Low-Overhead Security Protection for at-Rest PMOs

Derrick Greenspan, Naveed Ul Mustafa, Andres Delgado, Connor Bramham, Christopher Prats, Samu Wallace, Mark Heinrich, Yan Solihin
University of Central Florida

I. Motivation

Please click here [1] for PDF of the original paper. Persistent Memory Obcjects (PMOs) manage persistent data by holding them in pointer-rich data structures without the backing of a file. A PMO is mapped/unmapped to/from the address space of a user process by invoking `attach()`/`detach()` system calls. 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()`. 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 Persistent Memory (PM) for a long time. Therefore, protecting at-rest PMOs is as important as protecting in-use PMOs.

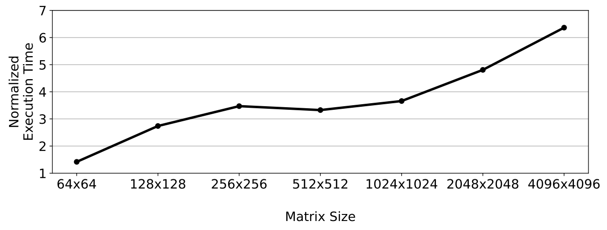

Fig. 1: Execution time of TMM workload.

To protect at-rest PMOs, previous work [2]encrypts every PMO on `detach()` and maintains a checksum that is updated on every `psync()`. Similarly, a PMO is decrypted and checksum comparison is performed on every `attach()`. This results in an increased latency of `attach()`/`detach()`/`psync()` proportional to the size of PMO. These latencies are on the critical path and exposed to the attaching process. Figure 1 shows that execution time of PMO-ported Tiled Matrix Multiplication (TMM) workload increases from 1.5× at 4096 bytes matrix to 6.4× at 16MB matrix size, when compared to no-protection case. The source of these problems, as illustrated in Figure 1 is that protections (security and integrity mechanisms) are enforced at the granularity of a *whole PMO* size regardless of the actual working set size of an `attach()`/`detach()` session.

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 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 [2], 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.

II. Threat Model

Our goal is to protect at-rest PMO data, ensuring confidentiality and integrity. Attackers aim to reveal or tamper with sensitive user-process data stored in at-rest PMOs, whose locations in memory they may identify. Similar to files, PMOs often remain at-rest, making them vulnerable to data remanence attacks, where plaintext data in persistent memory (PM) can be extracted from stolen or improperly discarded PM. We trust only the Linux Kernel Crypto API [4], critical kernel functions (e.g., memcpy, memset), and our PMO kernel subsystem. These components are assumed secure, as their small codebases are amenable to formal verification [3].

III. Low Overhead at-rest PMO Protection Design

Our design space exploration is guided by finding answers to the following challenges. **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 I.

The Whole PMO Encryption and Decryption (WED) design of [2]encrypts/decrypts whole PMO (C1), performs decryption

This work was supported in part by the U.S. Office of Naval Research through Grant N00014-23-1-2136 and Grant N00014-20-1-2750, and by the National Science Foundation through Grant 1900724 and Grant 2106629.

Naveed Ul Mustafa is currently associated with New Mexico State University and was at UCF when this work was performed.

TABLE I: 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

at `attach()` (C2.D) and encryption at `detach()` (C2.E). When providing Integrity (I) protection (WED/I_p), it verifies PMO-level checksum at `attach()` (C2.V) and updates it at `psync` (C2.U). 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).

A. Encrypting/Decrypting Both PMO Pages (BP)

The BP design aims at reducing the latency of `attach()`, `detach()`, and `psync()` by performing encryption/decryption at granularity of pages (both primary and shadow ones) (C1). After successfully attaching a PMO, a page is decrypted only on Page Fault (PF) and encrypted on `detach()` (i.e. C2.D and C2.E). BP/I_p additionally verifies the page-level checksum at PF (C2.V) and updates it at `psync()` (C2.U).

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. 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 full paper [1].

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

To further reduce latency of `attach()` and `detach()`, SP design encrypts and decrypts only shadow pages (C1). After successfully attaching a PMO, on a page fault only a shadow page is decrypted (C2.D). The 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*. When providing integrity protection (SP/I_d), the page-level checksum is verified at page-fault (C2.V) and updated on `psync()` (C2.U). 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).

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).

IV. Evaluation

Figure 2 compares the I/O bandwidth of different Filebench workloads [5]achieved by different PMO designs. Results are normalized to No Encryption Decryption and Integrity (NEDI) protection and reported for 8 threads with synchronization performed after every write or append operation. On average, SPI_p is $2.56\times$ faster than WEDI$_p$, while SP alone is $3.2\times$ faster.

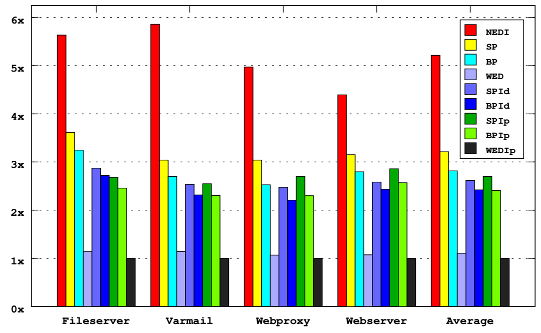

Fig. 2: Filebench [5]results, normalized to NEDI.

References

[1] D. Greenspan, N. U. Mustafa, A. Delgado, C. Bramham, C. Prats, S. Wallace, M. Heinrich, and Y. Solihin, "LOaPP: Improving the Performance of Persistent Memory Objects via Low-Overhead at-Rest PMO Protection," in *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*, IEEE, 2024, pp. 131–142.

[2] D. Greenspan, N. U. Mustafa, Z. Kolega, M. Heinrich, and Y. Solihin, "Improving the Security and Programmability of Persistent Memory Objects," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, IEEE, 2022, pp. 157–168.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 207–220.

[4] Kernel Development Community, "Block Cipher Algorithm Definitions," Available: https://www.kernel.org/doc/html/v5.14/crypto/api-skcipher.html#symmetric-key-cipher-api, Accessed: Mar. 1, 2023.

[5] V. Tarasov, "Filebench: A Flexible Framework for File System Benchmarking," *The USENIX Magazine*, vol. 41, p. 6, 2016.