Seeds of SEED: New Security Challenges for Persistent Memory

Naveed Ul Mustafa
Department of Computer Science
University of Central Florida
Orlando, FL, USA
0000-0002-0650-3464

Yuanchao Xu, Xipeng Shen
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
0000-0003-4165-9138, 0000-0003-3599-8010

Yan Solihin
Department of Computer Science
University of Central Florida
Orlando, FL, USA
0000-0002-8863-941X

2021 International Symposium on Secure and Private Execution Environment Design (SEED) | 978-1-6654-2025-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEED51797.2021.00020

***Abstract*—Persistent Memeory Object (PMO) is a general system abstraction for holding persistent data in persistent main memory, managed by an operating system. PMO programming model breaks inter-process isolation as it results in sharing of persistent data between two processes as they alternatively access the same PMO. The uncoordinated data-access opens a new avenue for cross-run and cross-process security attacks.**

In this paper, we discuss threat vulnerabilities that are either new or increased in intensity under PMO programming model. We also discuss security implications of using the PMO, highlighting sample PMO-based attacks and potential strategies to defend against them.

***Index Terms*—Persistent memory objects, Security attacks, PMO vulnerability**

I. Introduction

The release of DIMM-compatible Intel Optane DC Persistent Memory [1] in 2018 marked the beginning of the incorporation of Persistent Memory (PM) into main memory. Compared to DRAM, PM provides higher density, better scaling prospect, non-volatility, and lower static power consumption, while providing byte addressability and access latencies that are not much slower. Consequently, PM blurs the boundary of memory and storage.

There are different ways PM can be viewed as it is integrated into the computer system: as fast medium for hosting the file system or hosting memory-mapped files, as a large-capacity memory alternative to DRAM, or as persistent main memory. While file systems can benefit from PM (e.g., PMFS [5], BPFS [6], and NOVA [7]), the file system software overheads, or the need to reconcile virtual memory and file system semantics present formidable challenges. In this paper, we limit our discussion to the persistent main memory case, where PM is used to host persistent data structures encapsulated in *objects* that are managed by the OS. These persistent memory objects (PMO) were proposed initially in [4].

The security threats affecting PM fabric has been explored, for example, PM encryption was proposed to address data remanence [8], [12]–[14], and mechanisms to prevent early wearout from repeated writes by malicious programs were proposed [15]. However, said studies address security threats from the use of PM as main memory fabric, but do not address the PMO model, i.e. using PM as system objects hosting persistent data. Only two recent studies (e.g., [4], [16]) look into reducing exposure window of PMOs and PMO layout randomization. However, they did not analyze what threats were possible and under what situations the protection could be effective.

This paper discusses threat models and vulnerabilities that are either new or increased in intensity from the use of PMOs. For example, we discuss that the use of PMO may break inter-process isolation guarantee that is central to OS security protection through address spaces. It discusses security implications of using the PMO, presents how threats are affected by the underlying assumptions and the programming model. It highlights sample attacks made possible by the vulnerabilities and discusses potential strategies to defend against them.

The rest of the paper is organized as follows. Section II provides background on Persistent Memory Objects (PMOs): programming model and differences from those of files or DRAM. Section III discusses related work and Section IV presents the assumed threat models. Section V presents a discussion on merits and demerits of different pointer types from PMO security perspective. Section VI presents sample PMO-based control and non-control data attacks, while Section VII qualitatively analyzes defense mechanisms and briefly hints on strategies to detect attacks. Section VIII concludes the paper.

II. Background

A. Persistent Memory Object (PMO)

PMO is a general system abstraction for holding persistent data managed by operating system (OS) [4]. PMO data is not backed by files, and may permanently reside in physical memory. Data in a PMO is held in regular data structures, hence may contain complex data types and pointers, and is accessible directly with load and store instructions, unlike files[1] The OS may provide file system-like namespace and permission settings to PMOs so that data in a PMO can be reusable across process lifetimes and basic access control can be provided.

This work is supported in part by ONR through award N00014-20-1-2750 and NSF through award CNS-1717425.

[1]While some files may contain serialized objects and hence pointers, accesses require system calls and serialization/deserialization.

978-1-6654-2025-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SEED51797.2021.00020

Two key primitives for a PMO are *attach()* and *detach()* system calls [4]. As PMO already resides in physical memory, and its data is already in data structure form, for a process to work on PMO data, it calls *attach()* system call to map the PMO into its address space. Once attached, the process can access it with regular loads/stores, without involving the OS. Likewise, *detach()* unmaps the PMO from the address space, making it inaccessible. After detached, any laod/store to the address region where the PMO used to map result in protection faults.

Just like a file, a PMO may outlast process lifetime, it is conceivable that it will be attached and accessed by multiple processes at different times (or simultaneously read). Similarly, a single process may attach and access multiple PMOs simultaneously. When multiple processes alternatingly access a file, a process may make changes to the file that affect other processes. The same can occur to a PMO accessed by multiple processes. However, a PMO is mapped directly to the address space, hence a change to PMO data by one process directly affects the address space of another process.

B. Security Implications of PM vs DRAM data

There has been a lack of systematic studies on the security implication of PM-resident data in active use. A plausible reason is the perception that existing memory protection designed for DRAM is enough for PM; the whitepaper [17] published by the Storage Networking Industry Association (SNIA) is often taken as the evidence. The whitepaper [17] stated that "*memory protection practices for DRAM apply to persistent memory*". While the statement is true, it should not be interpreted as existing memory protection is sufficient for PM.

In fact, PM poses some principled differences for protection in comparison to DRAM in several ways:

1) **PMO corruption is persistent**. Since PMO keeps persistent data, any data corruption or bugs (dangling pointers, memory leak, etc.) are also persistent.
2) **Recovering from data corruption is challenging**. The effect of corrupted volatile data structures on process behavior can be erased by process relaunch. In contrast, process relaunch does not lead to persistent data recovery, and may cause recurrence of incorrect process behavior.
3) **PMO corruption is transmissible between runs**. Due to PMO data being long lived, its content and structure are reused across runs of the same application, and even across different applications. Data corruption caused by one run directly affects the security of other runs or even unrelated applications.
4) **Attackers can figure out the target locations incrementally**. Unlike DRAM data, the attacker can slowly and incrementally figure out the target locations of data to corrupt over a long period of time across different runs.

Overall, the *cross-run/process security vulnerabilities* are new due to the PMO model. When a PMO can be shared by multiple processes, fundamentally this breaks the inter-process isolation that is the staple security feature of address space isolation of the OS. Hence, memory protection practices common for DRAM require substantial boosting for PMOs.

C. Security Implications of PM vs Files

PMOs are expected to hold data structures, making them pointer rich. On the contrary, files are normally used to hold data (but can also be used to hold data structures, though less common). Since a file contains no pointers, cross-process attacks are much harder to carry out. Pointers are attractive targets for attacks, which makes PM protection more important.

Also, data placed in a PMO-resident data structure is more tightly coupled with execution flow of a process as it can be accessed with regular load/store instructions. Even non-pointer data in PMOs is more likely to be directly used to determine program control flow, making them attractive attack targets. In contrast, data from files is first de-serialized and placed in data structures before used in a process' execution flow.

Finally, unlike PMOs, file data is managed directly by the OS, any access (read/write) requires system calls, and the OS can perform security checks when serving the system calls. In contrast, PMO data can be manipulated directly by loads/stores transparent to the OS.

The combination of longevity, direct byte-addressability and uncoordinated shared access distinguishes PMOs from both DRAM and traditional storage for memory protection, in terms of vulnerability, consequences of security breaches, as well as opportunities for novel solutions.

III. Previous work

Most prior studies focused on the security vulnerabilities of PM fabric itself, rather than the PMO model. To address data remanence due to non-volatility, PM encryption was proposed, e.g. [8], [12], [13]. The limited write endurance of PM may lead to early wear out if the attacker is allowed to write to them excessively. Hence, preventing redundant writes [15] and wear leveling are critical.

However, PM vulnerabilities go beyond just the fabric itself; hosting persistent data as in the PMO model, introduces new vulnerabilities while PM data is in active use. Memory Exposure Reduction and Randomization (MERR) [4] protects PMOs by reducing their exposure window (by attaching only when needed for access and detaching afterward) and hence the attack surface. They proposed splitting the page table to accelerate attach, and PMO Space Layout Randomization (PSLR), where the PMO is mapped to a different randomized location at each attach. Another recent work [16] focuses on mapping PMOs into separate domains, in order to leverage domain protection such as Intel Memory Protection Key (MPK). The intent was to restrict accesses to PMOs only to threads that access them.

Both works [4], [16] seek to make unauthorized accesses to PMOs difficult *for the process accessing the PMO*. However, However, as discussed in Section VI, a vulnerable process with legit access permissions can be used by an attacker to launch a cross-process attack on a victim via shared PMO (i.e., accessed

by two processes at different times). In doing so, PMO itself becomes a new security vulnerability that can be exploited by an attacker irrespective of other known vulnerabilities (e.g., buffer/integer overflow). This paper presents sample PMO-based attacks and also briefly hints on mechanisms to detect such attacks before they are activated (Section VI and VII).

IV. Threat Model

We consider a threat model where two or more processes share a PMO, attaching it at different times. One of the process has no known memory safety vulnerabilities and is the process that an adversary would like to attack. The other process has memory safety vulnerabilities that the adversary can exploit. We will refer to them as the *victim* and *payload* process. The goal of an adversary is to use the payload process in order to compromise the victim process. We assume the adversary knows the fact that a PMO is shared by these two processes, and knows the data structure and layout of the PMO. This threat model is different from one assumed in [4], where they assume a single process that is both the victim and payload.

We assume data structures in PMOs may contain buffers and pointers and the payload process code may have regular known vulnerabilities (e.g., buffer-overflow, integer overflow, format string, etc.). We assume a trusted system software, such as the OS, which manages address space isolation between processes. PMOs are also managed by OS which applies permission checking while granting access to a PMO. This implies that access to a detached PMO is not permitted and results in segmentation fault. However, a process can read and write a legally attached PMO.

V. Pointer Classification

A PMO may hold a pointer-rich data structure, and the types of pointers it holds affects their security implications. Pointers can be categorized based on their direction (from volatile to persistent memory and vice versa) and type of address they hold (absolute or relative).

A. Direction-based pointer classification

Under the PMO programming model, persistent data resides in OS-managed PMOs hosted in PM while non-persistent data is placed in regular data structures in volatile memory (VM) that will be cleared out upon process termination. Accordingly, three pointer types are possible: VM2PM, PM2VM, and PM2PM (i.e., intra and inter-PMO pointers).

1) VM2PM pointers: A VM2PM pointer is necessary for normal operation of a PMO, for example a stack pointer that points to a root pointer in a PMO. VM2PM pointers are destroyed at process termination. The programmer needs to ensure that a VM2PM pointer is dereferenced only when the corresponding PMO is still attached to the process. Otherwise, a segmentation fault is incurred. Furthermore, we assume that only one process can attach a PMO at a given time. Hence, dereferencing a VM2PM pointer by an unauthorized process also results in segmentation fault.

2) PM2VM Pointers: Though in principle PM2VM pointers can be created, they should not be permitted because similar to a file, a PMO may be attached by multiple processes at different times, and may outlast a process lifetime, hence PM2VM pointers will not be valid across runs.

3) PM2PM Pointers: A PM2PM pointer can be one of two types: An intra-PMO pointer that does not cross the PMO boundary. It points to a location within the same PMO as the one holding the pointer. Such pointers are essential to build data structures (e.g a link list, skip list or tree) in a PMO. An inter-PMOs pointer that crosses the PMO boundary. It points to a location in a different PMO than the one holding the pointer. Such pointers might seem unnecessary at first glance, however they may be desired under certain situation.

Consider an example hash table that uses link lists to handle collisions. Hash table and linked lists can be placed in a single PMO as shown in Figure 1a. However, splitting them into multiple PMOs confers two benefits: 1) permitting different processes to attach and modify different lists concurrently, and 2) shortening the exposure window of PMO resident data. In Figure 1a, the PMO1 must be attached (and hence hash table and all linked lists are exposed) for the duration $T_1 + T_2$, where T_1 and T_2 are the times for accessing the hash table entry and traversing the linked list (of entry zero), respectively. In contrast, the separate PMO approach (Figure 1a(b)) allows PMO1 to be attached for only T_1 and PMO2 for only T_2 independently. This approach requires inter-PMO pointers.

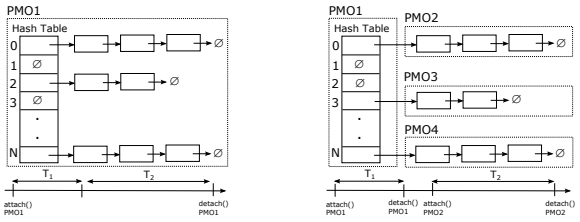


Fig. 1: Hash table and linked list placed in a) same PMO and b) different PMOs.

B. Address-based pointer classification

Based on the addressing mechanism, either absolute or relative pointers can be used to access PMOs and data-structures they hold. An absolute pointer contains virtual address, e.g. in Mnemsoyne [18] An absolute pointer is fast to dereference because it relies on the traditional address translation mechanism. However, it makes PMO Space Layout Randomization [4] costly; any time the PMO is mapped to a different virtual address region, pointers in the PMO must be rewritten accordingly. Finally, if multiple processes are allowed to simultaneously share a PMO, absolute pointers require the PMO to be mapped to the same virtual address range in all processes.

Alternatively, relative pointers can be used. A relative pointer contains a combination of PMO ID and offset. A relative pointer can use a regular 64-bit format or use a fat

pointer format where a pointer is represented by multiple fields. Figure 2 shows a regular relative pointer format where 32-bit PMO_ID with 32-bit offset form a 64-bit pointer. In order to dereference a pointer, a translation table is looked up to translate the system-wide unique PMO ID to its base virtual address [4], [19], and then the offset is added to it. Unlike absolute pointers, relocating such PMOs is straightforward to perform.

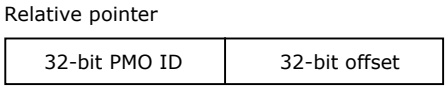


Fig. 2: Structure of a relative pointer [4], [19].

VI. Attack Types

Memory corruption attacks can be divide into two broad categories: 1) *control-data attacks* that alter target program's control data (e.g., return address and function pointer) in order to execute injected malicious code or stitched gadgets, and 2) *non-control-data attacks* that depend on specific semantics of target application and the source code to corrupt variety of application data such as configuration data, user identification data and decision-making data [20].

Since PMO data is long lived, a security attack in one run of a process affects future runs of the process or other processes. This section illustrates that an adversary can exploit a PMO to launch *cross-process/run* control-data and non-control-data attacks on a victim process under certain assumptions.

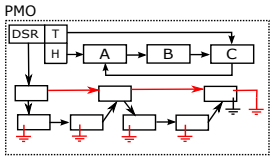
A. Control-Data Attacks

Figure 3a illustrates a PMO consisting of a main data structure (a skip list) and a free list that manages free nodes were deallocated. Skip pointers (of skip list) and next pointers (of both skip list and free list) are shown by in red and black, respectively. Data Structure Root (DSR) holds pointer to the start of the skip list. The free list is a circular linked-list of free nodes where separate head (*H*) and tail (*T*) pointers are maintained. Nodes are allocated from head of the free list.

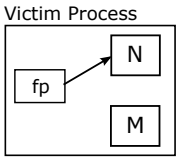
A security attack on the PMO may involve multiple steps as shown in parts 3b-3f of Figure 3. In Step 1, an adversary discovers a function pointer or return address (fp) in the volatile memory portion of the victim process address space. fp initially points to code N, and the code that the adversary wants to execute is denoted as M. M may be an out-of-context library code of victim process (as shown in Figure 3b) or code injected by the adversary.

In Step 2 (Figure 3c), the adversary uses payload process to attach the PMO, overwrite the forward pointer *fd* of first node such that it points to M. Also, the tail pointer is overwritten to point to location of fp minus a constant displacement Δ. The displacement is equal to the difference between address of a node and its *fd* pointer. [2] After this point, the adversary persists the PMO, detaches it, and waits.

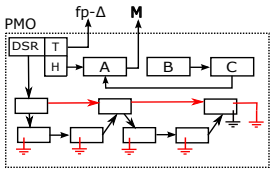
[2]In our example implementation of the attack, the constant displacement is 8 bytes.



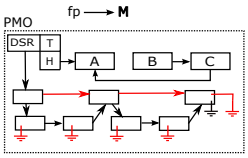
(a) PMO with skip list and free list.



(b) Step 1: Address discovery.



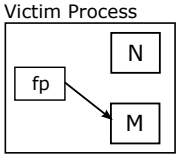
(c) Step 2: Exploit PMO.



(d) Step 3: Activation.

```
1 //C points to fp-Disp
2 last_node=*T;
3 first_node=*H;
4 //makes FP point to M
5 last_node->fd=
      first_node->fd;
6 *H=first_node->fd;
```

(e) Consolidation code.



(f) Step 4: Seize control.

Fig. 3: PMO-based cross-process/run pointer redirection attack.

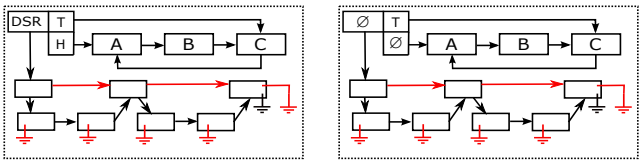
When victim attaches the same PMO again (in the same or a different run) and node A is allocated, the free list consolidation code (Figure 3e) removes node A from the list. Line 2 of the code gets *last_node* by dereferencing *T*. Since *T* was overwritten by adversary, *last_node* points to fp - Δ. The left side of the assignment statement in line 5, last_node→*fd*, points to a location at last_node plus difference between address of last_node and its *fd* pointer: (fp-Δ)+Δ = fp. Since first_node→*fd* was set by adversary to point to M, line 5 makes fp point to M (Step 3, Figure 3d). Finally, when the function pointer is used by the victim, the target code is executed (Step 4, Figure 3f), resulting in a successful attack. Such kind of attack can successfully alter the victim program's execution flow by invoking out-of-context or injected code M.

Note here is that despite not having an exploitable vulnerability, the victim process is successfully attacked because it shares the PMO with another process that has an exploitable memory vulnerability. The time to carry out the attack can span multiple attach/detach sessions, and can even span across multiple process lifetimes, as long as fp and M remain constant over the span.

B. Non-Control-Data Attacks

The severity of non-control-data attacks is equivalent to that of control-data attacks [20]. Examples attacks are denial of service and attacks based on corrupting decision-making data. Such attacks can be set up by an adversary using a similar approach as shown in Section VI-A.

1) Denial of Service Attack: A denial of service attack causes a victim process to produce wrong output, crash, or hang [24]. Figure 4 illustrates such an attack where an adversary uses the payload process to attach a PMO, set DSR and *H* pointers (in PMO) to null, persist the updates, and detach the PMO. Then adversary waits for the victim to attach again the same PMO (in the same or different run) and access the skip list or free list. When that happens, victim process crashes as result of segmentation fault.



(a) PMO with skip list and free list. (b) DSR & H set to null.

Fig. 4: PMO-based cross-process/run denial of service attack.

2) Attack corrupting decision-making data: Decision-making routines rely on several boolean variables (conjunction, disjunction, or combination of both) to reach the final verdict. An adversary can corrupt the values of these decision-making data to influence the eventual critical decision [20].

As an example from real application, Figure 5 shows a code snippet from *ZRANGE min max BYSCORE* command of REDIS [21], [22]. The code iterates over elements (shown as *ln* in the snippet) of a skip list and returns all the elements in the sorted set with a score between *min* and *max*. For example, invocation of *ZRANGE 4 8 BYSCORE* on the PMO shown in Figure 6a prints scores 4, 4, 6, 8.

Figure 6b shows an attack where an adversary uses a payload process to attach the PMO, corrupt the data by overwriting value of a skip list node from 6 to 9 (shown in red), persist the updates and detache the PMO. When the victim process, after attaching again the same PMO, invokes *ZRANGE*, the command returns only 4, 4 (instead 4, 4, 6, 8).

VII. Possible Defenses

Some of the attacks presented in Section VI can be mitigated by applying existing defenses. For example, the pointer redirection attack presented in Figure 3 can be successful only if addresses of fp and M (out-of-context code in victim process) are not changed between two successive runs of the victim process. If PSLR is enabled, the addresses of fp and M will be randomized on subsequent run and hence attack will fail to change the execution flow of victim. Similarly, with M as a code injected by adversary, the attack can be successful only if Data Execution Prevention (DEP) is not supported on the execution platform.

Some attack frameworks can breach PSLR or DEP defense schemes [23], [27]. Also, denial of service and (PMO-resident) data corrupting attacks are agnostic to PSLR and DEP. Therefore, additional steps are needed to detect and foil PMO-based attacks.

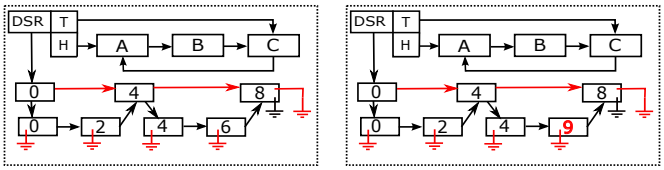
Figure 7 shows the timeline of the steps in the PMO-based pointer redirection attack (presented in Figure 3), illustrating a

```
while (ln && limit--) {
    /* Abort when score of the node is no
    longer in range. */
    if (reverse) {
        if (!zslValueGteMin(ln->score,range))
    break;
        } else {
        if (!zslValueLteMax(ln->score,range))
    break;
    }

    rangelen++;
    /* Send score to the handler */
    handler->emitResultFromCBuffer(handler, ln
    ->ele, sdslen(ln->ele), ln->score);

    /* Move to next node */
    if (reverse) {
        ln = ln->backward;
        } else {
        ln = ln->level[0].forward;
        }
    }
```

Fig. 5: A code snippet from *ZRANGE* command of REDIS [22]



(a) PMO with skip list and free list. (b) PMO data corruption.

Fig. 6: PMO-based attack to corrupt decision-making data.

PMO shared successively by two processes. The figure shows opportunities for detecting and foiling the attack.

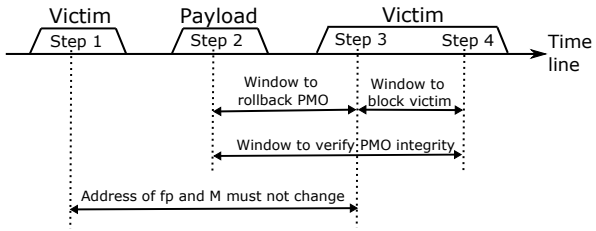


Fig. 7: Steps of cross-process/run PMO-based pointer redirection attack.

First, address of fp and M must remain the same between Step 1 and 3, for the attack to succeed. If the addresses change, the attack will corrupt PMO but not result in control flow hijacking. Second, the window of time between Step 2 and 4 is the window of opportunity to detect the attack by verifying the integrity of the data structures in the PMO. In other words, the integrity check must be performed before step 4 as the potential attacks get activated by that time. The integrity of data structure(s) can be checked by performing topology verification [25]. Third, in the window of time between Step

2 and 3, if PMO integrity problem is detected, and a non-corrupted previous version is restored, the PMO integrity can be restored and attack foiled. But, between Steps 3 and 4, to foil the attack, the victim process must be blocked/terminated.

Denial of service and data corrupting attacks (presented in Figure 4 and 6, respectively) overwrite the PMO-data instead of pointers. To detect such attacks, data integrity check (e.g., calculating and comparing against stored SHA256 hash code) is needed before a process accesses PMO-resident the data.

To foil the attack where new hash code is computed after corrupting data and stored in PMO, data invariance checks can be performed. For example, a PMO can be screened to ensure that data structure root (DSR) and head/tail pointers of free-list are not null. Similarly, data semantics may also be used for invariance checks, e.g. the values of skip list nodes must be monotonically increasing or decreasing. Depending on the application, several other data invariance checks can be applied [26]. Note that applying larger number of data invariance checks increases the probability to foil non-control-data attacks but at the cost of higher overhead of invariance checking.

Table I summarizes PMO-based attacks mentioned above, assumptions for setting up those attacks and strategies to detect them.

TABLE I: Summary of PMO attacks

Attack	Assumptions	Detection strategy
Pointer redirection to out-of-context code	No PSLR PM2NVM pointers are permitted	Topology verification
Pointer redirection to injected code	No PSLR No DEP PM2NVM pointers are permitted	Topology verification
Denial of service		Hash re-computation and comparison
Corrupting decision-making data		Data invariance checking

VIII. Conclusion

In this paper, we have shown that multiple processes may access persistent data in an uncoordinated way under PMO programming model. This makes PMOs a new tool for launching security attacks. We demonstrated PMO-based control-data and non-control-data attacks, and possible defenses. This paper makes the case for increased memory safety protection when persistent memory is used.

References

[1] Intel, "Intel® Optane™ DC Persistent Memory," Available at https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (2021/07/21).

[2] H. Taeho, L. Dokeun, N. Yeonjin, W.Youjip, "Designing persistent heap for byte addressable NVRAM," In 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA) 2017 Aug 16 (pp. 1-6). IEEE.

[3] K. Sudarsun, G. Ada, S. Karsten, "PVM: Persistent virtual memory for efficient capacity scaling and object storage," In Proceedings of the Eleventh European Conference on Computer Systems 2016 Apr 18 (pp. 1-16).

[4] X. Yuanchao, S. Yan, S. Xipeng, "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 2020 Mar 9 (pp. 987-1000).

[5] D. Subramanya R et al, "System software for persistent memory," In Proceedings of the Ninth European Conference on Computer Systems 2014 Apr 14 (pp. 1-15).

[6] C. Jeremy et al, "Better I/O through byte-addressable, persistent memory," In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles 2009 Oct 11 (pp. 133-146).

[7] X. Jian, S. Steven, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," In 14th USENIX Conference on File and Storage Technologies (FAST 16) 2016 (pp. 323-338).

[8] C. Siddhartha, S. Yan, "i-NVMM: A secure non-volatile main memory system with incremental encryption," In 2011 38th Annual international symposium on computer architecture (ISCA) 2011 Jun 4 (pp. 177-188). IEEE.

[9] L. Youyou, S. Jiwu, S. Long, M. Onur, "Loose-ordering consistency for persistent memory," In 2014 IEEE 32nd International Conference on Computer Design (ICCD) 2014 Oct 19 (pp. 216-223). IEEE.

[10] P. Steven, MC. Peter, FW. Thomas, "Memory persistency," In 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA) 2014 Jun 14 (pp. 265-276). IEEE.

[11] R. Andy, "Persistent memory programming," Login: The Usenix Magazine. 2017;42(2):34-40.

[12] Z. Pengfei, H. Yu, "SecPM: a secure and persistent memory system for non-volatile memory," In 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18) 2018.

[13] Z. Pengfei, H. Yu, X. Yuan, "Supermem: Enabling application-transparent secure persistent memory with low overheads," In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture 2019 Oct 12 (pp. 479-492).

[14] Y. Mao, AZ. Kazi, M. Aziz, A. Amro, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," IEEE Transactions on Dependable and Secure Computing. 2019 Sep 13.

[15] M. Sparsh, IA. Ahmed, "A survey of techniques for improving security of non-volatile memories," Journal of Hardware and Systems Security. 2018 Jun;2(2):179-200.

[16] X. Yuanchao, Y. ChenCheng, S. Yan, S. Xipeng, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) 2020 May 30 (pp. 680-692). IEEE.

[17] (SNIA), S. N. I. A. Persistent memory hardware threat model. Technical Whitepaper (2018).

[18] V. Haris, JT. Andres, MS. Michael Swift M, "Mnemosyne: Lightweight persistent memory," ACM SIGARCH Computer Architecture News. 2011 Mar 5;39(1):91-104.

[19] W. Tiancong, S. Sakthikumaran, S. Yan, T. James, "Hardware supported persistent object address translation," In 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) 2017 Oct 14 (pp. 800-812). IEEE.

[20] C. Shuo, X. Jun, CS. Emre, G. Prachi, KI. Ravishankar, "Non-Control-Data Attacks Are Realistic Threats," In USENIX Security Symposium 2005 Aug (Vol. 5).

[21] Redis, "ZRANGE - Redis," Available at https://redis.io/commands/zrange (2021/07/20).

[22] Redis, "redis/t_zset.c at unstable . redis/redis . Gitub," Available at https://github.com/redis/redis/blob/unstable/src/t_zset.c (2021/07/20).

[23] B. Dion, "Interpreter exploitation: Pointer inference and JIT spraying," BlackHat DC. 2010 Jan.

[24] MM. Kharbutli, "Improving the security of the heap through inter-process protection and intra-process temporal protection," North Carolina State University; 2005.

[25] C. Bor-Yuh Evan, R. Xavier, N. George C, "Shape analysis with structural invariant checkers," In International Static Analysis Symposium 2007 Aug 22 (pp. 384-401). Springer, Berlin, Heidelberg.

[26] DE. Michael et al, "The Daikon system for dynamic detection of likely invariants," Science of computer programming. 2007 Dec 1;69(1-3):35-45.

[27] ZS. Kevin et al, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," In 2013 IEEE Symposium on Security and Privacy 2013 May 19 (pp. 574-588). IEEE.