PMO Checker: A Framework for Protecting Persistent Memory Objects Against Security Attacks

NAVEED UL MUSTAFA, University of Central Florida, USA and New Mexico State University, USA
YAN SOLIHIN, University of Central Florida, USA
XIPENG SHEN, North Carolina State University, USA

As a new abstraction to manage persistent data in non-volatile memory, Persistent Memory Objects (PMOs) hold (pointer-rich) data structures that can be shared among processes over many runs and system boots. While convenient and fast, such sharing opens up a new class of attacks that attempt to break inter-process isolation, allowing the attacker to compromise a non-vulnerable process through a different process and a shared PMO. Due to the difficulty in completely preventing or avoiding security attacks, we present PMO Checker, a comprehensive framework for detecting and recovering from them. Our framework includes various invariant checking and checkpointing schemes. We implement them on a real system with Linux kernel on a real platform with Intel Optane PMem memory. Our evaluation shows the effectiveness of the attack detection and various performance optimizations that keep the overheads low.

CCS Concepts: • **Security and privacy** → *Systems security*; *Software and application security*.

Additional Key Words and Phrases: Persistent Memory Objects, Invariant Checking, Attack Detection, Attack Recovery

1 INTRODUCTION

Non-Volatile Memory (NVM) emerged as an attractive main memory in modern computer systems. While Intel Optane PMem was recently discontinued, others, such as memory-semantic SSD (e.g. Kioxia FL6 [27], Samsung's CXL-based SSD [36]), battery-backed DRAM [33, 42], as well as ReRAM and MRAM-based memories, provide alternatives. NVM provides a substrate that can be used to host a filesystem or memory-mapped files [8, 25, 43], or as a repository for hosting memory objects [5, 26, 29, 44, 45]. For the latter, the Operating System (OS) provides an abstraction of *persistent memory object* (PMO) which can wrap pointer-rich data structures and manage its namespace, permission, and sharing semantics. A user process invokes the `attach()\detach()` system calls to map\unmap a PMO to\from its address space [44]. Once attached, PMO data is directly accessed through loads and stores. The PMO Abstraction, similar to Intel PMem pool, stores persistent data in non-volatile memory, but without any file-backing. PMOs' basic protection and sharing mechanisms are similar to those of file systems: only processes of a valid user can attach a PMO and write is exclusive (a process can attach a PMO for write *only* when it is not attached to any other process) but read is not (multiple processes can attach a PMO to read). On top of the basic permission, a key-based permission can be used, where a process must produce the right key to attach a PMO.

Authors' addresses: Naveed Ul Mustafa, num@nmsu.edu, University of Central Florida, Orlando, Florida, USA and New Mexico State University, Las Cruces, New Mexico, USA; Yan Solihin, yan.solihin@ucf.edu, University of Central Florida, Orlando, Florida, USA; Xipeng Shen, xshen5@ncsu.edu, North Carolina State University, Raleigh, North Carolina, USA.

Manuscript submitted to ACM

Table 1. YCSB Workload A evaluation: single DB and single client (Left), and 4 DBs with 2 clients per DB (Right).

System	Pointer Type	Object Sharing	Inter-Object Pointers	PP2VM	Object Validation
Twizzler [5]	Relative	Allowed	Allowed	Possible	No
Mosiqs [26]	Relative	Allowed	Allowed	Possible	No
Puddles [29]	Native	Allowed	Allowed	Possible	No
GPMO [19]	Native	Allowed	Allowed	Possible	No

Current PMO designs [5, 19, 26, 29, 44] envision a PMO as a repository for holding pointer-rich data structures persistently. In addition, similar to a file, a PMO can be accessed by multiple processes over time, although generally only one process is allowed to update it at a time. These two characteristics together *greatly amplify* the security concerns of how one process may affect another process that shares common PMOs. Compared to a volatile memory region shared among processes, PMO-hosted data structures may outlive the sharing processes. Hence, an attack may proceed fast or slowly over time, over multiple attach/detach sessions or even across boots, due to PMO persistency. While the same inter-process sharing concerns also apply to files, in contrast to files, PMOs host pointer-rich data structures, which are high-value targets for the attacker to corrupt. Also, unlike files, attack transmission via PMOs is hard to detect/prevent as PMO accesses via loads/stores are invisible to the OS.

To make things worse, several common features in PMO systems increase the security vulnerabilities (Table 1). First, inter-PMO pointers are allowed, as they may be useful for flexibility and concurrency. They allow a data structure to grow beyond its originally allocated object size over into a new object. By splitting a large data structure into multiple PMOs with inter-PMO pointers connecting them, concurrency improves because multiple processes can attach different parts of the data structure by attaching a subset of the constituent PMOs. Second, they allow pointers from the PMO to volatile memory (PP2VM). While the use of such pointers are risky and not encouraged, they are not disallowed. One may imagine a potential use for performance optimization (e.g., a cache of indices maintained by a database application).

These common features pose potential security threats to sharing processes, e.g., an in-PMO pointer may be maliciously overwritten to point to a volatile memory space or even to an address in another PMO. If one process changes the value of a pointer and detaches the PMO, a process that attaches the PMO later may dereference the corrupted pointer, leading to control flow change or further (more serious) memory corruptions. The former allows the attack to proceed to overwrite data that is used for control flow decisions. The latter allows one process to affect another process who later uses the pointed-to PMO. To make it worse, it was discovered that even when a malicious process does not share the same PMO with another process, the former can still attack the latter by leveraging intermediate processes to transmit the attack along the chain of PMO accesses among them [31]. Despite these vulnerabilities, current PMO systems lack *mechanisms* to validate PMO updates.

The pointer-rich nature of PMOs and the potential for stitching together attack steps over a long period of time (detailed in Section 3), makes complete avoidance of attacks difficult on PMO sharing processes. Earlier works in PMO security protection have focused on *prevention* and *avoidance* of an attack [44, 45]. However, due to the broad attack surface and various sharing scenarios [31], in this paper we focus on providing an ability to detect anomalies that may indicate ongoing attacks, and to recover from them. To this end, we present PMO Checker, a framework for *detecting* and *recovering* from security attacks for the PMO abstraction. For detection, the framework relies on invariant checking to detect a potentially compromised PMO before letting it be attached to a user process. Since the security protection depends on the accuracy and coverage provided by invariants, we design PMO Checker to have the

flexibility to add/remove invariants, and low performance overheads that make the flexibility practical. For recovery, an already-checked PMO is checkpointed, enabling its restoration upon future detection of an invariant violation.

This paper makes the following contributions. First, we propose PMO Checker, the *first known framework* for detecting and recovering from attacks targeting PMO. The framework requires little input from a user program (i.e., PMO group ID, data structure type, and allow list as explained in Section 5). Beyond that, it is *transparent* to the user program as its scheduling and tracking is performed by a runtime system integrated into the kernel space. Although the current design of the PMO Checker focuses on not all but only some most important invariants detection, we note that it is extendable, allowing addition and deletion of invariants for other attacks. Second, we integrate *PMO invariants* into PMO Checker that are effective in detecting a broad class of known security attacks, and analyze their coverage. Third, achieving a high detection coverage with low overhead is challenging, because of: (a) high invariant checking and checkpointing latencies, (b) high frequency of invariant checking demand, and (c) high synchronization costs among multiple threads. To address them, our PMO Checker framework includes various *optimizations* to reduce performance overheads. They target the checking latency (invariant merging, pipelining, and criticality prioritization), checking throughput (multithreading, prediction), checkpointing latency (page-level checkpointing), and synchronization (thread localization). Finally, we integrated PMO Checker into Linux OS running on a *real system* with Optane Pmem memory platform. We evaluated both PMO Checker performance and security. For *performance*, our optimizations are greatly effective, e.g. for multi-client YCSB, the execution time overheads of PMO checker are reduced by 98.5%, resulting in an execution that is only 12% slower than an insecure PMO baseline). For *security*, we show that PMO Checker provides 100% detection coverage for all example attacks except for one case with 65% coverage.

2 RELATED WORK

2.1 Protecting PMOs

Memory integrity protection in secure enclaves can protect PMO data from unauthorized memory modifications caused by external events, such as RowHammer, or illegitimate parties. However, it is not designed to detect corruption by a process that is legitimately authorized to access a PMO. Because both PMOs and files rely on user-based permissions, all processes belonging to a valid user may access them. Therefore, memory integrity protection does not address the vulnerabilities targeted in this paper.

Prior work on PMO protection [44–46] has primarily focused on preventing or avoiding attacks. Table 2 summarizes these schemes in terms of protection model, attack coverage, hardware requirements, and reported performance overhead. For example, MERR [44] considers a threat model in which a victim process contains known memory vulnerabilities (e.g., buffer overflows), and an attacker exploits these vulnerabilities to inject or exfiltrate PMO data while the PMO is attached. MERR mitigates such *intra-process* PMO attacks by attaching a PMO only when it is needed, thereby minimizing its exposure window. To support frequent attach and detach operations, MERR reduces their overhead by embedding the PMO's page-table subtree within the PMO itself. However, this design increases the PMO's memory footprint and requires architectural support—specifically, a hardware permission matrix per core—to enforce process-specific PMO access permissions.

TERP [45] addresses the same threat model and similarly targets intra-process PMO attacks. TERP also focuses on reducing PMO exposure window and its main contribution is compiler and architectural support that enables automatic insertion of `attach` and `detach` primitives in both single-threaded and multi-threaded applications, without burdening the programmer.

Table 2. Summary of prior PMO protection schemes

Scheme	Protection	Coverage	Custom HW?	Avg. Overhead (%)
MERR [44]	Prevention	Intra-process	Yes	10.9
TERP [45]	Prevention	Intra-process	Yes	6
PDV [46]	Prevention	Cross-thread	Yes	23.97
PMO Checker	Detection & Recovery	Inter-Process	No	12

PMO Domain Virtualization (PDV) [46] considers a different threat model, focusing on *cross-thread* PMO attacks within a multi-threaded process. In this scenario, an adversary exploits memory vulnerabilities in one thread to gain unauthorized access to a PMO attached by another thread of the same process. PDV mitigates such attacks by introducing protection domains: when attached, a PMO is placed into a domain, and each access request is validated against the domain's access policy for the requesting thread. This approach requires architectural extensions to Intel Memory Protection Keys (MPK) in order to restrict PMO access at thread granularity.

In contrast to these prevention-oriented approaches, our work focuses on detecting and recovering from ongoing PMO attacks. Recent studies demonstrated security threats from a scenario where multiple processes share a PMO either simultaneously or *successively* over time, where the attacker may use vulnerabilities in one sharer to gain control of another [32]. Attacks can also happen when processes do not directly share a PMO [31]. Those studies motivated our effort in focusing on detection and recovery from attacks, rather than just prevention and avoidance.

PMO Checker is the first system to protect against inter-process PMO attacks, making it orthogonal to existing PMO protection schemes. As summarized in Table 2, prior PMO protection mechanisms rely on customized hardware support and/or compiler modifications, whereas PMO Checker is implemented entirely as a kernel module and does not require custom architectural support. Because prior schemes require architectural support and compiler changes that are unavailable in commodity systems, we rely on reported results for qualitative comparison rather than re-implementation.

2.2 Invariant Checkers

Invariant of an entity (e.g., a data structure or a program-code) is a property that must hold for its lifetime [4]. Invariants may specify data values (e.g. "variable x is non-zero" [3]), relational (e.g. "elements of a sorted list are ordered" [38]), or topological (e.g. "a tree does not contain cycles"). Invariants are known to be useful for a variety of purposes, such as security (e.g. detecting rootkit [4]), debugging [37], localizing faults [3], among others. Our PMO Checker is a framework to integrate various invariant checks for detection of attacks on PMOs with low performance overheads.

2.3 Checkpointing

It is a technique that saves a process' state and rolls back the process, in case of failure, to its most recent checkpoint to resumes its execution [28]. PMO Checker checkpoints already invariant-checked PMOs and then uses them for recovery from detected attacks in future.

3 BACKGROUND

3.1 GPMO System Overview

Our approach for invariant checking and checkpointing applies to any PMO systems in general [5, 26, 29], but for ease of discussion, we discuss it in the context of Greenspan et al. [19]'s PMO (or *GPMO* for short) system. GPMO provides a set of system calls for applications to use PMOs, as listed in Table 3. `pcreate()` and `pdestroy()` creates a

Table 3. PMO system calls, source: [19].

Primitive	Description
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.
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.

new PMO and deletes an existing one, respectively. A process must first attach a PMO to access data structure kept in the PMO. A process may attach multiple PMOs simultaneously. Multiple processes can attach the same PMO with read permission, but only one process can attach a PMO at a time with write permission, for consistency. Both attach and detach are non-blocking. A failed attach returns an error; the process can re-attempt to attach in the future. GPMO provides `psync` for managing crash consistency by preventing PMO modifications from persisting until a psync call, at which point all prior modifications are atomically saved. psync is blocking, so any stores following it do not execute until it is completed. In the event of a crash, GPMO ensures the PMO state matches that of the last successful psync. It implements psync through shadowing, where updates occur on a shadow copy until they are atomically saved to the primary copy during psync.

3.2 Inter-Process PMO Attacks

Prior works detailed inter-PMO [31] and inter-process [32] attacks that are possible as a result of multiple processes sharing PMOs over time. This work focuses on inter-process attacks which assume a threat model with two process: Payload (P) and Victim (V). P and V belong to the same user hence are legally allowed to attach PMOs that have access permitted for the user. P is assumed to have some known security vulnerabilities (e.g., buffer overflow) while V has none. The goal of attacker is to compromise V. To that end, an attacker exploits the security vulnerabilities of P to modify the shared PMO.

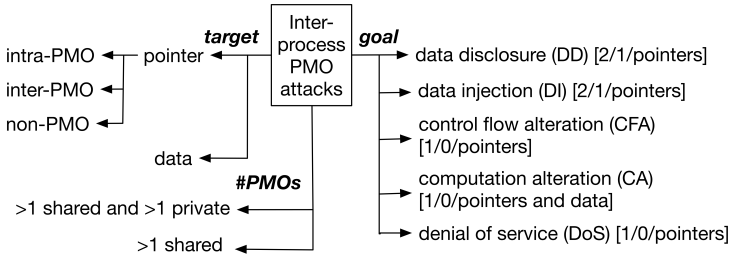


Fig. 1. Attack classification. Attacks need `X` shared and `Y` private PMOs and target pointer or data, shown as [X/Y/Pointers or Data].

3.3 Attack Classification

Figure 1 shows taxonomy of inter-process attacks. It illustrate various ways corrupted PMOs can be leveraged to launch such attacks (and not an exhaustive attack list). Based on their goal, attacks are categorized into breach of confidentiality (data disclosure or DD), data integrity (data injection or DI), control flow integrity (control flow alteration or CFA), computation integrity (computation alteration or CA), and denial of service (DoS). Note that while PMO Checker does not aim at providing full protection against DoS, we show attacks that can cause DoS and detection effective for a

subset of DoS attacks. DD, DI, and DoS are variants of CFA and CA (with different attack goals) identified in [32]. Inter-process attacks can also be classified based on the PMO overwrite target, i.e., PMO data or pointer. A pointer may be overwritten to point to an address within the same PMO containing the pointer (intra-PMO), in a different PMO (inter-PMO) or, outside PMO (non-PMO). Note that inter-PMO attacks [31] rely on creating inter-PMO pointers. Finally, based on the types of PMOs involved, we have two cases: the attacks that only involve one or more PMO(s) shared by the P and victim V processes, and attacks that involve additional one or more PMOs private to V.

3.4 Attack Exposition

The simplest inter-process attack involves one PMO shared among P and V over time, i.e., P attaches, modifies and detaches a PMO which is then subsequently attached and accessed by V. Such PMO sharing possibly even occur over multiple system boots. When attached, P may overwrite the PMO to accomplish different attack types. Control Flow Alteration (CFA), described in [32], is accomplished by targeting pointers with non-PMO targets. For DoS, P can overwrite a PMO's pointers. For example, overwriting a node's pointer in a linked-list to point to NULL makes a section of the list unreachable (and depending on the runtime system, may result in memory leaks). As another example, overwriting a node's pointer to point to the same node creates a cycle such that a traversal of an acyclic data structure never ends. For Computation Alteration (CA), P can overwrite a PMO's pointers (e.g., in a tree, redirecting a node's child pointer to its grand-child) or data to cause V unexpected execution outcome. Note that all these attacks require P to detach the PMO for them to transmit to V.

Figure 2 illustrates the DD attack using a PMO-ported SQLite database example with three PMOs, i.e., X, Y, and Z holding `Free`, `Most_active`, and `Premium` member tables (represented by B+ trees) of an online service company, respectively. P (e.g., an employee in the finance department) and V (e.g., head of the finance department) share PMO X and Y while Z is private to V. Note that GPMO manages a PMO's memory like a heap with deallocated or pre-allocated chunks placed in a circular linked-list. Head and Tail fields in Figure 2 point to head and tail nodes of the linked list (not shown due to limited space), while Data Structure Root (DSR) points to the root of the main data structure, i.e, B+ tree in this example.

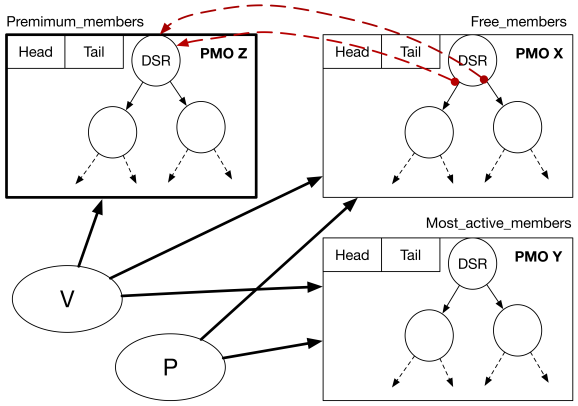


Fig. 2. Setup for Data Disclosure attack: PMO X and Y shared between *payload* and *victim* over time.

The DD attack shows that P can read PMO Z even though it does not have access permission to it and never attaches it. Consider that P and V independently execute a query to extract records from `Free_members` table (i.e., tree nodes)

with more than eight-hour active status a day and insert them into `Most_active_members` table. To launch the DD attack, P attaches X, corrupts its DSR by redirecting the left and right child pointers to the DSR of Z (shown by the red dotted arrows), then persists the PMO and detaches it. Under GPMO, assuming that P has means to discover the physical address of Z, it can easily find address of Z's DSR, i.e., $VA(DSR) = SPVAS + PA(Z) + Size(Head) + Size(Tail)$, where $VA(DSR)$ is virtual address of DSR, $SPVAS$ is Start of Persistent Virtual Address Space, and $PA(Z)$ is Physical Address of Z. Note that it is sufficient for the attacker to redirect pointers from X to any valid node in Z's data structure, not necessarily the DSR. Therefore, even when address space layout randomization (ASLR) is enabled and applied to the base addresses of PMOs, it does not sufficiently protect the security, if old and new address ranges of PMO mapping overlap. The larger the overlap, the higher the probability that addresses discovered by the attacker remain valid across address randomization. But even when there is no overlap, an attacker may apply existing techniques [6, 39] to break ASLR. Also, note that P must detach the PMO X. Otherwise, V cannot attach the PMO and attack is not transmitted. Let us assume that V subsequently attaches Z (e.g., for its private computation), X, and Y, and then executes the query. Since the DSR of X is corrupted, pointing to Z (i.e. `Premium_members` table), the query execution results in records of premium members inserted into Y, causing private data in Z to leak into Y. When P attaches the Y again, the attacker can access data that V has accidentally leaked from Z to Y, even though it never attached Z. Furthermore, the attack may leave no trace if the attacker removes the leaked data before persisting Y. The Data Injection (DI) attack can be carried out in a similar fashion, with some modifications.

4 THREAT AND TRUST MODELS

We consider a threat model that is uniquely introduced by the PMO model, i.e., inter-process sharing of PMOs. We do not consider threats that a process faces from its own vulnerabilities, as such threats are not unique to PMO. We denote the *victim* process V and payload process P sharing a PMO over time, both are legally allowed to attach the PMO. V has no exploitable security vulnerabilities, while P does. The attacker's objective is to eventually compromise V by exploiting vulnerabilities of P to corrupt the shared PMO as an intermediate step on the way to compromise V.

While there are many existing defenses to protect user-level process P from vulnerabilities e.g., StackGuard (using Canaries) [11], Data Execution Prevention (DEP)[2], and Address Space Layout Randomization (ASLR) [41], it is hard to catch all user-level vulnerabilities for two prime reasons: Current programming practices do not guarantee that user-level programs are free from vulnerabilities [34] and defense mechanism can be bypassed by attackers [1, 40]. For example, a recent work [7] shows step by step procedure for bypassing Canaries, DEP, and ASLR defenses. Besides, even if the large majority of processes are well protected, our threat model only requires that the attacker find one process that has exploitable vulnerabilities.

We consider the scenario when a PMO is first attached, modified, and then detached by P. Subsequently, the same PMO is attached and accessed by V. The adversary's objective is to exploit security vulnerabilities in P (e.g. buffer overflow, integer overflow, format string, etc.) to cause overwrites of the shared PMO's data or pointers, which V later accesses, eventually affecting V's execution. We assume that the adversary knows that the PMO is shared and knows the data structure layout in the PMO. The latter is a reasonable assumption as all sharing processes (including payload process with legit PMO access) need to know data structure layout to correctly perform operations (e.g., insertion, deletion, and searching etc). The latter is reasonable to assume as some persistency programming models require the programmer to distinguish pointer access from regular data access (e.g. Intel PMDK). We also assume that the adversary knows or has means to find out the target address in V that he/she wants to overwrite (e.g. function pointers, return

addresses, etc.). Denial of Service (DoS) attacks are not in scope of this work (though some may be detected by our invariants) and we trust system software, such as OS kernel.

5 DEFENSES FOR INTER-PROCESS PMO ATTACKS

5.1 Correctness Criteria for Defense Approach

Before presenting our defense (detection and recovery) methods, we first analyze timing criteria with regard to when invariant checking should be started and completed. Our trust model excludes user processes hence the OS is responsible to *launch* invariant checking. Since the OS cannot observe loads/stores and can only observe system calls (attach, psync, and detach), only at system calls the OS can launch invariant checking and checkpointing. psync is a point where updates are persisted, hence it is a possible launch point. However, the programmer may use multiple psyncs to persist incremental updates to the PMO data structure, hence using psync as a launching point may lead to excessive and redundant checkings. Therefore, we choose detach call as the earliest invariant checking launching point, and ensure that both invariant checking and checkpointing must be completed prior to the next attach. This defines the valid checking window.

PMO Checker framework's approach of relying on invariant checking to detect security attacks means that the detection coverage is only as good as invariants being checked. Therefore, it is important for the framework to be extensible, allowing invariants to be added or removed in the future as attacks may evolve. In addition, it is important that performance optimizations are as invariant-agnostic as possible.

5.2 Invariants for Multi-PMO Data Structures:

A pointer from one PMO to another PMO is referred as an inter-PMO pointer. As discussed in Section 3.4, an inter-PMO pointer may be leveraged by the attacker to launch several attacks, including Data Disclosure (DD) and Data Injection (DI) attacks (Figure 2). Thus, a naive solution may be to ban inter-PMO pointers to stop such attacks. However, this makes the use of PMOs too restrictive. For example, the programmer may find inter-PMO pointers useful for building large data structures that span multiple PMOs. Not only this may be useful for dynamically extending a PMO beyond its original size, splitting a large data structure into multiple PMOs may increase concurrency, reduce system call (attach/detach/psync) latencies, and enables more efficient memory management (e.g. some little-used PMOs may be swapped out to storage). Therefore, a smarter invariant is needed.

We extend GPMO to allow the user to specify a *PMO group*, consisting of inter-related PMOs that logically form a single (but potentially complex) data structure. Since the PMOs in a PMO group are related, we will allow inter-PMO pointers within a group, *but not across groups*. Sharing and permission settings are uniformly applied to a PMO group. For example, if two processes want to share a PMO in a group, our sharing semantic requires that all other PMOs in the group to be shared as well. In addition, a user process has the same level of permissions for all PMOs in a group. However, a process is not required to attach all PMOs in a group at once. It may dynamically attach only the needed PMO(s) and detach others. This allows better concurrency as other processes can modify other PMOs, avoids unnecessarily blowing up the address space of a process, and simplifies memory management. The OS maintains PMO group metadata (i.e., group ID, member PMOs, and permission).

Permitting inter-PMO pointers *only* within a group not only allows building multi-PMO data structures but also avoids attacks that rely on inter-PMO pointers. For example, the B+ trees of PMO X and PMO Z are unrelated (Figure

2) and hence are assigned to different PMO groups. As a result, checking for inter-group pointers from PMO X to PMO Z can detect DD attack.

5.3 PMO Pointers to Volatile Memory (PP2VM):

An interesting question is whether a PMO pointer should be allowed to point to volatile memory or not. Such (PP2VM) pointers are typically discouraged, because a PMO is persistent and may survive a process, leading to dangling persistent pointers when the process terminates. Furthermore, such pointers may enable security attacks, such as Control Flow Alteration (CFA) attack [32]. As a step in CFA attack, the payload may attach a shared PMO and modify pointers in the PMO free list to point to volatile function pointer (`fp`) and a target (injected or out-of-context) code block (i.e., `M`).

Therefore, we can consider using an invariant that bans PP2VM entirely. However, we also cannot rule out that a programmer may create a cache of PP2VM pointers as a performance optimization. Such an optimization is common in some database applications, where pointers to a large index tree are cached. If such applications are ported to use PMOs, barring PP2VM use may require significant code rewriting. Clearly, there is a flexibility-security trade off. To allow for caching of PP2VM pointers, but without sacrificing security, we extend GPMO such that PP2VM pointers are allowed, but only when the programmer provides an *allow-list*, i.e. a list of virtual volatile address ranges that can be pointed to by a persistent PMO pointer. We adopted the array-of-struct format for the allow-list from [30]. In our case, a user process can supply this list as an additional `attach()` call parameter. Each array element specifies the starting volatile address and size of allowed range of addresses for the attaching process. The allow-list is then maintained as a PMO kernel-level metadata and hence cannot be tampered by a user process. Furthermore, the programmer is responsible to ensure that the pointed volatile data is valid across multiple attach sessions, i.e. the pointed volatile data is not remapped or freed. For enhanced security, however, if a PMO is detached by a process that caches PP2VM pointers and then attached by a different process, we nullify those pointers and erase the old allow-list, preventing their use in transmitting an attack to another process.

Allowing PP2VM with an allow-list with resetting mechanism offers two benefits: 1) it allows software caching of PP2VM pointers for performance optimizations across multiple consecutive attach sessions by the *same process* and 2) it thwarts PP2VM pointers from being used to transmit security attacks to a different process.

5.4 Invariant Set Selection:

For PMO Checker to be effective against currently-known attacks, we need to identify invariants that are suitable for a PMO. Some aspects need to be considered for invariant inclusion: *ease of derivation*, *false positive rate*, *coverage*, and *universality*. Ease of derivation concerns with whether an invariant requires programmer's input or it can be derived automatically or semi-automatically, e.g. via compilation and profiling. Automatic or semi-automatic are preferred. Invariants should provide zero false positive rates, because detecting phantom attacks may cause unnecessary application crashes. Third, invariant checking is expensive as it often requires data structure traversal, hence selecting high-coverage invariants may permit fewer invariants to be checked. Finally, invariants that apply universally to all PMOs are preferred than those which are data structure-specific. However, because data structure-specific invariants tend to provide better coverage, we should support them. This requires metadata about data structure to be passed to PMO Checker. Fortunately, this is already supported in some systems, e.g. Intel PMOP already requires metadata that distinguishes pointers from regular data to be maintained [24]. Hence, we assume a modified `pcreate()` that specifies the type of data structures a PMO holds i.e., a template used by each node in the data structure, specifying node size, location of pointers and data. The template can be specified only at PMO creation and remains unchanged during a

PMO's lifetime. In the current version, we require all nodes to be homogeneous in their size and format, but this can be relaxed in future versions. `pcreate()` is also modified to let users specify the PMO group ID. Since the PMO Checker is a kernel module, it can access the information specified through `pcreate()` system call, map a PMO into kernel address space and then access its content.

With the above considerations in mind, we consider the following PMO invariants; Invariant 1-3 are new, while invariant 4-6 are adopted from literature.

① **PP2VM**. As discussed earlier, this invariant keeps track of the last detaching process for each PMO. If the next attach of the PMO is by the same process, PMO Pointer to Volatile Memory (PP2VM) are permitted, as long as the pointed-to addresses are found in the allow-list. However, when the attach is requested by a different process, the PP2VM pointers are nullified and the old allow-list erased before the PMO can be attached, as they may be used for attack transmission.

② **No Inter-Group Ptr.** In this invariant, inter-PMO pointers are permitted only when the source PMO and destination PMO belong to the same PMO group. As discussed earlier, checking a PMO for this invariant can detect a potential pointer attack e.g., DD and DI attacks (Figure 2 assuming PMO X, Y and Z belong to different PMO groups, each with a single member).

③ **PMO Size invariant** Checking the traversed number of nodes in the main data structure and the free list of a PMO against its declared size may detect attacks that rely on intra-PMO pointer modifications. To avoid traversal, we implement the free list as a kernel-managed data structure where nodes are allocated to a user space process from the list through `pmo_alloc()` and inserted back into the list on invocation of `pmo_free()` system call. OS maintains $\#allocated\_nodes$ and $\#free\_nodes$ metadata. Calling `pmo_alloc()` increments $\#allocated\_nodes$ and decrements $\#free\_nodes$, while calling `pmo_free()` decrements $\#allocated\_nodes$ and increments $\#free\_nodes$. Since allocation metadata is maintained in kernel space and updated only via trusted system calls, it cannot be directly modified by user processes and therefore does not require traversal-based validation. PMO size invariant requires that $size(\#allocated\_nodes + \#free\_nodes)$ must be equal to PMO's declared size.

An example attack that can be detected is one that redirects next pointer of a node to itself creating a cycle (i.e., DoS) or to a node different than its successor resulting in unexpected result of search operation (i.e., CA). This invariant assumes that a user process is free of memory leaks, e.g., allocating a node from the free list but not inserting it into the main data structure.

④ **Acyclic Topology.** If data structures are known to be acyclic, the presence of any cycle violates this invariant at any given time. In other words, the PMO data structure must stay acyclic over its lifetime. Checking a PMO for this invariant can detect some DoS attacks that create cycles on acyclic data structures. Checking for this invariant requires marking nodes as visited as the data structure is traversed, and any access to a node that was already visited indicates a cycle. The invariant requires the programmer's input (e.g., specifying data structure type through `pcreate()`) or shape analysis [15, 16].

⑤ **Data Structure-Specific Invariants (DSSI)** Each data structure may have its own specific invariant(s). For example, each list in a skiplist must be a subset of the list at lower levels [21], the number of black nodes on any path from the root node to a leaf must be same in a red-black tree [38], or next pointer of the tail node in a circular linked list must point to the head node, etc. DSSI was reportedly effective for detecting kernel-level rootkit attacks [4]. Applied to PMO, DSSI must be checked for both the free list and the main data structure. Checking a PMO for DSSI can detect illegal intra-PMO pointer modifications that violate at least one of them e.g., a CA attack that swaps left and right child pointers of a binary search tree or a DoS attack that disconnects a tail node from head node of a circular linked list.

Table 4. Summary of invariants and their attack coverage.

Inv	Name	Description	DD	DI	CFA	CA	DoS1	DoS2
①	PP2VM	Restricts PMO-to-volatile pointers (allow-list)			X			
②	Inter-group Ptr	Permits pointers only within same PMO group	X	X				
③	PMO Size	Ensures node count matches PMO size				X	X	
④	Acyclic	Detects cycles in data structures					X	
⑤	DSSI	Enforces data structure-specific properties				X		X
⑥	DIS	Enforces data properties (order, uniqueness)				X		

⑥ PMO data invariants There may be data-dependent invariants that the programmer intended, for example in a sorted linked list, keys must either be ascending or descending, keys must be unique, or key values must be in a certain range, etc. User processes sharing a PMO are expected to not cause deviation from these invariants. We collectively referred to these invariants as Data Invariant Set (DIS) which can vary across PMOs depending on its data. A key challenge to using DIS is how the invariants can be derived. In order to keep the programmer burden low, our design does not require users to manually specify data invariants. Instead, per-PMO likely data invariants are automatically inferred using Daikon [13], a dynamic analysis tool that generates a set of likely program invariants. For DIS generation, we use a dummy function as recommended by Daikon documentation [12].

Per-PMO DIS is generated after a PMO's creation through offline training. A PMO is not made available for actual use without completion of offline training and generation of DIS. In the offline training phase, a PMO is attached over time to multiple trusted kernel-level processes that capture the representative behavior of user processes expected to attach that PMO in its lifetime. We write a program (*DIS collector)* that runs on the PMO after every detach during the training phase. For example, to generate DIS for a PMO created to host a skiplist (as specified through `pcreate()`), *DIS collector* visits all nodes at all levels of the skiplist and while doing so invokes a dummy function on each node passing it node data-values and pointers. In this case, program invariants of the dummy function are the same as PMO data invariants since the function is invoked for all skiplist nodes. To increase the confidence in DIS, we set the confidence limit parameter (`daikon.inv.Invariant.confidence_limit`) to 1, and run *DIS collector* for the PMO after multiple runs of kernel processes using that PMO. Once generated, *DIS collector* registers DIS as per-PMO metadata with kernel. This allows the PMO checker to read DIS metadata and apply stored DIS on the PMO during invariant checking. Thus, DIS becomes a first-class, per-PMO kernel-maintained artifact that is generated once during offline training and enforced throughout the PMO's lifetime. Our current design assumes that per-PMO DIS is a static characteristic. While we recognize that DIS may evolve during the lifetime of a PMO, we leave dynamic generation of DIS as a future work.

Our focus is not on developing new data invariant specification or inference techniques, but on providing a framework that securely integrates externally supplied Data Invariant Set (DIS) into the PMO lifecycle and enforces them to detect corruption. PMO Checker can consume DIS produced by existing mechanisms, such as Daikon [13] or manually specified invariants. As a result, the degree of programmer involvement depends on the chosen invariant-generation mechanism rather than on our framework. In our current design, programmer burden is intentionally kept low by requiring only two inputs: (1) specifying the PMO's data-structure type at creation time via `pmo_create()`, and (2) providing representative kernel-level processes during an offline training phase. Beyond these inputs, the process of inferring likely data invariants using Daikon is automated, and invariant enforcement is transparent to users.

Table 4 summarizes the invariants, their purpose, and their attack coverage for clarity. DoS1 attacks change the number of nodes in the data structure (e.g. disconnecting nodes from a list), while DoS2 preserves them but change their connections.

Note that the invariants are independent of what operations are performed by P or V on a PMO. Since invariants are defined for a PMO, the checking is PMO-specific (and specific to the data structures it holds) irrespective of which process has last detached the PMO.

6 PMO CHECKER DESIGN

Before designing the PMO Checker, we need to set forth the design requirements. First, the design must meet the timing correctness criteria discussed in Section 5.1, i.e., checking must start after `detach` and complete before the next `attach`. Second, after a PMO has passed all invariants, checkpointing must be performed in a *crash consistent manner* to ensure a good checkpoint exists to recover from an attack. Third, the design should incur low latency and space overhead.

6.1 Design Approach

To meet the timing correctness criteria (i.e., checking must start after *detach* and complete before the next *attach*), we maintain states for each PMO. At creation, a PMO is checked for all applicable invariants, then checkpointed to provide an initial recovery capability. Then it enters the Checked and CheckPointed (CCP) state. A PMO can be attached by a user process only in this state. A PMO's state does not change during an attach-detach session. On detaching a PMO that attached with write permission (`detach(w)`), its state transitions from CCP to UnChecked (UC), necessitating invariant checking and checkpointing before it can be attached again. If invariant checking is successful, the PMO state transitions from UC to Checked (C), and after successful checkpointing, it transitions back to CCP. If any invariant check fails, the PMO enters CorRuPted (CRP) state, which triggers checkpoint recovery before it transitions back to CCP. Since a PMO can be attached only in CCP state, our approach meets the timing correctness criteria, where invariant checking and checkpointing can only start after a detach, and the next attach must wait until they are completed.

6.2 Crash-consistent Checkpointing

PMO checkpointing must be protected against partial updates. For example, when PMO checkpointing is in progress and a system/application crash happens, only partial updates are persisted in the checkpoint which renders it unusable for recovering a PMO to CCP state in case of an attack detected in future. To ensure a crash-consistent checkpointing mechanism, we could use in-place updates with logging or shadowing. We chose shadowing because it is simpler, especially checkpoint creation involves contiguous stores. Here the PMO Checker maintains a Primary CheckPoint (PCP) and a Shadow CheckPoint (SCP). Maintaining only a single checkpoint would risk losing the last consistent PMO state if a crash occurs during checkpoint persistence; alternating between PCP and SCP avoids this failure mode.

When checkpointing, the framework copies the PMO to its PCP, persists it by flushing, emits an sfence, and atomically sets an uncacheable flag called Checkpoint Validity Flag (CVF) to 0. If a crash happens while checkpointing PMO to PCP is in progress, SCP is available to recover PMO from attacks detected in future. On the next checkpointing, the same procedure is repeated but with PMO copied to the SCP, and the CVF atomically set to 1. Again, if a crash happens when checkpointing PMO to SCP is in progress, PCP can be used to recover PMO. This approach guarantees that at any time, one of PCP or SCP is always free of partial updates, allowing the PMO to be recovered from. As checkpointing needs both PCP and SCP, its storage overhead is roughly double the PMO size.

6.3 Low-Overhead PMO Checker Framework Design

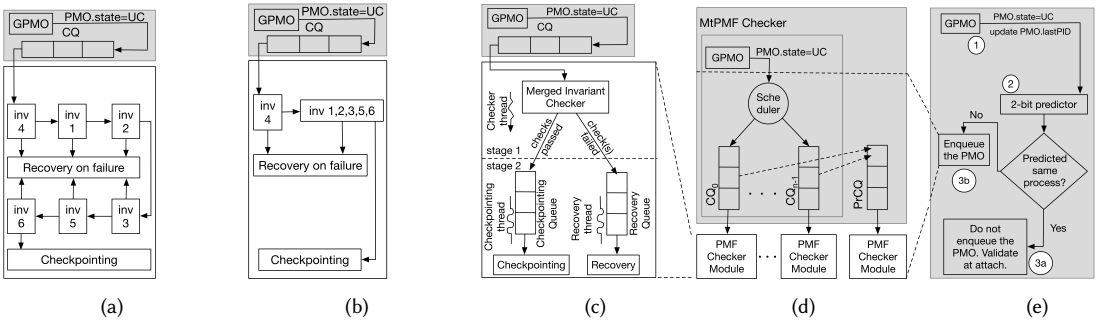


Fig. 3. PMO Checker Design. (a) Base Design: FIFO (F), with (b) Merging (MF), (c) Pipelining (PMF) (d) Multithreading (MtPMF) and Priority checking (MtPMF+Pr), and (e) Skipping check (MtPMF+Pr+S) optimizations.

We will discuss our PMO Checker framework starting from basic design to increasingly optimized versions.

6.3.1 FIFO invariant checks (F). Figure 3a shows our base design. PMO Checker kernel components are shown in grey and components as a loadable kernel module in white. We added a checking queue (CQ) to queue PMOs that got detached (and enter UC state). The CQ is a FIFO queue (except when priority checking optimization is enabled as explained in Section 6.3.5), hence we refer to this design as *F*. Invariant checking is performed for each PMO at the head of the queue, in the sequence of invariants 4, 1, 2, 3, 5, and 6. Invariant 4 is checked first because all others may not terminate if the data structure is cyclic. To perform an invariant check, the PMO checker traverses all nodes of the data structure, e.g., verifying that each pointer of a linked-list points to a node that belongs to a PMO within the same PMO group (i.e., no inter-group pointers). Any failed invariant check results in triggering recovery. After checking is complete, checkpointing is initiated.

6.3.2 Merging invariant checks (MF). We noted right away that the base *F* scheme is inefficient. Each invariant traverses data structures in the PMO once; repeated traversals incur a high latency. Thus, we explore merging multiple invariant checks into a single traversal, and refer to this as Merged F (*MF*) Checker (Figure 3b). Invariant ④ (cycle detection) is not merged with others because its traversal pattern is depth first, incompatible with other invariants. *MF* thus reduces the traversal count from 6 to 2.

6.3.3 Pipelined checking and checkpointing (PMF). The latency suffered by a PMO does not just depend on invariant checking and checkpointing, but also depends on the amount of time it spends in the checking queue, especially if there is a burst of many PMOs needing to be checked. In order to improve throughput, we pipeline the invariant checking and checkpointing or recovery, as illustrated in Figure 3c. We refer to this scheme as Pipelined MF (*PMF*). We added two additional queues (checkpointing and recovery queues), and assign separate threads for checking, checkpointing and recovery. With PMF, while one PMO is being checked for invariants, another PMO can move to either checkpointing (if invariant checks succeeded) or recovery (if failed). For pipelining to provide the best throughput improvement, pipeline stages need to be roughly equal in latencies. We found that invariant checking and checkpointing are not equal but reasonably close, e.g. for an 8-MB PMO, contributing almost 60% and 40%.

6.3.4 Multi-threading (MtPMF). There can be situations when the invariant checking thread of PMF can become a bottleneck. For example, when multiple processes attach different PMOs for a short period of time and then detach them,

the checking queue quickly accumulates pending PMOs putting more work pressure on the invariant checking thread. Alternatively, PMOs may be detached by processes less frequently but are relatively large in size, again increasing work pressure on the checker thread and converting it to a performance bottleneck. To mitigate the bottleneck, we can take advantage of multithreading by having multiple threads for checking, checkpointing, or recovery.

Two challenges need to be addressed. First, we need to decide whether the checking (checkpointing) threads should share a single checking (checkpointing) queue. Queue sharing is space efficient but incurs synchronization costs as multiple threads will access the same queues, and the cost increases with larger thread counts. Since each queue is small (small number of entries and small amount of metadata per entry), storage overhead is of little concern, hence we allocate each thread its own queue (Figure 3d). Evaluation results in Figure 5 of Section 8.2 show that under identical workloads (i.e., AVL and SL) stressing the performance, the multi-queue design (i.e., MtPMF) exhibits 41% (AVL) and 45% (SL) lower execution times compared to the single-queue design (i.e., PMF), indicating a corresponding reduction in effective PMO processing throughput due to queue contention and synchronization overhead. A second challenge is whether idle threads occupy resources such as cores. To avoid excessive resource consumption, we added a scheduler to put threads to sleep if their queues are empty. The scheduler also enqueues each incoming PMO to one of checking queues and awakens the corresponding thread. Races can still occur between the scheduler (which enqueues) and a checking thread (which dequeues), but the synchronization cost is low for two threads, and alternatively lock-free queue can be used.

To address the load balancing problem arising from usage of multiple checking queues, the scheduler tracks the number of waiting PMOs in each queue. When a new PMO is detached by a process, the scheduler enqueues it to a checking queue with the lowest count of waiting PMOs. This ensures that load is fairly distributed to checking threads working on their corresponding checking queues. The design is named Multi-threaded (*Mt*)PMF.

6.3.5 Priority Checking (MtPMF+Pr). Despite pipelining and multi-threading, when a small PMO is queued behind a large PMO, an `attach()` call will fail, and the user process making the call must wait until other PMOs ahead of it in the queue are completed before the PMO can even start invariant checking. This observation prompted us to optimize the design by prioritizing PMOs that are requested for attach over others. To achieve that, we create a priority queue set (one queue for checking, another for checkpointing, and another for recovery) with their own worker threads. Any waiting PMO that is requested for attach is moved from its queue (i.e., CQ_0 to CQ_{n-1}) to the Priority Checking Queue (PrCQ). Each checking queue operates in FIFO order under normal operation; except when priority checking is enabled. The optimized design, shown in Figure 3d, is referred as MtPMF with Priority (MtPMF+Pr) checking. The load of PrCQ is likely low as it holds only PMOs that are moved from checking queues.

6.3.6 Skipping Invariant Check (MtPMF+Pr+S). All previous optimizations aim at reducing checking *latency* or improving its throughput. Now we would like to see if some checkings can be skipped altogether. We note that if there is no attach by other processes between a PMO's detach and attach by the same process, then invariant checking can be skipped as there are no inter-process threats. However, a challenge is that this is unknown apriori, i.e. only known after an attach call is made. Therefore, employ an *attach predictor*. For each PMO, as shown in Figure 3e, (1), at each detach, we modify the GPMO to record the ID of the last detaching process. (2) GPMO then uses a per-PMO predictor to predict if the next attaching process will be the *same* as or *different* from that of the last detaching process. If the prediction is the *same* process, (3a) we do not enqueue the PMO for checking. Otherwise, (3b), we enqueue the PMO. We use a simple two-bit saturating counter predictor. On an attach request, we verify if the prediction is correct. If correct, the PMO is attached immediately; invariant checking was correctly skipped. Otherwise, misprediction has

occurred, and the PMO is enqueued in the priority queue for invariant checking and checkpointing. The scheme is referred to as *MtPMF+Pr+S*.

6.3.7 Page Level Checkpointing (MtPMF+Pr+S+PgLC). Previous optimizations focused on improving throughput of invariant checking and checkpointing. For our final optimization, we observe that not all pages of a PMO are always modified by a user process in an attach session. Therefore, the cost of checkpointing/recovery can be reduced if we can track pages that become dirty in a previous attach session. However, tracking such pages is not simple; an expensive page table walk is required to identify dirty pages before checkpointing, and the cost increases with larger PMOs. Here we could benefit from a simple hardware support that sets dirty bit of pages that are written in an attach session, and resets them at detach. However, it turns out that `psync` primitive in GPMO already prepares a list of dirty pages when invoked by a user process for persisting PMO updates. Information from the list can be accumulated over an entire attach session, and repurposed for our checkpointing. The optimized design is referred to as *MtPMF+Pr+S+PgLC*. This optimization is not shown in Figure 3 due to limited space.

6.4 Framework Security:

The PMO Checker's ability to detect a security attack depends on whether the attack violates an invariant. Our contribution is an extensible PMO Checker framework, hence a complete study of what invariants provide good coverage is beyond the scope of this paper, although we note that for attacks described in prior work [31, 32], we only found one instance of undetected corruption (Section 8.4). Note that our framework currently supports manual addition/removal of invariants which can be automated in the future.

A key is that the PMO Checker framework itself should not add security vulnerabilities that can be used to bypass the checking and checkpointing. Here the framework introduces a relatively small code base (6,636 lines of code) and small amount of metadata (PMO state, queues, ID of last detaching process, thread states, etc.) that are kept in the OS kernel or a kernel module. This keeps the attack surface small and our trust model assumes that the kernel and its modules are trusted. The only kernel data that the attacker may affect is the PMO's checkpoints. The attacker may attempt to corrupt a checkpoint, by using the payload P to attach the PMO, corrupt it, psync and detaches it. If the corrupted checkpoint is used to restore the PMO, the PMO may be corrupted as well, affecting victim V. However, this scenario is not possible because checkpointing is only performed after invariant checking passes. Furthermore, a checkpoint is kept in the kernel space hence the attacker must compromise the OS to be able to corrupt it. Our trust model assumes a trusted OS. The final vulnerability is when P launches a Denial of Service (DoS) through repeated PMO corruption. Our framework does not protect against DoS and it is beyond the scope of this paper.

6.4.1 False Positives and Negatives. Count of false positives and negatives may be reduced by revising or adding more accurate invariants, hence we design the PMO Checker and its optimizations to accommodate future invariants. Additionally, false positives can be addressed by one of or a combination of 1) creating a PMO-specific log that can be analyzed by users, 2) quarantining a violating PMO by not permitting further attaches, and 3) restoring the violating PMO to a prior checkpoint to allow the next process to resume computation. Both 2 and 3 prevent potential PMO corruption from spreading to other processes, including the victim.

Table 5. 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)
DRAM	4 × 32GiB DDR4 @ 2666MHz
PMEM	4 × 128GiB Intel Optane DC
OS and kernel	Linux 5.14.18

6.5 Discussion and Limitations

PMO Checker makes several assumptions that may influence deployment in production environments. First, we assume a trusted operating system and kernel modules. This assumption is shared by prior PMO protection mechanisms [17, 18, 44–46], and reflects the practical reality that production systems typically rely on kernel integrity as a foundational trust anchor. Attacks that compromise the kernel itself are outside the scope of this work. Second, while PMO Checker supports extensible invariant sets, including manual addition or removal of invariants, this capability is intended primarily for system integrators or developers rather than end users. In practice, invariant management is expected to be infrequent and driven by evolving attack knowledge or data-structure changes. Moreover, our design is invariant-agnostic and compatible with automated invariant-generation mechanisms (e.g., Daikon [13]), reducing the need for manual intervention. Overall, these assumptions trade off deployment simplicity against generality, but they do not preclude practical use: PMO Checker is implemented entirely in kernel space, requires no custom hardware or compiler support, and integrates with existing PMO systems with modest configuration effort (i.e., loading a kernel module).

7 EVALUATION METHODOLOGY

7.1 System and Hardware Platform

The GPMO and our PMO Checker module was implemented in Linux Operating System with Fedora 33 distribution with Kernel v5.14.18. We evaluate PMO Checker on an Intel Xeon-based system with 20 cores and Intel Optane DC persistent memory (Table 5).

7.2 Real Application Workloads

Prior work [5, 29, 44–46] uses YCSB workload on a key-value store for evaluation purpose. To evaluate PMO Checker on real applications, we similarly ported Lightning Memory-mapped Database Manager (LMDB) [20], a key-value store that uses B+ tree to store records, by allocating the tree on a PMO [23]. We refer to the port Persistent LMDB (PLMDB). We use the YCSB database initialized with 50,000 records in a 100MB PMO, and use a single thread default. For the first scenario, the server hosts a single database (DB) and deals with a single client performing 32 operations. For the second scenario, the server hosts four DBs, each serving two clients (8 clients total), with each client performing 16 operations (128 operations total). For both scenarios, we choose update-intensive workload A of YCSB [9], which have the highest fraction of transactions that are updates (50%), in order to stress test our PMO Checker. Following LMDB's default behavior [22], the DB is persisted after every update (i.e., `mdb_txn_commit`).

7.3 Microbenchmarks:

To evaluate PMO Checker broadly beyond YCSB and to study the performance sensitivity to various scenarios, we developed two microbenchmarks: a SkipList (SL) [35] (restricted to two levels with 10% level-1 nodes) and Adelson-Velskiy and Landis (AVL) tree [14]. Each microbenchmark creates multiple PMOs and allocate a free list and their respective main data structures in each PMO. Prior work also uses similar number and type of micrbenchmarks e.g., AVL tree [46], Linked-list [29, 46], Red-Black tree [5, 46], and B-Tree [29] etc.

For the base evaluation, we created 40 8-MB PMOs in total, divided into a group of eight shared PMOs, plus four eight-PMO groups (i.e. $8 \times 4 + 8 = 40$). Each run launches 4 processes, where each process performs 1,000 sessions, in each session attaching a PMO, modifying it (inserting/deleting a node), persisting it (via psync), and detaching it (Listing 1). Persisting after each operation is similar to the behavior of the YCSB workload, and is consistent with the recommendation by [44] for reducing the attack surface. There are a total of $4 \times 1,000 = 4,000$ attach sessions total. For each attach, the PMO is randomly selected from either its private or shared PMO group, based on the *contention level* (x), e.g. x% from the shared group and $(100 - x)$% from its private group.

```
while(count){count--;
    PMO_ID=get_random_PMO_ID();//get PMO ID
    do{ /*non-blocking PMO attach until success*/
    ret_val=attach(PMO_ID);
    }while(ret_val is failure);
    //modify PMO
    //persist and then detach PMO
    psync(PMO_ID); detach(PMO_ID);}
```

Listing 1. Pseudo code of a user process

7.4 Evaluated Schemes:

To use as a baseline for evaluation purpose, we run our benchmark on GPMO without PMO Checker enabled ((N)one). We report performance of our base design, i.e., FIFO (F) Checker, successively combined with optimizations i.e., Merging (MF), Pipelining (PMF), Multi-threading (MtPMF), Priority checking (MtPMF+Pr), Skipping checks (MtPMF+Pr+S), and Page Level Checkpointing (MtPMF+Pr+S+PgLC). For multi-threaded designs, we use only two FIFO queue/thread sets, and only a single queue/thread set is employed for priority checking. In case of misprediction, our Skip (S) checking optimization prioritizes checking of the PMO in UC state using Pr. Therefore, we do not evaluate S as a single optimization but only in combination with Pr.

8 EVALUATION RESULTS

We first evaluate the PMO Checker by running the four workloads and then analyze its performance sensitivity to number of workload processes.

8.1 Real Application Workload Performance

Figure 4 (left) shows the execution time of the YCSB's workload A with a single client and a single DB for various designs, normalized to no checking (*N*, the last bar), and broken down into the workload time (*no checking*), exposed

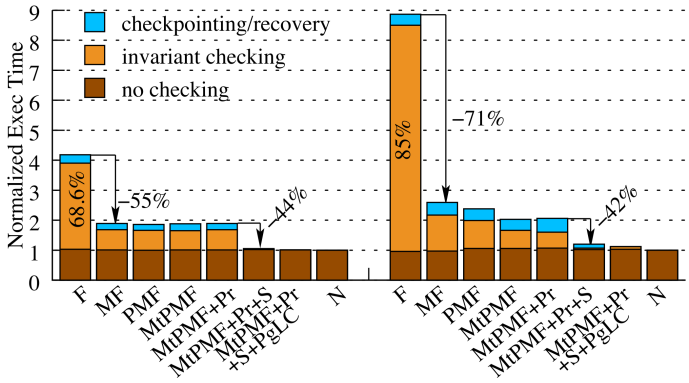


Fig. 4. YCSB Workload A evaluation: single DB and single client (Left), and 4 DBs with 2 clients per DB (Right).

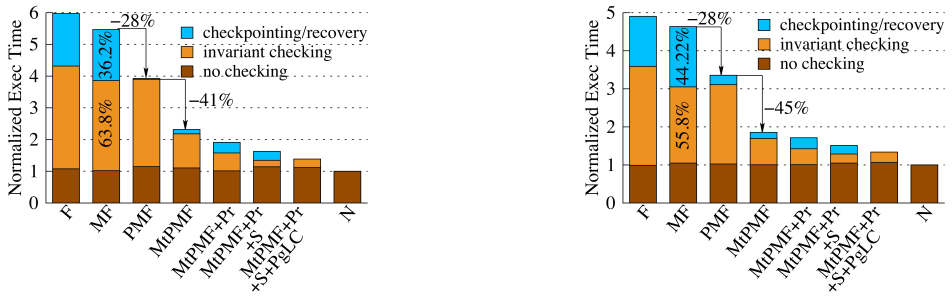
invariant checking time, and exposed *checkpointing/recovery* time. A latency is exposed if it incurs critical path delay in checking or checkpointing that delays a successful attach. The figure shows that the base design F incurs a slowdown of 4.1×. We checked for all five B+ Tree invariants , including Data Structure Specific Invariants (DSSI) [10], resulting in huge invariant checking time (68.6%). The MF optimization reduces the number of traversals from five to two, reducing the execution time by 55% [1]. Pipelining (PMF), multithreading (MtPMF) and priority checking (+Pr) have no impact as we only have a single PMO. Note that the PMO is repeatedly attached by the same client process, hence the addition of check skipping (MtPMF+Pr+S) reduces the execution time by 44% over MtPMF+Pr, where PMO Checker only incurs a 1% slowdown over N.

In order to test the effectiveness of PMO Checker for a situation with multiple PMOs and processes, Figure 4 (right) shows the average execution time for 8 clients running workload A on 4 DBs, where each DB is accessed by pair of two clients. The PMO Checker is more stressed as it now checks invariants for 8 clients and 4 PMOs, with the baseline F slowed down by 8.9×. Invariant merging (MF) reduces the execution time substantially, by 71%. Pipelined checking (PMF) slightly improves the performance by overlapping invariant checking and checkpointing/recovery. Multithreading (MtPMF) with two queue/thread sets reduces the invariant checking time by half over PMF. We observe that priority checking (+Pr) does not further reduce execution time. This is likely because of the benefit of priority checking materializes with a larger number of PMOs (as in our microbenchmark results in Figure 5). Adding check skipping (MtPMF+Pr+S) reduces the invariant checking time by 42% over MtPMF+Pr. This is because our 2-bit predictor is able to predict the process that attaches next by a prediction accuracy of 87%. At this point, the remaining overhead is dominated by the checkpointing time, hence MtPMF+Pr+S+PgLC further improves performance by checkpointing only dirty pages. Our data shows that, on average, an update operation dirties 11 pages out of 25,600 pages (0.04%) from a 100 MB PMO-resident DB. With all optimizations combined, our PMO Checker incurs a minor 12% slowdown over no checking. Overall, our combined optimizations are greatly effective. Most (98.5%) of the original overheads from the base design are eliminated for multi-client YCSB.

8.2 Microbenchmarks

We repeat the overhead measurements, for microbenchmarks with AVL (Figures 5a) and SL (Figure 5b) workloads. Compared to YCSB, we increase the performance stress by using short attach sessions leading to frequent attaches

[1]Invariant 4 uses one traversal, while four other invariants are combined into one traversal. Invariant 6 is omitted as Daikon breaks when inferring the likely data invariants of PLMDB's B+ tree.



(a) AVL (b) SL

Fig. 5. Performance evaluation on AVL Tree (5a) and Skip List (i.e., SL 5b) microbenchmarks.

and detaches, with each session modifying and persisting the PMO. The figure shows the base design F's significant slowdowns (6.0× for AVL and 4.9× for SL). While all optimizations are effective, a few optimizations are notably impactful. The first is pipelining, which reduces execution time by 28% going from MF to PMF, due to hiding the most of the checkpointing time behind invariant checking time. The second is multithreading, reducing the execution by 41% (for AVL) and 45% (for SL) over PMF. Both optimizations are notably more effective on the microbenchmarks than on YCSB, due to the larger number of PMOs (40 vs. 1-4), which provides more opportunities to benefit from pipelining and parallel processing. Priority checking (+Pr) and especially invariant check skipping (+S) substantially reduce invariant checking time due to the predictor's accuracy. As before, after check skipping, the remaining overheads are dominated by checkpointing, which is then nearly entirely removed by Page level checkpointing (+PgLC). Everything combined, the slowdowns are lowered to 1.4× (AVL) and 1.3× (SL). These overheads are reflective of high performance stress scenarios, due to the short attach sessions (each with little computation), large process counts, and large PMO counts being shared. Hence, in most real world scenarios, the performance overheads are likely to be substantially lower.

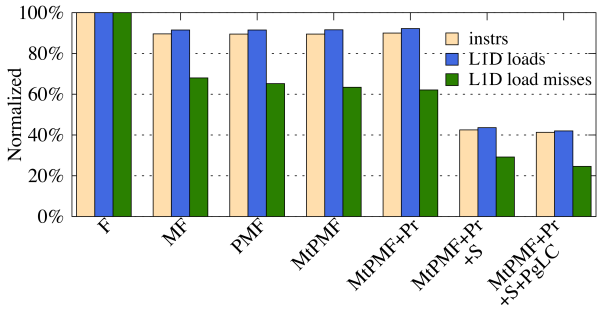


Fig. 6. PMO Checker's behavior for AVL.

To get a deeper insight, Figure 6 shows instruction counts, L1D loads, and L1D misses as experienced by different designs of PMO Checker while running AVL microbenchmark. Several observations are notable. MF reduces instruction counts slightly, but L1D misses substantially due to improved temporal locality. PMF, MtPMF, +Pr, improve performance via overlapping execution and reducing waiting time, hence microarchitecture statistics do not decline much. Check skipping (+S) reduces the number of invariant checkings and checkpointings, hence instruction counts, L1D loads, and L1D misses, are reduced by a little more than a half. Finally, the page-level checkpointing (+PgLC) reduces pages that

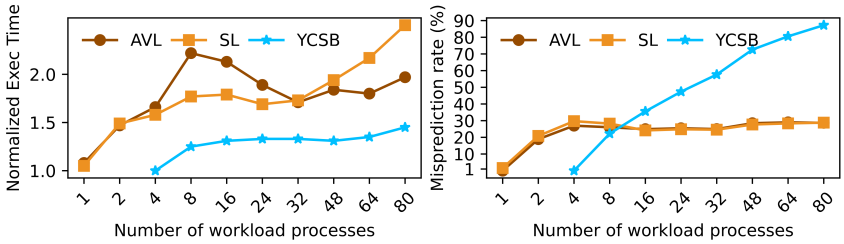


Fig. 7. Sensitivity of execution time workloads (left) and misprediction rate of 2-bit predictor (right) to count of workload processes. The pridictor is used by Skipping Check optimization (MtPMF+Pr+S).

# Clients/PMO	1	2	4	6	8	12	16	20
Misprediction (%)	0.0	22.22	35.58	47.37	57.64	72.6	80.55	87.24
Overhead (×)	1.00	1.25	1.31	1.33	1.33	1.31	1.35	1.45

Table 6. Predictor misprediction rate and normalized overhead of MtPMF+Pr+S under increasing YCSB concurrency.

are checkpointed, and since checkpointing has low temporal locality, its biggest impact is on reducing L1D load misses. We observe similar trends in SL and YCSB hence we do not plot them.

8.3 Sensitivity Studies

Performance Sensitivity to Number of Workload Processes. To evaluate the sensitivity of performance to the number of workload processes, we conducted experiments using the MtPMF+Pr+S scheme. The number of threads dedicated to invariant checking, checkpointing, and recovery was fixed at six—comprising three threads for the two regular checking queues and the priority queue. Although we used multi-process workloads in this sensitivity study, the concurrency patterns exercised are representative of multithreaded environments, as PMO Checker behavior is driven by the level of concurrent PMO accesses, i.e., `attach()` or `detach()`, rather than the execution abstraction (i.e., processes or threads).

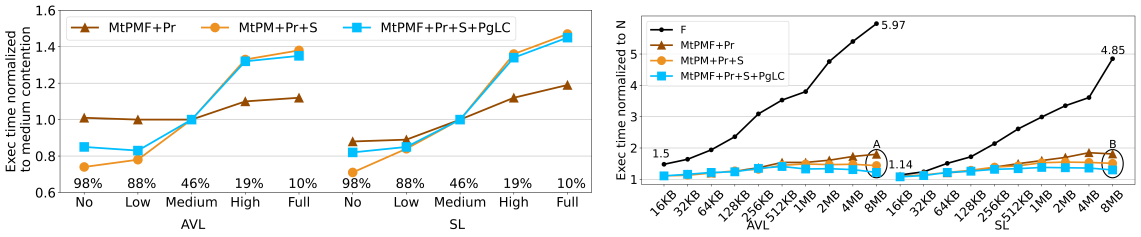
We ran the AVL, SL, and YCSB workloads while gradually increasing the number of processes. For the AVL and SL workloads, each process was assigned a private group of eight PMOs. In addition, a shared group of eight PMOs was accessible to all processes. Out of 1000 attach requests, each process directed 50% to its private PMO group and the remaining 50% to the shared group. For the YCSB workload, four databases were used. The number of client processes accessing each database was varied from 1 to 20, resulting in a total process count ranging from 4 to 80.

Figure 7 shows that the execution time (left) normalized to no checking exhibits stable trends with low-slope increases for SL and YCSB, demonstrating the scalability of the PMO Checker. Despite relying on multiple threads for invariant checking and checkpointing, it effectively handles an increasing number of user processes. The slow rise in normalized execution time with YCSB correlates with a higher misprediction rate (Figure 7, right), making it more difficult to predict that the next attaching process will be the same as the current one. This suggests that a more sophisticated predictor could enhance performance, which we will explore in future work.

To quantify the impact of mispredictions under high concurrency, Table 6 reports misprediction rate and normalized overhead for MtPMF+Pr+S as the number of YCSB clients increases. While the misprediction rate rises sharply, reaching 87% with 20 clients per PMO-hosted database (i.e., 80 total workload processes), the corresponding execution-time overhead increases only modestly from 1.0× to 1.45×. This shows that although mispredictions trigger additional invariant checking, their cost is bounded: a misprediction delays only the affected attach and does not stall concurrent

checking, checkpointing, or recovery of other PMOs. As a result, PMO Checker remains scalable even under highly concurrent, real-world workloads where predictor accuracy degrades.

Next, by running AVL and SL microbenchmarks, we evaluate the sensitivity of our three most performant designs.



(a) Contention Level

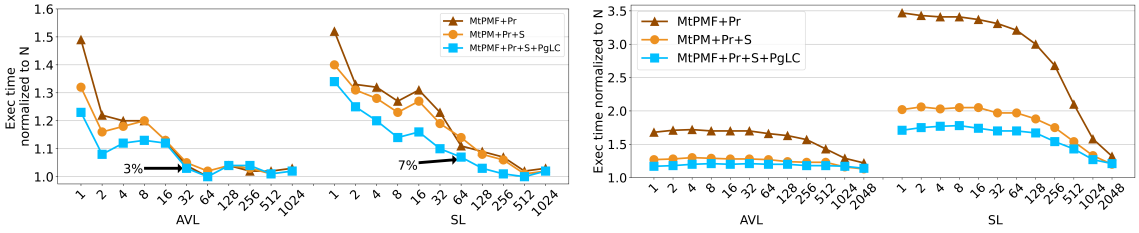
(b) PMO Size

Fig. 8. Performance sensitivity to contention level (8a) and PMO size (8b).

Contention Level Sensitivity. Contention level is defined as the percentage of attaches made to shared (rather than private) PMOs. Figure 8a shows the execution time for all five contention levels from No (0%), low (10%), medium (50%, default), high (90%), and full (100%), normalized to the medium level. The x-axes also shows numerical percentages of attach calls that are consecutively for the same PMO by the same process at each contention level. The percentage reduces with increasing contention. As expected, the execution time increases with higher levels of contention as we move to the right along the x-axes. Furthermore, the figure also shows that designs employing skipped checking optimization (i.e., MtPMF+Pr+S and MtPMF+Pr+S+PgLC) are more sensitive. Upon closer analysis, the reason for the heightened sensitivity is the percentage of consecutive attaches made by the same process which declines substantially with higher level of contention (down from 46% to 10% in full) and increases substantially with lower level of contention (up from 46% to 98% with no contention). Comparatively, MtPMF+Pr is less sensitive as it checks all detached PMOs (including those attached by the same process) at all contention levels.

PMO Size Sensitivity. Figure 8b shows the sensitivity of PMO Checker performance to the PMO size, varied from 16KB to 8MB. The figure reports execution time of workloads with four schemes including F and MtPMF successively combined with +Pr, +S, and +PgLC. For each design, the execution time is normalized to no checking N for respective PMO sizes. As expected, the execution time increases with the PMO size as invariant checking, which requires traversing the entire data structure, takes longer to perform. The base design F is the most sensitive to the PMO size as its slowdown increases from 1.5× to 5.97× (AVL) and from 1.14× to 4.85× (SL) going from 16KB to 8MB. Our optimizations are effective in reducing the performance sensitivity to the PMO size (shown in circles *A* and *B*). Furthermore, our most effective scheme (MtPMF+Pr+S+PgLC) no longer shows sensitivity once the PMO size reaches 1MB as their slowdowns are roughly constant or slightly decreasing beyond 1MB. This can be explained by two reasons: (1) page-level checkpointing is dependent on the number of dirty pages instead of the PMO size, and (2) while invariant checking time increases with the PMO size, its increase is slightly smaller than the increase of the workload time. These two factors led to the decreasing execution time overheads once the PMO reaches 1MB.

Sensitivity to Attach Session Length. Figure 9a shows the sensitivity of PMO Checker performance to the attach session length (number of operations, i.e., inserting/deleting a node, per attached session). We report the execution time of the workloads with three most performant schemes, MtPMF+Pr and its successive combination with S and PgLC



(a) Attach session length (b) Dirty page count

Fig. 9. Performance sensitivity to attach session length (9a) and number of dirty pages (9b).

optimization, when the attach session length is varied from 1 (default) to 1,024. All execution times are normalized to N for the respective session lengths.

With the increasing attach session lengths, all three designs show decreasing slowdowns compared to N. Not surprisingly, MtPMF+Pr is most sensitive as its initial slowdown was the highest, and MtPMF+Pr+S+PgLC the least sensitive as its initial slowdown was the smallest. One important observation is that at sufficiently sized attach session (32 for AVL and 64 for SL), the most optimized PMO Checker scheme reaches less than 10% overheads, and even less optimized schemes reach that threshold at the same or slightly longer attach session. Beyond those points, the overheads become nearly negligible. Thus, we expect that unless the programmer utilize many small attach sessions, the performance overheads of PMO Checker would be nearly negligible. This is good news because the programmer can adjust attach session length as a way to control the overheads of invariant checking and checkpointing.

Sensitivity to the Number of Dirty Pages. Figure 9b shows the sensitivity of execution time of three of our most performant designs to the number of dirty pages per attach session, varied from 1 to 2,048 pages for an 8MB PMO with a 4KB page size. Execution times are normalized to the respective no checking (N) case. The figure shows decreasing execution time slowdowns as the number of dirty pages increases. This is because the workload time to write to more pages increases while invariant checking and checkpointing times are relatively unaffected by the number of dirty pages. Hence, the relative execution time overheads decrease.

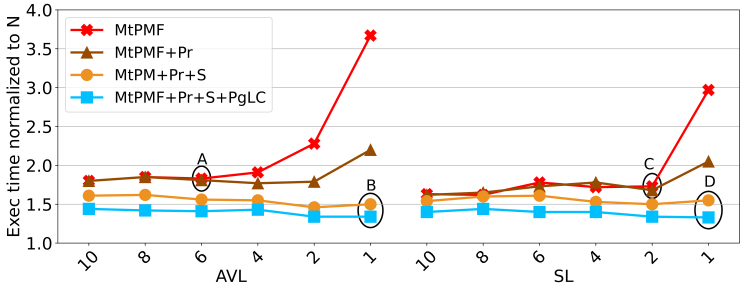


Fig. 10. Resource requirement of PMO Checker.

Resource Requirement. To study the resource requirement (i.e., thread count) sensitivity of our PMO Checker performance, we vary the number of queue and thread sets from 10 to 1. Each set contains a trio of queues and threads, i.e., one for checking, another for checkpointing, and the last for recovery. For any case, there is only one priority queue used. The workloads (AVL and SL) use four user processes running on four cores for all cases. With four workload

Table 7. Attack detection coverage for example attacks.

Attack	DD	DI	CFA	DoS1	DoS2	CA-P	CA-D
Detection coverage (%)	100	100	100	100	100	100	65

processes (AVL or SL), three priority-queue threads (checking, checkpointing, and recovery), and the number of FIFO thread sets varying from 10 to 1, the total number of threads and processes ranges from 37 to 9 on a platform with up to 40 cores. We select 10 thread sets as the upper bound to remain within the platform's core capacity; larger values would oversubscribe cores and confound performance analysis with scheduling effects rather than PMO Checker behavior.

Figure 10 shows the execution time of workloads with decreasing queue/thread sets, normalized to no checking (N) case. The figure shows that in general, as we use fewer threads, execution time increases due to additional queueing delays. However, the more optimized the scheme, the less sensitive it becomes to running on fewer threads. This is particularly true for our two most performant schemes: MtPM+Pr+S and MtPM+Pr+S+PgLC. The most important point from the figure, however, is what is the minimum thread set count needed for PMO Checker to perform well. Less optimized schemes (MtPMF and +Pr) required either 6 (AVL) or 2 (SL) thread sets (points A and C in the figure). In contrast, the two most optimized schemes (MtPM+Pr+S and +PgLC) do well with only one thread set (points B and D in the figure). Therefore, not only our optimizations reduce execution time overheads, they also reduce the number of thread sets needed to do well.

8.4 Security Evaluation

To evaluate the PMO Checker security, we built a simple persistent database modeled after SQLite. It consists of three tables shown in Figure 2, i.e., PMO X, Y, and Z holding `Free_members`, `Most_active_members`, and `Premium_members` tables, respectively, stored in data structures. Two clients, representing payload and victim processes of our threat model, access the database. Both share `Free_members` and `Most_active_members` tables (i.e., PMO X and PMO Y in Figure 2) while `Premium_members` (i.e., PMO Z) is private to the victim. In our security evaluation experiment, both clients perform 420 PMO attach sessions individually. The Payload (P) client corrupts the shared PMOs to launch a randomly selected attack on the Victim (V) client. A CA attack is distinguished based on whether it corrupts a PMO pointer (CA-P) or data (CA-D). Note that likely Data Invariant Set (DIS) for our PMOs (generated using Daikon tool [12]) states that record keys are sorted in ascending order and unique.

As a measure of security effectiveness of PMO Checker for example attacks, we report the *attack detection coverage*, defined as the number of detected attacks divided by the number of attacks launched by the payload client. Our results, shown in Table 7, report that our PMO Checkers provide 100% attack detection coverage except for CA-D which is detected with 65% coverage. This is because the random corruptions of key values of records do not violate DIS that was derived from Daikon [13] (Section 5.4), for 35% of CA-Data attacks.

8.4.1 Overhead of False Positives and Negatives. A false positive is when PMO Checker detects an attack while the detached PMO has no corruption. A false negative is when PMO checker detects no attack while the detached PMO is corrupted. For all tested attacks shown in Table 7, no false positive or false negatives are observed for DD, DI, CFA, DoS1, DoS2 and CA-P attacks. Likewise, no false positives are observed for CA-D attack. However, 35% (i.e., for 21 out of 60) false negatives are observed for CA-D attack. Remember that attack detection triggers PMO recovery while no attack detection triggers PMO checkpointing. In other words, for those 35% false negative CA-D attacks, the right cost of PMO-recovery is replaced by the false cost of PMO checkpointing (thus reducing protection effectiveness). Note

that the most optimized design of PMO Checker (*MtPMF+Pr+S+PgLC*) checkpoints only dirty pages of the detached PMO which incurs overhead as low as 12% in YCSB multi-client real workload (Figure 4) and almost entirely removes overhead in SL and AVL microbenchmarks (Figure 5). This overhead may vary depending on how many PMO pages are modified by attacker in the detached PMO.

Partial attack coverage and the cost of false negatives for CA-D attacks suggests that in some cases, automatically-derived invariants may not be sufficient, and may need to be augmented by programmer-specified invariants, to provide higher attack detection coverage. The key takeaway is that the PMO Checker must have the flexibility to add/remove/modify invariants, which is provided by our framework.

9 CONCLUSION

We presented a novel kernel-based PMO Checker framework that uses invariant checking for attack detection and checkpointing for recovery. To address high performance overheads in a naive design, we identified bottlenecks and proposed optimizations that reduce overhead to 1-12% across two real application workloads. Our evaluation confirmed effective attack detection coverage, achieving 100% in all but one case, which had 65% coverage.

REFERENCES

[1] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 263–277.

[2] Starr Andersen and Vincent Abella. 2004. Data execution prevention. changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies.

[3] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th international symposium on software testing and analysis*. 177–188.

[4] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2010. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (2010), 670–684.

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

[6] Dion Blazakis. 2010. Interpreter exploitation: Pointer inference and JIT spraying. *BlackHat DC* (2010).

[7] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. 2022. An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences* 12, 13 (2022), 6702.

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

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[11] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo*.

[12] Daikon. 2022. *The Daikon Invariant Detector User Manual*. Retrieved Sep 17, 2024 from http://plse.cs.washington.edu/daikon/download/doc/daikon.html

[13] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[14] Caxton C Foster. 1965. Information retrieval: information storage and retrieval using AVL trees. In *Proceedings of the 1965 20th national conference*. 192–205.

[15] Rakesh Ghiya and Laurie J Hendren. 1996. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–15.

[16] Rakesh Ghiya and Laurie J Hendren. 1998. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 121–133.

[17] Derrick Greenspan, Naveed Ul Mustafa, Jongouk Choi, Mark Heinrich, and Yan Solihin. 2025. Persistent Memory Objects on the Cheap. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 1234–1249.

[18] Derrick Greenspan, Naveed Ul Mustafa, Andres Delgado, Connor Bramham, Christopher Prats, Samu Wallace, Mark Heinrich, and Yan Solihin. 2024. 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, 131–142.

[19] Derrick Greenspan, Naveed Ul Mustafa, Zoran Kolega, Mark Heinrich, and Yan Solihin. 2022. Improving the Security and Programmability of Persistent Memory Objects. In *2022 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 157–168.

[20] Gavin Henry. 2019. Howard chu on lightning memory-mapped database. *Ieee Software* 36, 06 (2019), 83–87.

[21] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm. In *International Colloquium on Structural Information and Communication Complexity*. Springer, 124–138.

[22] Symas Corporation. Howard Chu. 2015. *LMDB: Lightning Memomr-Mapped Database Manager (LMDB)*. Retrieved Sep 17, 2024 from http://www.lmdb.tech/doc/index.html

[23] Symas Corporation. Howard Chu. 2023. *Read-only mirror of official repo on openldap.org*. Retrieved Sep 17, 2024 from https://github.com/LMDB/lmdb

[24] Intel. [n. d.]. *Persistent Memory Developmeent Kit (PMDK)*. Retrieved Sep 17, 2024 from https://pmem.io/pmdk/

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.

[26] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, Jinsuk Ma, Myeong-Hoon Oh, and Youngjae Kim. 2020. Persistent memory object storage and indexing for scientific computing. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 1–9.

[27] Kioxia. 2022. *SLC NAND Flash Memory | Kioxia - United States*. Retrieved Sep 17, 2024 from https://americas.kioxia.com/en-us/business/memory/slc-nand.html

[28] Richard Koo and Sam Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering* SE-13, 1 (1987), 23–31.

[29] Suyash Mahar, Mingyao Shen, TJ Smith, Joseph Izraelevitz, and Steven Swanson. 2024. Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 575–589.

[30] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[31] Naveed Ul Mustafa and Yan Solihin. 2023. Persistent Memory Security Threats to Inter-Process Isolation. *IEEE Micro* (2023).

[32] Naveed Ul Mustafa, Yuanchao Xu, Xipeng Shen, and Yan Solihin. 2021. Seeds of SEED: New Security Challenges for Persistent Memory. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 83–88.

[33] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 401–410.

[34] Krerk Piromsopa and Richard J Enbody. 2011. Survey of protections from buffer-overflow attacks. *Engineering Journal* 15, 2 (2011), 31–52.

[35] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[36] Samsung. 2023. *[Webinar] Memory-Semantic SSD™: Samsung's CXL-based SSD for the Memory-Centric Computing Era*. Retrieved Sep 17, 2024 from https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/

[37] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 69–80.

[38] Ajeet Shankar and Rastislav Bodik. 2007. DITTO: Automatic incrementalization of data structure invariant checks (in Java). *ACM SIGPLAN Notices* 42, 6 (2007), 310–319.

[39] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*. IEEE, 574–588.

[40] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[41] PAX Team. 2003. *PaX address space layout randomization*. Retrieved Sep 17, 2024 from https://pax.grsecurity.net/docs/aslr.txt

[42] An-I Wang, Peter L Reiher, Gerald J Popek, and Geoffrey H Kuenning. 2002. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System.. In *USENIX Annual Technical Conference, General Track*. 15–28.

[43] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 478–496.

[44] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 987–1000.

[45] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. 2022. Temporal Exposure Reduction Protection for Persistent Memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 908–924.

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