TENET: Memory Safe and Fault Tolerant Persistent Transactional Memory

R. Madhava Krishnan, *Virginia Tech;* Diyu Zhou, *EPFL;* Wook-Hee Kim, *Konkuk University;* Sudarsun Kannan, *Rutgers University;* Sanidhya Kashyap, *EPFL;* Changwoo Min, *Virginia Tech*

https://www.usenix.org/conference/fast23/presentation/krishnan

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

TENET: Memory Safe and Fault Tolerant Persistent Transactional Memory

R. Madhava Krishnan Diyu Zhou* Wook-Hee Kim† Sudarsun Kannan‡ Sanidhya Kashyap* Changwoo Min
Virginia Tech *EPFL** *Konkuk University†* *Rutgers University‡*

Abstract

Byte-addressable non-volatile memory (NVM) allows programs to directly access storage using memory interface without going through the expensive conventional storage stack. However, direct access to NVM makes the NVM data vulnerable to software bugs and hardware errors. This issue is critical because, unlike DRAM, corrupted data can persist forever, even after the system restart. Albeit the plethora of research on NVM programs and systems, there is little focus on protecting NVM data from software bugs and hardware errors.

In this paper, we propose TENET, a new NVM programming framework, which guarantees memory safety and fault tolerance to protect NVM data against software bugs and hardware errors. TENET provides the popular persistent transactional memory (PTM) programming model. TENET leverages the concurrency guarantees (*i.e.*, ACID properties) of PTM to provide performant and cost-efficient memory safety and fault tolerance. Our evaluations show that TENET offers an enhanced protection scope at a modest performance overhead and storage cost as compared to other PTMs with partial or no memory safety and fault tolerance support.

1 Introduction

Byte-addressable non-volatile memory (NVM) opens a new paradigm in designing storage stack. NVM provides byte-addressability and low-access latency like DRAM and it offers data persistence like storage. A program can directly map (`mmap`) an NVM region to its address space and access it using `load/store` instructions without storage stack overhead (referred to as *direct persistence*). Several works leverage NVM in the core storage stack, including file systems [34, 51, 88, 89, 96], key-value stores [52, 55, 57, 61, 65, 66, 86], and persistent transactional memory (PTM) [47, 56, 77, 87]. Although the first commercial NVM product, Intel Optane DCPMM, was discontinued recently [13],industry continues to explore various forms of direct persistence [42]. In particular, the emerging Compute Express Link (CXL) [12,31] opens new opportunities for byte-level persistence based on NAND flash [16,21], NRAM [39], battery-backed DRAM [41,78], and PRAM [22]. Also, many software-based solutions [53, 67, 91], which exploit direct persistence (DRAM along with in-rack battery), are being widely deployed in data centers [1, 11, 18, 45, 50].

However, the direct persistence of NVM opens several challenges in protecting data from software bugs (*e.g.*, "memory scribbles") and media errors. NVM data can be permanently corrupted due to a single memory scribble, which roots from a spatial safety violation (*e.g.*, buffer overflow) or a temporal safety violation (*e.g.*, use-after-free) in a program. Previous studies [26, 32, 35, 36, 59, 69–71, 73, 79, 81, 83, 84, 92, 95] have shown that such memory safety violations are prevalent in programs (*e.g.*, 70% of CVEs [5, 17, 25]). Since NVM is mapped to the same address space as DRAM, memory safety violations in NVM and DRAM can corrupt NVM data. Besides these software bugs, dense NVMs have a higher random raw bit error rate (RBER) than DRAMs, with RBER closer to NAND flash [85,93]. Hence, NVM (*e.g.*, Intel Optane) adopts stronger ECC for error correction. Unfortunately, certain hardware errors can still escape the error correction, leading to Uncorrectable Media Errors (UME) in NVM [4, 7].

PTMs [47, 56, 77, 87] are one of the most popular NVM programming models because of their ability to exploit direct persistence. A few recent PTM systems, such as SafePM [27] and Pangolin [94], attempt to provide NVM data protection by extending `libpmemobj` [47]. A desirable PTM system that offers NVM data protection should (1) offer extensive data protection: protect against both NVM media errors and software memory safety violations in both DRAM and NVM, and (2) incur lower performance overhead and storage costs.

Unfortunately, existing works fail to meet the above criteria. SafePM [27] provides NVM memory safety by instrumenting every NVM access. It does not protect against media errors and memory safety violations in DRAM. The memory instrumentation and the associated metadata incur high performance overhead and storage cost. Pangolin offers data protection with checksum and parity while `libpmemobj` provides fault tolerance by simply replicating the NVM data to a backup NVM region. However, both systems are still vulnerable to memory safety violations, incur high NVM storage cost, and suffer from high performance overhead. As further explained in §2.2, in summary, prior approaches compromise the protection coverage [27, 44, 94] while also incurring high storage cost and high performance overhead [27, 47, 94].

This paper proposes TENET, a principled PTM-based approach that offers an enhanced memory safety and fault tolerance guarantees at a significantly lower performance overhead and storage costs than prior works. Leveraging off-the-shelf hardware features and the concurrency properties of PTM, TENET reduces performance overhead and storage costs without compromising its protection coverage. We realize TENET's memory-safe design principles using the state-of-the-art and highly scalable PTM framework, TimeStone [56] that does not provide NVM data protection. In particular, key techniques of TENET are as follows:

- **Hardware-enforced memory domain separation.** Instead of instrumenting every memory access to check for memory safety violations, TENET exploits an existing hardware

feature: Intel Memory Protection Keys (MPK) [49, 74], to separate the address space into NVM domains and a DRAM domain. Only the trustworthy TENET library can write to the NVM domains. Thus, outside the TENET library, TENET offloads NVM data protection against memory scribbles to hardware. This enables data protection for most memory access with almost zero overhead.

- **On-first-read and on-commit memory safety enforcement.** Enforcing memory safety at every NVM access in the TENET library incurs high overhead. Instead, leveraging PTM semantics, TENET enforces the temporal safety violation only at the first reference of an NVM object and the spatial safety violation only at the commit of a persistent transaction. This, in tandem with the memory domain separation technique, prevents the corrupted data from reaching NVM with very low runtime overhead.
- **Asynchronous hybrid NVM-SSD replication.** Protecting against NVM media errors fundamentally requires creating redundancy. TENET asynchronously replicates the NVM data to SSD off the critical path to tolerate any number of NVM media errors. It thus offers low storage cost fault tolerance without hindering performance.
- We design TENET using the above approaches, which to the best of our knowledge is the first high-performance PTM with memory safety and fault tolerance guarantees.
- We evaluate two different versions of TENET– (1) memory safety only (TENET-MS) and (2) memory safety and fault tolerance (TENET) with key data structures and real-world workloads. Our results indicate that TENET offers enhanced protection at a modest performance overhead and storage cost as compared to state-of-the-art systems.

2 Background and Motivation

This section first introduces NVM media errors (§2.1) and memory safety violation in NVM programs (§2.2), followed by discussing the prior PTM works that address the media errors and memory safety violations (§2.3).

2.1 NVM Media Errors

Figure 1 shows the classification of potential errors in NVM. These errors can be classified into hardware errors and software errors. Hardware errors can be further classified into media errors (MEs) and silent data corruptions (SDCs). Media errors are caused by faults in the NVM media such as exceeding the write endurance, power spikes, soft media faults etc that directly corrupt data in the NVM media [85, 93]. SDCs are caused by faults that occur outside NVM media, which indirectly causes data corruption. Examples of SDCs are buggy NVM firmware, faults in CPUs, memory controllers, or other hardware components [28, 54]. Handling SDCs is a separate research area and it is out of scope of this paper.

Hardware media error (ME) correction. Commercially available NVMs implement error-correction code (ECC) in hardware to detect and correct media errors. For example, Intel Optane DCPMM uses hardware parity to detect any-bit

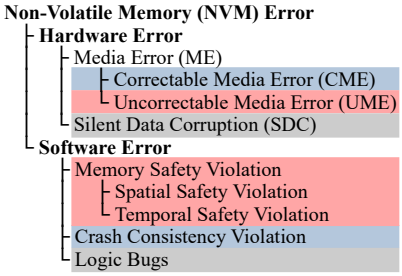

Figure 1: Classification of errors in NVM. TENET handles *UME*, *Spatial and Temporal Safety Violation* bugs (red). TENET relies on the hardware ECC to fix *CME* and the underlying PTM to handle *Crash Consistency Violations* such as atomicity and persistence ordering (blue). *Silent Data Corruption* in the hardware (*e.g.*, CPU faults) and *logical bugs* in the application are out of scope (grey).

errors, and it can correct up to two 2-bit errors [10]. The NVM hardware transparently fixes such correctable media errors (CMEs). However, uncorrectable media errors (UMEs) will be reported for software intervention as detailed below.

Reporting uncorrectable media error (UME) to software. The OS receives the reports of UMEs; and it can pass it to the application. Specifically, when a CPU accesses an NVM page affected by UMEs, the NVM hardware sends a poison bit along with the relevant data to the CPU. Upon encountering the poison bit, the CPU raises a memory check exception (MCEs) for the OS to handle. Currently, Linux handles the MCE by adding the corrupted page to the bad block list and sends a `SIGBUS` signal to the application [4, 9]. Then the OS leaves the responsibility to the application for fixing UMEs during the recovery phase [7]. *We note that, although the NVM is byte-addressable, UMEs are reported to the software at the page granularity due to the blast radius effect [3].*

2.2 Memory Safety in NVM Programs

We categorize software "scribbles", which corrupt NVM data, as spatial and temporal memory safety violations (Figure 1). *Spatial safety violations* happen when memory is accessed beyond its allocated range. Buffer overflows and array out-of-bound accesses are classical examples. *Temporal safety violations* happen due to dangling pointers; *i.e.*, when accessing an already freed (*use-after-free*) or accessing a reallocated address range (*use-after-realloc*). These memory safety bugs are even more dangerous in NVM than DRAM because the NVM data will be corrupted forever and a simple system restart would not fix these issues. *Note that memory safety bugs on either DRAM or NVM region of an application can cause NVM data corruption since the NVM region is mapped directly to application's address space*.

2.3 Prior NVM Data Protection Approaches

Memory safety in NVM programs. Prior works – Pangolin [94], SafePM [27], and Corundum [44] – include mechanisms to protect NVM data from memory safety violations. Pangolin extends `libpmemobj` [47] and uses per-object checksum to detect spatial safety violations. SafePM adds Address-

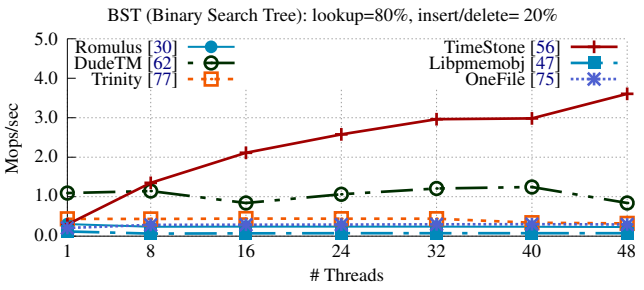

Figure 2: Performance of TimeStone against other PTMs. None of the PTMs are memory safe or fault tolerant against UME.

Sanitizer [80] to `libpmemobj` transaction to detect spatial and temporal safety violations on NVM data. Corundum is a Rust-based NVM programming library and leverages Rust's type system to statically enforce spatial and temporal memory safety. However, they have some critical limitations. *First, none of these approaches prevent NVM data corruption due to memory safety violations on "DRAM data"*. Suppose that the buggy code inside a transaction causes a buffer overflow on DRAM data; such spatial safety violations on DRAM can scribble arbitrary memory location, including NVM data. *Moreover, none of them guarantee to protect NVM data from temporal safety violations.* Pangolin does not check temporal safety violations. Meanwhile, SafePM does not detect use-after-realloc bugs. Even with Corundum, the developers still have the responsibility to guarantee type and memory safety for the "unsafe" Rust code, both can result in spatial and temporal safety violations.

Both Pangolin and SafePM suffer from high performance overhead and introduce additional performance bottlenecks. Pangolin calculates and verifies checksums on the critical path, imposing high performance overhead. Furthermore, it verifies checksum only for write transactions (*i.e.*, read transactions are unprotected). SafePM instruments every NVM access to check for memory safety violation, which is costly. SafePM further introduces extra UNDO logging overhead over the already existing expensive logging in the `libpmemobj` to guarantee crash consistency for its memory safe metadata.

Fault tolerance against UME. To protect against UME, `libpmemobj` supports replicating data on NVM. However, it replicates data on the write critical path, leading to high performance overhead. Furthermore, storing the replicated data on NVM wastes the precious NVM space, doubling (2×) storage cost. Pangolin uses parity for fault tolerance; however, *parity calculation on the critical path causes high performance overhead and it unnecessarily serializes the transactions which affects the write scalability*. Further, Pangolin can recover up to one page within a parity region; a data loss will happen if UME occurs on more than a page. SafePM and Corundum do not provide any fault tolerance against UME.

2.4 Prior PTMs for NVM

`Libpmemobj` [47] has been the de-facto PTM. However, it suffers from high performance overhead and poor scalability. Thus, several new PTMs focus on addressing its limitations [30,40,62,68,75–77,87]. Figure 2 shows that none of the existing PTMs, except TimeStone [56], scale beyond 8 cores even for a read-intensive workload. Further, TimeStone performs up to 8× better than the existing PTMs. Based on this observation, we chose TimeStone [24, 56] as the transaction abstraction for TENET. Moreover, designing memory safety and fault tolerance techniques for such a high performance PTM is challenging as even a small bottleneck can compromise its original scalability and performance. We introduce the relevant design aspects of TimeStone below.

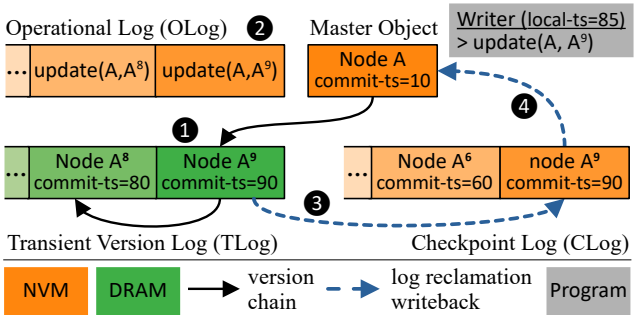

Figure 3: An illustrative example of updating `Node A` to its 9th version (`A`^{9}) in TimeStone.

Multi-version concurrency control. TimeStone follows multi-version concurrency control (MVCC). With MVCC, TimeStone supports non-blocking reads and concurrent disjoint writes, achieving high concurrency. For each object created by the application (*e.g.*, B-tree node), TimeStone allocates a *master object* on NVM (see Figure 3). On updating a master object, TimeStone creates a new version (❶) on DRAM, chaining multiple version objects from new to old object's age. TimeStone dereferences the right version object during the dereference phase with the help of timestamps. Each version object gets assigned a timestamp when it is committed (`commit-ts`). Also, each transaction gets a timestamp (`local-ts`), which denotes the transactions' start time. TimeStone traverses the version chain and chooses the most recent version of an object based on these timestamps (*i.e.*, `commit-ts<=local-ts`). This guarantees a consistent snapshot of NVM data for all transactions at any given time.

Operational log based immediate durability. TimeStone uses a DRAM-NVM hybrid logging technique, named TOC logging for efficient crash consistency. The TOC logging consists of Transient Version Log (`TLog`) on DRAM, Operational log (`OLog`) and Checkpoint log (`CLog`) on NVM, as illustrated in Figure 3. TimeStone creates a new version on `TLog` (❶), and logs the performed operation to the `OLog` (❷) for immediate durability. An operational log entry is typically much smaller than the conventional undo/redo logging, which duplicates the data, thus making crash consistency efficient.

Asynchronous log reclamation and replay based recovery. As more versions are created, `TLog` eventually becomes full,

triggering log reclamation. When TLog is reclaimed, the latest version of an object on TLog (A^9 over A^8) is checkpointed to the CLog (❸). Similarly, when CLog becomes full, the latest checkpoint (A^9 over A^6) is written back to the master object (❹). To recover from a crash, TimeStone first applies the checkpoints in CLog to the respective master objects and reverts them to a consistent snapshot. Then the OLog is executed to recreate all the updates that are lost on TLog.

3 Overview of TENET

3.1 Threat Model and Assumptions

TENET aims to protect against spatial and temporal memory safety violations in buggy application code. Furthermore, TENET considers the possibility of a memory safety violation on DRAM data corrupting NVM. TENET also aims to guarantee fault tolerance for NVM data against the uncorrectable media errors (UMEs). PTMs in general and TimeStone in particular cannot guarantee **ACID** properties for the application code that is outside the transaction or when the PTMs' APIs are misapplied. This applies to TENET as well, *i.e.*, it cannot guarantee memory safety and fault-tolerance for the code outside the transaction. TENET is not designed to handle SDC that occur outside the NVM media. Protection against the adversarial attacks (*e.g.*, control-flow attacks) is out-of-scope. However, the protection techniques and mechanisms against SDC and control-flow attacks can be orthogonally deployed to TENET. In TENET, application code is distrusted while TENET library code and OS kernel are considered as a trusted computing base (TCB).

3.2 Design Goals

- **Protect NVM data from memory safety violations.** TENET should detect all spatial and temporal safety bugs not only from NVM but also from DRAM. Any memory safety bugs either in DRAM or NVM code should not corrupt NVM data.
- **Protect NVM data against UMEs.** TENET should provide a robust fault tolerance mechanism to recover and restore NVM data from UMEs transparently.
- **Low performance and storage overhead.** TENET aims to be a *practical* system that offers an enhanced protection scope and strong fault tolerance at a minimal performance and storage overhead.

3.3 Design Overview

TENET re-purposes the multi-versioning and transactional semantics of TimeStone to achieve its design goals. Below we introduce TENET's main techniques as illustrated in Figure 4.

(1) Separation of NVM protection domain from DRAM. A memory safety bug (*e.g.*, out-of-bound write) either in DRAM or NVM can result in NVM data corruption. Enforcing full memory safety in every single memory access incurs prohibitive runtime overhead as prior studies show [27, 94].

To prevent unauthorized NVM writes without checking every single memory access, TENET grants the write permission to the NVM region only for the TCB *i.e.*, the TENET library code. In other words, the application code has read-only permission for the NVM, and consequently, it only writes on DRAM. When the application commits its transaction, writer thread gets write permission to execute the TENET library code which propagates the updates on DRAM to the NVM.

TENET completely segregates DRAM and NVM regions so that all new version and master objects are created on DRAM (referred to as *DRAM Objects*). Therefore, TENET application code does not require write access to NVM, as it writes only to the DRAM region. If a buggy application code tries to write to the NVM region, it will receive an exception (SIGSEGV) from TENET and will be terminated. TENET exploits Intel Memory Protection Keys (MPK) [49, 74] to efficiently switch NVM permissions for each thread.

(2) On-commit spatial safety enforcement. As applications can always write to the TLog (*i.e.*, DRAM), it is vulnerable to arbitrary memory scribble. A corrupted DRAM object can be eventually propagated to CLog (❻ in Figure 4) and the master object (❽), consequently corrupting the NVM data. We propose *on-commit spatial safety enforcement* to prevent corrupted DRAM objects from reaching NVM. TENET adds eight byte canary values at the start and at the and of a DRAM object during its creation (❷). Specifically, TENET assigns a random value to C0, and the hash of C0 and its location (xor(C0,&C1)) to C1. When an application commits the transaction, TENET inspects the integrity of canary values of all DRAM objects in that transaction (③ and ④). If the canaries are compromised (*i.e.*, C0 != xor(C0,&C1)), then TENET aborts the transaction and gracefully terminates without propagating the corrupted objects to NVM.

Our on-commit spatial safety enforcement is efficient with minimal performance overhead. Unlike the prior approaches [27, 69, 79, 95], our technique avoids reading additional metadata, and it checks the integrity only once during the transaction commit. Note that NVM objects do not have canary values and thus no NVM space overhead.

(3) On-first-dereference temporal safety enforcement. Even after an NVM (master) object is freed (and then reallocated), a program still can reference it via dangling pointers which can corrupt the NVM data in unintended ways.

We propose *on-first-dereference temporal safety enforcement* to efficiently enforce temporal safety of NVM objects with a minimal runtime overhead. TENET uses a tag-based approach, which essentially checks if a pointer points to the right object by comparing tags associated with the pointer and the pointed object. When TENET creates an NVM (master) object, it assigns a 2-byte random integer as a tag of the object (*e.g.*, 0xCAFE for Node A in Figure 4). We encode this 2-byte tag in the upper 16-bit of a pointer, which is unused in the x86 architecture. When the object is freed, its associated tag on the header set to zero for detecting use-after-free. When the object is dereferenced *first time* in a TENET transaction (①), TENET checks whether the encoded tag in the pointer matches

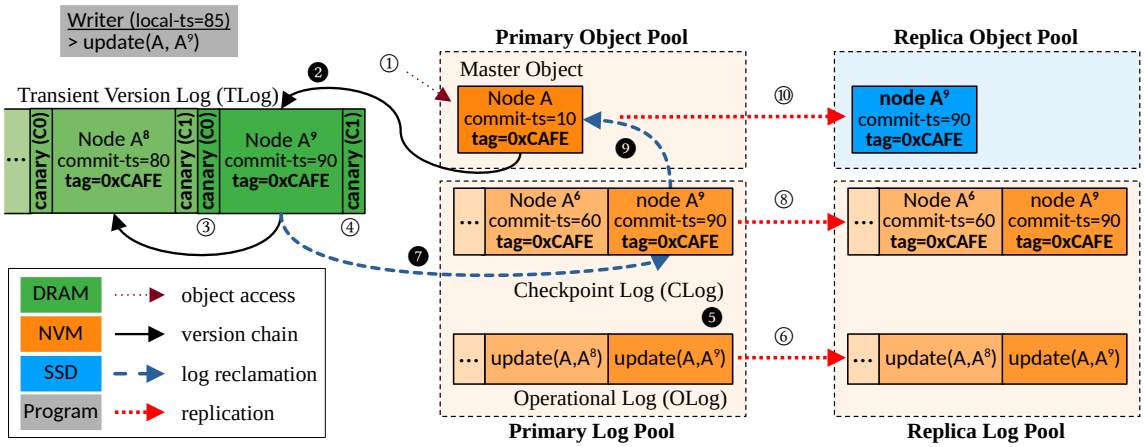

Figure 4: Overall architecture of TENET with an example of updating `Node A` to its 9th version (`A`^{9}). ⓝ denotes the newly added memory safety checks and replication to the TimeStone transaction. Note that the application has read/write access to DRAM and read-only permission for NVM. When accessing `Node A`, TENET validates its temporal safety by comparing the tags, `0xCAFE` (①). If the tags do not match, the transaction is aborted. Otherwise, the writer proceeds to traverse the `Node A`'s version chain, makes a copy of the latest version (`A`^{8}) in its `TLog` and updates it to `A`^{9} (❷). Upon commit, `Node A`^{9} is validated for spatial safety by checking the canary values (③ and ④). The transaction is aborted if the validation fails. Otherwise, the writer commits the transaction by updating its `OLog` (❺) for durability and it also synchronously updates the replica `OLog` for fault tolerance (⑥). When reclaiming the `TLog`, `Node A`^{9} is once again validated for spatial safety before checkpointing it to the `CLog` (❼) followed by synchronously updating the replica `CLog` (⑧). Similarly, when the `CLog` is full, TENET writes back the latest checkpoint (`Node A`^{9}) to the original master object `Node A` (❾). The updated Node A is then *asynchronously* replicated to the disk (⑩).

with the tag in the pointed object. If the tags do not match, it means the pointer points to the already-freed/-reallocated object (*i.e.*, dangling pointer), which violates temporal safety. In this case, TENET aborts the transaction immediately.

Our approach is efficient and imposes minimal performance overhead because it checks the temporal safety of each object only once in a transaction. Also, accessing the inlined tags is cache-friendly, which, unlike prior approaches [27,71,79,95], requires no additional metadata lookup.

(4) Off-critical path NVM replication to SSD. TENET replicates all NVM data; in the case of a UME, corrupted NVM pages can be restored using the replica. The main challenge in designing a replication scheme is minimizing the performance overhead and storage cost. While replication to another NVM region can be performance efficient, it incurs 2× higher capacity cost. Instead, we propose a hybrid NVM-SSD replication technique; TENET *asynchronously* replicates the master objects to SSD (⑩) and *synchronously* replicates the transaction logs (`CLog` and `OLog`) to NVM (⑥, ⑧). Master objects, are application data structures, which can be large and also potentially occupy the entire NVM space. Hence, TENET replicates master objects to the SSD off the critical path to reduce both storage cost and performance overhead. Although the replication is asynchronous, TENET guarantees *loss-less* NVM data recovery by prudently leveraging the transaction logs and grace period semantics. Meanwhile, transaction logs are small and finite, so TENET replicates them to NVM to reduce performance overhead. Further, TENET is also capable of recovering from multiple simultaneous UMEs occurring in one or multiple NVM pages. We explain this design, its correctness and recovery guarantees in §4.4 and §4.5.

3.4 Putting It All Together For TimeStone

TENET makes the NVM read-only for all except the TENET's library code. So the NVM objects in TimeStone do not need spatial safety checks as they are read-only objects. TENET enforces temporal safety checks for all NVM objects (using pointer tags) during the object dereferencing to detect dangling pointers. On the contrary, DRAM objects are vulnerable to application scribbles (due to write permission) hence TENET enforces on-commit spatial safety checks using the canary bits. DRAM objects do not need separate temporal safety checks as they are managed internally by TENET; *i.e.*, as DRAM objects are accessed via the respective NVM object, enforcing temporal safety for NVM objects indirectly guarantees it for DRAM objects. We discuss the correctness of these techniques in §4.3. TimeStone can not handle UMEs, so TENET proposes to replicate master objects and transaction logs to SSD and NVM respectively; in the event of a UME, NVM data can be restored using the NVM/SSD backup. In a nutshell, we optimally apply TENET's memory safety techniques to the vulnerable parts of TimeStone and organically redesigned it to guarantee full memory safety. If TENET was to be used for other PTMs, then its techniques can well be applied, albeit it may require some engineering effort. We discuss this further in §6. Refer to Figure 4 for a summary on lifecycle of a TENET transaction.

4 TENET Design

In this section, we first describe TENET transaction design (§4.1) followed by the design of memory safety (§4.2-§4.3),

fault tolerance replication (§4.4), and recovery (§4.5).

4.1 TENET Transaction

Below we explain how TimeStone transaction is redesigned using TENET to enforce memory safety and fault tolerance.

4.1.1 NVM Object Dereference

Object dereferencing in TimeStone (§2.4) only traverses the version chain and returns the correct version, whereas in TENET, object dereferencing is a two-step process.

(1) Temporal safety validation. TENET validates the master object pointer for temporal safety (§4.3.2) to detect dangling pointers; transaction aborts if the validation fails (§4.1.4).

(2) Version chain traversal. If the object passes the validation, then TENET dereferences the correct DRAM object or directly the master object if the version chain does not exist.

4.1.2 Updating an Object

In TENET, a writer updates a master object by creating a new DRAM object as done in the TimeStone. However, TimeStone allows its users (application) to allocate and write to the NVM when creating new master objects. Thus, a buggy application can easily corrupt the NVM region. In TENET, this is restricted to prevent direct NVM writes; so the application allocates and writes to a new master object (*shadow master object*) on the DRAM and then during the commit phase TENET library creates a corresponding NVM copy only if the writes pass the spatial safety violation checks.

4.1.3 Committing a Transaction

In TimeStone, the commit procedure updates the `OLog` to guarantee durability and then makes all the updates atomically visible. TENET's commit procedure happens in three phases:

(1) Spatial safety validation. All the new versions and shadow master objects created in a transaction are validated for spatial safety violations (§4.3.1). Upon successful validation, TENET allocates and updates the persistent master object from the corresponding shadow master object.

(2) Transaction durability and replication. Updating `OLog` guarantees durability, and replicating it ensures fault tolerance (§4.4.1). Also, TENET adds all the newly created master objects in **(1)** to the replica buffer to trigger async disk writes using background workers (§4.4.2).

(3) Publishing the updates atomically. TENET makes the updates atomically visible by adding the new versions to their respective version chain, and this procedure is exactly the same as TimeStone. Additionally, TENET frees all the shadow master objects, if any, and exits the critical section.

4.1.4 Aborting a Transaction

Common abort procedure. TENET rolls back any used log space, lock status, and reclaims all the shadow master objects and also its NVM counterpart if one exists. This is common for all three abort cases described below.

Abort due to lock conflict. During the object update (§4.1.2), if the writer fails to acquire a lock, it aborts the transaction. This is a benign abort *i.e.*, no memory safety violations, so TENET performs the common abort procedure and retries the transaction after the backoff period.

Abort due to memory safety violation. All ongoing transactions are aborted if a transaction aborts due to spatial safety or temporal safety violation. TENET executes the common abort procedure and returns an exception.

Abort due to a UME. The OS notifies a UME by sending a `SIGBUS` signal. TENET's signal handler catches the signal, returns a UME exception to notify the application, and gracefully terminates the process. TENET fixes the affected NVM region during the recovery process (§4.5).

4.2 Unauthorized NVM Write Prevention

TENET already prevents application code from directly writing to the NVM by using DRAM objects for the updates. However, a buffer overflow on DRAM can corrupt the NVM data as NVM is directly mapped to the applications' address space. TENET employs Memory Protection Keys (MPK), a hardware feature available in the Intel systems [33, 34, 43, 74, 82] to detect NVM writes out of TENET library code.

Using MPK to enforce read-only NVM access. With MPK, a page can be assigned to one of the 16 available protection domains. The assigned protection domain is encoded in the page table entry. A thread's access permission to the protection domains is controlled at the per-thread level via a user-accessible register, `PKRU`. A thread can switch its access permissions to the protection domains by writing to the `PKRU` register, which only costs 20 CPU cycles. In TENET, each NVM pool is assigned a unique protection key during pool creation. Only the TCB (*i.e.*, TENET library code) is allowed to write to the NVM pool. Thus, a thread grants itself read-write permissions to the corresponding NVM pool during the library code execution and revokes it before exiting the library. As a result, if the application writes to NVM (*e.g.*, due to buffer overflow), MMU prevents the access and OS sends a `SIGSEGV` signal. Thus, any spatial safety violations due to a buggy write is contained within the DRAM region.

4.3 Enforcing Memory Safety

In this section, we explain the spatial (§4.3.1) and temporal safety design (§4.3.2). In §4.3.3, we explain the array interface as an example, and how the interface provides memory safety.

4.3.1 On-commit Spatial Safety Design

TENET enforces spatial safety for all DRAM objects to prevent NVM data corruption due to a buggy DRAM write.

Technique. As illustrated in the Figure 4, all DRAM objects are assigned two 8-byte canaries at the start `C0` and at the end `C1`. Specifically, `C0` is a random value and `C1` is the hash of `C0` and its location (`xor(C0,&C1)`). TENET inspects the integrity of canary bits to detect buffer overflows and underflows.

On-commit validation. When the application commits its writes (§4.1.3), TENET inspects canary bits for all the newly created DRAM objects. A transaction is committed only when both `C0` and `C1` are intact in all the DRAM objects. Otherwise, the transaction aborts and discards all the corrupted objects. An erroneous transaction can corrupt the DRAM objects outside of the current transaction *i.e.*, the ones that are part of other concurrent transactions or the ones that are not part of any ongoing transactions at all. To detect such cases, TENET places an 8-byte canary at the start and the end of the transactions' write set. Note that all the DRAM objects including the shadow master objects are part of a transactions' write set. TENET validates the write set canaries before and after each step of the commit process (§4.1.3). This ensures that a transactions' write set (*i.e.*, DRAM object) has not been corrupted by an erroneous concurrent transaction, particularly between the initial validation (**(1)** in §4.1.3) and the publication of the updates (**(3)** in §4.1.3). However, if the write set canaries are found to be compromised then TENET aborts all the transactions as explained in §4.1.4.

Correctness. Deferring spatial safety checks until the commit time does not violate the correctness as the other concurrent transactions *can not* observe any uncommitted DRAM objects. Although a rare case, to avoid reading a DRAM object that is corrupted (after it commits), TENET performs spatial safety check before dereferencing a committed DRAM object. Subsequently, the DRAM objects (❼ in Figure 4) and the shadow master objects are re-validated before and after copying to the NVM to prevent *Time-of-Check-Time-of-Use (TOCTOU)* bugs [6]. If an DRAM object is found to be corrupted post the copy operation then the corresponding NVM object will be safely reclaimed as part of the transaction abort procedure. Finally, TENET cannot detect the corruptions that occur without overwriting the canaries, aka intra-object overflows. We discuss this further in §6.3.

4.3.2 On-first-dereference Temporal Safety Design

TENET enforces temporal safety for all NVM (master) objects to detect dangling pointer dereference. Accessing an already free-ed (or reallocated) address can corrupt the NVM data due to use-after-free (or use-after-realloc) bugs.

Technique. To detect dangling pointers, TENET assigns an *unique 2-byte tag* for all the master objects, which is stored in the object's header (`0xCAFE` in Figure 4) at the time of its creation. A copy of this tag is also encoded in the *unused* upper 16-bits of the master objects' address. On deallocating the master object, the tag in the objects' header is set to zero.

On-first-dereference validation. When the application accesses a master object for *the first time in a transaction*, TENET validates the pointer to the master object before traversing the version chain (§4.1.1). TENET extracts the tag encoded in the master objects' pointer and compares it with the tag stored in the respective master objects' header. If they match, then it is a valid pointer. When an application

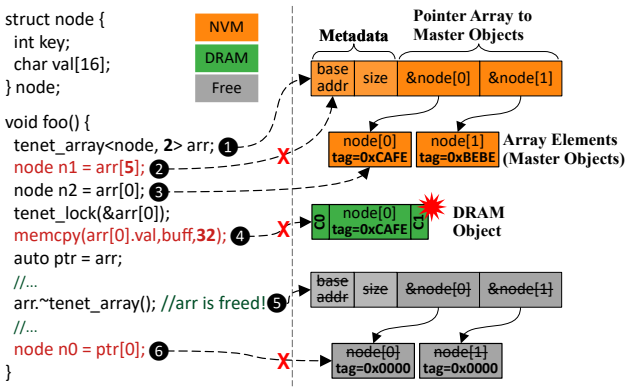

Figure 5: Memory safety design for arrays. ❷ and ❹ are spatial safety violations due to out-of-bound read (detected by bounds checking) and write (detected using canaries), respectively. ❻ is temporal safety violation due to use-after-free (detected using pointer tags).

accesses a master object with a dangling pointer, tag matching would fail; the tag in the header would either be zero (if the address is already freed) or different random value (if the free-ed address is reallocated). In that case, TENET cuts the version chain access and aborts the transaction.

Correctness. Once a master object is successfully dereferenced, it can be safely used without any further temporal safety checking *within the transactions' lifetime*. This is because TENET (and TimeStone) uses an RCU-style, epoch-based garbage collection scheme so it never frees an object (and its versions) with live references from other transactions; *i.e.*, an object will be free-ed only when all the transaction that has live references exits. Also, a DRAM object can be dereferenced only via its NVM object and TENET cuts the version chain access upon detecting a dangling pointer, which indirectly guarantees temporal safety for DRAM objects.

4.3.3 Spatial and Temporal Safety for Array Objects

In TimeStone, an array is stored and accessed as a single pointer. Even if the application just reads/writes to one array element, TimeStone dereferences the entire array. Such a design is highly unsafe. For instance, once the entire array is dereferenced, a buggy application can read/write out-of-bounds resulting in an undetected corruption. This is a notoriously hard problem even in the DRAM world. To address this, we redesigned the array interface in TENET. *An array is internally represented as an array of pointers where each array index stores a pointer to its element*. With this design, TENET dereferences only the array index that the application intends to read/write. If the application accesses an index that is out-of-bound, TENET aborts the transaction.

Array interface. ❶ in Figure 5 presents the TENET's array interface. In TENET, each array element is a master object; and an array consists of pointers to these master objects along with the base address and size information. This representation is internal and the application accesses its array in the traditional *C semantics*. We do not present the pseudocode

for our interface due to space limitations. Essentially, TENET retains the C-style semantics by leveraging C++ operator overloading. `tenet_array` class overloads the necessary operators to hide the internal representation. For instance, the array access operator (`[]`) is overloaded to perform bounds checking, then access the master object at the index. Similarly, other operators (=, +, -, etc) are also appropriately overloaded to retain the programmability and to make the interface transparent. However, this representation requires additional memory to maintain pointers to the array elements. An `N` element array requires a space of `N*sizeof(N)`, whereas TENET requires an additional `sizeof(void*)*N` space to maintain the pointers.

Memory safety validations. Figure 5 illustrates how TENET enforces memory safety for arrays (`arr` with two elements). The canary-based spatial safety and the tag-based temporal safety apply to every array element. In addition, TENET performs bounds checking for every array dereference using the base address and size metadata *i.e.*, `index > size` (❷). In ❹, a transaction writes to the `val` out-of-bounds, TENET detects this violation by inspecting the corrupted canary bits in the commit phase. In ❻, transaction dereferences a dangling pointer (freed in ❺) and TENET detects it by comparing the tags (`0xCAFE` $\neq$ `0x0000`) during the object dereference.

4.4 Enforcing fault tolerance Against UMEs

This section explains the synchronous log replication and the off-critical path master object replication design to guarantee fault tolerance against UMEs.

4.4.1 Transaction Log Replication

As illustrated in Figure 4, the primary log pool on the NVM consists of all the transaction logs (`OLog` and `CLog`). TENET maintains a consistent backup of the primary log pool by *synchronously* replicating the logs on the critical path, *i.e.*, when an `OLog` or a `CLog` in the primary log pool is updated, the corresponding log in the replica log pool is also updated. Atomicity for primary and replica log writes is inherently guaranteed by the transactions' commit protocol (§4.1.3); *i.e.*, TENET commits a transaction only when both the logs are updated. So, if a crash happens before updating the replica log, then the transaction is considered to be aborted and the partially written log entries are discarded during the recovery phase. Similarly, during log reclamation, the primary log is reclaimed first and the replica log is reclaimed up to the same point to maintain consistency. TENET ensures that pages in the primary and the replica log pool do not overlap by maintaining two disjoint NVM pools for the primary and replica log pool. In this way, TENET *can recover from multiple UMEs even if it spans across many pages within a log pool.*

Why replicate logs on the critical path? TimeStone buffers the updates to the master objects in the `OLog` and `CLog` to maximize the write coalescing. Hence, if the logs in the primary pool are corrupted, it may cause a significant amount of data loss during the recovery. As a result, TENET replicates the primary log pool synchronously to ensure that there is always a consistent backup. Thus, TENET can simply use the replica log pool to recover the NVM data *without losing any committed updates*. TENET uses NVM to reduce the performance overhead as the replication is done in the critical path.

4.4.2 Off-critical Path NVM Replication to SSD

TENET makes three critical design choices for a performant and cost-efficient NVM (master) objects replication: (1) objects are replicated to SSDs instead of NVM to reduce the storage cost overhead, (2) replication is performed out of the critical path to reduce the performance overhead (§4.4.3), and (3) TENET uses grace period semantics to enforce NVM-SSD consistency to guarantee loss-less recovery (§4.4.4).

4.4.3 Off-critical Path Writes to SSD

TENET leverages `io_uring` [8] for accelerating SSD writes. `io_uring` is a high-performance asynchronous IO framework. `io_uring` maintains two queues, a submission queue (SQ) where the TENET adds its disk write requests and a completion queue (CQ) where TENET can poll for the completed disk writes. Both queues are shared between the kernel and the user space, which further reduces the context-switching overhead for request submission and polling.

Technique. TENET maintains a per-writer replica buffer in the NVM, where writers enqueue the new master objects that are created in the ongoing transaction and the objects that are updated with the latest checkpoints from the `CLog` (❽ in Figure 4). TENET then spawns multiple workers to visit the per-thread replica buffer and issue the disk writes using `io_uring`'s submission queue. The workers then poll for the request completion in the `io_uring`'s completion queue and exit only when all the requests are completed. TENET creates a separate disk file for each master object pool; during replication, TENET *writes a master object at the disk file offset, same as the objects' corresponding NVM file offset.* This is critical to correctly roll back the corrupted page from the disk to the NVM during the recovery.

4.4.4 Enforcing NVM-SSD Consistency

Although replication is asynchronous, TENET *guarantees that no committed data will be lost upon either a crash or a UME.* TENET accomplishes this by leveraging the `OLog`, `CLog`, and grace period detection.

Grace period detection in TimeStone. A grace period is the quiescence period, in which all application threads that entered the critical section (since the start of detection), finish, and exit their respective critical section. A background thread (`gp-thread`) continuously detects the grace period, and publishes the detected grace period timestamp. TimeStone uses the timestamp to safely reclaim/free the obsolete entries/objects in the `TLog`, `CLog`, and the `OLog`. TENET extends this design to enforce NVM-SSD consistency.

Modified grace period detection in TENET. To detect a

grace period, the gp-thread not only waits for all the threads to exit the critical section but also waits for all master objects that are created/updated by these threads to be written to the SSD. The key invariant is that *when a grace period is detected, it guarantees that all master objects created/updated in that window are persisted to the SSD*. This means that the TLog, OLog, and the CLog will not be reclaimed until the disk writes are guaranteed to be persisted. That is because gp-thread will not publish the grace period timestamp unless the disk writes are completed and without it the logs can not be reclaimed. *In a nutshell, all the updates that are not persisted in the SSD are guaranteed to be either in the OLog (newly created master objects) or in the CLog (updates to the existing master object).*

Guaranteeing consistent loss-less recovery. If a UME occurs before the SSD writes finish, during recovery, TENET can restore the NVM objects with the stale SSD replica (from the previous grace period). Then it uses the CLog to update the existing master objects with the latest checkpoints and uses OLog to recreate the new master objects that are missing in the stale replica. Note that TENET maintains a consistent backup of OLog and CLog at all times (§4.4.1). Also, the OLog and CLog execution are idempotent *i.e.*, re-executing the same log entries multiple times does not violate the consistency. TENET can tolerate multiple UMEs across any number of pages in a master object pool as it replicates to the SSDs. Given at least one of the log pools is consistent, TENET can recover up to the last committed transaction. Note that even if both the log pools are affected by UMEs, TENET can still recover the master objects to the state of last grace period.

4.5 Recovery

(1) Recovering from non-UME crashes. This recovery includes recovering from a system crash or a memory safety violation. Upon restart, the recovery procedure is of two steps: (1) CLog replays, where all the entries in the CLog are replayed to set the master objects to a consistent state. This step is necessary to bring all the master objects to the latest checkpointed state. (2) Then all OLog entries are sorted based on their commit-ts and replayed sequentially in the exact sorted order. This will bring the master objects to the last committed state before the crash occurs. Note that, if the crash happens due to a memory safety violation, a developer should fix the bug to avoid repetitive non-UME crashes.

(2) Recovering from a UME crash. Upon restart, if TENET cannot open its NVM pools, it indicates a UME has occurred. The recovery steps depend on the victim pools' type.

UME in the master object pool. TENET identifies the corrupted physical offset using the ndctl tool [9] and then extracts the corresponding logical file offset. TENET brings the entire page where the corrupted offset belongs from the replica disk file. Then TENET allocates a new NVM page using fallocate and updates it using the disk replica. Finally, it deallocates the corrupted page and removes it from the operating system's bad block list. Once NVM is restored, TENET recovers similar to the non-UME crash as explained in **(1)**, *i.e.*, CLog replay followed by the OLog replay.

UME in a log pool. TENET does not need to access the disk to fix the bad page. Instead, it fixes the affected NVM page by allocating a new empty page. Then TENET uses the uncorrupted backup log pool to perform CLog and OLog replay. At the end of the recovery, it frees all the CLogs and OLogs, and new logs are allocated during the normal execution.

5 Implementation

TENET library is implemented in C and C++ which is ∼11K LoC. The core TENET library includes the TimeStone PTM (∼7K LoC), memory safety checks (∼1.5K LoC), and the NVM-SSD replication (∼2.5K LoC). We rigorously tested TENET with a carefully curated set of unit tests, functional tests, and integration tests along with the offline testing tools such as the Pmemcheck [48], Address sanitizer [80] to ensure correctness of our implementation.

6 Discussion

In this section, we discuss the key takeaways in TENET (§6.1) and the applicability of TENET's ideas on ARM architecture (§6.2). We also discuss the limitations and potential future research directions in §6.3.

6.1 Leveraging the Concurrency Guarantees of PTM

Enforcing low overhead spatial safety. Most PTMs perform out-of-place updates to enforce the Isolation property (ACID) [30, 40, 56, 62, 68, 87], to support concurrent read and write [30, 40, 56], and to enable write batching [30, 56, 87]. These PTMs have at least two separate domains: one in which new updates are made and buffered, and another that contains consistent data (*i.e.*, old updates) to which the new updates are eventually merged. TENET leverages this property to enforce a separate protection domain, such a design enables it to use light-weight techniques such as MPK and canaries to enforce spatial safety *without having to check every access*.

PTMs such as the libpmemobj [47] that perform in-place updates can be modified to perform out-of-place updates as done in Pangolin [94]. Although Pangolin uses microbuffering to perform out-of-place updates, it relies on expensive data checksum to enforce spatial safety *i.e.*, checksum is calculated and verified every time the data is moved to and from the microbuffers. SafePM [27] relies on compiler instrumentation of loads and stores and hence it needs to perform spatial and temporal safety checks at every access resulting in a high performance overhead (§7.3).

Enforcing low overhead temporal safety. Almost all PTMs support a stronger Consistency (ACID) guarantee such as linearizability or serializability. Such PTMs usually perform conflict checks (*i.e.*, read/write set validation) during the commit phase and the transactions are aborted if a read-write conflict is observed during the validation. In the context of temporal safety, this means that objects with live references

in any on-going transaction will not be freed until those transactions finish. Unlike the prior PTM works, TENET leverages this property to perform temporal safety checks only at the first dereference and avoids redundant checks during every pointer deference in a transaction. This is because, once an object is dereferenced, it can not be freed by concurrent transactions, a inherent guarantee provided by PTM.

6.2 TENET's Ideas on ARM Architecture

ARM processors support memory domains [2], which is similar to Intel MPK except that the permission switch happens in the OS kernel. Moreover, ARM processors have been supporting virtual address (pointer) tagging (upper 12-16 bits) at the hardware level and it is shipped with the *top byte ignore (TBI)* feature [14, 19, 23]. Therefore, we believe that TENET's ideas can be applied beyond x86 architectures.

6.3 Limitations and Future Work

Protecting against intra-object overflow. Protecting against intra-object overflow is a hard, open research problem. Even the state-of-the-art techniques, such as BOGO [95] do not protect against intra-object overflow. We believe that protecting against intra-object overflow with reasonable performance overhead would require significant architectural changes and/or compiler-level instrumentations because of the fine granularity of protection [46, 90]. However, TENET protects the transactional metadata which are essential for correct execution and recovery from the intra-object overflow. We do this by placing an additional intra-object canary between the metadata section and the application data section in a DRAM object (not shown in the figures). This restricts the corruption to only the application data section of an object.

Protecting against the code outside the transaction. TENET already protects the NVM data from spatial safety violations due to the code outside the transaction by using MPK. However, it is possible to corrupt the DRAM objects outside the transaction and TENET may not detect such corruption, particularly the ones that do not overwrite the canaries. One way to protect the DRAM objects is to protect all the `TLogs` using the MPK and allow to switch permission only within the TENET library. However, as `TLog` is per-thread and there are only 16 MPKs available, we may need to employ MPK virtualization [74] to offer a more fine-grained protection.

Impact of shorter tags. In TENET, we use all the upper 16-bits to store the pointer tag; expansion of address space in the future will reduce the number of available bits thus making the tag range shorter. TENET allows to reuse of duplicate tags across different pointers, but if the bits are too few (e.g., only 4 bits are available), reusing tags may cause false negatives. In TENET, tag reuse becomes a problem, only if the reallocated pointer is assigned with the same tag (that it had before last free), which makes TENET 's temporal safety detection probabilistic. Reusing tags across different pointers or the same pointer with non-consecutive reallocations results in a deterministic detection. As the CPU vendors are extending hardware support for pointer tagging, we believe that expanding this idea to overcome bit limitations (e.g., similar to x86 segmentation overcoming 64KB address limitation) will be an interesting future work.

7 Evaluation

We evaluate TENET by answering the following questions, (1) what are the performance overhead of TENET's memory safety and off-critical path disk replication techniques (§7.1)? (2) How does TENET perform in comparison with the other state-of-the-art memory safe PTMs (§7.3)? (3) What is the tail latency of TENET (§7.4)? (4) How does TENET fare in the bug detection, correction, and recovery stress tests (§7.5)?

Evaluation platform. We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 Physical cores, 256GB of NVM (2×128GB), 32 GB of DRAM (2×16GB) per socket, and 2×1TB M.2 SSDs (Samsung 970 EVO). We used GCC 11.2.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 5.16.12 with `io_uring` support.

Configuration. We preset the size of `TLog` and `OLog` to 8 MB and `CLog` to 32 MB, respectively. We also present the performance analysis for varying log size in §7.4. We use two SSDs for NVM replication *i.e.*, one SSD per socket. Throughout our evaluation, we present two versions of TENET: (1) **TENET-MS** – *which enforces only memory safety (i.e., no NVM/SSD replication)*, and (2) **TENET** – *which enforces both memory safety and NVM/SSD replication for fault tolerance*. For microbenchmarks, we initially warm up the data structures with 1 Million (M) keys followed by executing a mix of lookup, insert, update, and delete operations for 60 seconds as done in the prior PTM works [30, 40, 56, 75–77, 87]. For the real-world evaluation, we use the YCSB benchmark [29] to evaluate TENET's B+Tree based key-value store engine for 10M keys, we use 8 bytes integer keys and 100 bytes values with Zipfian distribution. We present the average performance of 10 runs, with an average error rate of ±1.8%.

7.1 Performance Analysis of TENET

Figure 6 compares the performance of TENET-MS and TENET against the TimeStone for three different workloads with varying read/write ratios. Comparing TENET-MS and TENET with TimeStone will enable us to quantify the overheads due to memory safety and fault tolerance techniques.

7.1.1 TENET-MS vs TimeStone

For the read-dominated workloads, TENET-MS performs mostly on-par (< 5% overhead) or slightly better than the TimeStone. This is because reads in TENET-MS require only temporal safety checks and the overhead from spatial safety checks are negligible due to the lower write ratio. The low overhead temporal safety checks can be attributed to our in-place pointer tagging technique wherein it only requires one shifting operation for extracting the tag from the pointer and one compare operation for validating the extracted tag.

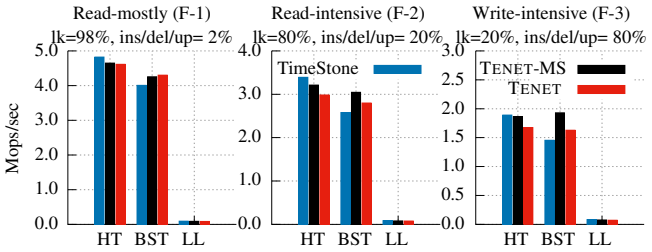

Figure 6: Performance comparison of TENET-MS and TENET against TimeStone for Hash Table (HT), Binary Search Tree (BST), and Linked List (LL) for 24 threads.

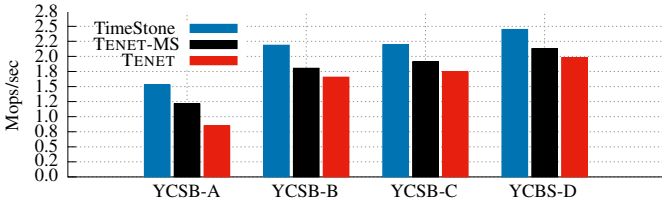

Figure 7: Performance comparison of TENET-MS and TENET against TimeStone for the B+tree key-value store with 24 threads.

For write-intensive workload, TENET-MS performs on par with TimeStone; this shows that our canary based spatial safety checks incur only a minimal overhead. For BST, TimeStone suffers from high transaction aborts due to lock conflicts on parent nodes. Unlike the BST, hash table is inherently more concurrent and incurs lower aborts due to less lock conflicts. Memory safety validation steps in TENET-MS reduce the aborts; our further analysis revealed that TimeStone incurs about 3.5× more aborts than TENET-MS for BST. Consequently, TENET-MS performs on par with TimeStone for hash table and slightly faster in case of a BST.

7.1.2 TENET vs TimeStone

In addition to memory safety, TENET guarantees fault tolerance by performing NVM/SSD replication. For read-mostly workloads, TENET performs on par with that of TimeStone and TENET-MS. Due to a lower write ratio, the number of log writes, and master object writes are less; consequently replication does not add any significant overhead. However, the replication overhead becomes evident as the write ratio increases from 20% to 80% and TENET performs up to 12.6% and 18% slower than the TENET-MS and TimeStone, respectively. As the master objects are inserted/deleted/updated frequently, the replica writes to SSD also increases. Therefore, grace period detection is relatively longer in TENET as the `gp-thread` has to wait for all the SSD writes to complete. A longer grace period detection increases traffic in the `TLog` as the log reclamation becomes slower. Overall, TENET adds a modest overhead (< 18%) over TimeStone while enforcing memory safety and fault tolerance.

7.2 Real-world Workload Evaluation

We built a B+tree-based key-value store using TENET; we chose B+tree (fanout=64) to test and evaluate our array interface but any other data structures can also be used. Figure 7 compares the performance of TENET-MS and TENET key-value store against the TimeStone key-value store.

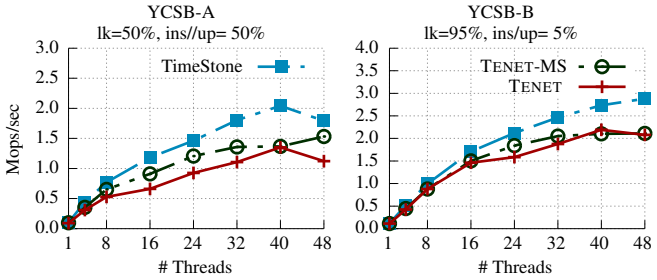

Figure 8: Scalability of TENET-MS and TENET for B+tree

PTM	Spatial Safety	Temporal Safety	UME	NVM Cost
Libpmemobj [47]	No	No	Yes	High
TimeStone [56]	No	No	No	None
SafePM [27]	Yes	Yes	No	Moderate
Pangolin [94]	Partial	No	Yes	Moderate
TENET-MS	Yes	Yes	No	None
TENET	**Yes**	**Yes**	**Yes**	**Low**

Table 1: Comparison of TENET against other PTMs.

TENET-MS. TENET-MS is 17% slower than the TimeStone across all YCSB workloads. For data structures (that do not use an array), such as the hash table, every read to a hash node requires only one object dereference because each hash node is a master object. But for a B+tree, reading one leaf node requires a 2× fanout (2×64) number of dereferences as each array element (of the key-value array) is a master object. Although TimeStone incurs the same number of object dereference, the additional temporal safety checks during the object dereferencing in TENET-MS causes a 17% slowdown.

TENET. For write-intensive YCSB-A, TENET performs 41% slower than TimeStone. This is because of lower chances of write coalescing in the `TLog` and `CLog`. As the writes happen at the array element level, the chances of an array index being repeatedly written to is less. This is the worst-case scenario for TimeStone as it relies on maximizing write coalescing on DRAM objects to reduce NVM writes. Lower write-coalescing causes frequent checkpoints (from `TLog`) on `CLog` and frequent checkpoint writebacks (from `CLog`) to the NVM object. TimeStone just performs frequent writebacks to the NVM object; for TENET, increase in the number of writebacks also increases the SSD writes due to replication. This trend is corroborated by the performance of TENET for read-intensive YCSB workloads (B, C, and D), where it exhibits only a 21% slowdown against TimeStone. This is almost half of the slowdown experienced for the YCSB-A workload (41%) as the number of SSD writes are lower in read-intensive workloads. In a nutshell, TENET guarantees memory safety for arrays (TENET-MS) with a modest 17% overhead and providing fault tolerance adds an additional 24% overhead due to the reduced write coalescing in TimeStone.

7.3 Comparison with Other PTMs

Table 1 compares the protection scopes of PTMs; TENET is the only PTM to offer full memory safety and cost-efficient

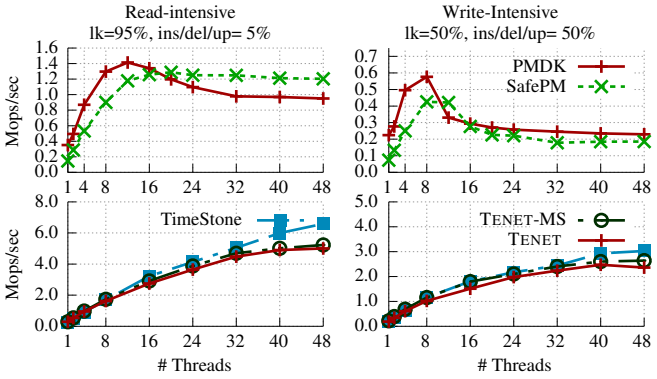

Figure 9: TENET-MS vs SafePM: performance overhead study with hash table for read-intensive and write-intensive workloads.

fault tolerance. We have discussed the limitations in the protection scope of prior works in §2.3. Moreover, TENET incurs a relatively minimal performance overhead as compared to SafePM (Figure 9) and Pangolin which incurs up to 60% and 67% overhead over the `libpmemobj`.[1] To ensure fairness, we compare SafePM and TENET-MS on basis of performance overhead incurred over their respective baseline PTM. Note that SafePM does not guarantee fault tolerance against the UMEs, so we use only TENET-MS for comparison.

As shown in Figure 9, SafePM performs up to 67% slower than the `libpmemobj` across both the workloads. When the `libpmemobj`'s performance saturates after 16 threads, SafePM performs on-par; this is because the high contention overhead in the `libpmemobj` amortizes the memory safety overhead in SafePM. SafePMs' overheads come from: (1) additional undo logging to guarantee crash consistency for the memory safety metadata. Note that this undo logging is in addition to the ones performed by the `libpmemobj` transaction, (2) the memory safety metadata must be accessed for every read and write which further slows down the performance.

Unlike the SafePM, TENET-MS guarantees memory safety with a modest 5%-8% performance overhead; because, (1) it does not require additional crash consistency for memory safety metadata as the pointer tags are embedded in the objects, and (2) memory safety checks are performed only once per transaction (on-commit and on-first-dereference).

7.4 Other Evaluations and Analysis

Scalability analysis. Figure 8 and Figure 9 shows the read and write scalability of TENET-MS and TENET for hash table and B+tree, respectively. Both TENET-MS and TENET show good read and write scalability for B+tree and hash table. The performance difference across thread counts are consistent with what is observed for 24 threads in Figure 6 and Figure 7. For read-intensive workloads, TENET-MS and TENET show less than 5% performance slowdown for a hash table and a 17% (TENET-MS) and 24% (TENET) slowdown for a B+tree. For a write-intensive hash table, TENET-MS and TENET exhibit a 5% and 18% slowdown respectively, while for B+tree, TENET-MS and TENET exhibit a 17% and

[1] Directly referenced from the paper as Pangolin is not open-sourced.

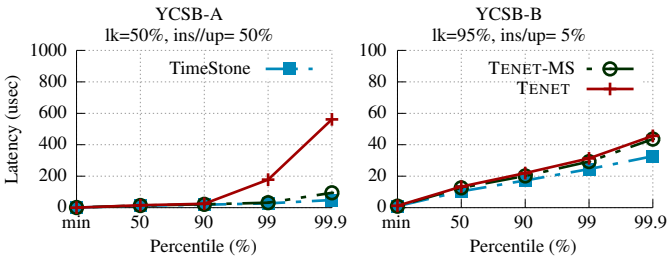

Figure 10: Tail latency comparison of TENET-MS and TENET against TimeStone for B+tree with 24 threads.

44% slowdown. Overall, both TENET-MS and TENET scales on-par with TimeStone; this shows that the TENET*'s memory safety and fault tolerance techniques does not impede the original scalability of TimeStone*.

Storage cost analysis. With TENET, the DRAM space usage is bounded by the size of `TLog` (8MB). TENET stores the replica logs in the NVM and this is bounded by the size of `OLog` and `CLog`. TENET replicates the application data structure to the SSD; given the $/GB of SSD ($0.15) and the NVM ($10) [15, 20], TENET saves ∼60× on storage cost when replicating the entire NVM space (512GB) to the SSD as opposed replicating to the NVM. In addition to the cost benefits, TENET can recover from multiple UMEs spanning across multiple pages while Pangolin can recover only from a single page is corruption.

Tail latency. Figure 10 shows the tail latency of TENET-MS and TENET compared against the TimeStone. As done in prior works [55, 60], we sample 10% of operations so that the tail latency calculation does not overshadow the performance. TENET-MS performs on-par with TimeStone, which shows the efficacy of our memory safety techniques. However, for write-intensive YCSB-A, TENET's tail latency spikes up at the 99th and 99.9th percentile. This is because of the additional writes incurred while performing replication to the NVM/SSD for fault tolerance. For read-intensive workload, TENET's tail latency is almost on par with TimeStone as lower ratio reduces the number of SSD writes. TENET-MS shows similar tail latency to that of the TimeStone across workloads as it does not perform replication. We believe our fault tolerance design can be further optimized for tail latency by making log writes asynchronously, which would be an interesting future work.

Log size sensitivity. To study the impact of log size on the performance, we present the relative performance of TENET for varying log sizes using a concurrent hash table with 1 and 24 threads (Figure 11). We show the performance only for write-intensive workloads as read-intensive workloads are less sensitive to the log size. The X-axis represents the log size, and the Y-axis represents the relative performance normalized to the default log size used in all the previous evaluations. TENET's performance increases up to 21% with the increasing log size. As the log size is decreased, the performance drops to 38%. As the log size increases, the writers spend less time reclaiming log space and hence better performance.

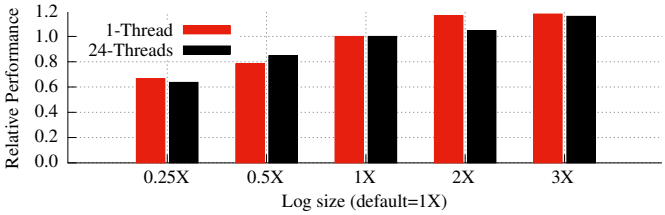

Figure 11: Performance sensitivity of TENET for varying log sizes.

Alternatively, for smaller log sizes, the writers spend more time reclaiming log space. TENET requires all SSD writes in a grace period window to be persisted before reclaiming the log space, further increasing the pressure on the writers. So we observe a larger performance drop (38%) for a smaller log size and relatively a smaller performance gain (21%) when the log size is increased. We confirmed that this behavior is consistent across different thread counts and data structures.

7.5 Error Detection and Correction

Spatial safety test. Our test cases select transactions at random to intentionally cause *buffer overrun* bugs on a B+tree leaf nodes' value pointer (`p_val`) and to access the key array (in a B+tree node) *out-of-bounds*. For the buffer overflow bug, the erroneous transactions execute a `memcpy` on the `p_val` for 1KB where the `p_val` pointer is of size 100 bytes. We also tested intra-array overflow with a smaller size of 128 bytes. For out-of-bound access, the erroneous transactions access the key array at index *96*, which is beyond the original fanout (64). For all test cases, TENET detected spatial safety violations in the commit phase and aborted the transactions, returning an exception to the B+tree code. In our 200 random tests, TENET detected spatial safety violations 100% of the time.

Temporal safety test. We modified the delete function in our open-chaining hash table benchmark to free the target node and *not update the previous nodes' next pointer (`p_next`)*. A randomly chosen transaction executes the buggy delete logic and spawns read transactions to access the dangling `p_next`. TENET detected the dangling pointer access during the object dereferencing phase and returned an exception to the application. Further, to test the case where a free-ed address may be reallocated again, we kept allocating a new hash node until the free-ed NVM address was reallocated. Our test case then waits for a transaction to access the *dangling `p_next` (reallocated)*. We repeated both the temporal safety tests 200 times, and TENET detected dangling pointer access and returned an exception to the application.

UME Test. We used the `ndctl` utility tool (*`ndctl inject-error`*) for injecting a UME at a specified offset [9]. While running the benchmark, we first injected a UME in the log pool, particularly on a randomly chosen `CLog`. TENET's `SIGBUS` handler received the OS notification and terminated the program gracefully. Upon restart, TENET rightly identified the corrupted log pool and successfully recovered using the replica log pool. We also injected UME in one of the master object pools and observed that TENET restored the NVM status successfully using the SSD replica. Both these tests were repeated multiple times and TENET successfully recovered the hash table without losing any data. The recovery time for TENET and TimeStone are similar, bounded by `OLog` and `CLog` size (not shown due to space constraints). The SSD access is performed in the background using `io_uring` and the cost is relatively small. Our future work will develop techniques to accelerate recovery.

8 Related Work

DRAM based memory safety techniques. Memory safety violation in the DRAM has been extensively studied in the security community [26, 32, 35, 36, 59, 69–71, 73, 79, 81, 83, 84, 92, 95]. In fact, our work was inspired by this line of research which essentially conveys that memory safety violations are the source of all evils. But the downside of these techniques is that they suffer from high performance overhead (up to 200%). In TENET, we reduce the performance overhead by leveraging the concurrency properties of the PTM and also by limiting our scope of protection (*e.g.*, no support for control flow attacks). Moreover, applying these DRAM based techniques to NVM is non-trivial as they are not designed to be crash consistent and adding crash consistency to these techniques comes with its own set of challenges and may potentially increase the performance overhead.

NVM bug finding techniques. There are a plethora of works on detecting crash consistency bugs in the NVM software [37, 38, 58, 63, 64, 72]. These techniques primarily focus on detecting bugs that violate crash consistency correctness such as atomicity, linearizability, and persistence ordering bugs; they neither focus on memory safety nor UMEs.

9 Conclusion

In this paper, we propose TENET. TENET enforces DRAM/NVM memory domain separation using MPK to prevent NVM writes out of TENET library. Additionally, TENET uses canary values and in-place pointer tagging to guarantee on-commit spatial safety and on-first-dereference temporal safety. Further, TENET proposes off-critical path NVM/SSD data replication to guarantee a performance and cost-efficient fault tolerance for the NVM data against the UMEs. Our evaluations showed the performance efficiency of TENET's techniques along with a thorough analysis on scalability, storage cost, and tail latency. Overall, TENET provides enhanced NVM data protection at a modest performance and storage cost as compared to the other state-of-the-art PTMs.

Acknowledgments

We thank the anonymous reviewers and Adam Morrison (our shepherd) for their insightful comments and feedback. This work was partly supported by the National Science Foundation under the grants CCF-2153748, CNS-1910593, and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2022R1F1A1076373).

References

[1] Amazon Signs up for Another 450MW of Solar, Giant Batteries. https://www.datacenterknowledge.com/energy/amazon-signs-another-450mw-solar-giant-batteries.

[2] ARM Developer Suite Developer Guide: Memory access permissions and domains. https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains.

[3] Blast Radius. https://pmem.io/glossary/#blast-radius.

[4] Build Persistent Memory Applications with Reliability Availability and Serviceability. https://www.intel.com/content/www/us/en/developer/articles/technical/build-pmem-apps-with-ras.html.

[5] Chrome: 70% of all security bugs are memory safety issues. https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[6] CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html.

[7] Dealing with Uncorrectable Errors. https://www.intel.com/content/www/us/en/developer/articles/technical/pmem-RAS.html.

[8] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[9] Error Recovery in Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/error-recovery-in-persistent-memory-applications.html.

[10] Frequently Asked Questions for Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/support/articles/000056000/memory-and-storage/intel-optane-persistent-memory.html.

[11] Google Thinks Data Centers, Armed with Batteries, Should 'Anchor' a Carbon-Free Grid. https://www.datacenterknowledge.com/google-alphabet/google-thinks-data-centers-armed-batteries-should-anchor-carbon-free-grid.

[12] Intel Donates Compute Express Link, a High-Speed Protocol for PCIe 5.0. https://www.tomshardware.com/news/intel-compute-express-link-pcie-5.0,38786.html.

[13] Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good.

[14] Intel Linear Address Masking "LAM" Ready For Linux 6.2. https://www.phoronix.com/news/Intel-LAM-Linux-6.2.

[15] Intel Optane DCPMM Cost. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[16] Last week Intel killed Optane. Today, Kioxia and Everspin announced comparable tech: Rumors of storage-class memory's demise may have been premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[17] Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[18] Microsoft slashes backup power costs with lithium-ion batteries. https://www.computerworld.com/article/2895064/microsoft-slashes-backup-power-costs-with-lithiumion-batteries.html.

[19] Pointer tagging for x86 systems. https://lwn.net/Articles/888914/.

[20] Samsung EVO NVMe M.2 SSD Cost . https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-1tb-mz-v7e1t0bw/.

[21] Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift.

[22] SMART brings Optane memory to AMD and Arm. https://blocksandfiles.com/2022/04/13/smart-brings-optane-memory-to-amd-and-arm/.

[23] The Arm64 memory tagging extension in Linux. https://lwn.net/Articles/834289/.

[24] Timestone Source Code. https://github.com/cosmoss-jigu/timestone/tree/master.

[25] Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[26] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[27] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. Safepm: A sanitizer for persistent memory. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, Rennes, France, April 2020.

[28] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, online, April 2021.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM.

[30] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, Vienna, Austria, July 2018.

[31] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[32] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[33] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st ACM/IFIP International Middleware Conference*, Virtual, December 2020.

[34] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[35] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Brining Bounded Model Checking to Heap Implementation Security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[36] Chris Evans. The poisoned NUL byte, 2014 edition, 2014. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html.

[37] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, online, October 2021.

[38] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, November 2022.

[39] Bill Gervasi. A Persistent CXL Memory Module with DRAM Performance. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/persistent-cxl-memory-module-dram-performance.

[40] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 913–928, Renton, WA, July 2019.

[41] Pekon Gupta. CXL Attached Persistent Memory: Implementing NVDIMM-N Like Architecture. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[42] Jim Handy and Thomas Coughlin. Persistent Memories Without Optane, Where Would We Be? In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[43] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[44] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[45] IBM. Power Failure Handling - IBM i in a Hosted Environment. https://www.ibm.com/support/pages/power-failure-handling-ibm-i-hosted-environment.

[46] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *Proceedings of the 48th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2021.

[47] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016.

[48] INTEL. Valgrind: an enhanced version for pmem, 2019.

[49] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[50] Raghunathan Modoor Jagannathan, Sulav Malla, and Parimala Kondety. Power Loss Siren: Making Meta resilient to power loss events, 2021. https://engineering.fb.com/2021/12/16/data-center-engineering/power-loss-siren/.

[51] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[52] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[53] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[54] Rajat Kateja, Nathan Beckmann, and Gregory R. Ganger. Tvarak: Software-managed hardware offload for redundancy in direct-access nvm storage. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2020.

[55] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, online, October 2021.

[56] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[57] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, online, July 2021.

[58] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, June 2014.

[59] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[60] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, August 2019.

[61] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[62] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[63] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[64] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[65] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[66] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Virtual, February 2021.

[67] Sulav Malla, Qingyuan Deng, Zoh Ebrahimzadeh, Joe Gasperetti, Sajal Jain, Parimala Kondety, Thiara Ortiz, and Debra Vieira. Coordinated priority-aware charging of distributed batteries in oversubscribed data centers. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 839–851. IEEE, 2020.

[68] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys17.

[69] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[70] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[71] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[72] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, November 2020.

[73] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, page 573–584, Chicago, IL, November 2010.

[74] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 241–254, Renton, WA, July 2019.

[75] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Onefile: A wait-free persistent transactional memory. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN)*, June 2019.

[76] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Persistent memory and the rise of universal constructions. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, April 2020.

[77] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In *Proceedings of the 24th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, online, March 2021.

[78] Arthur Sainio and Pekon Gupta. Scaling NVDIMM-N Architecture for System Acceleration in DDR5 and CXL-Enabled Applications. In *PM+CS Summit*. SNIA, 2022. https://www.snia.org/educational-library/scaling-nvdimm-n-architecture-system-acceleration-ddr5-and-cxl-enabled.

[79] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, Boston, MA, June 2012.

[80] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[81] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[83] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[84] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[85] Haris Volos. The case for replication-aware memory-error protection in disaggregated memory. *IEEE Computer Architecture Letters*, 2021.

[86] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, Dallas, TX, April 2020.

[87] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, high performance transaction for persistent memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[88] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2016.

[89] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[90] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[91] Yuanchao Xu, Wei Xu, Kimberly Keeton, and David E. Culler. Scaling NVDIMM-N Architecture for System Acceleration in DDR5 and CXL-Enabled Applications. In *Non-Volatile Memory Workshop (NVMW)*, 2022. http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-final5.pdf.

[92] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, August 2020.

[93] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka, Japan, October 2018.

[94] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[95] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.

[96] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.