Anchor: A Library for Building Secure Persistent Memory Systems

DIMITRIOS STAVRAKAKIS, TU Munich & University of Edinburgh, Germany & United Kingdom
DIMITRA GIANTSIDI, University of Edinburgh, United Kingdom
MAURICE BAILLEU, University of Edinburgh, United Kingdom
PHILIP SÄNDIG, TU Munich, Germany
SHADY ISSA, TU Munich, Germany
PRAMOD BHATOTIA, TU Munich, Germany

Cloud infrastructure is experiencing a shift towards disaggregated setups, especially with the introduction of the Compute Express Link (CXL) technology, where byte-addressable persistent memory (PM) is becoming prominent. To fully utilize the potential of such devices, it is a necessity to access them through network stacks with equivalently high levels of performance (e.g., kernel-bypass, RDMA). While, these advancements are enabling the development of high-performance data management systems, their deployment on untrusted cloud environments also increases the security threats.

To this end, we present 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 reasonable overheads, while providing strong security properties.

CCS Concepts: • **Security and privacy → Systems security**; **Tamper-proof and tamper-resistant designs**; • **Information systems → Data management systems**; **Information storage systems**; • **Hardware → Memory and dense storage**; • **Networks → Network security**.

Additional Key Words and Phrases: Persistent memory; Dependable systems; Trusted execution environments; Secure data management systems; Secure kernel-bypass networking

Authors' addresses: Dimitrios Stavrakakis, TU Munich & University of Edinburgh, Munich & Edinburgh, Germany & United Kingdom, dimitrios.stavrakakis@tum.de; Dimitra Giantsidi, University of Edinburgh, Edinburgh, United Kingdom, d.giantsidi@sms.ed.ac.uk; Maurice Bailleu, University of Edinburgh, Edinburgh, United Kingdom, M.Bailleu@ed.ac.uk; Philip Sändig, TU Munich, Munich, Germany, philip.saendig@tum.de; Shady Issa, TU Munich, Munich, Germany, shadyalaa@gmail.com; Pramod Bhatotia, TU Munich, Munich, Germany, pramod.bhatotia@cit.tum.de.

2836-6573/2023/12-ART231 $15.00
https://doi.org/10.1145/3626718

ACM Reference Format:
Dimitrios Stavrakakis, Dimitra Giantsidi, Maurice Bailleu, Philip Sändig, Shady Issa, and Pramod Bhatotia. 2023. Anchor: A Library for Building Secure Persistent Memory Systems. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 231 (December 2023), 31 pages. https://doi.org/10.1145/3626718

1 INTRODUCTION

Cloud storage and networking infrastructure is going through a dramatic shift to favor the design of modern disaggregated data management systems [44, 52, 77, 81, 139], especially with the recent introduction of the Compute Express Link (CXL) technology [36]. On the storage front, byte-addressable Persistent Memory (PM) aims to bridge the gap between volatile main memory and SSDs [57, 90, 105], providing opportunities for high-volume pools of low-latency non-volatile memory. Similarly, on the networking front, kernel-bypass I/O based on RDMA [49, 80] or DPDK [5] offers superior throughput and low latency [29, 73, 78], and is necessary to efficiently use byte-addressable storage in disaggregated system setups. To leverage these hardware advancements, the research community is actively working on combining PM with kernel-bypass networking to build high-performance storage systems [30, 99, 121, 124].

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 [117, 118]. Prior work has shown that software bugs, configuration errors and security vulnerabilities pose a real threat for storage systems [39, 45, 47, 88, 118].

In the context of PM-based systems, attackers can tamper 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 [106]. Based on this promise, TEEs are now available in all major commodity CPUs [1, 7, 19, 21, 66, 91], and are offered by major cloud providers [35, 48, 54, 107].

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 the 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 [3, 73]—especially in the context of TEEs due to switches between the trusted and untrusted world [25, 132]. While direct I/O vastly optimizes network operations, it is incompatible with TEEs as untrusted DMA operations are prohibited in the protected memory [25]. On top of that, ensuring security and crash consistency when accessing PM via RDMA is another major challenge [77]. 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 [62]. 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.

Overall, we make the following contributions.

- **Secure data management APIs (§ 4):** We expose generic APIs for secure data management within the realms of the established PM programming model [62], 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 (§ 5):** 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 (§7.2).
- **System operations (§ 6):** 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 (§7.2), for ensuring crash consistency and data freshness.

Based on these contributions, we implement a prototype leveraging Intel SGX [7], and integrate with our PM engine based on PMDK [62] and secure network stack based on eRPC [78], a direct I/O networking library. We evaluate Anchor with the YCSB benchmark suite [15, 37]. Our evaluation shows that Anchor incurs reasonable overheads considering its strong security properties.

2 BACKGROUND

2.1 Disaggregated Systems & Persistent Memory

Disaggregated cloud systems, where compute, memory and storage resources are decoupled, benefit from the use of high-speed interconnects, such as the newly introduced CXL [36], to provide low-latency, cache-coherent access to byte-addressable storage, like PM, allowing for efficient data access and processing in a scalable and flexible manner. For such systems, high-speed networking [43, 59, 77, 144] 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., PM.

PM is byte-addressable and can be accessed via *ld*/*st* instructions with performance properties close to DRAM, while ensuring durability [60, 101]. Existing PM technology primarily interfaces with the OS in the "app direct mode" [61] via PM-aware direct access file systems (DAX) [56, 85]. However, PM is susceptible to two important issues in case of system failures, which can lead to inconsistent state [115]: *(i)* atomicity is not guaranteed for updates larger than 8 B; and *(ii)* cache lines can be written back to PM out-of-order.

To address these issues, PMDK [62] offers the *libpmemobj* library. *libpmemobj* contains a PM allocator [64] and implements software-based transactions to ensure crash consistency. *libpmemobj* maps a PM-file into a contiguous region in the application's virtual address space. This file is called *PM pool* and contains a metadata header, the transaction logs and the persistent heap.

In particular, *libpmemobj* exposes a transactional API [63] with durability, consistency and atomicity semantics. PMDK transactions do not provide data isolation; applications need to resolve any data races themselves. 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 can replay any live redo/undo logs to recover PM to a consistent state.

To enable developers to build end-to-end systems, PMDK further provides networking support through *librpma* [67]. It allows for accessing remote PM over Remote Direct Memory Access (RDMA). Importantly, high-performance networking based on kernel-bypass abstractions, such as DPDK [5] and RDMA [49, 80], eschews the OS and alleviates any bottleneck in the kernel network stack by directly interacting with the NIC hardware. To simplify kernel-bypass network programming, remote procedure call (RPC) frameworks such as eRPC [78] provide a general, yet performant, API for asynchronous RPCs while hiding the complexities of managing the low-level transport layer interfaces. RDMA-based RPCs have been demonstrated through research and industry efforts [20, 44, 55, 70] to be the most efficient programming paradigm for high-performing cloud systems that incorporate byte-addressable storage [31, 67, 72].

2.2 Confidential Computing

Trusted Execution Environments (TEEs) [7, 21, 91] provide a hardware protected *enclave* that ensures the security of code and data residing in this isolated volatile memory region. Additionally, there exists Virtual Machine (VM)-Based TEEs, such as AMD SEV [19] and Intel TDX [66]. They offer a different layer of protection by creating a virtualized secure environment within each VM instance, which encrypts data-in-use and data-in-transit, and further ensures the isolation of sensitive data and applications.

In Anchor, we build on Intel SGX [7], a set of x86 ISA extensions for TEEs. The enclave memory in SGX is mapped in the physical memory as *Enclave Page Cache (EPC)*, where the pages are protected by an on-chip Memory Encryption Engine (MEE). However, the EPC is limited to 128 MiB—256 MiB for SGX-v1/v2. To accommodate larger enclaves, SGX offers secure paging; however, it incurs prohibitive overheads (up to 2000× [22]). In the context of Anchor, we need to consider two

important architectural aspects of SGX. Firstly, normal syscall-based I/O operations require an expensive (5×) world switch [131]. Secondly, while SGX offers a hardware trusted monotonic counter [6], it is extremely slow (60 −250 m s) and can wear out after some days of continuous use [102].

Anchor is built on SCONE [22], a shielded execution framework that leverages SGX [22, 27, 111, 113, 123, 131]. SCONE links the application against a modified libc version confining its address space inside the secure enclave memory region.

3 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 [42] 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 attacks 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 [50, 87, 94, 108, 133–135, 142].

Fault model. Anchor mandates crash consistency [38, 89, 122], 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 [53, 59, 77].

Programming model. Anchor offers a transactional programming model based on PMDK [62]. 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 [63] 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 [127], byte-addressable storage devices should share a similar programming model. Especially due the current emergence of the Compute Express Link (CXL) technology [36], there will be an upsurge in vendors providing non-volatile memory devices. On top of that, CXL will be combined with Confidential Computing [40, 58, 74] to build end-to-end secure systems. In such setups, Anchor can be used to securely manage PM devices due to their compatibility with the existing PM concepts and libraries [2].

4 OVERVIEW

4.1 System Overview

Anchor offers a PM library with the following properties:

- *Security*: Anchor ensures the confidentiality, integrity, authenticity and freshness for 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 PMDK programming model.

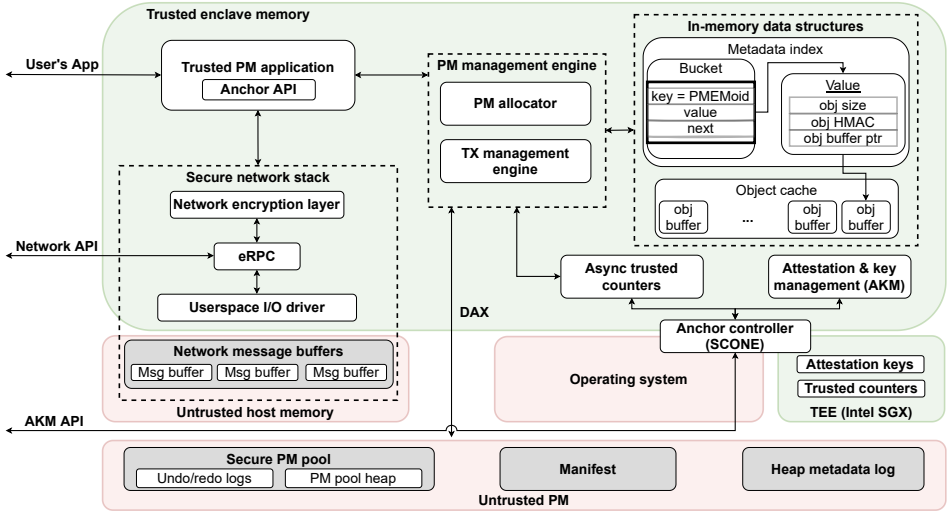

Fig. 1. System overview *(green regions are trusted and red regions are untrusted)*

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 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 data (red regions) resides in the untrusted host memory and PM.

Anchor's core component, the PM management engine (§ 5.2), 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 (§ 5.5). We also design in-memory data structures (§ 5.3), consisting of an index and an object cache optimizing the read path. Anchor also exposes remote access to PM through a secure network stack (§ 5.4). Lastly, we offer a remote attestation and key management (AKM) service for clients to ensure the trustworthiness and authenticity (§ 5.6).

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 enclave to generate a signed measure of its identity, whose authenticity is verified by a trusted third party entity [4]. 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. Afterwards, 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 (§ 6.2).

Pool mgmt. 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 mgmt. 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 1. Anchor's secure data management APIs

4.2 Anchor System APIs

Anchor exposes secure data management APIs (shown in Table 1) by adopting and extending the well-established APIs of PMDK [62]. 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.

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*). Afterwards, 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's programming model, 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 [78]. 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) [4] 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.

4.3 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/wite 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, 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 [63] 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 favour of latest trusted and correct content.

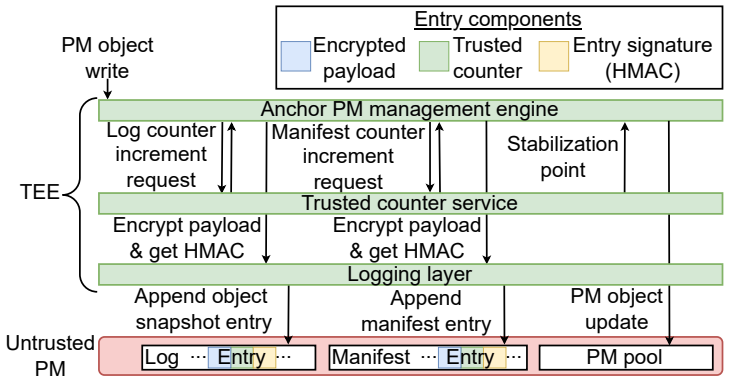

Fig. 2. Anchor logging protocol & log structure

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 [26]. 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 rollback 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 execution [22] further increase the latency compared to conventional approaches (e.g., sockets) that are already the bottleneck in networking systems [3, 51, 73]. While direct I/O networking solutions such as RDMA are prominently deployed in data centers to overcome the I/O bottlenecks [77], 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 [25] and *(ii)* RDMA operations might lead to inconsistent PM state.

Approach: Anchor overcomes these limitations by integrating userspace networking [78], 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.

5 SYSTEM DESIGN AND IMPLEMENTATION

5.1 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 2). 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 2), 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.

5.2 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 [8] 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, 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 (§ 6.1). 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's logs.

5.3 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 (§ 6.1). The index is trusted at run-time since it resides in the enclave; we populate the metadata index entries during a successful attested bootstrap (§ 6.2).

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 eliminates the decryption calls and access to 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 [93]. 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 on-going transaction.

5.4 Network Stack

Since networking consists 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 [5, 78] 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) in the untrusted host memory. Anchor stores the messages encrypted in DMA-capable buffers in the untrusted host memory satisfying two requirements (§ 4.3, #3): *(i)* DMA-ed memory cannot be inside the enclave and *(ii)* EPC usage is optimized. We integrate eRPC [78], with DPDK [5] 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*.

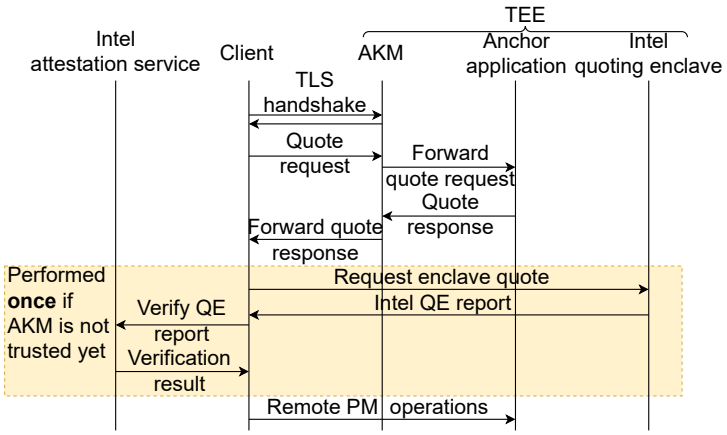

Fig. 3. Anchor attestation protocol

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 from 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 [8]. 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, 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 the PM security and crash consistency properties are ensured via the server's PM management engine.

5.5 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 [26], 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 [6]. While 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

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
```

stabilisation point and occurs when the SGX counter is successfully increased after an increment request (60 −250 m s). Anchor's recovery mechanism only trusts entries with stable counter values (§ 6.2).

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 [102]) that implement a trusted counter as a service in distributed settings [46] to reduce the stabilization time and ensure longevity.

5.6 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 [4] and integrates a key management system, which provisions clients with keys (e.g., for communication).

Attestation protocol. Figure 3 demonstrates Anchor's attestation protocol. More precisely, clients connect to the AKM service via a secure TLS channel. Following, they 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 the 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 [17]. 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.

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
```

6 SYSTEM OPERATIONS

6.1 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 enclave. Otherwise, Anchor fetches the PM object inside the enclave, and 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 (§ 6.2) 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 2. 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 2. After the snapshot, Anchor searches for

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
```

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.

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
```

6.2 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 are originated from an authentic Anchor instance.

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.

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
```

6.3 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 latest entries.

6.4 Network Operations

During the initialisation, 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 (§ 5.4) and its payload. Anchor encrypts the message and places it in a buffer residing in the untrusted host memory with its authentication tag. Then, the message is ready to be sent. 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 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.

7 SECURITY ANALYSIS

Anchor extends the standard SGX threat model, as described in Section 3, 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* [129] 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., AddressSanitizer [120] and Valgrind [14], to verify the memory safety of Anchor's enclave code and Anchor's crash consistency property (§7.1).

Importantly, we further formally prove the security principles of Anchor's remote attestation and secure logging protocol using the Tamarin prover [103] (§7.2). 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 for Tamarin to efficiently exhaust the search space and terminate without the need for oracles.

7.1 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) [120], 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 [14] and pmreorder [12] 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 [10] to Anchor. All the test cases [9] passed without indicating any crash consistency violation. Lastly, we adapt a pmreorder test of PMDK [11] and the core pmreorder example of the PM book [119] 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.

7.2 Formal Verification of Security Protocols

Remote attestation protocol. We model Anchor's attestation protocol (Figure 3), described in §5.6, using Tamarin [103]. 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 [13] 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 2. 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 (§6.2), *(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 to 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.

8 EVALUATION

8.1 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 writing this paper, processors could not support SGX and PM simultaneously[68, 69], therefore we opted for emulating PM backed by DRAM. We inject write latency on cacheline flushes similarly to previous works[76, 83, 109, 136]. For the network experiments, the nodes are equipped with Intel Corporation Ethernet Controller XL710 network card and are connected over a 40GbE QSFP+ network switch.

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) [15, 37] with varying R/W ratios. For our client-server evaluation, we use iPerf [71] and YCSB.

8.2 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.

Figure 4 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

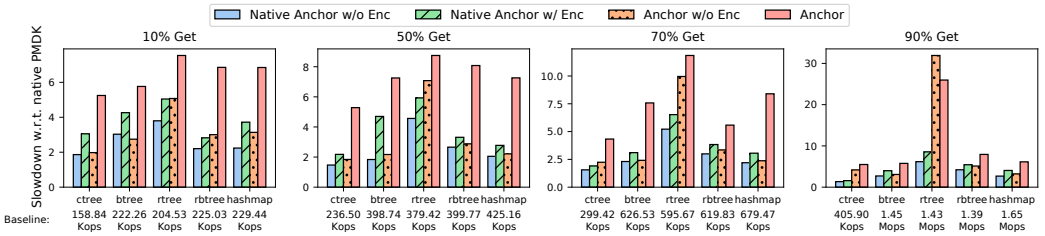

Fig. 4. Performance of PM indices under YCSB workloads for secure, native, w/ and w/o encryption Anchor versions.

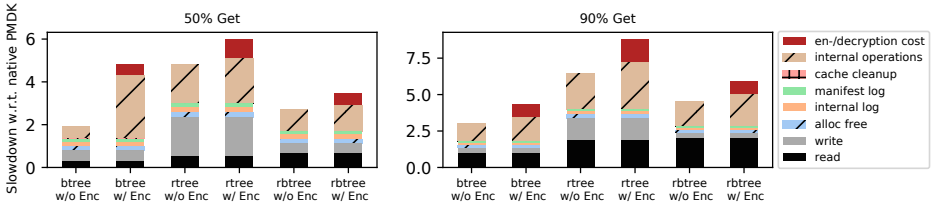

Fig. 5. Overhead breakdown in native Anchor version for PM indices under varying YCSB workloads.

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 2. rtree overhead for 10k keys

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 5. We further observe that the application integration into SCONE leads to a slowdown of 1.45-3.47×, depending on the workload. 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 that reduces the EPC pressure and decreases the frequency of the cache cleanup.

Moreover, we note that the average overhead slightly increases when increasing the read ratio. 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 5 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 this behaviour 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 result in the significantly higher overheads for rtree when running in SCONE. The EPC paging effect is highlighted through Table 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 [58]) will eliminate the limited EPC issue, thus leading to reduced overheads.

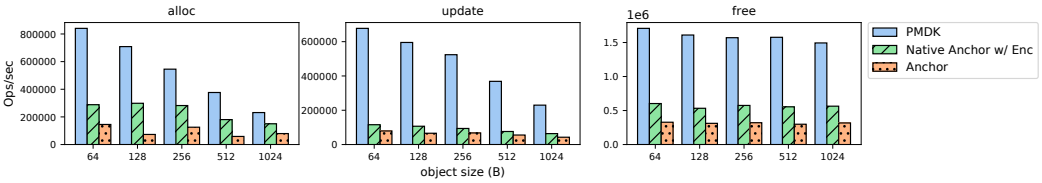

Fig. 6. Performance evaluation for TX memory operations for secure & native Anchor versions w/ encryption.

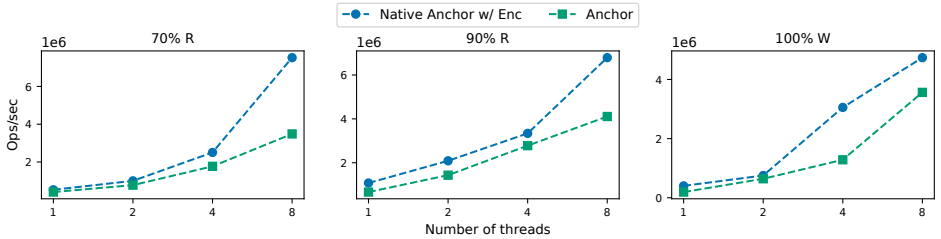

Fig. 7. R/W performance evaluation of secure and native Anchor versions w/ encryption with varying number of threads.

8.3 Operation Performance

We evaluate Anchor using a microbenchmark based on *pmembench* [65] 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 6 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 the native version and 4.7-5.3× compared to PMDK. For both alloc and free, we observe that Anchor's behaviour is similar to PMDK's with increasing object sizes. This is expected as de-/allocations only perform metadata modifications which 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.

8.4 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 7 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.

8.5 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.

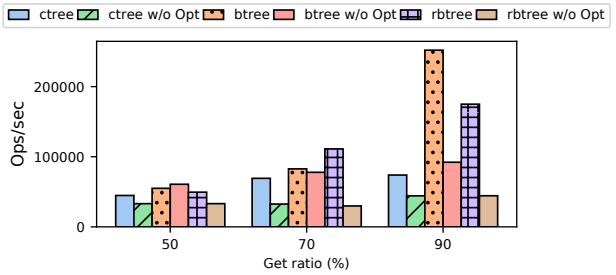

Fig. 8. Effectiveness of optimizations w/ YCSB.

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 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. Write amplification normalised per object

Figure 8 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 3 shows the cache hit ratio for the PM-indices averaged across the three different workloads shown in Figure 4. We notice that all workloads achieve more than 80 % hit rate, confirming the usefulness of this optimization.

8.6 PM 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 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 a small addition of the trusted counter value and size to ensure the integrity and freshness properties doubles 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' entries, metadata modifications need to be tracked in the manifest as well as entries indicating the transactions' phase.

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

Table 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 6. Recovery time w/ varying manifest & log size

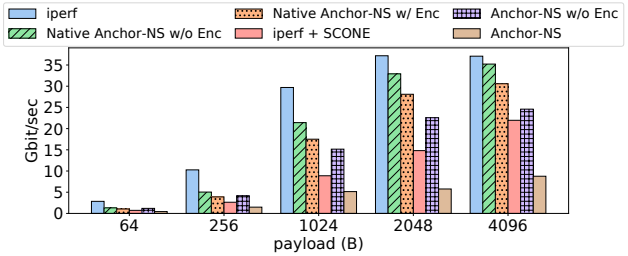

Fig. 9. Network stack throughput with iperf.

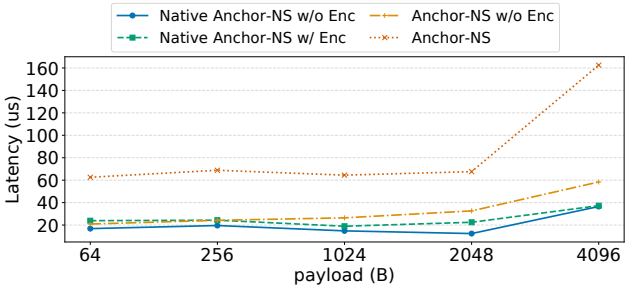

Fig. 10. Network stack latency.

8.7 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 logs sizes.

Table 5 and Table 6 show the required time to open a pool, scan the manifest, reconstruct the metadata index and perform the recovery, if needed. Table 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 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.

8.8 Anchor's Network Stack

Next, we evaluate the performance of Anchor's Network Stack (NS). We use 6 different setups: *(i)* iperf [71], *(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 behaviour 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 timespan.

Figure 9 and 10 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 for 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.

9 RELATED WORK

Persistent memory. PM systems are actively researched across several dimensions, such as filesytems [32, 75, 82, 141], KV-stores [30, 76, 92, 140, 148], crash consistency & reliability [34, 110, 115, 146, 147, 149] and testing tools [28, 96–98]. 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 [23, 24, 33, 95, 143, 145, 150, 151]. Unlike these systems, Anchor ensures data freshness, does not require 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 accesses to sensitive data. Several works pursue this goal for single-node [16, 18, 26, 41, 86, 104, 114, 128, 138] and distributed settings [25, 100, 112, 137] 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 [126]. Asynchronous system calls [22, 111] alleviate these overheads, but still require copying data to and from the trusted environments. Further works [104, 116, 125] 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 [78–81].

On the storage front, while the modern shielded execution frameworks have employed direct I/O in the context of TEEs [25, 26, 104, 131, 132], they are incomptabile 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 PM to include durability semantics; performance and crash consistency for remote operations [52, 53, 77, 84, 144]. 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 [125, 130].

10 CONCLUSION

In this paper, we present Anchor, a secure persistent memory library. Anchor allows for building secure PM data management systems by offering a programming model similar to PMDK, 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 which 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.

Acknowledgements. We would like to sincerely thank Julian Pritzi for his immense assistance in formally verifying the security protocols. This work was partially supported by a Schwerpunktprogramm (SPP) (ID: 2377) from Deutsche Forschungsgemeinschaft (DFG), an ERC Starting Grant (ID: 101077577), and a UK RISE Grant from NCSC/GCHQ at the University of Edinburgh, UK.

REFERENCES

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

[2] [n. d.]. CXL Software ecosystem. https://github.com/pmem/pmem.github.io/blob/main/content/blog/2023/cxl-blog-post.md.

[3] [n. d.]. 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.

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

[5] [n. d.]. Intel DPDK. http://dpdk.org/. Last accessed: Jan, 2021.

[6] [n. d.]. 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.

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

[8] [n. d.]. OpenSSL library. https://openssl.org. https://openssl.org Last accessed: Jan, 2021.

[9] [n. d.]. PMDK - object and pool recovery tests. https://github.com/pmem/pmdk/tree/stable-1.8/src/test/obj_recovery.

[10] [n. d.]. PMDK - unit test for pool recovery. https://github.com/pmem/pmdk/blob/stable-1.8/src/test/obj_recovery/obj_recovery.c.

[11] [n. d.]. PMDK - unit test for store reordering. https://github.com/pmem/pmdk/blob/stable-1.8/src/test/obj_reorder_basic/obj_reorder_basic.c.

[12] [n. d.]. The pmreorder utility. https://pmem.io/pmdk/pmreorder/.

[13] [n. d.]. Tamarin TLS handshake proof. https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/classic/TLS_Handshake.spthy.

[14] [n. d.]. Valgrind: an enhanced version for pmem. https://github.com/pmem/valgrind.

[15] [n. d.]. YCSB. https://github.com/brianfrankcooper/YCSB. Last accessed: Jan, 2021.

[16] 2022. 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)*. USENIX Association, Santa Clara, CA, 363–380. https://www.usenix.org/conference/fast22/presentation/kim-igjae

[17] October 27, 2023. 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. Last accessed: October 27, 2023.

[18] Adil Ahmad, Kyungtae Kim, Muhammad Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. https://doi.org/10.14722/ndss.2018.23296

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

[20] Apache. October 27, 2023. Apache Crail (incubating). https://github.com/apache/incubator-crail.

[21] ARM. [n. d.]. 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. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf Last accessed: Jan, 2021.

[22] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 689–703.

[23] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 263–276. https://doi.org/10.1145/2980024.2872377

[24] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 104–115.

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

[26] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. Speicher: Securing LSM-Based Key-Value Stores Using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 173–190.

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

[28] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. 2022. SafePM: A Sanitizer for Persistent Memory. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. ACM, NY, NY, USA, 506–524. https://doi.org/10.1145/3492321.3519574

[29] Ruining Chen and Guoao Sun. 2018. A Survey of Kernel-Bypass Techniques in Network Stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence* (Shenzhen, China) *(CSAI '18)*. ACM, NY, NY, USA, 474–477. https://doi.org/10.1145/3297156.3297242

[30] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory *(ASPLOS '20)*. ACM, NY, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[31] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. ACM, NY, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[32] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Trans. Storage* 14, 1, Article 4 (April 2018), 30 pages. https://doi.org/10.1145/3204454

[33] Siddhartha Chhabra and Yan Solihin. 2011. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 177–188. https://doi.org/10.1145/2000064.2000086

[34] Brian Choi, Randal Burns, and Peng Huang. 2021. Understanding and Dealing with Hard Faults in Persistent Memory Systems. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. ACM, NY, NY, USA, 441–457. https://doi.org/10.1145/3447786.3456252

[35] Alibaba Cloud. [n. d.]. 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.

[36] CXL™ Consortium. October 27, 2023. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

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

[38] George Copeland, Tom Keller, Ravi Krishnamurthy, and Marc Smith. 1989. The case for safe RAM. VLDB.

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

[40] Dana Neustadter - Synopsys. [n. d.]. 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. https://www.chipestimate.com/Protecting--Data--over--PCIe--and--CXL---in--Cloud-Computing/Synopsys/Technical-Article/2021/08/10

[41] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. 2018. 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)*.

[42] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* (1983).

[43] Chet Douglas. 2020. 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. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf

[44] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

[45] Gunawi et al. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[46] Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia. 2022. Treaty: Secure Distributed Transactions. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 14–27. https://doi.org/10.1109/DSN53405.2022.00015

[47] Garth Goodson and Bianca Schroeder. 2008. An Analysis of Data Corruption in the Storage Stack. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-08/analysis-data-corruption-storage-stack

[48] Google Confidential VMs [n. d.]. Introducing Google Cloud Confidential Computing with Confidential VMs. https://cloud.google.com/blog/products/identity-security/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.

[49] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. 202–215.

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

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

[52] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, 17–33. https://www.usenix.org/conference/nsdi18/presentation/honda

[53] Haixin Huang, Kaixin Huang, Litong You, and Linpeng Huang. 2018. Forca: Fast and Atomic Remote Direct Access to Persistent Memory. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 246–249. https://doi.org/10.1109/ICCD.2018.00045

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

[55] IBM. October 27, 2023. Analytics infrastructure: High-performance I/O architecture. https://www.zurich.ibm.com/cci/analytics/crail.html?lnk=hm.

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

[57] Intel. [n. d.]. Intel Optane Technology. https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/.

[58] Intel. [n. d.]. 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. https://cdrdv2-public.intel.com/742542/software-enabling-for-tdx-tee-io-fixed.pdf

[59] Intel. 2019. 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. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html

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

[61] Intel. 2021. Intel® Optane™ Persistent Memory Product Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html

[62] Intel. 2021. Persistent Memory Development Kit. https://pmem.io/pmdk/. https://pmem.io/pmdk/

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

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

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

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

[67] Intel. 2023. The librpma library. https://pmem.io/rpma/. https://pmem.io/rpma/

[68] Intel. October 27, 2023. 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.

[69] Intel. October 27, 2023. 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.

[70] Intel. October 27, 2023. 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.

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

[72] Nusrat Sharmin Islam, Md. Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. ACM, NY, NY, USA, Article 8, 14 pages. https://doi.org/10.1145/2925426.2926290

[73] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. 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)*.

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

[75] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, NY, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631

[76] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[77] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. ACM, NY, NY, USA, 105–119. https://doi.org/10.1145/3419111.3421294

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

[79] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, NY, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[80] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[81] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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)*.

[82] Chandan Kalita, G. Barua, and Priya Sehgal. 2018. DurableFS: A File System for Persistent Memory. *ArXiv* abs/1811.00757 (2018).

[83] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[84] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J. Marathe, and Sanketh Nalli. 2019. Correct, Fast Remote Persistence. *CoRR* abs/1909.02092 (2019). arXiv:1909.02092 http://arxiv.org/abs/1909.02092

[85] The Linux kernel archives. 2021. DAX - Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt. https://www.kernel.org/doc/Documentation/filesystems/dax.txt

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

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

[88] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-Assisted Fault Tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. ACM, NY, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/2901318.2901339

[89] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. 1991. The ObjectStore database system. *Commun. ACM* 34, 10 (1991), 50–63.

[90] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (2010), 143–143. https://doi.org/10.1109/MM.2010.24

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

[92] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, NY, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[93] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 185–196. https://doi.org/10.1109/ICDE.2018.00026

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

[95] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 310–323. https://doi.org/10.1109/HPCA.2018.00035

[96] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. 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* (Virtual, USA) *(ASPLOS 2021)*. ACM, NY, NY, USA, 487–502. https://doi.org/10.1145/3445814.3446691

[97] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. ACM, NY, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[98] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. ACM, NY, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[99] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[100] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. In *ACM Transactions on Computer Systems*. http://dl.acm.org/citation.cfm?doid=2063509.2063512

[101] PCPerspective Allyn Malventano. 2017. How 3D XPoint Phase-Change Memory Works. https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works/. https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works/

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

[103] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*.

[104] Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk, and Rüdiger Kapitza. 2021. Precursor: A Fast, Client-Centric and Trusted Key-Value Store Using RDMA and Intel SGX. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. ACM, NY, NY, USA, 1–13. https://doi.org/10.1145/3464298.3476129

[105] Micron. October 27, 2023. NVDIMM. https://www.micron.com/products/dram-modules/nvdimm.

[106] Jakub Szymaszek Microsoft. 2021. 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. https://techcommunity.microsoft.com/t5/azure-sql/always-encrypted-with-secure-enclaves-now-generally-available-in/ba-p/2502560

[107] Microsoft Azure. [n. d.]. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

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

[109] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[110] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. 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* (Virtual, USA) *(ASPLOS 2021)*. ACM, NY, NY, USA, 401–414. https://doi.org/10.1145/3445814.3446694

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

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

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

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

[115] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, NY, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[116] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4277–4292. https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger

[117] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. 2009. Towards Trusted Cloud Computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing* (San Diego, California) *(HotCloud'09)*. USENIX Association, USA, Article 3.

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

[119] Steve Scargall. 2020. *Debugging Persistent Memory Applications*. Apress, Berkeley, CA, 207–260. https://doi.org/10.1007/978-1-4842-4932-1_12

[120] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[121] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, NY, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[122] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*. 170–185.

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

[124] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. 2020. TH-DPMS: Design and Implementation of an RDMA-Enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage* 16, 4, Article 24 (Oct. 2020), 31 pages. https://doi.org/10.1145/3412852

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

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

[127] Xinyang (Kevin) Song, Sihang Liu, and Gennady Pekhimenko. October 27, 2023. 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..

[128] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[129] László Szekeres, M. Payer, Tao Wei, and D. Song. 2013. SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy* (2013), 48–62.

[130] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 691–704. https://www.usenix.org/conference/atc20/presentation/taranov

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

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

[133] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. 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)*.

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

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

[136] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 91–104. https://doi.org/10.1145/1961295.1950379

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

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

[139] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. 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* (Portland, Oregon) *(DaMoN '20)*. ACM, NY, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/3399666.3399898

[140] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[141] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

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

[143] Fan Yang, Youmin Chen, Haiyu Mao, Youyou Lu, and Jiwu Shu. 2020. ShieldNVM: An Efficient and Fast Recoverable System for Secure Non-Volatile Memory. *ACM Trans. Storage* 16, 2, Article 12 (May 2020), 31 pages. https://doi.org/10.1145/3381835

[144] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 111–125. https://www.usenix.org/conference/nsdi20/presentation/yang

[145] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, NY, NY, USA, 33–44. https://doi.org/10.1145/2694344.2694387

[146] L. Zhang. 2019. Building Reliable Software for Persistent Memory.

[147] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 897–911.

[148] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. ACM, NY, NY, USA, 194–209. https://doi.org/10.1145/3447786.3456237

[149] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 3–18. https://doi.org/10.1145/2786763.2694370

[150] Pengfei Zuo and Yu Hua. 2018. 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)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/zuo

[151] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling Application-Transparent Secure Persistent Memory with Low Overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, NY, NY, USA, 479–492. https://doi.org/10.1145/3352460.3358290

Received April 2023; revised July 2023; accepted August 2023