MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization

Yuanchao Xu
North Carolina State University
Raleigh, North Carolina, USA
yxu47@ncsu.edu

Yan Solihin
University of Central Florida
Orlando, Florida, USA
Yan.Solihin@ucf.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Abstract

This paper proposes a new defensive technique for memory, especially useful for long-living objects on Non-Volatile Memory (NVM), or called Persistent Memory objects (PMOs). The method takes a distinctive perspective, trying to reduce memory exposure time by largely shortening the overhead in attaching and detaching PMOs into the memory space. It does it through a novel idea, embedding page table subtrees inside PMOs. The paper discusses the complexities the technique brings, to permission controls and hardware implementations, and provides solutions. Experimental results show that the new technique reduces memory exposure time by 60% with a 5% time overhead (70% with 10.9% overhead). It allows much more frequent address randomizations (shortening the period from seconds to less than 41.4us), offering significant potential for enhancing memory security.

CCS Concepts • **Security and privacy → Systems security**; *Hardware-based security protocols*; • **Hardware → Non-volatile memory**.

Keywords persistent memory objects; memory exposure reduction; runtime randomization

ACM Reference Format:
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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378492

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378492

1 Introduction

Despite decades of security research, unauthorized memory reads and writes are still problematic to security. Attackers may exploit unauthorized memory writes to cause *memory corruption*, that lead to various attacks (e.g., code-reuse [48, 50, 54], code-injection [46], data-oriented [26] attacks, etc.). Attackers may use unauthorized memory reads to perform *memory disclosure* that can be used to defeat certain security techniques such as address space layout randomization (ASLR). Many techniques have been proposed to mitigate unauthorized memory reads and writes, including Control Flow Integrity (CFI) [2], Code-Pointer Integrity (CPI) [38], and Data Flow Integrity [5, 14]. However, they incur large performance overheads when enabled.

In this paper, we propose MERR, a new approach for reducing memory disclosure and corruption vulnerabilities for data. The key idea is to reduce *memory exposure* by making data accessible only when the program needs it by *attaching* data to the process address space, while making it inaccessible at other times by *detaching* (i.e. removing) it from the address space. Detaching data from address space provides a very strong protection because even virtual memory (VM) implementation vulnerabilities cannot be exploited by the adversary. For example, one aspect that enables Meltdown attack [41] is the failure of the VM implementation in enforcing privilege access check for out-of-order executed instructions.

Several challenges arise when we want to achieve said protection. One challenge is that supporting the primitives of attaching and detaching data requires data to be encapsulated and managed by the Operating System (OS), provided with namespace and permission mechanisms. Currently, an object that fits the attach/detach model includes memory-mapped files, which are mapped to and unmapped from the process address space, allowing process to directly access file data. With the rise of persistent memory, we expect that persistent data structures without file backing will also be commonplace. We will refer to OS-managed data object supporting attach/detach primitives as *persistent memory object* (PMO). In this paper, we focus on a PMO that permanently resides in physical memory, hence no high latency I/O operations are needed to attach it to process address space. To provide the protection, a process only attaches a PMO

when it needs to access it, and detaches as soon as it finishes accessing it.

A key challenge for our strategy of memory exposure reduction is the high performance overheads that come with attaching and detaching a PMO. Traditional memory mapping mechanism is exceedingly expensive. When data with multiple pages is mapped, multiple page table entries (PTEs) must be initialized by the kernel. For each page, a victim page is selected, TLB shootdown is initiated, and page fault occurs on the first access. A TLB shootdown serially interrupts each core which must acknowledge the TLB invalidation, costing thousands of clock cycles per core [22]. After mapping, an access to a recently-mapped page incurs a page fault, that requires the page fault handler to read a page-size region from the file and copy it to the mapped page. Many page faults may occur to populate the memory-mapped region. Much of the costs also apply to unmapping.

To reduce the expense, we exploit the fact that a PMO already resides in the physical memory. Thus, with the right mechanism, page faults are not necessary. To attach a PMO to process address space, we propose to simply initialize PTEs to point to the physical memory where the PMO already resides. However, for a large PMO, initializing many PTEs is still prohibitively expensive. To avoid this cost, we propose embedding PTEs into a PMO, if the PMO is larger than a page. An x86 page table is hierarchical and forms a radix tree. We propose storing a page table subtree in the PMO itself. Thus, attaching a PMO requires initializing only one PTE at the appropriate level to point to the subtree. With this method, only one PTE is initialized and only one TLB shootdown is involved. With these optimizations, we reduce the cost of attaching a PMO such that it becomes feasible to perform them frequently. For example, we can wrap any functions that access the PMO with attach and detach.

However, naively embedding PTEs into a PMO violates process-specific permission semantics in current VM system, because it forces permissions to be PMO-specific regardless of the process that attaches it. In order to allow process-specific permissions for a PMO, we propose a novel process-specific permission matrix hardware support for keeping PMO-wide permission. An access is valid when it is determined to be legal by both the permission matrix and TLB.

While detached, a PMO is not vulnerable to memory disclosure and corruption. However, the adversary can still target the PMO when it is attached. To improve security further, each time we attach a PMO, we can change the virtual address region where the PMO is attached at each attach session. We refer to this scheme as *PMO space layout randomization* (PSLR). PSLR makes it harder for attackers to figure out which address location it must attack, as it frequently changes. The window in which the attacker must successfully probe the system and deploy the attack is limited to the length of time a PMO is attached.

Overall, this paper makes the following contributions:

1. We propose a new approach to reduce memory disclosure/corruption vulnerabilities by **reducing memory exposure time using attachment and detachment of a PMO**.
2. We propose a novel architecture support to make attachment/detachment fast by **embedding a page table subtree into a PMO**. This reduces the number of PTEs to modify and number of page faults to only one.
3. We propose an **architecture support for providing process-specific PMO-wide permission**.
4. We take the advantage of frequent PMO attachment by deploying **PMO Space Layout Randomization** (PSLR) which randomizes the location of a PMO at every attachment session.

We refer to our approach Memory Exposure Reduction and Randomization (MERR). MERR carries unique benefits. The reduction in memory exposure reduces the window of vulnerability in which the adversary can mount an attack. Furthermore, by wrapping only functions that access the PMO with attach and detach, the code attack surface is reduced to just these functions. By reducing the attack surface to much smaller code, it is easier to ensure code security, and the overheads of deploying protection techniques are only incurred for this code base. In addition, deploying randomization at every attach further reduces the window of time available to the adversary. The adversary must, within a single attach/detach session, probe the system and deploy an attack. Finally, the novel page table subtree and permission matrix mechanisms can be thought of as enabling technology that reduces the latency of attach and detach. In this paper, we use it to reduce memory exposure time, but they will likely also open up future uses that rely on it.

Our experiment results using Whisper benchmark suite [45] indicate that on average, MERR only incurs 10.9% slowdown while reducing memory exposure by 70% and randomizing PMO address every 41.4us or less.

The remainder of the paper is structured as follows. Section 2 describes background knowledge. Section 3 motivates the problem tackled in the paper. Section 4 presents our design. Section 5 describes the randomization technique. Section 6.1 presents the evaluation methodology. Section 6.2 discusses our evaluation results and findings. Finally, Section 9 concludes the paper.

2 Background

2.1 Memory Disclosure and Corruption

Despite decades of security research, memory attacks are still problematic to security. Memory attacks may be unauthorized memory writes (*memory corruption*) or reads (*memory disclosure*). The use of memory-unsafe languages such as C/C++ contributes to such a situation. A notorious example memory corruption is buffer overflow, which occurs when

the program misses a bounds check when accessing a buffer, causing corruption in values adjacent to the buffer. Format string vulnerability can also be used to cause memory corruption. Memory corruption may affect the stack region (e.g. stack buffer overflow), the heap (e.g. heap buffer overflow), or other regions. It may lead to various attacks, such as code-reuse [48, 50, 54], code-injection [46] and data-oriented [26] attacks, and so on. Typical mitigation for memory corruption is address randomization.

Memory disclosure is a case where memory content is read or leaked. Various vulnerabilities may lead to this, including format string [51] and buffer overread [59]. Memory disclosure can be used to defeat randomization strategies designed to prevent memory corruption from being exploited. More recently, Spectre/Meltdown [34, 41] show that memory disclosure can result even through speculative instructions.

Many techniques have been proposed to mitigate unauthorized memory reads and writes, including Control Flow Integrity (CFI) [2], Code-Pointer Integrity (CPI) [38], and Data Flow Integrity [5, 14]. However, they incur large performance overhead when all of them are enabled. For example, it was reported that CFI incurs 21% overhead [2] while CPI incurs 8.4% overhead [38] on average. Applying data flow integrity to all the data incurs 50%-100% overhead [14].

Static address randomization defenses, e.g., address space layout randomization (ASLR), can be used as a first line of defense against memory attacks. ASLR works by randomizing the start address of various segments. However, the effectiveness of ASLR lies on the assumption that the layout remains secret. This assumption is often incorrect. Memory disclosure can be used to figure out the location of certain datum, leading to the disclosure of the random offset. As a result, newer work proposed re-randomization every certain interval. Unfortunately, re-randomization involves copying entire memory segments, hence it increases execution time substantially even when performed every 5 seconds [54]. This compromises security as the attacker may succeed within 5 seconds (e.g. meltdown can bypass KASLR to read data successfully within seconds [41]). Therefore, we conclude that memory attacks are still deeply problematic in terms of costs to performance.

MERR differs from prior protection mechanisms by detaching data (or PMO) entirely from process address space when the program does not need to access it. MERR provides stronger protection than merely re-randomizing its location. On top of that, MERR assigns a PMO to different virtual memory address each time a process attaches the PMO, without physically relocating/copying data. Note that MERR only protects data in PMO, so the rest of the address space should still rely on traditional techniques such as ASLR or KASLR.

2.2 Persistent Memory Programming Support

Non-volatile memories (NVMs), such as Intel Optane DC Persistent Memory, provides high capacity at low cost, low idle power, byte-addressability, persistence, and performance closer to DRAM than SSD or disks [4, 32, 37, 39]. For these reasons, they are considered a contender for future main memory fabric.

There are at least two paradigms for using NVM. One uses it as storage to host a file system, the other uses it via a new abstraction where a data structure is wrapped into a *persistent memory object* (PMO), which allows the data structure to be hosted persistently in physical memory without involving a file system. PMOs may combine some features of a file system (naming, permission, durability, and sharing) and some features of data structures (pointer-rich, address space mapping, purely load/store access). In this paper, we assume the latter.

A PMO may be a container for a data structure that lives beyond process termination and system reboots. A PMO requires several properties to be supported: *crash consistency* allows a PMO to remain in a consistent state even when the process that accesses it crashes or system power is lost, *system naming and permission* allows a PMO to be found by processes and the system to manage its use by processes, *attach/detach* primitives where a PMO can be attached to a process address space when needed and detached when not needed, *relocatability* where a PMO can be attached at different parts of a process address space at different times. A PMO may be implemented as pools [29, 60**?** , 61] and given a unique identifier. Pools can be organized as a collection. There is a root object from which all other objects in the pool can be reached. In this paper, we use the term PMO as a general concept, and pool as a specific implementation of a PMO, which may not have all features a PMO should support.

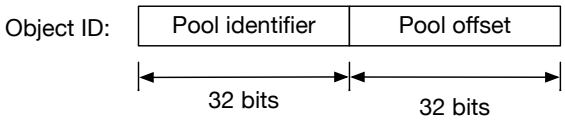

Figure 1. Structure of pool pointer [15, 60, 61]

To support relocatability, each pointer (64-bit) used in a data structure is split into a 32-bit pool ID (ObjetID) concatenated with a 32-bit offset within the pool (Figure 1). To address a pointer, the base address for the ObjectID is added to the offset. PMDK [29] and other prior works [17, 60] have described interfaces for manipulating pools and objects. We adopt the interface proposed by Wang *et al.* [60]. It supports functions for creating pools (analogous to files), objects within pools, support for persisting objects, and failure-safety through durable transactions. Table 1 shows a subset of their interface.

One thing we point out is that *pool_open* will look for a pool of the given name. If it exists and if the calling process has permission to access this pool, this pool is mapped into

Table 1. Pool APIs described in prior work [29, 60].

Function	Description
pool* pool_create (name, size, mode)	Create a pool with the specified size and associate it with a name. The running process is the owner.
pool* pool_open (name, mode)	Reopen a pool using name that was previously created. Permissions will be checked.
pool_close(pool* p)	Close a pool p
OID pool_root(pool* p, size)	Return the root object of the pool p with specific size. The root object is intended for programmers to design as a directory of the contents in the pool.
OID pmalloc (pool* p, size)	Allocate a chunk of persistent data with the given size on pool p and return the ObjectID of the first byte.
pfree(oid)	Free persistent data pointed to by the ObjectID.
void* oid_direct(oid)	Translate an ObjectID to a virtual address. Used when there is no hardware translation.

the process' address space. In our case, a pool is always mapped to a page-aligned contiguous virtual address range large enough to fit the pool.

2.3 Hardware-Supported Address Translation

Table 2. Instructions to support translation in hardware [60]

Instruction	Description
nvld rd, rs1, imm	rd = MEM[Lookup(rs1)+imm]
nvst rs1, rs2, imm	MEME[Lookup(rs2) + imm] = rs1

Translation from a pool pointer to virtual address places a burden on the programmer and incurs significant performance overhead if performed in software. We adopt the architectural design proposed by Wang *et al.*[60] to accelerate ObjectID translation. In order to distinguish pool pointer format from regular pointers, accesses to a pool must be made using special load and store instructions (nvld and nvst) as shown in Table 2. When a pointer is accessed with nvld or nvst, the ObjectID is used to index a table called *persistent object table* (POT); a POT entry specifies the base (virtual) address of the pool. A recently-used subset of POT entries can be cached in a hardware structure called *persistent object lookaside buffer* (POLB). The location pointed to by the pointer is obtained by adding the base address (obtained from the POT/POLB) with the 32-bit offset. Figure 2 shows the brief system design of POLB and POT.

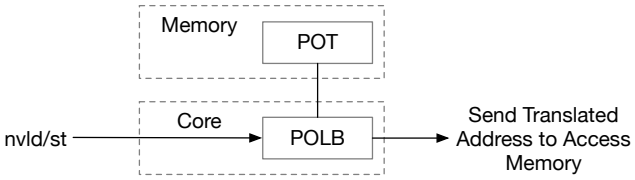

Figure 2. Design of POT and POLB in [60]

3 Premises

This section describes some premises necessary for following the rest of the discussions.

The PMO is a general system abstraction of data, and can be used to wrap any data, residing in volatile memory (DRAM) or persistent memory (NVM). However, supporting attach and detach primitives that require system namespace and permission mechanism make it more useful and applicable for persistent data, such as memory-mapped files or persistent data structures. It is likely that the PMO abstraction will be used to hold data that is small/medium in size (e.g., sub-GB), while large data will still utilize files, considering that the cost per byte of disks/SSD is substantially lower than even NVM and that NVM capacity is limited. A PMO will also tend to be small because it will be data structure-centric, hence data requiring dissimilar data structures is likely to be split into multiple PMOs for easier maintenance. A corollary of the PMO being small/medium is that a process will likely attach multiple, perhaps many, PMOs to its address space during its execution. As a result of multiple PMOs accessed per process, it is unlikely for all the PMOs to be accessed all the time throughout a process execution.

Furthermore, *we hypothesize that due to convenience, a process will attach a PMO in its address space much longer than needed.* It is easy to figure out when to attach a PMO since it precedes access. However, it is less straightforward to determine when to detach a PMO. A process is supposed to detach a PMO after it finishes using it. However, since there is no obvious penalty for detaching late, it is likely that a PMO is attached far longer than needed, evidenced by the fact that in existing persistent memory benchmarks (e.g., WHISPERS [45]), persistent data structures are kept in the process address space for nearly entire executions of the programs.

Therefore, we advocate an automated approach where code analysis is performed to figure out which functions contain code that access PMO data. Once these functions are found, they become candidate to wrap by attach and detach. Nested attach/detach is avoided by wrapping only the outermost function. We introduce a term, *Attached Memory Exposure Time* (AMET), which measures the total length of the time when a PMO is being attached to a process. Reducing AMET is important to reduce the exposure of PMO to

security attacks, but also to reduce the code attack surface to only that which wrapped by attach and detach.

Threat Model Just like any other data structures, data structures in PMO may contain buffers and pointers. Code that access PMO may contain regular known vulnerabilities. So traditional memory vulnerabilities are assumed to exist. We do not seek to protect against specific vulnerabilities. Instead, our focus on making unauthorized reads or writes to data in PMOs difficult. MERR only protects the PMO hence other parts of the process code or data would still need traditional protection against memory disclosure/corruption.

We assume trusted system software, such as the OS, which manages address space isolation between processes. With that, read or write to data to memory region that is not mapped in the page table will not be permitted and will generate segmentation fault exception. Furthermore, we assume that the processor memory management unit (MMU) is implemented correctly, in that it will not allow access of memory that is not mapped to the page table.

While a PMO is attached, the attacker can probe, read or write the data in the PMO. Our PMO permission mechanism allows to mark PMO as non-executable, hence we assume that the OS sets up each PMO as non-executable, hence if the attacker injects code into the PMO, the code cannot be executed.

4 Reducing Memory Exposure

This section describes our proposed techniques to mitigate the vulnerability of PMOs to memory attacks by reducing the attached memory exposure time (AMET) of PMOs.

4.1 Reducing AMET

As we argued earlier, it is likely that when a process attaches multiple PMOs, it will work on a particular PMO for only a small fraction of time. Keeping the PMO attached from the start of the process until process termination unnecessarily exposes the PMO to memory attacks. Thus, our first strategy is to reduce AMET by tightening up the window of time when the PMO is attached.

Reducing AMET improves the security of using PMOs. If an unauthorized memory read or write occurs when a PMO is not detached, the read or write will trigger a segmentation fault, prohibiting the attack from succeeding.

In thinking about how small AMET can be made, one extreme is to wrap each load or store instruction with attach() and detach() system calls. However, system calls incur substantial overheads, due to pipeline flush, mode switch, and cold cache effects. Furthermore, since attach() maps a PMO to the calling process address space, if the PMO spans over multiple pages, multiple page table entries (PTEs) must be initialized by the kernel. For each page, two types of cost are incurred: TLB shootdown and page fault, the latter dominating the cost. A TLB shootdown is initiated, by serially interrupting each core which must acknowledge the TLB invalidation. The cost of TLB shootdown is in the order of thousands of clock cycles per core [22]. Even for a single page, malloc takes 2μs. As the number of pages increases, the latency increases super-exponentially, reaching several milliseconds for 1GB data (256K pages $\times$ 4KB). On top of that, after mapping, an access to recently-mapped page incurs a page fault that requires the page fault handler to read a page size region from the file and copy it to the mapped page. Many page faults must occur to populate the memory mapped region. For a PMO, since it already resides in memory, a page fault does not need to perform any copying other than initializing the PTE to correctly point to the correct page of the PMO with the right permission. Thus, it is clear that we cannot use attach()/detach() too frequently.

Reducing attach/detach frequency could reduce the overhead but enlarge exposure time. Fundamentally, minimizing the cost of PMO attach/detach is the key to this tradeoff.

4.2 Fast and O(1) PMO Attachment

In our model, PMO size must be specified at creation. Furthermore, PMO is always mapped to a contiguous virtual address (VA) space, but may not be contiguous in physical memory. If during execution a PMO runs out of space, a system call to increase its size is required. If the current VA range cannot be extended, the process must detach it and re-attach it at a new, larger, VA range.

As discussed earlier, there are three types of costs associated with a memory map: TLB shootdown, PTE initialization, and page faults. Let us examine these costs, starting with the simplest case. We assume hierarchical page table (PT) organization, similar to one used in x86 systems. Suppose that a PMO fits in a page (i.e. its size is smaller than 4KB). In this case, to attach the PMO to a process address space, we need to simply find an unassigned PTE in the process PT, and initialize it to point to the physical page frame where the PMO is currently residing (Figure 3(a)). The permission in the PTE is initialized to match the process' request for the PMO. The initialization of the PTE requires TLB shootdown to enforce TLB coherence. Subsequently, no page fault will be incurred as the PMO is already in the physical memory and the PTE is valid. Thus, we only incur the costs of TLB shootdown and PTE initialization.

Now let us consider a large PMO, spanning multiple pages, say 256K pages, for a total PMO size of 1GB. In this case, attaching the PMO involves allocating at least one L3 PTE, 512 L2 PTEs, and 256K L2 PTEs, a TLB range flush being initialized. This is clearly prohibitively expensive. Thus, we propose to store a PT subtree in the PMO itself as metadata. The PMO PT subtree is hierarchical just like the process PT, with hierarchy depth depending on the PMO size. If a PMO is 4KB or smaller, it has no subtree. If it is 2MB or smaller, the subtree starts from level 1. If it is larger than 2MB but 1 GB or smaller, the subtree starts from level 2, etc.

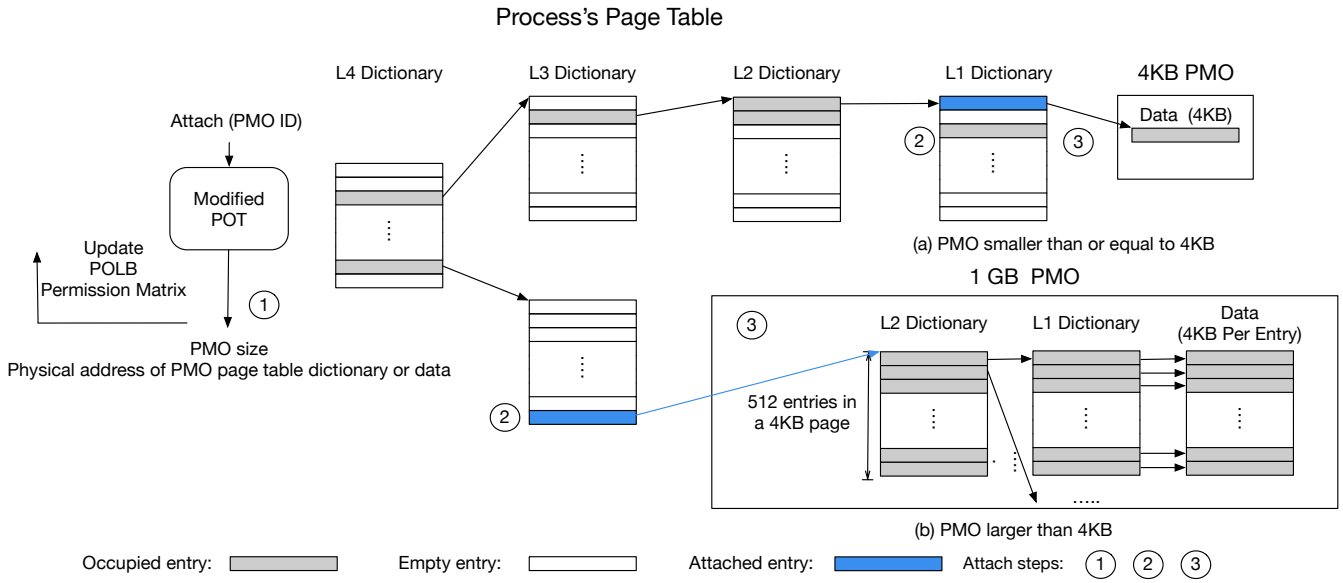

Figure 3. Fast PMO attachment mechanism for a PMO size ≤ 4KB (a), and for PMO size > 4KB (b).

Thus, attaching the example PMO would require initializing only one (parent) L3 PTE, to point to the existing PMO subtree. Figure 3(b) illustrates this case. Hence, only one PTE entry is initialized and one TLB shootdown is involved when attaching a PMO. Consequently, PMO size no longer determines the cost of attaching it. A drawback of this approach is that PMO will occupy a VA range that is one of PT granularity increments, e.g. 4KB, 2MB, 1GB, etc., introducing VA (not PA) fragmentation. A similar optimization was proposed for memory-mapped files [27] and file-only memory [56]. However, there are differences between files and PMOs. A file is viewed uniformly by all sharing processes, whereas a PMO allows a process-specific view. For example, one process may attach a PMO in a read-only mode, while another process may attach the same PMO with read and write mode in its address space. We discuss process-specific view of PMO in Section 4.3.

To manage PMOs, the system manages a structure called persistent object table (POT) (Section 2). A POT entry contains PMO ID, PMO size, physical address (PA) of the PMO PT subtree, permission, etc. Figure 3 illustrates the steps for attaching a PMO. When the system receives an attach() request with a PMO ID, first it enters kernel mode. The PMO ID is then used by hardware MMU to perform POT walk to find the PMO, including its size and the PA of the PMO PT subtree. Second, the system finds an empty PTE at the appropriate level to accommodate the PMO size. For example, if the PMO is 512MB, the next size up in the PT hierarchy is 1GB, hence a level 3 (parent) PTE is needed to point to the PMO PT subtree. If no such entry is found, the system allocates a new page of level 3 PTEs. Third, the empty PTE is initialized to point to the PA of the PMO PT subtree, and its valid bit set. After these three steps, the PMO is accessible without further page faults. The PMO attachment latency thus only takes a constant time, unaffected by PMO size.

When accessing the PMO, the process may miss in the TLB, and PT walk is performed, involving the process PT and PMO PT subtree. The PT walk transitions seamlessly between the PT and the PMO PT subtree. The final VA-to-PA translation is then placed into the TLB. While the PT walk does not distinguish between the process PT and PMO PT subtree, the PMO PT subtree is managed differently. The PMO PT subtree is initialized when the PMO is created. Furthermore, the PMO PT subtree must be persistent and crash atomic, so that the PMO can be recovered and addressed correctly after power failure.

Initializing PTEs in PMO PT subtree takes time, but it is a one-time cost. Once the subtree is constructed, we can reuse it cross runs. It can also be reused by different processes if the process passes permission check.

In contrast to the attach() system call, the detach() system call looks more expensive: It requires shooting down all TLB entries of the PMO. Since the VA of a PMO is consecutive, TLB entires of a PMO can be flushed through a TLB range flush. Meanwhile, unlike attachment, PMO detachment is typically off the critical path. Hence, detach() latency scales up with PMO size, but bounded by the size of the TLB.

With these optimizations, we reduce the cost of attaching a PMO such that it becomes feasible to perform them quite frequently. Programmers can simply use the O(1) attachment-based memory management to improve the security of using PMO with low performance overhead.

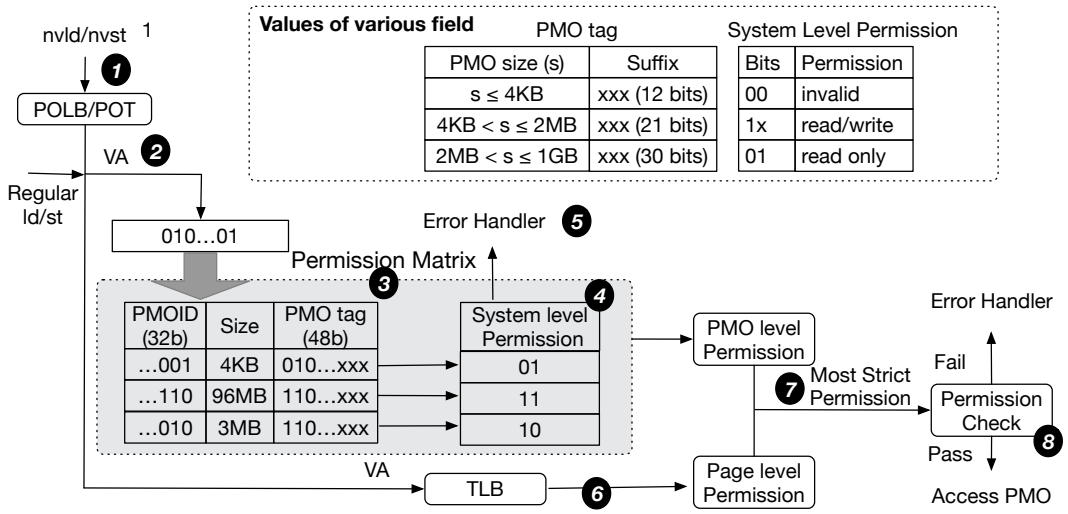

Figure 4. The design of permission matrix

4.3 PMO Access Permission Design

One challenge of embedding PT subtree in a PMO is that permissions of each PTE is PMO-specific, rather than process-specific. While utilizing a PMO PT subtree for attachment is fast, the process must rely on permission bits already set in the PT subtree. This does not allow a process-specific view of a PMO. For example, one process may want to attach a PMO in a read-only mode, while another process may want to attach the same PMO with read/write mode.

In order to allow process-specific permissions for a PMO, we propose a permission matrix structure for setting, maintaining, and validating PMO-wide permission. Each process has its own permission matrix, allowing process-centric view of PMOs. PMO-level permission lets the system to treat the entire PMO as the basic unit to manage rather than 4KB pages, which makes it convenient and cheap to manage PMOs spanning multiple pages. A permission matrix entry will be created when a PMO is attached and will be deleted when the PMO is detached. The permission matrix is a system table for the process, but cached for quick lookup. A memory access is valid when it is determined to be legal by both the permission matrix and PT/TLB. Permission matrix also allows process-specific view of a PMO since each process has its own permission matrix. Permission matrix becomes a part of the process state and is included in the process context switch. Permission matrix does not need to be persistent because upon a crash or power failure, the process lost its state, including attachment of all PMOs it had access to. When the process restarts, it needs to re-attach all PMOs that it wants to access.

Figure 4 illustrates the design and mechanism of the permission matrix. The permission matrix has one entry for each PMO. It contains a 32-bit PMO ID, PMO size, a 48-bit PMO tag, and system level permission. The PMO ID uniquely identifies each PMO. The size encodes how large the PMO is. The PMO tag identifies VA range of a PMO. It consists of a prefix whose length depends on the size of the PMO. For example, in the table above the diagram, a PMO that is up to 4KB has a 12-bit suffix, which means the prefix is $48 - 12 = 36$-bit long. A larger PMO that is up to 2MB has a 21-bit suffix, which means the prefix is $48 - 21 = 27$-bit long. An even larger PMO that is up to 1GB has a 30-bit suffix, which means the prefix is $48 - 30 = 18$-bit long. System-level permission requires kernel privilege to set. The total size of the permission matrix depends on the number of PMOs that can be cached there; it is less than 1KB to cache up to 32 PMOs.

The permission matrix is used in the following way. When a nvld/nvst is executed, the object address involved is translated into VA by POLB/POT (Step 1). Discussion on POLB and POT and how they support PMO relocatability was presented in Section 2. For regular ld/st instruction, the address is already in the form of VA (Step 2). The VA is then compared against all PMO tags up to the length of the PMO tag prefix (Step 3). If a match is found, the access is to an address covered by a PMO PT subtree. Otherwise, the access is not to a PMO; no further steps are needed. If an address matches a PMO tag prefix (suffix is ignored), we check the load/store against the system-level permission (Step 4). A load (or store) is not legal if the system-level permission does not allow read (or write) access, and in this case an exception is triggered (Step 5). If the load/store passes through system-level permissions, the legality of the load/store must still wait until permission at the page level at the TLB occurs (Step 6). If the load/store is legal according to both the permission matrix and the TLB (Step 7), the load/store can commence (Step 8). Otherwise, an exception is raised.

For fast access, the PMO tag prefix checking utilizes a CAM tag array, and only a limited number of PMO entries are cached. The permission matrix can also perform an additional security check. When a VA prefix matches a PMO tag prefix, we can retrieve the PMO ID recorded in the matching entry and compare it against the nvld/nvst's upper 32-bit object ID. If they match, the nvld/nvst is intended to access a particular PMO and accesses the address range of the PMO. If they mismatch, there is something wrong, where it is accessing a particular PMO but with an address range of a different PMO. This situation could occur when a pointer intended to access a particular PMO has been overwritten by the attacker to point to a different PMO.

The PMO level permission check and page permission check are performed in parallel, so the permission matrix latency is hidden. However, nvld/nvst must still go through POT/POLB to generate VA before the VA can be checked against the permission matrix. Hence, an additional clock cycle delay is added to nvld/nvst.

5 PMO Layout Randomization

5.1 Motivation

In the previous section, we have discussed how we can improve the security of PMO by (1) reducing AMET using frequent use of fast O(1) attach() and detach() system call. However, this protections in some cases will still be insufficient. First, even though a PMO may be attached and detached many times, if every time it is attached, it is attached in the same address range, the attacker can construct an attack targeting the address range slowly. Second, data in PMO is reused many times in different parts of a program. The attacker can combine the information obtained in different parts of this program to launch a successful attack. Finally, a PMO has a long life time, even small memory disclosure or information leak per run can be *aggregated* across many runs of the same or different programs.

To avoid such aggregation of knowledge across attach sessions or across runs, we propose PMO Space Layout Randomization (PSLR). PSLR changes the address where a PMO is attached at every attached session. PSLR makes it difficult for attackers to figure out which address location it must attack, as it frequently changes. It provides stronger protection than address space layout randomization (ASLR) which randomizes the start address of segments at the start of program, but generally leaves them unchanged during the process life time. Unlike segments, PMO can be attached and detached, so we exploit this fact to re-randomize PMO address at every attach.

Re-randomization in ASLR is possible, but substantially increases execution time even when performed every 5 seconds [54]. This is due to (1) relocating a segment involves copying a huge chunk of memory from one place to another and rewriting pointers, (2) invalidating all PTEs at the old location and initializing all PTEs at the new location, and (3) many page faults result after re-randomization. Thus, frequent re-randomization in ASLR is not feasible due to the high overheads. With such high overheads, existing re-randomization focuses only at a low frequency fine-grained randomization [54], coarse-grained randomization [2], or permutation [3, 33]. The security level is also limited due to low frequency and small entropy. In contrast, we will discuss that unlike segments, PMO can be moved more easily and cheaply.

5.2 PSLR Design

Our PSLR combines the O(1) attachment with re-randomization. For each attach() system call, the system selects a random empty PTE to attach the target NVM. Traditionally, to find an empty PTE, the PT needs to be locked to avoid races, which may create a critical path. To avoid coarse-grain locking, we maintain three separate free lists to record several random PTEs at different PT hierarchy levels, representing 4KB, 2MB, 1GB regions. Each free list must have a sufficient number of PTEs to choose randomly that are allocated but have not been assigned, meaning that the PTE's ancestors (parent, grandparent, etc.) have all been allocated and assigned to point to the page where this PTE is located, but the PTE itself is not assigned a valid PA yet. Among PTEs in the free list, a random PTE is selected for the PMO to be attached to.

PSLR avoids the high cost of re-randomization. First, pointers in PMO are relocatable. Second, the embedded PT subtree in PMO also allows quick relocation, because none of PTE in the PMO PT subtree needs to be modified. All that needs to be modified is the parent PTE. Hence, relocation of a PMO at attachment time incurs a constant cost involving only one PTE initialization. Finally, page faults are also averted.

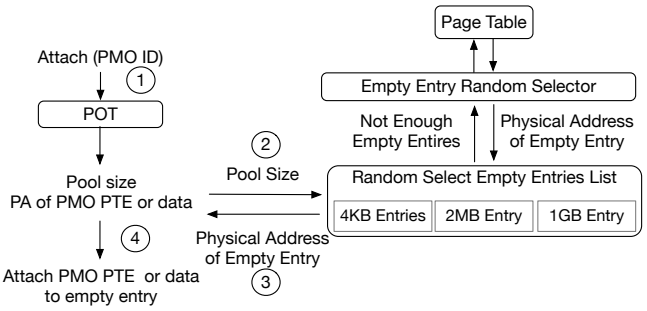

Figure 5. The design of PSLR with attach system calls

Figure 5 shows PSLR design. First, the system will use the PMO ID in the attach system call parameter to find the pool in POT. The system retrieves the PMO size and the physical address (PA) of this PMO. Second, the system uses the size to check the list corresponding to the correct size bin (4KB, 2MB, or 1GB), and retrieves a random PTE from the list. The PTE is then initialized with the PA of the target PMO PT

subtree. Offline, the random selector pre-selects a random PTE for following attach requests. It relies on the random number generator from Intel AES-NI instructions. If the list does not have a sufficient number of PTEs, the selector locks the page table to find the empty PTEs and add them into the list.

6 Evaluation

6.1 Methodology

Processor and system environment. We implement the APIs as a kernel module in Linux OS. In the kernel mode, the system maintain pre-allocated PMO lists with different size bins: 4KB, 2MB, and 1GB. To serve an attach() system call, a pre-allocated page is randomly selected. Permission check is performed in kernel mode. Upon a detach() system call, PMO is unmapped from a process PT but we still maintain this PMO at the system level.

Table 3. Experimental Setting

Processor	Intel(R) Xeon(R) Silver 4114 CPU 10 Cores @2.20GHz
Cache	L1D cache 8-ways 32KB L1i cache: 8-ways 32KB L2 cache: 16-ways 1024KB
Memory	128GB DDR4 (2666 MHz)
TLB	L1 data TLB: 4KB pages, 4-way, 64 entries L2 4KB/2MB pages, 6-way, 1536 entries
OS	Linux kernel version: 4.4.0-145-generic
Measured	POLB Translation: 0.5ns; PM Check: 1ns; TLB invalidation: 130ns; TLB miss: 11.4ns; Cache miss: 16.7ns

To evaluate the performance of our design, we use a system with DRAM as main memory, shown in Table 3. The PMOs are implemented as memory-mapped region in DRAM. We execute the default workloads to get the baseline performance. Then we insert attach() and detach() into the code, and execute the workload to measure the overhead from attach() and detach(). After that, we re-execute the workloads again with pin [43] to record the memory traces. We use these traces to estimate the overhead from architectural designs.

To estimate the overhead of attach() and detach() system calls, we use a combination of things. We create a new system call and place it where attach and detach would have been located. In the system call, we first check whether this PMO is already attached in this process within mutex protection, then we perform a copy_from_user to copy parameters of system call from user space to kernel space, kmalloc involving a single page, virt_to_phys for physical address, remap_pfn_range to remap a kernel page to the user space including PTE initialization, flush_tlb_range to TLB shootdown when detach(). Modification to the PTE is also protected using mutex lock. Thus, even though we did not fully implement the functionality of attach/detach, we model its overheads, including PTE initialization, TLB shootdown, system call, mode switch, the ensuing TLB miss, cache miss, and page fault. Because the entire embedding page table is attached, there is no subsequent page faults when the program is accessing the PMO data. We use a similar approach to model the detach() system call performance.

To model the overhead of permission matrix check (that incurs additional latency beyond the TLB), we add a constant 2ns latency to every ld/st to PMOs. This is likely too high because in an out-of-order processor, not all such latencies are exposed. However, in this paper, we are more interested in estimating the upperbound performance overheads instead of lowerbound or averages.

To account for additional TLB misses and cache misses that come from PSLR, we use Intel Pin instrumentation tool [43] to produce memory traces that are then modified and replayed on a TLB simulator and a cache simulator to obtain each benchmark's new TLB miss rate and new cache rate. Each additional TLB miss or cache miss is then multiplied by the average cost of TLB misses or cache misses and added to the execution time.

Workloads. To demonstrate our techniques on real world applications, we use six benchmarks (tpcc, ycsb, echo, hashmap, ctree, redis) from Whisper benchmark suite [45], which were derived from real world applications. The suite was built assuming the architecture has a mixture of persistent and volatile memory. In Whisper, a program usually places operations into an epoch, and multiple epochs execute in a transaction. It sets a ratio of update/read operation for each transaction or directly use insert operations within a transaction.

In our experiment, we run WHISPER benchmarks for 100k transactions or operations in a 2GB PMO. Table 4 shows a brief description of Whisper benchmarks; more details can be found in [45].

Table 4. WHISPER Benchmarks [45].

Benchmark	Description
Echo	echo test, 100k transactions in total
YCSB	YCSB like test, 80% writes, 100k transactions in total
TPCC	TPC-C like test, 80% writes, 100k transactions in total
C-tree	100K insert operations
Hashmap	100K insert operations
Redis	redis server/ lru-test, 1 million gets/puts

6.2 Experimental Results

This section first reports the overhead of MERR in a spectrum of attach/detach frequencies, and the corresponding reduction of memory exposure time, and then provides the analysis of the benefits MERR brings to memory security.

6.2.1 Memory Exposure and Overheads

This part uses four metrics: 1) AMET: the attached memory exposure time as defined earlier; 2) Attached Memory Exposure Rate (AMER): AMET over the total execution time; 3) Memory Exposure Window (MEW): the length of an attach/detach session; 4) Average PLSR period length: the average re-randomization period length, equal to the average time between two adjacent attach() calls.

In our experiments, for evaluation purpose, we adjust the insertions of attach/detach calls in the programs such that the programs exhibit a spectrum of AMER values. We first report the detailed measurements when the AMERs of the program executions are about 30%, and then report the results in other AMER settings.

Table 5. Measurements of MERR when AMER is about 30%. The overhead section reports the overhead percentage from each of the sources and the total.

Benchmark		Echo	YCSB	TPCC	CTree	HM	Redis	Avg
AMER (%)		30.1	30.6	31.6	29.2	29.9	30.0	30.2
PSLR Period (us)		31.7	29.8	19.3	59.4	36.6	71.5	41.4
AMET (us)	MIN	5.1	2.7	3.0	3.4	3.4	3.1	3.45
	AVG	9.6	9.9	5.7	12.9	11.7	12.1	10.3
	MAX	10.4	11.8	8.5	27.3	13.6	19.7	15.2
Overhead								
Attach (%)		6.3	6.7	10.4	3.4	5.5	2.8	5.8
Detach (%)		5.3	5.7	8.8	2.8	4.6	2.4	4.9
POLB (%)		0.011	0.005	0.018	0.004	0.007	0.001	0.008
Permission Matrix (%)		0.019	0.005	0.024	0.006	0.011	0.001	0.011
TLB misses (%)		0.036	0.038	0.133	0.019	0.031	0.017	0.024
Cache misses (%)		0.063	0.067	0.259	0.034	0.055	0.031	0.085
Total (%)		11.72	12.52	19.63	6.26	10.2	5.25	10.9

Table 5 shows the observations when AMER is around 30%. As the programs by default keep PMOs open throughout their executions, a 30% AMER means an around 70% reduction of the memory exposure time. The average memory exposure window (MEW) is only 10.3us. The average re-randomization period length is 19.3us to 71.5us, and the average time overhead is 10.9%. Putting the results into perspective, while not directly comparable in goal and the scope of address space protection, existing ASLR re-randomization was reported to add 50% overhead if re-randomization occurs every 1 second [25]. Compared to ASLR re-randomization, PSLR re-randomization is 24, 154× more efficient, while offering the additional protection of reducing memory exposure. The *overhead* section in Table 5 reports the detailed breakdown of the time overhead, showing the overhead from each of the sources. Attach and detach dominate the overhead. The NVM pointer address translation (POLB) and the operations on the permission matrix weigh no more than 0.04%, the use of re-randomization incurs some extra TLB and cache misses, but the total performance penalty from them is no more than 0.5%.

The effectiveness of PSLR is attributed to the removal of most page faults, PTE initializations, and TLB shootdowns,

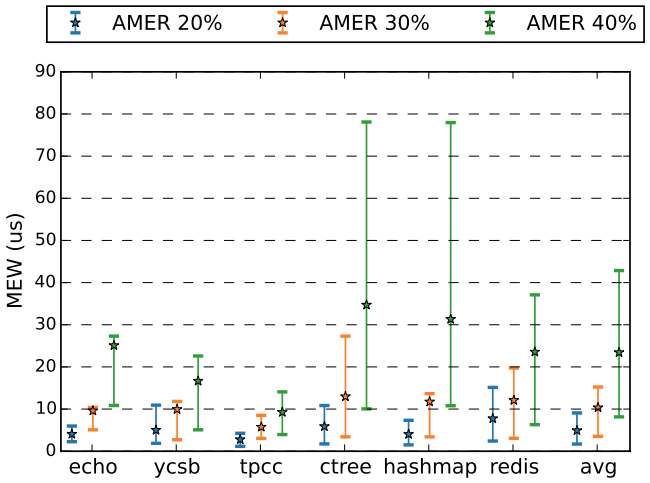

Figure 6. MEW of MERR at three different AMER values.

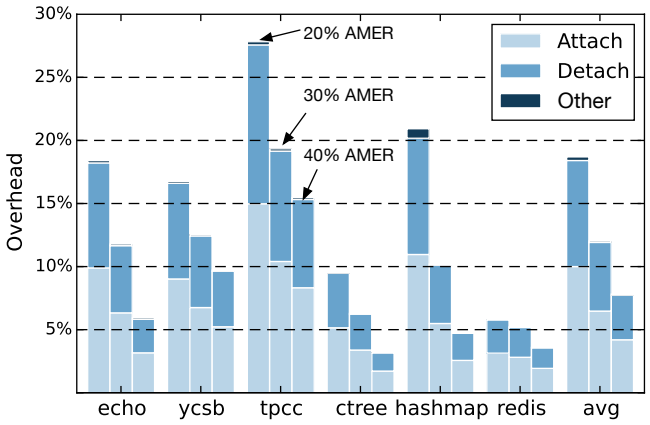

Figure 7. Overhead of MERR at three different AMER values.

as well as PMO relocatability which removes the need to rewrite pointers when a PMO is moved.

The variations of overhead across the benchmarks are largely caused by the differences in the density of PMO accesses in the programs. A denser distribution entail the need for more attach/detach sessions to get the same AMER. TPCC, for instance, has a 14.2X higher PMO access density than Redis has, which leads to a correspondingly higher density of attach/detach calls, and hence the larger time overhead.

Figures 6 and 7 report the MEW and overhead of MERR on the benchmarks when we vary the frequency of attach and detach calls such that the AMER of the benchmarks are at about 20%, 30%, and 40% level. In the overhead breakdown in Figure 7, we use "Other" to represent the total overhead of all the sources besides attach and detach. The results show that at AMER=20%, MERR reduces average MEW to 5us while incurring about 18% average overhead; the numbers change

to 10us and 10.9% for AMER=30%, and 12us and 7.5% for AMER=40%.

6.2.2 Security Analysis

The reduction of AMET provides a high-level memory protection as while detached, a PMO is not in the process address space, and hence its data cannot be accessed at all (read, written, or executed) by any instructions, even speculative ones. One implementation vulnerability that enables Spectre/Meltdown-style attacks is memory access by speculative instruction which is allowed to proceed and alter the cache state. With MERR, even such vulnerability is protected against because there is no valid PTE entry that maps the virtual address that is being accessed to a valid physical address. In comparison, if the PMO is always attached at the same virtual address across all attach sessions, then the adversary can aggregate these sessions and perform various steps of the attack across sessions. MERR deploys PSLR to randomize the virtual address at which a PMO is attached, at each attach session. In this case, to succeed, the adversary would need to perform the attack in one attach session. Recent exploits require several seconds to succeed [54]. Meltdown attack still needs hundreds of microseconds to test one possible randomized layout [41]. Our PSLR re-randomizes the layout more than 24154 times per second, and each attach session lasts for less than 20 us, creating an attack window that is very difficult to exploit by currently known attacks.

Table 6 enumerates vulnerabilities that we protect against, divided into two stages (attached or detached). All of the listed attacks require some access to PMO data which is not possible during the time PMO is detached, which in our experiments is 70% of the time. We discuss each of the vulnerabilities next.

Table 6. Security of the PMO under MERR protection. X for prevented completely, H for hindered.

Attacks	Stage	
	PMO detached	PMO attached
Meltdown [41]	X	H
Spectre v1, v1.1, v1.2 [34]	X	H
Spectre v2, v4, v5 [36, 44]	X	H

Meltdown Meltdown requires three steps to succeed in reading secret data: probing to read secret data address, transmitting the secret, receiving the secret. The probing step is when the adversary probes a virtual address to read secret data. While detached, the probing step will fail as no valid PTE exists, hence the program will receive a page fault. Because both the user space and the kernel space cannot read this data due to hardware valid bit protection, this data is fully inaccessible. With the page fault, the probed virtual address of secret data cannot be put into the cache or the register. The adversary may use error handling to fork another process for the next probe. But the next probe will also generate page fault as long as the PMO is detached. Such a page fault can be used to raise an alarm to alert user or developer of potential ongoing attacks or bugs.

Spectre Spectre attack relies on mistraining the branch prediction including branch target buffers and return address stacks, and leaves architectural information of mispredicted branch to reveal secret data. MERR does not protect against mistraining. In the next step, Spectre also probes a virtual address to read data. This probe is not permitted when a PMO is detached, but may not necessarily raise an exception if the instruction ends up flushed from the pipeline. Another difference is that the Spectre has lower chance to launch a successful attack than Meltdown due to the time needed to mistrain the branch predictor.

7 Discussions

In our experiments, we varied the frequency of attach/detach calls to evaluate the sensitivity. In actual programming, their insertions could be determined by programmers or compilers; details are outside of the scope of this paper.

To support multi-threading, each PMO in each process will have an attached counter initialized as 0 when a PMO is opened. This data is maintained as metadata in each process. For every attach or detach, the operating system first checks this bit via mutex, increasing or decreasing the counter by one for attach and detach respectively. At an attach call, if the counter is larger than 0, the counter increases but other operations of the call are skipped. At a detach call, if the counter is greater than one, the counter decreases by one, and other operations of the call are skipped.

8 Related Work

Persistent Memory. In this paper, we discussed primitives (attach and detach) for PMOs and architecture support for them. There is a rich set of papers in literature covering other aspects of persistent memory, including but not limited to, memory-mapped files [17, 57], file system [18, 20, 64, 65], physical organization [7, 8], persistency models [6, 18, 35, 47, 53, 55, 57], logging [52], checkpointing [21], memory encryption [9–11, 16], and GPU [40].

Memory Protection/Isolation. Software-fault isolation techniques (SFI) [49, 58] create a separate protected memory region by instrumention at every memory access instruction. This ensures that the instrumented instruction can only access the designated memory segment. SFIs incurs large overhead and can be bypassed through Meltdown and Spectre. ISboxing [19] separates address space to allow untrusted code to access only 32-bit address space. The available address space reduction could limit practical usage of NVM. Another way to protect data is Data Flow Integrity (DFI) [5, 14]. DFI static analysis creates data-flow and enforces the data flow

during runtime by instrumenting memory access instructions. However, it incurs 50–100% overhead if it is applied to all data. Jang et al. [30] propose to provide a heterogeneous isolated execution. Another work that tries to hinder the probe step of attacks [12] transforms the program whenever it detects probes. It focuses on only code reuse attacks.

In the hardware aspect, Frassetto *et al.* [23] try to provide in-process memory isolation; Spectre and Meltdown remain threats. CHERI [62] introduces two stages to provide fine-grained memory isolation. But the switching overhead is high and it relies on intensive static analysis. Intel TSX groups several instructions into an atomic transaction. Exception would be raised at an invalid memory access. But Meltdown can completely surpass this protection [31]. Intel CPU implements a variety of new memory protection, including Memory Protection Extension (MPX) and Memory Protection Keys (MPK) [28]. MPX is to provide hardware-assisted checks to avoid buffer overflow. The programmers can specify bounds using dedicated registers which can be checked with newly introduced instructions. But the library code can still access secret data. Using MPK to protect CPI will incur 12.43% average overhead [23]. The technique proposed in this work is complementary to the prior work, in that it shortens the exposure time of memory to possible attacks.

Table 7. Randomization Techniques Comparison. DP for data pointers, D for data.

Methods	Scope		Entropy (bits)	Runtime Randomization Freq	Avg. Overhead
	DP	D			
64-bits Pax [1]	✓	✓	29-30	No	3.6%
TASR [13]	✓	×	29-30	At I/O Only	30-40%
Runtime ASLR [42]	✓	×	28-48	At fock() Only	0.5%
Shuffler [63]	✓	×	27	Fix Interval, 50ms	14.9%
Morpheus [24]	✓	×	60	Fix Interval, 10ms	4.4%
Enhance ALSR [25]	✓	✓	18-36	Fix Interval, 5s	10%
Ours	✓	✓	18-36	Programmer inserted & Runtime, 41.4us	10.6%

Randomization. Randomization and runtime re-randomization try to hinder attacks by randomizing the code and data. As shown in Table 7, ASLR [1] serves as the first-level defence, but can be easily bypassed as a static one-time randomization. TASR [13] and Runtime ASLR [42] improve ASLR by providing re-randomization at sensitive system calls. Shuffler [63] and Morpheus [24] further augment ASLR by reducing overhead and adding encryption. Shuffler re-randomize the code, code pointers, and data pointers at a 50 ms period with 14.9% overhead. Morpheus re-randomize the code pointers and data pointers at a 10 ms period with 5% overhead. Those methods do not support data re-randomization. Enhanced ASLR [25] provide both data and data pointers runtime randomization but suffer large overhead, 50% overhead at a randomization every second and 10% overhead every five seconds. Our method provides both data and data pointers runtime randomization with only 10% overhead at a frequency of every 41.4us.

9 Conclusion

This paper has proposed MERR, a new way to enhance memory protection with high efficiency, especially useful for NVM. Complementary to existing approaches, this new method takes a unique perspective, reducing memory exposure time by enabling fast attach and detach of NVRegions by embedding embedding page table into a PMO, coupled with a fast O(1) PMO attach mechanism and other techniques. We demonstrate the use of the technique for enhancing the frequency of address randomization, and describes the enabled randomization technique named PSLR. PSLR perform randomization at every PMO attach session. Experiments show that MERR can reduce memory exposure time by 60% with a 5% overhead (70% with 10.9% overhead). The randomization period is shortened from seconds in prior work to less than 41.4us, offering significant potential benefits for enhancing memory security.

Acknowledgements

We thank all the anonymous reviewers whose feedback is helpful for improving the final version of the paper. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609, CNS-1717425, CCF-1703487, and Office of Naval Research (ONR) under grant No. N00014-20-1-2750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] PaX Team. 2003. [n.d.]. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[2] Martn Abadi and Mihai Budiu. 2005. lfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*.

[3] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: thwarting DOP attacks with runtime stack layout randomization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 26–36.

[4] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[5] 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.

[6] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique. In *Proc. of the International Symposium on Computer Architecture*.

[7] A. Awad, S. Blagodurov, and Y. Solihin. 2015. Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems. In *Proc. of the International Symposium on Performance Analysis of Systems and Software*.

[8] A. Awad, S. Blagodurov, and Y. Solihin. 2016. Write-Aware Management of NVM-based Memory Extensions. In *Proc. of the International*

Conference on Supercomputing.

[9] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. In *Proc. of the International Symposium on Architecture Support for Programming Language and Operating Systems*.

[10] A. Awad, Y. Wang, D. Shands, and Y. Solihin. 2017. ObfusMem: a Low-Overhead Access Obfuscation for Trusted Memories. In *Proc. of the International Symposium on Computer Architecture*.

[11] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-NVM: persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 104–115.

[12] Koustubha Bhat, Erik Van Der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2019. ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 545–558.

[13] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[14] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 147–160.

[15] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 191–203.

[16] S. Chhabra and Y. Solihin. 2011. i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption. In *Proc. of the International Symposium on Computer Architecture*.

[17] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[18] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.

[19] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 555–566.

[20] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*.

[21] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*.

[22] Bogdan F. Romanescu, Alvin Lebeck, Daniel Sorin, and Alecia Bracy. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. 1–12. https://doi.org/10.1109/HPCA.2010.5416643

[23] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. {IMIX}: In-Process Memory Isolation EXtension. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 83–97.

[24] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 469–484.

[25] C. Giuffrida, A. Kuijsten, and A.S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*.

[26] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[27] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. 2015. Unified address translation for memory-mapped SSDs with FlashMap. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 580–591.

[28] Intel. [n.d.]. Intel 64 and IA-32 architectures software developerâĂŹs manual, combined volumes 3A, 3B, and 3C: System programming guide.

[29] Andy Rudoff Intel. [n.d.]. Persistent Memory Programming. http://pmem.io/.

[30] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 455–468.

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 380–392.

[32] Takayuki Kawahara, Riichiro Takemura, Katsuya Miura, Jun Hayakawa, Shoji Ikeda, Y Lee, Ryutaro Sasaki, Yasushi Goto, Kenchi Ito, Toshiyasu Meguro, et al. 2007. 2Mb spin-transfer torque RAM (SPRAM) with bit-by-bit bidirectional current write and parallelizing-direction current read. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 480–617.

[33] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 339–348.

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[35] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. *ACM SIGPLAN Notices* 51, 4 (2016), 399–411.

[36] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.

[37] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 256–267.

[38] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer integrity. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 147–163.

[39] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

[40] Zhen Lin, Mohammad Alshboul, Yan Solihin, and Huiyang Zhou. 2019. Exploring Memory Persistency Models for GPUs. In *Proc of International Conference on Parallel Architectures and Compilation Techniques*.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.

[42] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization.. In *NDSS*.

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[44] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2109–2122.

[45] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 135–148.

[46] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[47] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 265–276.

[48] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[49] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).

[50] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and communications security*. New York,, 552–561.

[51] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David A Wagner. 2001. Detecting format string vulnerabilities with type qualifiers.. In *USENIX Security Symposium*. 201–220.

[52] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 178–190.

[53] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the long latency of persist barriers using speculative execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 175–186.

[54] 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.

[55] Yan Solihin. 2019. Persistent Memory: Abstractions, Abstractions, and Abstractions. *IEEE Micro* 39, 1 (2019), 65–66.

[56] Michael Swift. 2017. Towards O(1) Memory Proceedings of the Workshop on Hot Topics in Operating Systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*.

[57] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[58] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1994. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, Vol. 27. ACM, 203–216.

[59] Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. 2015. Risk assessment of buffer" Heartbleed" over-read vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 555–562.

[60] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. 2017. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 800–812.

[61] Tiancong Wang, Sakthikumaran Sambasivam, and James Tuck. 2018. Hardware supported permission checks on persistent objects for performance and programmability. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 466–478.

[62] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[63] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 367–382.

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

[65] 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*.