LOaPP: Improving the performance of Persistent Memory Objects via Low-Overhead at-rest PMO Protection

Derrick Greenspan, Naveed Ul Mustafa, Andres Delgado,
Connor Bramham, Christopher Prats, Samu Wallace, Mark Heinrich, Yan Solihin
{derrick.greenspan, unknown.naveedulmustafa, andres.delgado,
co317248, Christopher.Prats, sa214201, heinrich, yan.solihin}@ucf.edu

***Abstract*—Persistent Memory (PM) is nearly as fast as traditional volatile memory while being denser and having the capability to retain data indefinitely. However, the long-lasting nature of PM means that, without encryption, it is vulnerable to data disclosure attacks. Recent research has introduced Persistent Memory Objects (PMOs) as an abstraction for PM that is both crash consistent and has the capacity to be secure against corruption and data disclosure attacks for PMOs at-rest. Previously presented PMO designs to protect against corruption and data disclosure attacks use whole PMO encryption/decryption with integrity verification (WEDI). While this design works, it is slow and inefficient.**

We address this problem with WEDI for the first time. First, we observe that a PMO can be broken into pages and that we can adopt demand paging for PMO encryption (per-page encryption). Second, we explore the design space of per-page PMO encryption and integrity verification, which we refer to as Low Overhead at-rest PMO Protection (LOaPP), and we discuss the trade-offs of each design. Third, we introduce a crash handler to ensure that PMOs are always secure, even in the face of crashes. Our new design, with per-page encryption alone, outperforms whole-PMO encryption without integrity verification (WED) by 1.4× and 2.6× for two sets of evaluated workloads. Adding per-page integrity verification on top of per-page encryption outperforms the original WEDI design by 2.19× and 2.62×.

I. Introduction

Persistent Memory (PM) offers byte-addressability, high data densities, and low latency access [35]. These features make PM a preferred choice to hold persistent data compared to a storage device. While the first commercial PM (Intel Optane) was recently discontinued, other products are entering the space, including CXL Persistent Memory [8], [2] and a battery-backed SoftPM approach [31].

To manage PM-resident data, prior work has proposed two main approaches: either host a file system or memory mapped files within PM [17], [29], or consider PM as a repository of Persistent Memory Objects (PMOs) [30], [32], [1].

A PMO holds persistent data in pointer-rich data structures without the backing of a file, while the operating system kernel manages the namespace, permissions, and sharing semantics of PMOs. A PMO is mapped/unmapped to/from the address space of a user process by invoking `attach()`/`detach()` system calls [13]. Once attached, PMO data is accessed directly as if it were traditional volatile memory. PMO systems provide the `psync()` system call as a primitive to persist data and to manage crash consistency: any modifications to a PMO are not made durable until `psync()`, and a crash will result in the PMO being restored to the last durable state of the most recent `psync()`. PMOs support key-based protection where a successful attach requires that a user-supplied key must match the system-stored key. Once attached, the granted access permissions are enforced by the kernel throughout the attach session. Like a file, a PMO can outlive the process that created it and survive system boots; like traditional pointer-rich data, a PMO can store buffers, pointers, and other assorted data structures.

A PMO can be shared among multiple reader processes simultaneously, but a writer process must have exclusive PMO access. Once created, a PMO is either *in-use* i.e., attached to a user process or *at-rest* i.e., not attached to any user process. Like files, PMOs are likely to spend most of their lifetime at-rest, holding the persistent data of user processes. This makes PMOs susceptible to data remanence attacks, where unless deleted, PMO data remains in plaintext in PM for a long time. Another vulnerability is when the adversary compromises the OS kernel to steal or corrupt a PMO's data. This results in the adversary stealing or corrupting a PMO's data without being noticed by the user processes accessing those PMOs. Therefore, protecting at-rest PMOs is as important as protecting in-use PMOs.

To protect at-rest PMOs, prior work [13] proposed *Whole*-PMO Encryption, Decryption and Integrity verification (WEDI). Encryption protects the PMO data from unauthorized reads, while integrity verification protects the PMO data from unauthorized writes while at rest. The approach assumes a trusted kernel implementing the cryptographic primitives for the system calls. For WEDI, on `detach()`, the kernel encrypts the whole PMO, computes and stores its checksum. On `attach()`, the whole PMO is decrypted and its checksum is computed and validated against the stored checksum; any modifications to the PMO while at rest will be detected, causing the attach to fail. Since a crash may restore the PMO state to the one from the last `psync()`, the PMO state after each `psync()` must be recoverable, hence the kernel must also compute and update the PMO checksum at each `psync()`.

A key problem with WEDI is that the latency of `attach()`/`detach()`/`psync()` system calls scales proportionally to the size of a PMO [13], and the latency is difficult to hide. An `attach()` call is on the critical path of the requesting process as a user process cannot access the PMO until it is mapped to its address space. The latency of `psync()` is also in the critical path because a consistent state must be achieved prior to continuing computation past a `psync()`. While the latency of `detach()` may be off the critical path, a subsequent `attach()` must be delayed until `detach()` completes, hence potentially exposing it. These mostly exposed and non-scalable latencies present an impediment to achieving better security even for in-use PMOs, since a prior work proposed frequent attaches and detaches to improve security [30]. They result in a higher total execution time of a user application with WEDI as compared to No Encryption, Decryption and Integrity verification (NEDI).

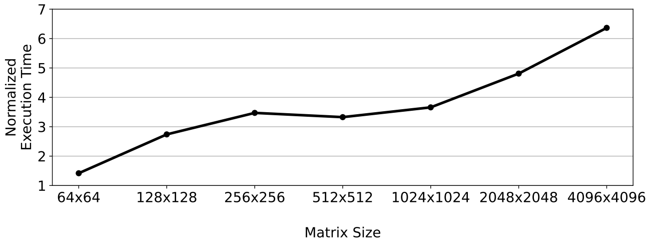


Fig. 1: Execution time of the Tiled Matrix Multiplication benchmark with WEDI, with a matrix size from 64×64 to 4096×4096 elements, normalized to that of NEDI.

To illustrate the extent of the problem, Figure 1 shows the execution time of a PMO-ported Tiled Matrix Multiplication (TMM) benchmark [9] with WEDI, normalized to those of NEDI for each corresponding size. The x-axis shows the effective size of the PMO holding the input matrix for the workload. The matrix size within the PMO increases from 64×64 items to 4096×4096 items (i.e., the PMO increases from $64 \times 64 = 4096$ bytes to $4096 \times 4096 = 16$MB). Each iteration of the performance critical loop attaches the PMO, updates an array element, psyncs the PMO and finally detaches it. The figure shows that the overhead of WEDI rapidly increases along the PMO size; from 1.5× at 4096 bytes to 6.4× at 16MB.

The source of WEDI's non-scalable performance is that it enforces protections (encryption, decryption, and integrity mechanisms) at the granularity of a whole *PMO* size irrespective of the actual working set size of a single attach session.

Thus, in this paper, we propose a new approach that we refer to as *Low-Overhead at-rest PMO Protection (LOaPP)* ("Low-App") scheme without lowering the security level. The key idea is to reduce the PMO overheads by protecting PMO data at a finer granularity (i.e. pages) and paying the protection costs only when data are actually accessed. While conceptually simple, we have to deal with several challenges. Note that PMOs use a shadowing approach, where each primary page is backed by a shadow page [13], to guarantee crash-consistent atomic updates. An important question is *what* to decrypt/verify/encrypt for low-overhead? I.e, shadow pages, or both shadow and primary pages? Another challenge, *when* to decrypt/verify/encrypt a PMO during an attach session. Finally, there is the question of how to ensure that LOaPP maintains the *checksum integrity* of PMOs. We conduct an extensive design-space exploration to address these challenges.

This paper makes the following **contributions**:

1) We propose a novel, *Low-Overhead at-rest PMO Protection (LOaPP)* scheme.
2) We present an exploration of LOaPP's design space with several performance optimizations to reduce the protection overhead.
3) We implement LOaPP on a Linux kernel, with a real system equipped with Intel Optane PMem. We extensively evaluate LOaPP on several workloads and find that our most performant design is 2.19× faster than WEDI. For FileBench, our most performant design is 2.62× faster than WEDI.

The rest of this paper is organized as follows: Section II discusses background knowledge and Section III discusses related work. Section IV describes our threat model, while the design of LOaPP and our performance optimizations are presented in Section V, while its implementation is detailed in VI. Evaluation methodology and results are presented in Section VII and VIII, respectively. Finally, Section IX concludes and discusses future work.

II. Background

This section discusses the necessary concepts and related work required to more fully understand per-page encryption with PMOs, including an overview of PM and the PMO subsystem design that we are using.

A. Persistent Memory

Persistent Memory (PM) is a class of memory technologies that retains data after power loss, is byte addressable, has access latency closer to volatile memory than storage devices, and is more dense than volatile memory. Given its characteristics, PM becomes attractive for hosting small and medium sized persistent data, while large data is likely to still rely on block-based storage. To manage persistent data residing on PM, one approach is to use a filesystem. Designs based on this approach store persistent data in files and then map them to the address space of a user process [5], [17], [29]. However, this approach must keep both memory metadata and file metadata consistent and reconcile their semantics. Furthermore, the approach creates large (≈ 13×) overheads compared to raw persistent memory device write bandwidth [17].

B. Persistent Memory Objects

An alternative approach of managing PM is to view it as a collection of Persistent Memory Objects (PMOs) [1], [19], [30], [32]. A PMO holds persistent data stored in potentially pointer-rich data structures, while the kernel manages the namespace, permissions, and sharing semantics of PMOs. Though our design of LOaPP is applicable to any PMO system, we implement LOaPP on the Greenspan PMO (GPMO) system[13]) The GPMO system is the only available PMO system that can work with a real operating system on real hardware. PMO system calls for the GPMO system are shown in Table I. A user process creates a PMO of a given name, size and key by invoking the `pcreate()` system call, while `pdestroy()` deletes a PMO and reclaims its space. Once created, a PMO is mapped into the address space of a user process by invoking `attach()`, making it accessible to the process. Conversely, invoking `detach()` unmaps a PMO from the address space of a process, making it inaccessible. Any PMO updates, performed after attaching it, are persisted in a crash-consistent way by invoking the `psync()` system call. When `psync()` is invoked, the kernel uses shadowing to achieve crash-consistent persistency of PMO updates.

TABLE I: PMO system calls.

Primitive	**Description**
pcreate(name,size,key)	Create a PMO `name` of `size` and `key`.
pdestroy(name,key)	Given a `key`, delete PMO `name`, and reclaim its space.
attach(name,perm,key)	Render accessible the PMO `name`, given a valid `key` with permissions `perm`.
detach(addr)	Render inaccessible the PMO `addr` points to.
psync(addr)	Persist updates to the PMO `addr` points to.

C. Crash-Consistency

Crash-consistency is an important requirement for managing persistent data on PM regardless of whether data is stored in memory-mapped files or persistent memory objects. An application, system, or power failure may cause partial or unordered writes to persistent data; in the absence of crash consistency, this can lead to the data being in some inconsistent state from which it cannot be restored. Logging and shadowing are two popular approaches for achieving crash-consistency. The GPMO system uses the latter whereby it maintains a primary and shadow copy of each PMO. The system provides a fundamental guarantee that, even in the case of a crash, either the primary or the shadow PMO remains consistent (i.e, free of partial and unordered writes). Based on the PMO's state at the time of a crash, one of the two copies are used to restore the PMO to a consistent state afterwards.

III. Related Work

A. Hardware support for memory encryption

There is hardware support for memory encryption; some examples include AMD's Secure Memory Encryption (SME) [18] and Intel's Total Memory Encryption - Multi Key (TME-MK) [15]. Additional related work include Secure Enclaves such as Intel's Software Guard Extensions (SGX) [7] and AMD's Secure Encrypted Virtualization (SEV). While theoretically these solutions could be used to accelerate PMO performance and improve security, this paper does not use them for several reasons.

First, AMD's SME is lacking because it uses a single securely generated key and encrypts the entire virtual address space. Under this scheme, the whole PM may be encrypted, but isolating one PMO from another through whole-memory encryption keys is impossible.

Second, although Intel's TME-MK design seems more promising, since different parts of memory can be encrypted with different keys, it suffers from a limited number of supported domains. For example, according to Intel's own specifications [15], a system supporting TME-MK has an *absolute theoretical limit* of $31,999$ domains that may be supported; but since it is anticipated that PMOs will be small, this limit is likely to be easily met and exceeded (for example, if each PMO is 4096 bytes large, the limit is exhausted after 128 GB). Furthermore, this limit is described by Intel as the *maximum* possible value supported by the CPU model specific register (MSR) for TME-MK; currently available systems support several orders of magnitude fewer domains [34]. Finally, TME-MK is only available on 3rd Generation Scalable Xeons or newer, and it is exclusive to Intel, whereas we are aiming for a design that is platform agnostic. For all of these reasons, we do not use TME-MK.

Finally, while secure enclaves provide the ability to completely isolate processes from one another, AMD's SEV and Intel's SGX solutions are lacking. First, they are vulnerable to Spectre and Meltdown-style attacks [4], [23]; second, Intel's current implementation is limited to only 93MB, third; Intel's support for it has been deprecated on modern processors [14], while AMD's SEV is designed per virtual machine rather than per process [18].

B. PMO Security

Prior work with PMOs has largely focused on protection of PMOs *in-use* (i.e., while they are attached). For example, [34], [30], [33] all focus on protecting in-use PMOs from attacks. MERR [30] proposes attaching a PMO only when needed and otherwise keeping a PMO in a detached state, while Xu et al. [34] proposes extending the MERR approach to use Intel Memory Protection keys (MPK) to limit access of PMOs only to those threads that require access (i.e., isolating PMOs between threads). TERP [32] expands on MERR by providing a compiler pass to perform automatic attaches and detaches.

Mustafa et al. [22] demonstrated that the use of PMOs break inter-process isolation, at least so long as a PMO is shared between multiple processes; Mustafa and Solihin [21] expanded on this to show that inter-process isolation can be broken even without sharing a PMO over time, so long as the processes are linked in some way.

IV. Threat Model

For our threat model, our goal is to protect at-rest PMO data to provide confidentiality and integrity. Note that our threat model is different from [30], [22], [21], as they require a PMO to be in-use to be exploited for a security attack. To fully protect a PMO, our protection for at-rest PMO data can be paired with other protection schemes for protecting in-use PMO data.

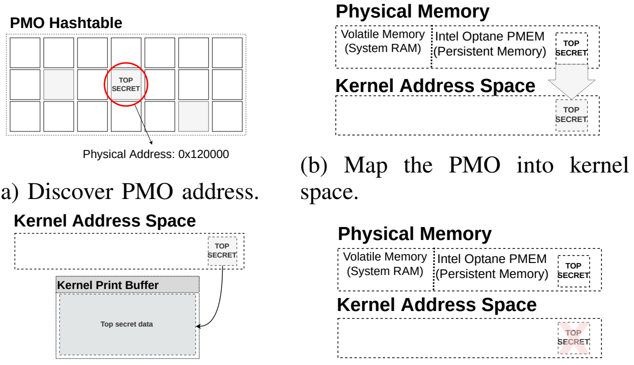
The attacker's goal is to either reveal or tamper with the confidential data belonging to a user-process stored within an at-rest PMO. We assume that the attacker knows or has the capability to find the location of the target PMO in memory.

Like files, PMOs are likely to spend most of their lifetime at-rest, holding the persistent data of user processes. One attack a PMO is susceptible to is data remanence, where unless deleted, PMO data remains in plaintext in PM for a long time. A stolen or improperly disposed of PM may be analyzed by the attacker to obtain sensitive data. Such attacks have been documented in data stored in files in hard drives [25]. Since then, filesystem encryption has become widely used to protect at-rest data in files (but no similar mechanism exists to protect at-rest data in PM).

Another attack we consider is an attack where the adversary compromises the OS kernel to steal or corrupt a PMO's data. We assume that the data structures residing in the PMO contain buffers and pointers, which may be targeted for overwrites by the compromised OS. This results in the adversary stealing or corrupting a PMO's data without being noticed by the user processes accessing those PMOs.

Our trust is limited to specific components of the system software, notably the Linux Kernel Crypto API [6], crucial kernel memory functions like memcpy and memset, and our PMO kernel subsystem. We assume that these components are devoid of any code vulnerabilities, a plausible belief since their code sizes are small enough to undergo formal verification, which has been done on similarly sized programs [20]. To illustrate, the Linux Kernel Crypto API for version 5.14.18 encompasses approximately 82500 source lines of code (SLOC), while our LOaPP PMO system contains about 2700 source lines of code, and the kernel memory functions comprises roughly 100 lines of architecture-specific inline assembly. This contrasts with the entirety of the kernel, which contains about 2.2 million lines. Consequently, our kernel subsystem, the critical memory functions (specific to architecture and written in assembly), and the Crypto API collectively contribute to a mere 0.4% of the overall kernel. Furthermore, we trust the encryption hardware of the CPU.

A. *Example Attack*



(a) Discover PMO address.

(b) Map the PMO into kernel space.

(c) Read mapped PMO.

(d) Unmap PMO.

Fig. 2: Steps of PMO example attack, from [13].

Figure 2, reproduced from [13], demonstrates a data-disclosure attack on an at-rest PMO. We assume the attacker has already exploited a kernel code vulnerability to manipulate its control flow. In step a), the attacker locates the physical address of the targeted PMO by navigating through the GPMO metadata hashtable. In step b), attacker maps the target PMO into the kernel virtual address space (e.g., through `vmalloc`/`ioremap` in Linux). In step c), the attacker copies the PMO's contents into the kernel print buffer, thereby exposing confidential information. Finally, in step d), the attacker clears the print buffer and unmaps the PMO from the kernel address space, effectively erasing any trace of the attack. Note that an attacker can also perform a data-injection attack by writing into the mapped PMO in step c) and persisting the updates (e.g., by flushing the updated pages).

V. Low Overhead at-rest PMO Protection Design

From the perspective of our threat model, we consider two scenarios: 1) when at-rest PMOs need to be protected only against unwarranted reads and 2) when protection against both unwarranted reads and writes is required. When a design D protects against unwarranted writes through Integrity (I) verification with a checksum updated at a system call c, we denote it as D/I_c. Otherwise, we denote the design only as D.

Our design space exploration is guided by finding answers to the two design challenges for at-rest PMO protection. **C1:** *what* should be decrypted, encrypted, or verified? **C2:** *when* to decrypt (D) a data item, verify (V), and update (U) its checksum, and encrypt (E) it during a PMO's attach session? **C3:** How to ensure the checksums are in a consistent state in the case of an application or system crash? This design space is shown in Table II.

The Whole PMO Encryption, Decryption, and Integrity verification (WEDI) design of [13] provides at-rest PMO protection at the granularity of an entire PMO (C1). Since the approach provides integrity verification and updates the checksum of an attached PMO on a `psync()` system call, we refer it as WED/I_p for the rest of the paper and is summarized by the first row of Table II. WED/I_p decrypts the entire PMO on an `attach()` request (C2.D). Recall that a PMO has both a primary copy and a shadow copy. Therefore, to achieve crash-consistent atomic PMO updates, the decryption is performed in two steps: first, the primary copy is decrypted and persisted into the shadow copy. Then, the shadow copy is copied back and persisted to the primary copy. The two-step process ensures that if a crash happens during the decryption step, at least one valid copy is available to recover the PMO to a consistent state. After decrypting, the checksum is computed on the decrypted PMO and verified against the stored one (C2.V). In the case of a successful verification, the PMO is mapped into the requesting process' address space. On `psync()`, after persisting all updates in the shadow copy, the PMO's stored checksum is updated (C2.U). On `detach()`, the shadow copy is first encrypted and persisted into the primary copy, and then the shadow copy is memset to zero and persisted (C2.E). Since the design always encrypts/decrypts a PMO in two steps, it offers an innate guarantee that the stored checksum will always match the data within the PMO (C3). More specifically, since the checksum is updated whenever `psync()` is, the checksum and the data within the primary copy will always match.

The rest of this section discusses the design of LOaPP starting from the base design to incrementally optimized versions.

TABLE II: LOaPP's design-space exploration.

Design	C1	C2				C3
		D	V	U	E	
WED/I$_p$	PMO	attach	-/attach	-/psync	detach	innate
BP/I$_p$	Both Pages	PF	-/PF	-/psync	detach	innate
BP/I$_d$	Both Pages	PF	-/PF	-/detach	detach	CrH
SP/I$_p$	Shadow Page	PF	-/PF	-/psync	psync	innate
SP/I$_d$	Shadow Page	PF	-/PF	-/detach	psync	CrH

A. Encrypting/Decrypting Both PMO Pages (BP)

WED/I$_p$ decrypts, verifies and encrypts all pages of a PMO, including those that are never accessed by the attaching process. This contributes to increasing the latency of all three system calls: `attach()` decrypts all the pages and then verifies the checksum at the level of the PMO, `psync()` updates this checksum, and `detach()` encrypts all the pages. To avoid paying the extra cost, our Low Overhead at-rest PMO Protection (LOaPP) design reduces the protection granularity to the *page-size* and decrypts PMO pages only on-demand. On an `attach()`, if the requested PMO is not already attached to another process and the user-supplied key matches the system-stored key, the call returns control immediately to the requesting process. However, the PMO is not mapped into the process' address space until a Page Fault (PF) happens. On a PF, the fault handler routine decrypts both the requested primary page and its shadow copy in a two-step crash consistent way (same as described for WED/I$_p$) i.e., C1 and C2.D. Since this design decrypts both pages, we refer it as BP. On `detach()`, the dirty pages that were already persisted in the shadow copy of the PMO by the last `psync()` are then encrypted and persisted into the corresponding primary copy (C2.E). The shadow copy is then destroyed, by calling `memset()` on the shadow pages, setting their values to 0, and then persisting. Since BP, like WEDI/$_p$, always perform decryption/encryption of PMO pages in two steps, it always maintains a valid copy of each PMO page, a PMO page that is free of any partial updates. Therefore, the design provides innate guarantees of crash-consistency (C3). Note that BP alone *does not provide integrity protection* against unwarranted writes.

BP offers several performance benefits: 1) By adapting on-demand paging, it does not decrypt/encrypt a PMO page until the Page Fault (PF) happens. This not only significantly reduces the *attach()/detach()* latency but also avoids unnecessarily delaying any non-PMO computation following an *attach()*. 2) By lowering protection granularity to the sizes of pages, unlike WED/I$_p$, BP decrypts/encrypts only those pages that are actually accessed in an attach session and avoids paying the extra protection cost for unaccessed pages.

a) BP with Integrity verification, checksum updated on `psync()` *(BP/I$_p$):* To protect against unauthorized writes in addition to unauthorized reads, we add integrity verification to the BP design. However, unlike the PMO-level checksum of WED/I$_p$, the new design maintains a per-page checksum. The design allows a decrypted page to be accessed by the requesting process only when the page fault handler verifies that the faulted (and decrypted) page's computed checksum matches with its stored checksum (C2.V). The checksum of *only* dirty PMO pages are updated on a `psync()` (C2.U), hence the design is referred as BP/I$_p$. Since per-page checksums are updated by means of a crash-consistent `psync()`, the design maintains the innate guarantee that the checksum and the data within the primary copy will always match (C3).

Note that maintaining per-page checksums increases the size of metadata per PMO (See SubSection V-D). However, the cost of maintaining extra metadata is outweighed by the significantly lower latency of `psync()`, compared to WED/I$_p$, as it only updates the checksums of dirty pages instead of updating a single PMO-level checksum. The optimization is likely to result in significantly lowering the protection overhead for applications frequently invoking `psync()` in an attach session.

b) BP with Integrity verification, checksum updated on `detach` *(BP/I$_d$):* While BP/I$_p$ significantly lowers the latency of `psync()` as compared to WED/I$_p$, it is still high compared to BP (which provides no integrity protection), especially when `psync()` is invoked more frequently in an attach session, as shown in Figure 7 of Section VIII. We ask the following question: Can we design a protection scheme that still provides integrity verification, ensures crash-consistency but further lowers the latency of `psync()`? One option is to update the checksum of dirty pages on `detach()` (C2.U) while the dirty pages themselves are still persisted by `psync()`. This design is referred as BP/I$_d$. While BP/I$_d$ reduces the `psync()` latency, it creates a checksum-consistency problem: If a crash happens between a `psync()` and the `detach()`, on reboot, there is a mismatch between between a page's data (persisted at `psync`) and its checksum (updated at `detach()`). To address the issue, we equip the BP/I$_d$ with a Crash Handler (CrH) routine that restores the checksum-consistency guarantee for BP/I$_d$ design (C3). The crash handler is discussed in Subsection V-F.

B. Encrypting/Decrypting only Shadow PMO Page (SP)

Unlike WED/I$_p$ that decrypts/encrypts the whole PMO on `attach()/detach()`, BP/I$_p$ and BP/$_d$ aim at reducing the `attach()/detach()` latency by decrypting/encrypting PMO pages only on-demand. BP/I$_p$ and BP/$_d$ have both the primary and shadow copies in decrypted form after a PMO Page Fault (PF). To reduce the latency of the two system calls even further, we propose a design that decrypts the primary page on PF into the shadow page (C2.D) and persists it in the shadow page but *does not* copy back and persist the shadow page to the primary page. In other words, only the Shadow Page is modified (C1), hence the design is referred to as SP. Like BP, SP also verifies the checksum of a page on PF (C2.V) and updates it on `psync()` (C2.U).

The SP design involves a trade-off: Since a shadow page is in decrypted form while its corresponding primary page is in encrypted form, a `psync()` operation *after* persisting a shadow page *must encrypt* and persist the shadow page into the primary page (C2.E). The additional operation of encryption *increases `psync()` latency*. Since updates are always guaranteed to be encrypted and persisted in the primary page by `psync()`, SP can simply zero the shadow pages on `detach()` and free them. As the updates are persisted by already crash-consistent `psync()`, SP maintains the innate guarantee of crash-recovery (C3). Note that the SP design is likely to reduce the protection overhead when an application attaches/detaches a PMO more often than psyncing a PMO (e.g., frequent read-only PMO attach-sessions).

a) SP with Integrity verification, checksum updated on `psync` (SP/I_p: Just as BP/I_p adds integrity protection to BP, SP/I_p adds integrity protection to SP with a page's checksum updated on `psync()` (C2.U). The SP/I_p design maintains an innate guarantee that the checksum will always match (C3).

b) SP with Integrity verification, checksum updated on `detach()` (SP/I_d): SP/I_d differs from SP/I_p in that it updates a page's checksum on `detach()` (C2.U), to further reduce `psync()` latency, but relies on the Crash Handler (CrH) routine to guarantee that the checksums on a detached PMO will always match the data stored within it (C3). SP/I_d performs integrity verification in the same way as BP/I_d.

C. PMO State Transition Diagram

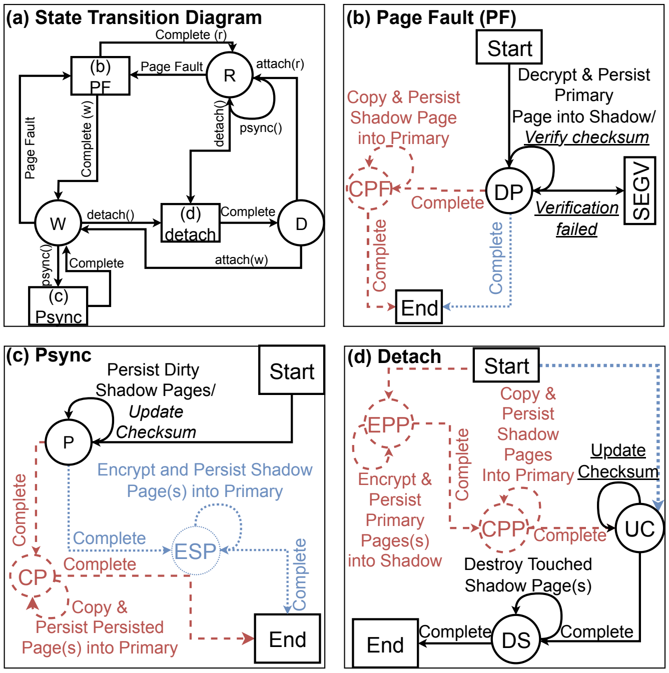

(a) State Transition Diagram
Complete (r)
(b) PF
Page Fault
R
attach(r)
Page Fault
Complete (w)
detach()
psync()
W
detach()
(d) detach
Complete
D
psync()
Complete
attach(w)
(c) Psync
(b) Page Fault (PF)
Start
Decrypt & Persist Primary Page into Shadow/ Verify checksum
Copy & Persist Shadow Page into Primary
CPF
Complete
DP
Verification failed
SEGV
Complete
Complete
End
(c) Psync
Persist Dirty Shadow Pages/ Update Checksum
Start
P
Encrypt and Persist Shadow Page(s) into Primary
Complete
Complete
ESP
Complete
CP
Complete
Copy & Persist Persisted Page(s) into Primary
Complete
End
(d) Detach
Start
EPP
Copy & Persist Shadow Pages Into Primary
Update Checksum
Encrypt & Persist Primary Pages(s) into Shadow
Complete
CPP
Complete
UC
Destroy Touched Shadow Page(s)
End
Complete
DS
Complete

Fig. 3: High-level state transitions for **BP** (red/dashed), and **SP** (blue/dotted)

Our design of Low Overhead at-rest PMO Protection (LOaPP) keeps per-PMO state information. PMO states and transitions among them are shown in Figure 3 for all designs i.e., BP/I_p, BP/I_d, SP/I_p, and SP/I_d.

a) State transition for BP, BP/I_p and BP/I_d: For the PMO state transitions in the BP, BP/I_p and BP/I_d designs, consider only the black/solid and the red/dashed parts of Figure 3. A PMO is initially in D (detached). On `attach(w)` or `attach(r)`, the PMO transitions D → W (write) or D → R (read). Note that BP maps PMO pages into the address space of the requesting process only on demand. That is to say, on a page fault, the PMO transitions W → DP or R → DP (decrypt and persist). In the DP state, the kernel decrypts and persists a primary PMO page into its corresponding shadow page, computes the checksum over the decrypted page and compares it to its stored checksum. In the case of a mismatch, a segmentation fault is reported (Note that checksum verification is not applicable to the BP design). Otherwise, the PMO transitions DP → CPF (copy and persist faulted) where the decrypted (and faulted) shadow page is copied back and persisted into the primary page. On completion, PMO transitions to the R or W state.

A `psync()` on a PMO in R triggers no state transition and nothing happens. A `psync()` on a PMO in W transitions the state W → P (persist), where all dirty shadow pages are persisted and their checksum is updated (only for BP/I_p). Once completed, the PMO transitions P → CP (copy and persist) where all persisted shadow pages are copied and persisted into their associated primary pages. Upon completion, the PMO transitions back to W.

On `detach()`, the PMO states transitions in the same way for BP and BP/I_p design, but differently for the BP/I_d design. For BP and BP/I_p, `detach()` transitions a PMO R → EPP or W → EPP (encrypt and persist) where all touched primary PMO pages are encrypted and persisted into corresponding shadow pages. On completion, the PMO transitions EP → CPP (copy and persist to primary) where all persisted pages in the shadow PMO are also copied back and persisted into associsaated primary pages. On completion, PMO transitions to Ds (destroy) where now all uncessary shadow pages are zeroed. Finally, the PMO transitions Ds → D.

For the BP/I_d design, `detach()` transitions a PMO R → UC (update checksum) where checksum of all PMO pages written in the attach session (i.e., zero for detach(r)) is updated. Once done, the PMO transitions to Ds (destroy) where now all unnecessary shadow pages are zeroed. Finally, PMO transitions Ds → D.

b) State transition for SP, SP/I_p and SP/I_d: For the PMO state transitions in the SP design, SP/I_p and SP/I_d designs, consider only the black/solid and the blue/dotted parts of Figure 3. For sake of brevity, we explain only those transitions that are different from the corresponding BP designs.

On a page fault, after decrypting and persisting a faulted primary page to associated shadow page in DP, the PMO transitions DP → W or DP → R. The decrypted shadow page is not copied and persisted back to the primary page (i.e., there is no DP → CPF transition). Recall that checksum verification is not applicable for the SP design. On `psync()` for PMO in Ⓦ, after persisting all dirty pages in the shadow (and updating their checksum for SP/I_p) in Ⓟ, the PMO transitions Ⓟ → EPP (encrypt and persist). In EP, all the touched shadow pages are encrypted and persisted into associated primary pages. This is required to ensure that the primary PMO pages always remain encrypted. Upon completion, the PMO transitions EPP → Ⓦ. On `detach()`, for SP and SPI_p, the PMO transitions Ⓡ → Ds or Ⓦ → Ds to zero all touched PMO pages. Note that in all SP designs, it is guaranteed by the last `psync()` that all touched shadow pages have been encrypted and persisted into associated primary pages. Therefore, all shadow pages can be safely zeroed. However, on `detach()` for SP/I $_d$, the PMO transitions Ⓡ → UC or Ⓦ → UC (update checksum) where all checksums are updated for all touched PMO pages before the PMO transitions to Ds state to zero the pages.

D. PMO System Layout with Integrity Verification

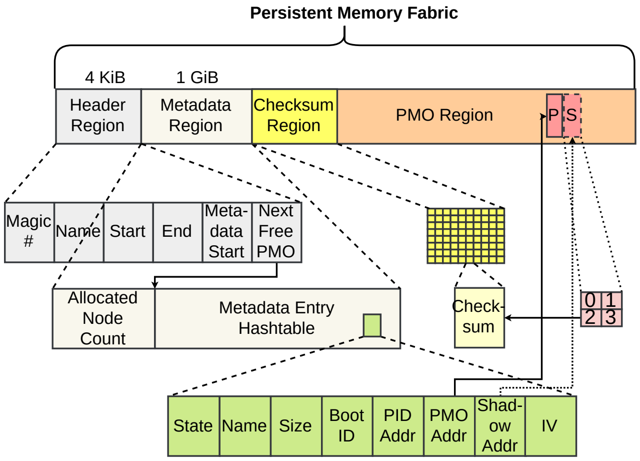


Fig. 4: Enhanced PMO System Layout for per-page checksums. Modified with permission from [13].

Regardless of whether we use I_p or I_d, it is necessary to store the checksums of each page somewhere. As depicted in Figure 4, we decided to add a new region, the *checksum region*, to the PMO system. Each page has an associated checksum that points to an entry in this region. Since each checksum is 32 bytes (we use SHA-256), this means that each 4096 byte page has an additional 32 byte overhead. We could have instead added the hash to the end (or beginning) of each page, but for simplicity and for performance reasons, we decided to store them in a separate region of the PM. This overhead can be reduced with the use of compression.

E. Performance analysis

Since detach and page faults are far simpler (simply drop the shadow, or decrypt the primary into the shadow), we expect both operations to be much faster with SP when compared to BP. On the other hand, encryption is not an entirely free operation, so encrypting the shadow into the primary at psync should take slightly longer. Section VIII evaluates the performance impact of each design.

F. Crash handler

In the prior PMO design [13], the PMO system only ensured that *crash consistency* was guaranteed in all circumstances. A PMO was only ensured to be encrypted if a PMO was detached successfully. This means that in prior work, a PMO at-rest was *not* guaranteed to be secure or to have its latest checksum (although it was always guaranteed to be crash consistent): a PMO, at-rest or in-use, was always crash consistent, but only secure when at-rest after a successful detach.

In contrast, we seek to extend the PMO security domain to include some types of abnormal process termination, such as if power is lost or a process dies. In this case, we propose a **crash handler**. The crash handler performs encryption routines that would have been performed by `detach()` if the process had terminated normally. For brevity, only the crash handler with SP will be explored.

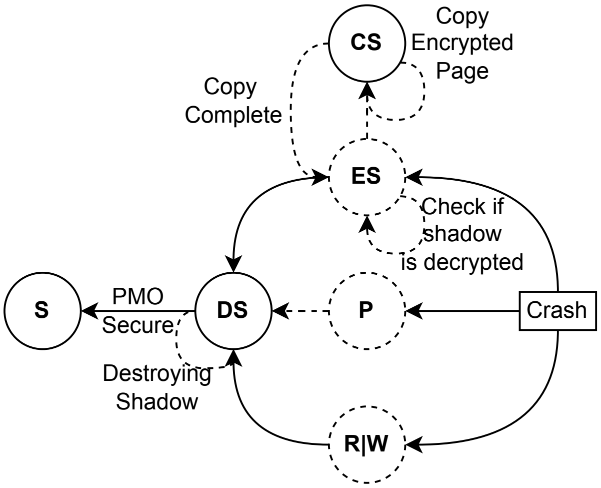


Fig. 5: State transitions for the crash handler

Figure 5 illustrates the crash handler's state transitions. Since the crash handler will run in the case of a process failure, it cannot know which pages are dirty and which are not. All the crash handler knows is the PMO state at the time of the crash (which is saved within the PMO metadata) and the data within each page. In the most common case, the PMO is in the read or write state (R/W) and is not being synchronized; in that case it is safe to simply drop all of the extant shadow pages (R/W → DS). Similarly if the PMO is persisting the shadow copy, those pages can simply be dropped without concern (Ⓟ → DS)

On the other hand, if the shadow is being encrypted into the primary (as a result of a `psync()`), then the crash handler will have to check whether a particular shadow page has data (and thus was faulted in). If the shadow page is not null, the crash handler will encrypt and copy the page from the shadow into the primary for every page within the PMO (ES → CS →

ES), which may mean overwriting primary pages that have already been copied over, and then drop the shadow (ES → DS). Since the threat model assumes that any crashes are not caused by hardware failures and does not consider attacks against a PMO in-use, it can be safely assumed that copying and encrypting the shadow pages will not preserve corruption. In either the case of I_p or I_d, the crash handler will update the checksums on these pages (i.e., the crash handler effectively performs a `detach()`).

A simple test demonstrates that the crash handler is a feasible way to ensure that data within a PMO are always secure: a 1GB PMO where every shadow page has data and has been decrypted takes approximately 2.2 seconds to perform a `detach()` that encrypts each page. Assuming a PMO system that is 120GB large, this would mean that it would take about 264 seconds to encrypt each page of the PMO system (a worst case scenario), or about 4 minutes. This is more than long enough to withstand a power fault when using an uninterruptible power supply (UPS). For example, a high-end UPS can last for more than 6 minutes at a full-load of 2700 Watts. [27].

Note that, although we do not implement it, it is also possible to further extend the crash handler to handle circumstances such as a kernel panic. For example, we could use a crash kernel that runs in the case where the kernel panics [11]. With this, as long as the system's hardware is functioning correctly, a PMO can be rendered secure.

VI. Implementation

A. Linux Crypto Subsystem

Performing encryption on individual pages, rather than on the entire PMO, allows the kernel to take advantage of multiple threads. The kernel encryption subsystem is multithreaded and asynchronous. Specifically, the Linux kernel crypto API's documentation [6] states that with regard to the Symmetric Key Cipher API, "Asynchronous cipher operations imply that the function invocation for a cipher request returns immediately before the completion of the operation. The cipher request is scheduled as a separate kernel thread and therefore load-balanced on the different CPUs via the process scheduler." When an encryption request is submitted to the subsystem, the subsystem assigns it to a kernel thread; that thread is then free to run on any available CPU core. This means that if multiple pages are waiting to be encrypted at `psync()` time and we are using SP, all the pages waiting can be encrypted and copied into the primary copy in parallel. Similarly, `detach()`'s performance with BP can be accelerated.

B. Crash Handler

To ensure that the stored checksum always matches a valid copy of the PMO data, we implement the crash handler described in the previous section. The crash handler is invoked whenever a process dies (either normally or abnormally), through `do_exit()` (which the Linux kernel *always* invokes whenever a process dies, regardless of cause).

C. Nonblocking Detach

In the original implementation described in [13], detach is blocking; i.e., the kernel does not return control back to the process until `detach()` completes. This design is inefficient, as the user process is stuck waiting on `detach()` even if it does not need it. Hence, the design here is nonblocking. When `detach()` occurs, a flag is set and control is immediately returned to the user process. However, any future `attach()` must wait until `detach()` completes.

VII. Evaluation Methodology

All of the designs described in Section V are evaluated. The baseline design is the Whole Encryption + Integrity Verification (WHOLEIVp) scheme, the same scheme originally described in the Greenspan PMO system [13]. The system used to evaluate these different designs is found in Table III. This enhanced PMO system is a modified version of the Greenspan PMO system described in [13], [21], which is itself a modified version of the Linux Kernel Version 5.14.18.

TABLE III: Configuration of the PM system used for evaluation

Component	Specifications
Motherboard	Dual socket Supermicro X11DPi-NT (w/ADR)
CPU	2×Intel Xeon Gold 6230, 20 cores, 40 threads
CPU Clock	2.1GHz (3.9GHz Boost)
CPU Cache	L1: 32KiB; L2: 1MiB; L3: 27.5MiB
DRAM	4 × 32GiB DDR4 @ 2666MHz
PMEM	3 × 128GiB Intel Optane DC (PMem)
OS and Kernel	64-bit AlmaLinux 9.0; Linux 5.14.18

To switch between different PMO designs, the kernel exposes a procfs file, `/proc/pmo` that can be read or written to. Writing to the file changes the PMO design scheme, reading from it produces the currently selected scheme. This allows for running all the different PMO designs in a script, without having to reboot to switch between schemes.

A. Evaluated Benchmarks

The following microbenchmarks are used for this evaluation: 2d Convolution (2dConv), Gaussian Elimination (Gauss), LU decomposition (LU) and Tiled Matrix Multiplication (TMM), all from [13] and originally obtained from [10]. These microbenchmarks were ported to use PMOs by designing and implementing a user-space free-list allocator and API. This allocator implements substitutes for standard dynamic memory functions, such as `malloc()` and `free()`, with its own versions, `p_malloc()` and `p_free()`, respectively.

The FileBench benchmarks [28] (FileServer, VarMail, WebProxy, and WebServer) are also used, adopted from [13] with no modifications.

1) Microbenchmarks: Each of the microbenchmarks invoke `psync()` after a specified number of iterations of the performance-critical loop, set to occur approximately every second when using the original, no encryption, case. This tempo (once per second) was chosen because Append on File Redis (AOF Redis)'s suggested default policy is to invoke fsync once per second [26], which is also used in other works utilizing Redis such as NVMove [3]. Additional works such as [12], [24] persist once per second and hence it is a common standard. The benchmarks use iterations instead of timers for consistency: using timers means that the amount of work between synchronization calls varies based on the PMO design, which negatively impacts the quality of the results.

Each of the microbenchmarks detach after a specified number of `psync()`s, from 1 – 16. For our overall evaluation of each PMO design, we start with 2 `psync()`s per detach. We do this because we anticipate that in the real-world, most attach/detach session will not fault every page within a PMO, but only a subset of them. Table VII-A1 shows the configurations of each benchmark.

TABLE IV: Microbenchmark Configuration

Benchmark	Configuration	PMO Size
2dConv	N=6144, M=104	32MB
Gauss	N=8192	256 MB
LU	N=7168	756 MB
TMM	N=4096, Tile Size=16	196 MB

2) FileBench benchmark configuration: Filebench [28] represents I/O intensive real-world applications and is designed for measuring I/O bandwidth performance. These benchmarks were ported to use PMOs by [13], and we adopt them here. It is important to note that synchronization points (`psync()` invocations) are invoked at every update and a pthread barrier is emitted before and after each `psync()` with the goal of avoiding data races. Each workload was run 5 times for 30 seconds, and the result is the average between the runs. Each workload has a different percentage of read/write operations. FileServer (FS) uses 67% writes, VarMail (VM) has 50%, WebProxy (WP) is 16%, and finally, WebServer (WS) is 9%.

VIII. Evaluation

This section attempts to answer several questions: How much more performant are the per-page designs compared to the baseline Whole Encryption design? Is BP or SP more performant, and in which workloads? How scalable are BP and SP as the attach/detach size increases, and how is thread scalability impacted? How much performance is gained with the designs supporting integrity verification compared to the original WEDI_p design? What is the performance impact of I_p compared to I_d?

A. Microbenchmark performance evaluations of per-page designs

Figure 6 compares the execution times of different PMO designs. Results are normalized to 8 threads. The results are split into different categories: **Psync Per+IV**, **Psync Enc/Cpy**, **Detach** , **PF Overhead**, **Attach Other**, **Attach Stall**, and **Compute**.

Psync Per+IV consists of the components of `psync()` that are not rendering the primary PMO crash consistent i.e., they include invalidating the cache lines to render the shadow copy valid and updating checksums when using the IVp designs. **Psync Enc/Cpy** includes updating the primary copy via encryption (Enc) or memcpy (Cpy) calls. **Detach** is the time it takes between when the detach system call is invoked and when it returns control back to the user process; since it is non-blocking, the time spent doing this is very small. **PF Overhead** is the additional time over the no-encryption case spent on servicing page faults (i.e., the time spent decrypting the page and rendering it available to the calling process). **Attach Other** is the time spent performing all attach operations that are independent of detach (i.e., if using the Whole design, the time spent decrypting the PMO), while **Attach Stall** is the time spent waiting for the detach thread to complete (since a detach call is immediately followed by an attach call, this is a proxy for the time it takes for the detach thread to complete). Finally, **Compute** is all other time: the time spent not waiting on the PMO system (this includes the base latency for servicing page faults, and the time spent performing computation).

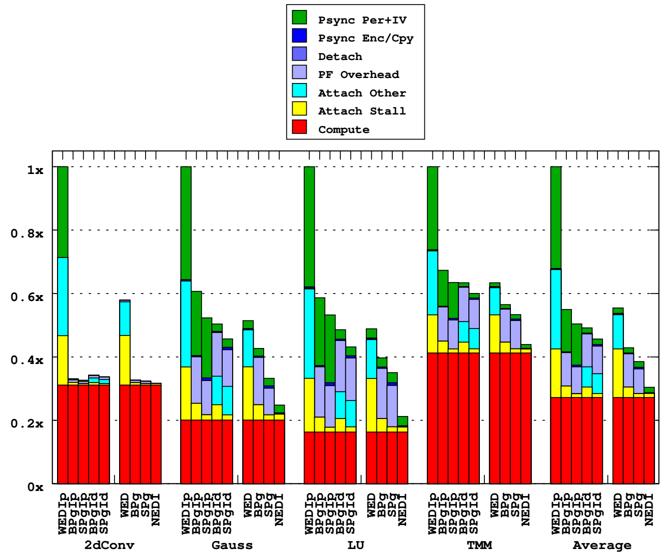


Fig. 6: Execution time by design, with attach session size of 2, normalized to WEDI_p

The performance of the individual categories matches expected performance; e.g., it makes sense that `psync()` calls will be more expensive with SP rather than BP (since SP requires encrypting the page into the primary rather than simply copying). Similarly, attach stall times should be more expensive since detaching a PMO using the BP method takes additional time (the primary must be encrypted into the shadow and then copied back, to ensure no loss of data).

With these four designs, the best performing IV design (SP/I_d) is, on average, 2.1× faster than the WEDI_p design. The second best design (BP/I_d) is 2.03× faster; a difference of 3%. For the non-IV designs, the performance of SP is 1.4× and BP is 1.3× faster than WED.

1) Sensitivity Study: Figure 7 shows `attach()` session sensitivity by growing the attach/detach size. The attach session size is the number of `psync()`s before a `detach()`. For example, we perform `psync()` once per second. If after each `psync()` we perform a `detach()`, then the attach session size is 1. On the other hand, if we only perform a `detach()` after 16 `psync()`s, then the attach session size

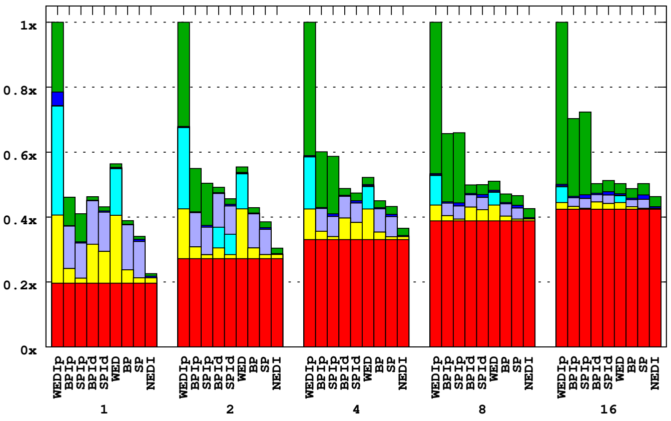


Fig. 7: Average execution time for all the microbenchmarks with different attach sizes, normalized to WEDI_p

is 16.

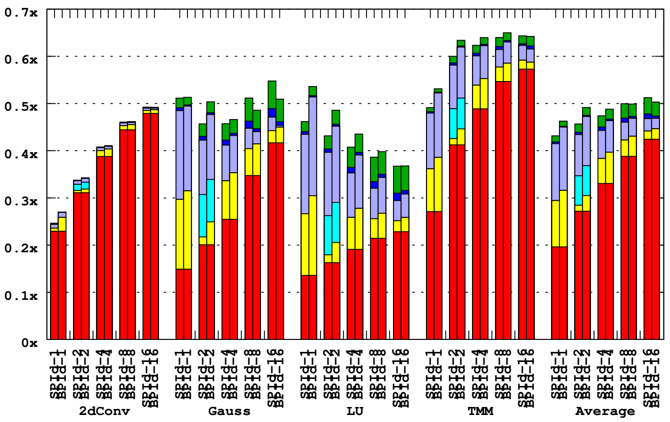


Fig. 8: Execution time of BP/I_d and SP/I_d with different attach session sizes, normalized to WEDI_p

Figure 8 focuses on the two most performant designs (BP/I_d and SP/I_d) as the attach session size increases. The results are normalized to SP/I_d at 1. At a size of 16, BP/I_d is slightly faster, 2.75%. What this demonstrates is that the BP/I_d designs are superior to SP/I_d when performing attach/detach infrequently, which is the expected behavior since `psync()` is more expensive with SP/I_d (encrypting the shadow into the primary vs. simply copying the shadow into the primary). While page faults are more expensive with BP/I_d, a large attach/detach size means that there are fewer pages being faulted in for the first time out of a total number of pages accessed. This is illustrated by Figure 9. Figure 9 shows the number of pages touched between an attach/detach session for each benchmark by attach/detach session size (dark colored bars, a/d) and shows that as the number of pages touched gets larger, the number of page faults declines relative to the number of `psync()`s (light colored bars, p). All of these facts combined means that BP tends to have better performance in these cases.

It is important to note that in an actual real-world application, the cost of psync may well prove to be much more important than the cost of detach, since detach is done off the critical path. In an application that attaches a PMO, performs multiple operations on it, detaches it, and does not use it for a long-time, BP is likely to be the more performant choice. Yet, as described in Section II, prior work by Xu, et. al. [33] propose reducing the amount of time a PMO is attached to a bare minimum; if this suggestion is followed, then SP is the better option.

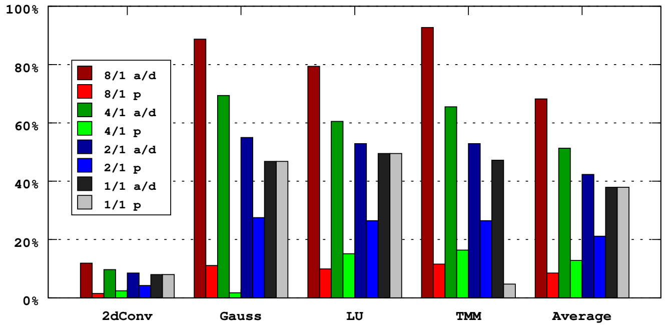


Fig. 9: Pages accessed between attach/detach calls and psyncs, by benchmark and attach/detach size. a/d is the average number of dirtied pages between attach/detach calls, while p is the average number of dirtied pages between psyncs.

B. Filebench

Figure 10 compares the I/O bandwidth of different Filebench workloads achieved by different PMO designs. Results are normalized to $WEDI_p$ and reported for 8 threads with synchronization performed after every write or append operation. On average, SP/I_d is 2.56× faster than WEDI_p, while SP alone is 3.2× faster.

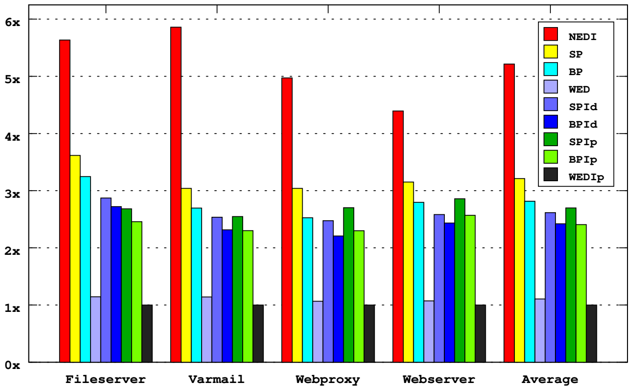
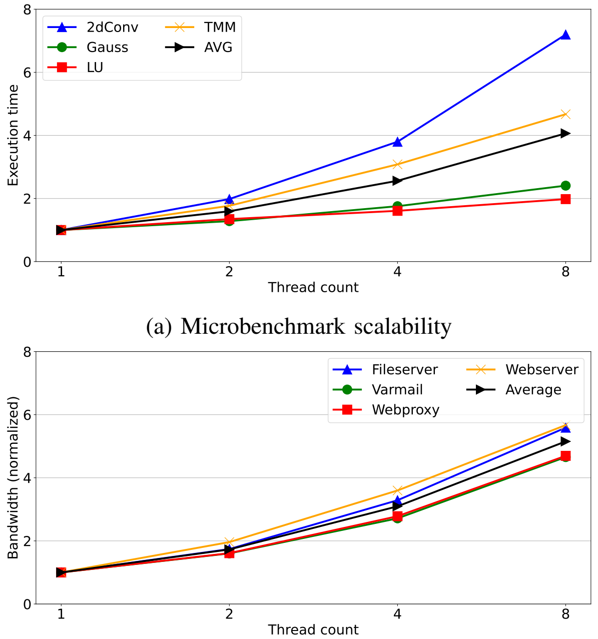


Fig. 10: Filebench results, normalized to NEDI_p.

Interestingly, with WebServer, I_p is slightly more performant than I_d because WebServer calls detach much more often than psync (9% writes, 91% reads). This allows us to conclude that in scenarios where `psync()` is infrequent or does little, I_d is worse than I_p, which is preferable in this scenario.

C. Thread scalability of LOaPP

Figure 11a shows the thread-scalability of the PMO system's performance when using the best-case design (SP/I_d). Results, normalized to a single thread, are shown for all of the



(a) Microbenchmark scalability

(b) Filebench scalability

Fig. 11: Thread scalability results with SP/I_d, from 1-8 threads.

microbenchmarks when executed with $N = (1, 2, 4, 8)$ threads and synchronized once per second, with an attach/detach size of 2. Results show that performance scales for an increasing number of threads, but Gauss and LU at 8 threads are only about 2× faster than at 1 thread. This behavior is expected and unrelated to PMOs; as it is an effect of the physical property of the PM fabric (Optane), as originally discovered by [16], [35]. Excessive numbers of writer threads slow the PM fabric down. To verify for ourselves that this is not related to our PMO system, we tested the microbenchmarks with a filesystem (ext4-dax) on top of the PM; and found that the thread scalability results remain the same. However, this behavior does *not* occur when testing the microbenchmarks with volatile memory.

Figure 11b shows the thread-scalability of Filebench with SP/I_d; these results are as expected; each thread is independent of each other, since Filebench is really testing sustained I/O bandwidth.

D. Correctness and Crash Consistency

To verify that the per-page encryption design is crash consistent and secure even in the face of crashes, two different tests are performed. First, `do_exit()` is inserted after a random number of PMO page faults are handled. This causes the associated user space process to terminate abnormally, and invokes the PMO crash handler. Examining the contents of the PMO after the crash handler has completed reveals that none of the faulted pages are visible, and that the shadow copy of the PMO is empty. Performing this same test within `psync()` after the persist stage but before the copy stage similarly reveals that the data within the shadow are gone, and the data from before the `psync()` are still in the primary copy, as expected.

IX. Conclusion

Security and performance are both important design considerations in persistent memory abstractions. To improve performance, this paper introduced per-page encryption to PMOs, and discussed the design and implementation of a PMO system with high performance, and high reliability (crash consistency) without reduction in security. Results show that, compared to the prior PMO abstraction, per-page encryption with integrity verification yields performance 2.19× and 2.62× faster without sacrificing security or reliability. For future work, we plan to investigate predictive decryption; where pages that have been used often in the past can be decrypted ahead of time off the critical path, before a page fault uses them.

Acknowledgements

This work is supported in part by the Office of Naval Research (ONR) under grant N00014-20-1-2750, and by the National Science Foundation (NSF) under grant 1900724.

References

[1] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell DE Long, and Ethan L Miller. Twizzler: a data-centric os for non-volatile memory. *ACM Transactions on Storage (TOS)*, 17(2):1–31, 2021.

[2] David Boles, Daniel Waddington, and David A Roberts. Cxl-enabled enhanced memory functions. *IEEE Micro*, 43(2):58–65, 2023.

[3] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. {NVMOVE}: Helping programmers move to {Byte-Based} persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, 2016.

[4] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[5] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software {IO} path with {Failure-Atomic}{Memory-Mapped} interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16, 2020.

[6] Kernel Development Community. Block Cipher Algorithm Definitions. https://www.kernel.org/doc/html/v5.14/crypto/api-skcipher.html#symmetric-key-cipher-api .

[7] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[8] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. Persistent memory research in the post-optane era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, pages 23–30, 2023.

[9] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–329. IEEE, 2017.

[10] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–329, 2017.

[11] David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Mini-ckpts: Surviving os failures in persistent memory. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–14, 2016.

[12] Adriano Marques Garcia, Dalvan Griebler, Claudio Schepke, and Luiz Gustavo Fernandes. Spbench: a framework for creating benchmarks of stream processing applications. *Computing*, 105(5):1077–1099, 2023.

[13] Derrick Greenspan, Naveed Ul Mustafa, Zoran Kolega, Mark Heinrich, and Yan Solihin. Improving the security and programmability of persistent memory objects. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 157–168, 2022.

[14] Rev. 010 Intel. 12th generation intel core processors, 2023.

[15] Revision 1.4 Intel. Intel architecture memory encryption technologies, 2022.

[16] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[17] 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*, pages 494–508, 2019.

[18] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[19] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, and Youngjae Kim. Mosiqs: Persistent memory object storage with metadata indexing and querying for scientific computing. *IEEE Access*, 9:85217–85231, 2021.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[21] Naveed Ul Mustafa and Yan Solihin. Persistent memory security threats to inter-process isolation. *IEEE Micro*, 2023.

[22] Naveed Ul Mustafa, Yuanchao Xu, Xipeng Shen, and Yan Solihin. Seeds of seed: New security challenges for persistent memory. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 83–88. IEEE, 2021.

[23] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.

[24] Liam Patterson, David Pigorovsky, Brian Dempsey, Nikita Lazarev, Aditya Shah, Clara Steinhoff, Ariana Bruno, Justin Hu, and Christina Delimitrou. Hivemind: a hardware-software system stack for serverless edge swarms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 800–816, 2022.

[25] P Roberts. Mit: Discarded hard drives yield private info. *ComputerWorld*, 16, 2003.

[26] S Sanfilippo and P Noordhuis. The redis documentation, 2016.

[27] CyberPower Systems. CyberPower PR3000 Sinewave UPS Specifications. https://www.cyberpowersystems.com/product/ups/smart-app-sinewave/pr3000/.

[28] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[29] 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 Symposium on Operating Systems Principles*, pages 478–496, 2017.

[30] Yuanchao Xu, Yan Solihin, and Xipeng Shen. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. New York, NY, USA, 2020. Association for Computing Machinery.

[31] Yuanchao Xu, Wei Xu, Kimberly Keeton, and David E Culler. Softpm: Software persistent memory. In *13th Non-Volatile Memories Workshop (NVMW)*, 2022.

[32] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. Temporal exposure reduction protection for persistent memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–924. IEEE, 2022.

[33] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. Temporal exposure reduction protection for persistent memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–924. IEEE, 2022.

[34] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692, 2020.

[35] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.