping characteristics and to maintain crash consistency across the CXL pipeline.

Bibliography

[1] Developer Guide and Reference for Intel® Integrated Performance Primitives Cryptography. https://www.intel.com/content/www/us/en/docs/ipp-crypto/developer-guide-reference/2021-12/overview.html.

[2] Ahmad, A., Kim, K., Sarfaraz, M. I., and Lee, B. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (2018), The Internet Society.

[3] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium (USENIX Security 09)* (Montreal, Quebec, August 2009), USENIX Association.

[4] Aldershoff, J. W. Intel reveals App Direct mode and Memory mode for Optane DC Persistent Memory, 2021. Accessed 27-09-2021.

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

[6] SEV-SNP Platform Attestation Using VirTEE/SEV. https://www.amd.com/content/dam/amd/en/documents/developer/58217-epyc-9004-ug-platform-attestation-using-virtee-snp.pdf.

[7] Queue the hardening enhancements. `https://security.googleblog.com/2019/05/queue-hardening-enhancements.html`, 2019. Accessed: 2021-02-27.

[8] 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.

[9] Antici, F., Bartolini, A., Kiziltan, Z., Babaoglu, O., and Kodama, Y. Mcbound: An online framework to characterize and classify memory/compute-bound hpc jobs.

[10] Apache. Apache crail (incubating). `https://github.com/apache/incubator-crail`, December 4, 2024.

[11] Ardelean, D., Diwan, A., and Erdman, C. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 405–417.

[12] ARM. Building a secure system using trustzone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`. Last accessed: Jan, 2021.

[13] ARM. Introducing arm confidential compute architecture. `https://developer.arm.com/documentation/den0125/0300`.

[14] Memory tagging extension: Enhancing memory safety through architecture. `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety`, 2019. Accessed 27-09-2021.

[15] Arm Confidential Compute Architecture. `https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture`. Last accessed: May 2021.

[16] 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* (USA, 2016), OSDI'16, USENIX Association, p. 689–703.

[17] Assa, G., Friedman, M., and Lahav, O. A programming model for disaggregated memory over cxl, 2024.

[18] Awad, A., Manadhata, P., Haber, S., Solihin, Y., and Horne, W. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. *SIGARCH Comput. Archit. News 44*, 2 (March 2016), 263–276.

[19] Awad, A., Ye, M., Solihin, Y., Njilla, L., and Zubair, K. A. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)* (2019), pp. 104–115.

[20] Aws nitro system. `https://aws.amazon.com/ec2/nitro/`.

[21] Aws nitro enclaves. `https://aws.amazon.com/ec2/nitro/nitro-enclaves/`.

[22] 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.

[23] 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 (ATC'21)* (2021).

[24] Bailleu, M., Stavrakakis, D., Rocha, R., Chakraborty, S., Garg, D., and Bhatotia, P. Toast: A heterogeneous memory management system. In *2024 33rd International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2024), IEEE. Paper accepted for publication.

[25] 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* (USA, 2019), FAST'19, USENIX Association, p. 173–190.

[26] Pmdk - unit test for store reordering. `https://github.com/pmem/pmdk/blob/stable-1.8/src/test/obj_reorder_basic/obj_reorder_basic.c`.

[27] 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).

[28] Berger, E. D., and Zorn, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, Association for Computing Machinery, p. 158–168.

[29] Bhardwaj, A., Thornley, T., Pawar, V., Achermann, R., Zellweger, G., and Stutsman, R. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (New York, NY, USA, 2022), HotStorage '22, Association for Computing Machinery, p. 37–44.

[30] Bozdoğan, K. K., Stavrakakis, D., Issa, S., and Bhatotia, P. Safepm: A sanitizer for persistent memory. In *Proceedings of the Seventeenth European Conference on Computer Systems* (NY, NY, USA, 2022), EuroSys '22, ACM, p. 506–524.

[31] Brandenburger, M., Cachin, C., Lorenz, M., and Kapitza, R. Rollback and forking detection for trusted execution environments using lightweight collective memory. *CoRR abs/1701.00981* (2017).

[32] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 209–223.

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

[34] Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, Association for Computing Machinery, p. 433–452.

[35] Checkoway, S., and Shacham, H. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, p. 253–264.

[36] CHEN, R., AND SUN, G. A survey of kernel-bypass techniques in network stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence* (NY, NY, USA, 2018), CSAI '18, ACM, p. 474–477.

[37] CHEN, Y., LU, Y., YANG, F., WANG, Q., WANG, Y., AND SHU, J. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 1077–1091.

[38] CHEN, Y., SHU, J., OU, J., AND LU, Y. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage 14*, 1 (April 2018).

[39] CHENG, W., ZHAO, Q., YU, B., AND HIROSHIGE, S. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)* (2006), pp. 749–754.

[40] CHHABRA, S., AND SOLIHIN, Y. i-nvmm: A secure non-volatile main memory system with incremental encryption. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011), pp. 177–188.

[41] CHOI, B., BURNS, R., AND HUANG, P. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the Sixteenth European Conference on Computer Systems* (NY, NY, USA, 2021), EuroSys '21, ACM, p. 441–457.

[42] The chromium projects - memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety`, 2021. Accessed 27-09-2021.

[43] Clang 13 Documentaion - AddressSanitizer, 2021. Accessed 27-09-2021.

[44] 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`. Last accessed: Jan, 2021.

[45] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News 39*, 1 (March 2011), 105–118.

[46] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*

(New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 133–146.

[47] CONSORTIUM, C. Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, December 4, 2024.

[48] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)* (2010).

[49] COPELAND, G., KELLER, T., KRISHNAMURTHY, R., AND SMITH, M. The case for safe ram. In *Proceedings of the 15th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1989), VLDB '89, Morgan Kaufmann Publishers Inc., p. 327–335.

[50] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

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

[52] 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. Last accessed: Dec, 2018.

[53] Cxl software ecosystem. https://github.com/pmem/pmem.github.io/blob/main/content/blog/2023/cxl-blog-post.md.

[54] DANA NEUSTADTER - SYNOPSYS. Protecting Data over PCIe & CXL in Cloud Computing. https://www.chipestimate.com/Protecting--Data--over--PCIe--and--CXL---in--Cloud-Computing/Synopsys/Technical-Article/2021/08/10.

[55] DANG, T., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (04 2015), 555–566.

[56] DEBIAN. Debian manpages - libpmemobj-dev, 2021.

[57] DEMERI, A., KIM, W.-H., KRISHNAN, R. M., KIM, J., ISMAIL, M., AND MIN, C. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings*

of the 21st International Middleware Conference (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, p. 207–220.

[58] DESNOYERS, P., ADAMS, I., ESTRO, T., GANDHI, A., KUENNING, G., MESNIER, M., WALDSPURGER, C., WILDANI, A., AND ZADOK, E. Persistent memory research in the post-optane era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (New York, NY, USA, 2023), DIMES '23, Association for Computing Machinery, p. 23–30.

[59] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M. K., AND ZDANCEWIC, S. Hardbound: architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, Association for Computing Machinery, p. 103–114.

[60] DHURJATI, D., AND ADVE, V. *Backwards-Compatible Array Bounds Checking for C with Very Low Overhead.* Association for Computing Machinery, New York, NY, USA, 2006, p. 162–171.

[61] DHURJATI, D., KOWSHIK, S., AND ADVE, V. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM, pp. 144–157.

[62] DI, B., LIU, J., CHEN, H., AND LI, D. Fast, flexible, and comprehensive bug detection for persistent memory programs. 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. 503–516.

[63] DICKENS III, B., GUNAWI, H. S., FELDMAN, A. J., AND HOFFMANN, H. Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2018).

[64] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on Information Theory* (1983).

[65] DOUGLAS, C. RDMA with PMEM, Software mechanisms for enabling access to remote persistent memory. `https://www.snia.org/sites/default/files`

/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2020.

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

[67] 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).

[68] Duck, G. J., Yap, R., and Cavallaro, L. Stack bounds protection with low fat pointers. In *Network and Distributed System Security Symposium* (2017), NDSS.

[69] Duck, G. J., and Yap, R. H. C. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction* (New York, NY, USA, 2016), CC 2016, Association for Computing Machinery, p. 132–142.

[70] Duck, G. J., and Yap, R. H. C. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, Association for Computing Machinery, p. 181–195.

[71] Eigler, F. Mudflap: Pointer use checking for c/c++. In *GCC Developers Summit* (01 2003).

[72] et al., G. 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).

[73] Cxl software ecosystem. EverspinAnnouncesNewSTT-MRAMEM128LXxSPIMemory .

[74] Fink, M., Stavrakakis, D., Sprokholt, D., Chakraborty, S., Ekberg, J.-E., and Bhatotia, P. Cage: Hardware-accelerated safe webassembly. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2025), CGO '25, Association for Computing Machinery. Paper accepted for publication.

[75] Fu, X., Kim, W.-H., Paddayuru Shreepathi, A., Ismail, M., Wadkar, S., Lee, D., and Min, C. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 1–15.

[76] GCC - Program Instrumentation Options, 2021. Accessed 27-09-2021.

[77] Ghose, S., Gilgeous, L., Dudnik, P., Aggarwal, A., and Waxman, C. Architectural support for low overhead detection of memory violations. In *2009 Design, Automation & Test in Europe Conference & Exhibition* (2009), pp. 652–657.

[78] 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.

[79] GNU Binutils. `https://sourceware.org/binutils/`, 2022. Accessed 20-07-2022.

[80] 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, February 2008), USENIX Association.

[81] Dataproc Confidential Compute. `https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/confidential-compute`.

[82] Introducing Google Cloud Confidential Computing with Confidential VMs. `https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms`. Last accessed: Jan, 2021.

[83] Gorjiara, H., Xu, G. H., and Demsky, B. Jaaru: Efficiently model checking persistent memory programs. 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. 415–428.

[84] Gueron, S. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Paper 2016/204, 2016. `https://eprint.iacr.org/2016/204`.

[85] Gulati, A., Shanmuganathan, G., Holler, A., and Ahmad, I. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing* (USA, 2011), HotCloud'11, USENIX Association, p. 3.

[86] 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.

[87] Hähnel, M., Cui, W., and Peinado, M. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[88] Haller, I., van der Kouwe, E., Giuffrida, C., and Bos, H. Metalloc: efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security* (New York, NY, USA, 2016), EuroSec '16, Association for Computing Machinery.

[89] 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).

[90] Handy, J. Understand how the CXL SSD can aid performance. December 4, 2024.

[91] Handy, J. Understand how the CXL SSD can aid performance. December 4, 2024.

[92] Hasabnis, N., Misra, A., and Sekar, R. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, Association for Computing Machinery, p. 135–144.

[93] Hastings, R., and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference* (1991), pp. 125–138.

[94] Hat, R. Ghost: glibc vulnerability (cve-2015-0235). `https://access.redhat.com/articles/1332213`, 2015. Accessed 27-09-2021.

[95] Homomorphic Encryption. `https://homomorphicencryption.org/`.

[96] The heartbleed bug. `https://heartbleed.com/`, 2020. Accessed 27-09-2021.

[97] Homomorphic Encryption. https://chain.link/education-hub/homomorphic-encryption.

[98] Honda, M., Lettieri, G., Eggert, L., and Santry, D. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), pp. 17–33.

[99] 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.

[100] HUANG, H., HUANG, K., YOU, L., AND HUANG, L. Forca: Fast and atomic remote direct access to persistent memory. In *2018 IEEE 36th International Conference on Computer Design (ICCD)* (2018), pp. 246–249.

[101] The Kernel Address Sanitizer (KASAN). `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html#hardware-tag-based-kasan`, 2021. Accessed 27-09-2021.

[102] Intel Corporation. Attestation Service for Intel Software GuardExtensions (Intel SGX): API Documentation. `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`.

[103] IBM. Confidential computing on IBM Cloud. `https://www.ibm.com/cloud/confidential-computing`, 2021.

[104] IBM. Analytics infrastructure: High-performance i/o architecture. `https://www.zurich.ibm.com/cci/analytics/crail.html?lnk=hm`, December 4, 2024.

[105] INFRADEAD. Direct Access for files. `https://www.infradead.org/~mchehab/kernel_docs/filesystems/dax.html`, 2021.

[106] INTEL. Intel optane technology. `https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/`.

[107] INTEL. Software Enabling for Intel® TDX in Support of TEE-I/O. `https://cdrdv2-public.intel.com/742542/software-enabling-for-tdx-tee-io-fixed.pdf`.

[108] INTEL. Persistent Memory Replication Over Traditional RDMA Part 1: Understanding Remote Persistent Memory. `https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html`, 2019.

[109] INTEL. An introduction to pmemobj (part 4) - transactional dynamic memory allocation, 2021. Accessed 27-09-2021.

[110] INTEL. An introduction to pmemobj (part 5) - atomic dynamic memory allocation, 2021. Accessed 27-09-2021.

[111] INTEL. Benchmarking tools for pmemkv, 2021. Accessed 27-09-2021.

[112] INTEL. Documentation for ndctl and daxctl, 2021. Accessed 27-09-2021.

[113] INTEL. Intel® Optane™ Persistent Memory. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`, 2021.

[114] INTEL. Persistent Memory Development Kit. `https://pmem.io/pmdk/`, 2021.

[115] INTEL. Persistent Memory Development Kit : libpmemobj - persistent memory transactional object store. `https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html`, 2021.

[116] INTEL. Persistent Memory Development Kit : librpmem - remote persistent memory support library. `https://pmem.io/pmdk/manpages/linux/master/librpmem/librpmem.7.html`, 2021.

[117] INTEL. Persistent Memory Development Kit : pmemkv, 2021. Accessed 27-09-2021.

[118] INTEL. Persistent Memory Development Kit : The libpmemblk library. `https://pmem.io/pmdk/libpmemblk/`, 2021.

[119] INTEL. Persistent Memory Development Kit : The libpmemlog library. `https://pmem.io/pmdk/libpmemlog/`, 2021.

[120] INTEL. Persistent Memory Development Kit : The libpmemobj library. `https://pmem.io/pmdk/libpmemobj/`, 2021.

[121] INTEL. Persistent Memory Development Kit : The libpmempool library, 2021. Accessed 27-09-2021.

[122] INTEL. Persistent Memory Development Kit : The libvmem library, 2021. Accessed 27-09-2021.

[123] INTEL. Persistent Memory Development Kit : The libvmmalloc library, 2021. Accessed 27-09-2021.

[124] INTEL. Persistent Memory Development Kit : The pmempool utility, 2021.

[125] INTEL. PMDK Internals: Important Algorithms and Data Structures. `https://link.springer.com/chapter/10.1007/978-1-4842-4932-1_16`, 2021.

[126] INTEL. pmembench: PMDK benchmark framework. https://github.com/pmem/pmdk/blob/master/src/benchmarks/pmembench.cpp, 2021.

[127] INTEL. The Challenge of Keeping Up with Data, 2021. Accessed 27-09-2021.

[128] INTEL. Discover Persistent Memory Programming Errors with Pmemcheck, 2022.

[129] INTEL. Persistent Memory Development Kit : The C++ bindings to libpmemobj, 2022. December 4, 2024.

[130] INTEL. Persistent Memory Development Kit : the libpmemobj array example, 2022. December 4, 2024.

[131] INTEL. Persistent Memory Development Kit : The libpmemobj examples, 2022. December 4, 2024.

[132] INTEL. Intel® Trust Domain Extensions (Intel® TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[133] INTEL. The librpma library. https://pmem.io/rpma/, 2023.

[134] INTEL. Can intel® sgx and intel® optane™ persistent memory 200 series be used on the same platform? https://www.intel.com/content/www/us/en/support/articles/000059500/memory-and-storage/intel-optane-persistent-memory.html, December 4, 2024.

[135] INTEL. Intel sgx + dc persistent memory. https://community.intel.com/t5/Intel-Software-Guard-Extensions/Intel-SGX-DC-Persistent-Memory/td-p/1286085?profile.language=en, December 4, 2024.

[136] INTEL. Leveraging rdma technologies to accelerate ceph* storage solutions. https://www.intel.com/content/www/us/en/developer/articles/technical/leveraging-rdma-technologies-to-accelerate-ceph-storage-solutions.html, December 4, 2024.

[137] Intel® Inspector: Deliver reliable applications. Locate and debug threading, memory, and persistent memory errors early in the design cycle to avoid costly errors later. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html#gs.cdp2vf, 2021. Accessed 27-09-2021.

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

[139] Intel® Trust Authority. https://docs.trustauthority.intel.com/main/articles/introduction.html.

[140] Intel software guard extensions remote attestation end-to-end example. `https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html`, December 4, 2024. Last accessed: December 4, 2024.

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

[142] Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing* (NY, NY, USA, 2016), ICS '16, ACM.

[143] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., Zhao, J., and Swanson, S. Basic performance measurements of the intel optane DC persistent memory module. *CoRR abs/1903.05714* (2019).

[144] Jenkins, L., and Scott, M. L. Persistent Memory Analysis Tool (PMAT), 2022.

[145] 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).

[146] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. Cyclone: A safe dialect of c. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)* (Monterey, CA, June 2002), USENIX Association.

[147] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging* (1997).

[148] Jérôme Glisse / Google. CXL Confidential Computing. `https://lpc.events/event/16/contributions/1250/attachments/1125/2158/LPC2022%20CXL%20Confidential%20Computing.pdf`, 2023.

[149] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (NY, NY, USA, 2019), SOSP '19, ACM, p. 494–508.

[150] Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., and ri Choi, Y. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, February 2019), USENIX Association, pp. 191–205.

[151] Kalbfleisch, S., Werling, L., and Bellosa, F. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 933–950.

[152] Kalia, A., Andersen, D., and Kaminsky, M. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 105–119.

[153] 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).

[154] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (NY, NY, USA, 2014), SIGCOMM '14, ACM, p. 295–306.

[155] 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.

[156] 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).

[157] Kalita, C., Barua, G., and Sehgal, P. Durablefs: A file system for persistent memory. *CoRR abs/1811.00757* (2018).

[158] Kalita, C., Barua, G., and Sehgal, P. Durablefs: A file system for persistent memory. *ArXiv abs/1811.00757* (2018).

[159] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 993–1005.

[160] Kashyap, S., Qin, D., Byan, S., Marathe, V. J., and Nalli, S. Correct, fast remote persistence. *CoRR abs/1909.02092* (2019).

[161] kernel archives, T. L. DAX - Direct access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`, 2021.

[162] Keystone Enclave - SDK Overview. https://docs.keystone-enclave.org/en/latest/Keystone-Applications/SDK-Basics.html.

[163] Kim, I., Kim, J. H., Chung, M., Moon, H., and Noh, S. H. A Log-Structured merge tree-aware message authentication scheme for persistent Key-Value stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 363–380.

[164] 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).

[165] Kim, W., Park, C., Kim, D., Park, H., ri Choi, Y., Sussman, A., and Nam, B. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 161–177.

[166] Kim, Y., Lee, J., and Kim, H. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), pp. 1153–1166.

[167] Cxl software ecosystem. `KioxiaLaunchesSecondGenerationofHigh-Performance,Cost-EffectiveXL-FLASHStorageClassMemorySolution`.

[168] 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).

[169] Kozyrakis, C. Phoenix benchmark overflow bug location. `https://github.com/kozyraki/phoenix/blob/1276c8d8f3b82050071d0a7a4b8a352a05d1faab/phoenix-2.0/tests/string_match/string_match.c#L158`. December 4, 2024.

[170] Kroes, T., Koning, K., van der Kouwe, E., Bos, H., and Giuffrida, C. Delta pointers: buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[171] 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* (NY, NY, USA, 2016), EuroSys '16, ACM.

[172] 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 Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 205–221.

[173] Kuvaiskii, D., Stavrakakis, D., Qin, K., Xing, C., Bhatotia, P., and Vij, M. Gramine-tdx: A lightweight os kernel for confidential vms. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security* (2024), CCS '24, Association for Computing Machinery. Paper under shepherding.

[174] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The objectstore database system. *Communications of the ACM 34*, 10 (1991), 50–63.

[175] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (USA, 2004), CGO '04, IEEE Computer Society, p. 75.

[176] Lee, B. C., Zhou, P., Yang, J., Zhang, Y., Zhao, B., Ipek, E., Mutlu, O., and Burger, D. Phase-change technology and the future of main memory. *IEEE Micro 30*, 1 (2010), 143–143.

[177] 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).

[178] Lee, S. K., Mohan, J., Kashyap, S., Kim, T., and Chidambaram, V. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (NY, NY, USA, 2019), SOSP '19, ACM, p. 462–477.

[179] LEIS, V., HAUBENSCHILD, M., KEMPER, A., AND NEUMANN, T. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 185–196.

[180] The libpmem webpage. `https://pmem.io/pmdk/libpmem/`, 2021. Accessed: 2021-02-27.

[181] LIN, Z., YU, Z., GUO, Z., CAMPANONI, S., DINDA, P., AND XING, X. CAMP: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)* (Philadelphia, PA, August 2024), USENIX Association.

[182] LING, H., HUANG, H., WANG, C., CAI, Y., AND ZHANG, C. Giantsan: Efficient memory sanitization with segment folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)* (2024).

[183] 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).

[184] LIU, S., KOLLI, A., REN, J., AND KHAN, S. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), pp. 310–323.

[185] LIU, S., MAHAR, S., RAY, B., AND KHAN, S. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (NY, NY, USA, 2021), ASPLOS 2021, ACM, p. 487–502.

[186] LIU, S., SEEMAKHUPT, K., WEI, Y., WENISCH, T., KOLLI, A., AND KHAN, S. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (NY, NY, USA, 2020), ASPLOS '20, ACM, p. 1187–1202.

[187] LIU, S., WEI, Y., ZHAO, J., KOLLI, A., AND KHAN, S. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (NY, NY, USA, 2019), ASPLOS '19, ACM, p. 411–425.

[188] The LLVM gold plugin. `https://llvm.org/docs/GoldPlugin.html`, 2022. Accessed 20-07-2022.

[189] 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.

[190] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: Trading address space for reliability and security. *SIGOPS Oper. Syst. Rev. 42*, 2 (mar 2008), 115–124.

[191] 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).

[192] MALVENTANO, P. A. How 3D XPoint Phase-Change Memory Works. `https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works/`, 2017.

[193] MARATHE, V. J., SELTZER, M., BYAN, S., AND HARRIS, T. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, July 2017), USENIX Association.

[194] 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).

[195] MATSAKIS, N. D., AND KLOCK, F. S. The rust language. *Ada Lett. 34*, 3 (oct 2014), 103–104.

[196] 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).

[197] MEMARIAN, K., MATTHIESEN, J., LINGARD, J., NIENHUIS, K., CHISNALL, D., WATSON, R. N. M., AND SEWELL, P. Into the depths of c: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, Association for Computing Machinery, p. 1–15.

[198] Memcheck: a memory error detector. `https://valgrind.org/docs/manual/mc-manual.html`, 2021. Accessed 27-09-2021.

[199] Memhive: Scale applications with persistent memory! `https://www.memhive.io/`, 2021. Accessed 27-09-2021.

[200] 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* (NY, NY, USA, 2021), Middleware '21, ACM, p. 1–13.

[201] MICRON. Nvdimm. `https://www.micron.com/products/dram-modules/nvdimm`, December 4, 2024.

[202] MICROSOFT, J. S. Always Encrypted with secure enclaves now generally available in Azure SQL Database. `https://techcommunity.microsoft.com/t5/azure-sql/always-encrypted-with-secure-enclaves-now-generally-available-in/ba-p/2502560`, 2021.

[203] MICROSOFT AZURE. Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/`.

[204] MISONO, M., STAVRAKAKIS, D., SANTOS, N., AND BHATOTIA, P. Confidential vms explained: An empirical analysis of amd sev-snp and intel tdx.

[205] 2023 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html`, 2023. Accessed: 21-07-2024.

[206] A proactive approach to more secure code. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`, 2019. Accessed 27-09-2021.

[207] 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).

[208] 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.

[209] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)* (2012), pp. 189–200.

[210] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, Association for Computing Machinery, p. 175–184.

[211] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)* (Dagstuhl, Germany, 2015), T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., vol. 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 190–208.

[212] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, Association for Computing Machinery, p. 245–258.

[213] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management* (New York, NY, USA, 2010), ISMM '10, Association for Computing Machinery, p. 31–40.

[214] Nam, M., Cha, H., ri Choi, Y., Noh, S. H., and Nam, B. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, February 2019), USENIX Association, pp. 31–44.

[215] Nam, M. J. Inline and sideline approaches for low-cost memory safety in C. Tech. Rep. UCAM-CL-TR-954, University of Cambridge, Computer Laboratory, February 2021.

[216] Nam, M. J., Akritidis, P., and Greaves, D. J. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th*

Annual Computer Security Applications Conference (New York, NY, USA, 2019), ACSAC '19, Association for Computing Machinery, p. 612–626.

[217] NEAL, I., QUINN, A., AND KASIKCI, B. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (NY, NY, USA, 2021), ASPLOS 2021, ACM, p. 401–414.

[218] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst. 27*, 3 (May 2005), 477–526.

[219] NETAPP. What is persistent memory? `https://www.netapp.com/knowledge-center/what-is-persistent-memory/`, 2021.

[220] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

[221] What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel® Xeon® Scalable Processor Platforms? https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html.

[222] Pmdk - unit test for pool recovery. `https://github.com/pmem/pmdk/blob/stable-1.8/src/test/obj_recovery/obj_recovery.c`.

[223] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., FELBER, P., AND FETZER, C. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR abs/1702.00719* (2017).

[224] 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).

[225] OpenSSL library. https://openssl.org. Last accessed: Jan, 2021.

[226] 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).

[227] PANDEY, S., KAMATH, A. K., AND BASU, A. *GPM: Leveraging Persistent Memory from a GPU*. Association for Computing Machinery, New York, NY, USA, 2022, p. 142–156.

[228] Park, S., Lee, S., Xu, W., Moon, H., and Kim, T. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 241–254.

[229] Parno, B., Lorch, J., Douceur, J. J., Mickens, J., and McCune, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011), IEEE.

[230] Pci express 6.0 specification. `https://pcisig.com/pci-express-6.0-specification`.

[231] What's new in the pcie 6.0 specification: Bandwidth & security. `https://www.synopsys.com/blogs/chip-design/whats-new-in-pcie-6.html`.

[232] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2017), SLE 2017, Association for Computing Machinery, p. 256–267.

[233] Heap overflow bug in the string_match benchmark of the phoenix benchmark suite. `https://github.com/kozyraki/phoenix/issues/9`, 2024. December 4, 2024.

[234] Suggested fix for the heap overflow bug in the string_match benchmark of the phoenix benchmark suite. `https://github.com/kozyraki/phoenix/pull/10`, 2024. December 4, 2024.

[235] GitHub issue on the missing abort. XXX, 2021. Accessed 05-10-2021.

[236] GitHub issue on the btree overflow. XXX, 2021. Accessed 05-10-2021.

[237] An introduction to pmemobj (part 2) - transactions. https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-2-transactions/.

[238] pmem-valgrind. `https://github.com/pmem/valgrind`, 2021. Accessed 27-09-2021.

[239] The pmdk webpage. `https://pmem.io/pmdk/`, 2021. Accessed: 2021-02-27.

[240] Valgrind: an enhanced version for pmem. `https://github.com/pmem/valgrind`.

[241] pmemkv engines. `https://github.com/pmem/pmemkv/blob/master/doc/libpmemkv.7.md#cmap`, 2022. Accessed 04-07-2022.

[242] pmemobj API. `https://pmem.io/pmdk/manpages/linux/v1.1/libpmemobj.3/`, 2022. Accessed 04-07-2022.

[243] AN INTRODUCTION TO PMEMOBJ (PART 3) - TYPES. `https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-3-types/`, 2021. Accessed 22-07-2022.

[244] The pmreorder utility. `https://pmem.io/pmdk/pmreorder/`.

[245] Five-level page tables. `https://lwn.net/Articles/717293/`, 2022. Accessed 04-07-2022.

[246] 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).

[247] 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.

[248] PRIEBE, C., VASWANI, K., AND COSTA, M. EnclaveDB: A Secure Database using SGX (S&P). In *IEEE Symposium on Security and Privacy* (2018).

[249] Code Sample: Intel® Software Guard Extensions Remote Attestation End-to-End Example. https://www.intel.com/content/www/us/en/developer/articles/code-sample/software-guard-extensions-remote-attestation-end-to-end-example.html.

[250] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), pp. 135–148.

[251] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007), pp. 13–24.

[252] Pmdk - object and pool recovery tests. `https://github.com/pmem/pmdk/tree/stable-1.8/src/test/obj_recovery`.

[253] Pmem-Redis. `https://github.com/pmem/pmem-redis`, 2021. Accessed 27-09-2021.

[254] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTLU, O. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (NY, NY, USA, 2015), MICRO-48, ACM, p. 672–685.

[255] A 64-bit port of the RIPE benchmark. `https://github.com/hrosier/ripe64.git`, 2019. Accessed 27-09-2021.

[256] pmem-rocksdb. `https://github.com/pmem/pmem-rocksdb`, 2021. Accessed 27-09-2021.

[257] ROTHENBERGER, B., TARANOV, K., PERRIG, A., AND HOEFLER, T. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (August 2021), USENIX Association, pp. 4277–4292.

[258] RUAN, C., ZHANG, Y., BI, C., MA, X., CHEN, H., LI, F., YANG, X., LI, C., ABOULNAGA, A., AND XU, Y. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 498–512.

[259] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium* (2004), NDSS.

[260] Samsung CXL Solutions – CMM-H. https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.

[261] 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.

[262] 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).

[263] SCARGALL, S. *Debugging Persistent Memory Applications*. Apress, Berkeley, CA, 2020, pp. 207–260.

[264] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association, pp. 309–318.

[265] Serebryany, K., Kennelly, C., Phillips, M., Denton, M., Elver, M., Potapenko, A., Morehouse, M., Tsyrklevich, V., Holler, C., Lettner, J., Kilzer, D., and Brandt, L. Gwp-asan: Sampling-based detection of memory-safety bugs in production, 2024.

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

[267] Intel® Software Guard Extensions SDK for Linux* OS. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html.

[268] Shan, Y., Tsai, S.-Y., and Zhang, Y. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (NY, NY, USA, 2017), SoCC '17, ACM, p. 323–337.

[269] Shapiro, J. S., Smith, J. M., and Farber, D. J. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles* (1999), pp. 170–185.

[270] Shi, J. Exadata with Persistent Memory:An Epic Journey. `https://www.snia.org/educational-library/exadata-persistent-memoryan-epic-journey-2020`.

[271] Shilov, A. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. December 4, 2024.

[272] 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).

[273] Shu, J., Chen, Y., Wang, Q., Zhu, B., Li, J., and Lu, Y. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage 16*, 4 (October 2020).

[274] Simpson, A. K., Szekeres, A., Nelson, J., and Zhang, I. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2020).

[275] SIMPSON, M. S., AND BARUA, R. K. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation* (2010), pp. 199–208.

[276] NVM Programming Model (NPM). https://www.snia.org/tech_activities/standards/curr_standa

[277] 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).

[278] SOLUTIONS, P. Leveraging Persistent Memory for Real-Time Analytics Workloads. `https://www.penguinsolutions.com/company/resources/newsroom/leveraging-persistent-memory-for-real-time-analytics-workloads`.

[279] SONG, X. K., LIU, S., AND PEKHIMENKO, G. Persistent memory – a new hope. `https://www.sigarch.org/persistent-memory-a-new-hope/#:~:text=Recently%2C%20Intel%20announced%20the%20cancellation,the%20consumer%2Dgrade%20Optane%20SSDs.`, December 4, 2024.

[280] Hardware-Assisted Checking Using Silicon Secured Memory (SSM). `https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html`, 2021. Accessed 27-09-2021.

[281] Security protocol and data model (spdm) specification. `https://www.dmtf.org/dsp/DSP0274`.

[282] Positioning of SPP LTO pass in LLVM pipeline. `https://github.com/dimstav23/llvm-project/blob/965cc46a7b34a60d6114698c311aa99d459ee726/llvm/lib/Transforms/IPO/PassManagerBuilder.cpp#L539`, 2024. December 4, 2024.

[283] Positioning of SPP transformation pass in LLVM pipeline. `https://github.com/dimstav23/llvm-project/blob/965cc46a7b34a60d6114698c311aa99d459ee726/llvm/lib/Transforms/SPP/spp.cpp#L560C20-L560C63`, 2024. December 4, 2024.

[284] Poject zero - stagefrightened? `https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html`, 2015. Accessed 27-09-2021.

[285] STAVRAKAKIS, D., GIANTSIDI, D., BAILLEU, M., SÄNDIG, P., ISSA, S., AND BHATOTIA, P. Anchor: A library for building secure persistent memory systems. *Proc. ACM Manag. Data 1*, 4 (dec 2023).

[286] STAVRAKAKIS, D., PANFIL, A., NAM, M., AND BHATOTIA, P. Spp: Safe persistent pointers for memory safety. In *2024 The 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (6 2024), IEEE Computer Society.

[287] 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.

[288] SUN, Y., WANG, S., LI, H., AND LI, F. Building enclave-native storage engines for practical encrypted databases. *Proc. VLDB Endow. 14*, 6 (February 2021), 1019–1032.

[289] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. *2013 IEEE Symposium on Security and Privacy* (2013), 48–62.

[290] Tamarin tls handshake proof. `https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/classic/TLS_Handshake.spthy`.

[291] TARANOV, K., ROTHENBERGER, B., PERRIG, A., AND HOEFLER, T. srdma – efficient nic-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 691–704.

[292] Tee device interface security protocol (tdisp). `https://pcisig.com/tee-device-interface-security-protocol-tdisp`.

[293] 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).

[294] Trusted Computing Group. `https://trustedcomputinggroup.org/`.

[295] 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).

[296] ARM Security Technology Building a Secure System using TrustZone Technology. https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Software-Architecture/The-TrustZone-API.

[297] 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).

[298] Type safety MACROS in libpmemobj. `https://pmem.io/blog/2015/06/type-safety-macros-in-libpmemobj/`, 2022. Accessed 04-07-2022.

[299] Unterguggenberger, M., Schrammel, D., Lamster, L., Nasahl, P., and Mangard, S. Cryptographically enforced memory safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 889–903.

[300] Iterating over def-use & use-def chains. `https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains`. December 4, 2024.

[301] Enable faster network packet processing with DPDK. https://cloud.google.com/compute/docs/networking/use-dpdk#gcloud.

[302] Alibaba Builds High-Speed RDMA Network for AI and Scientific Computing. https://www.alibabacloud.com/blog/alibaba-builds-high-speed-rdma-network-for-ai-and-scientific-computing_594895.

[303] Coroutine Made DPDK Development Easy. https://www.alibabacloud.com/blog/coroutine-made-dpdk-development-easy_599986.

[304] RDMA for Cloud Computing. https://www.microsoft.com/en-us/research/project/rdma-for-cloud-computing/.

[305] Set up DPDK in a Linux virtual machine on Azure. https://learn.microsoft.com/en-us/azure/virtual-network/setup-dpdk?tabs=redhat.

[306] Enhancing Network Performance with RDMA on Microsoft Azure Network Adapter. https://www.snia.org/educational-library/enhancing-network-performance-rdma-microsoft-azure-network-adapter-2023.

[307] 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).

[308] van der Veen, V., dutt Sharma, N., Cavallaro, L., and Bos, H. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions,*

and Defenses (Berlin, Heidelberg, 2012), D. Balzarotti, S. J. Stolfo, and M. Cova, Eds., Springer Berlin Heidelberg, pp. 86–106.

[309] van Schaik, S., Kwong, A., Genkin, D., and Yarom, Y. SGAxe: How SGX fails in practice. `https://sgaxeattack.com/`, 2020.

[310] van Schaik, S., Minkin, M., Kwong, A., Genkin, D., and Yarom, Y. Cache-Out: Leaking Data on Intel CPUs via Cache Evictions, 2020.

[311] 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).

[312] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.

[313] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. *SIGARCH Comput. Archit. News 39*, 1 (March 2011), 91–104.

[314] Wang, C., He, K., Fan, R., Wang, X., Kong, Y., Wang, W., and Hao, Q. Cxl over ethernet: A novel fpga-based memory disaggregation design in data centers, 2023.

[315] Wang, J., and Zhang, Q. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data* (New York, NY, USA, 2023), SIGMOD '23, Association for Computing Machinery, p. 37–44.

[316] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (USA, 2018), USENIX ATC '18, USENIX Association, p. 133–145.

[317] Wang, T., Sambasivam, S., Solihin, Y., and Tuck, J. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017* (2017), H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, Eds., ACM, pp. 800–812.

[318] 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).

[319] 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).

[320] WILANDER, J., NIKIFORAKIS, N., YOUNAN, Y., KAMKAR, M., AND JOOSEN, W. Ripe: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, Association for Computing Machinery, p. 41–50.

[321] WILL, N. C., AND MAZIERO, C. A. Intel software guard extensions applications: A survey. *ACM Comput. Surv. 55*, 14s (jul 2023).

[322] WOODRUFF, J., WATSON, R. N., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News 42*, 3 (June 2014), 457–468.

[323] WU, Y., PARK, K., SEN, R., KROTH, B., AND DO, J. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (NY, NY, USA, 2020), DaMoN '20, ACM.

[324] XIA, F., JIANG, D., XIONG, J., AND SUN, N. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 349–362.

[325] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, February 2016), USENIX Association, pp. 323–338.

[326] XU, S., HUANG, W., AND LIE, D. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, p. 224–240.

[327] 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 (Oakland)* (2015).

[328] YANG, F., CHEN, Y., MAO, H., LU, Y., AND SHU, J. Shieldnvm: An efficient and fast recoverable system for secure non-volatile memory. *ACM Trans. Storage 16*, 2 (May 2020).

[329] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, February 2020), USENIX Association, pp. 111–125.

[330] YCSB. `https://github.com/brianfrankcooper/YCSB`. Last accessed: Jan, 2021.

[331] YE, D., SU, Y., SUI, Y., AND XUE, J. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (2014), pp. 88–99.

[332] YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Computing Surveys - CSUR 44* (06 2012), 1–28.

[333] YOUNG, V., NAIR, P. J., AND QURESHI, M. K. Deuce: Write-efficient encryption for non-volatile memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (NY, NY, USA, 2015), ASPLOS '15, ACM, p. 33–44.

[334] YU, J. Z., WATT, C., BADOLE, A., CARLSON, T. E., AND SAXENA, P. Capstone: A capability-based foundation for trustless secure memory access. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, August 2023), USENIX Association, pp. 787–804.

[335] ZACARIAS, F. V., CARPENTER, P., AND PETRUCCI, V. Memory demands in disaggregated hpc: How accurate do we need to be? In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (2021), pp. 1–6.

[336] ZHANG, L. *Building Reliable Software for Persistent Memory*. PhD thesis, UC San Diego, 2019.

[337] ZHANG, L., AND SWANSON, S. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (USA, 2019), USENIX ATC '19, USENIX Association, p. 897–911.

[338] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, February 2022), USENIX Association, pp. 51–68.

[339] ZHANG, W., ZHAO, X., JIANG, S., AND JIANG, H. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (NY, NY, USA, 2021), EuroSys '21, ACM, p. 194–209.

[340] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. *SIGARCH Comput. Archit. News 43*, 1 (March 2015), 3–18.

[341] ZHOU, D., QIAN, Y., GUPTA, V., YANG, Z., MIN, C., AND KASHYAP, S. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 179–193.

[342] ZUO, P., AND HUA, Y. Secpm: a secure and persistent memory system for non-volatile memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, July 2018), USENIX Association.

[343] ZUO, P., HUA, Y., AND XIE, Y. Supermem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (NY, NY, USA, 2019), MICRO '52, ACM, p. 479–492.