Temporal Exposure Reduction Protection for Persistent Memory

Yuanchao Xu*, Chencheng Ye†, Xipeng Shen*, Yan Solihin‡
*North Carolina State University, †Huazhong University of Science and Technology,‡University of Central Florida,

I. Introduction

The emerging persistent memory (PM) is increasingly supplementing and substituting DRAM as main memory, due to PM's higher density, better scaling, lower idle power, and non-volatility, while retaining byte addressability and random accessibility [2]. The long-living nature and byte-addressability of persistent memory (PM) amplifies the importance of strong memory protections. First, persistent data in Persistent Memory Object (PMO) is now subject to memory safety vulnerabilities (e.g. accidental/malicious read/write). Second, worse than DRAM, data corruption is permanent in a PMO. Third, data in a PMO is long lived; its existence and structure are preserved across process runs. The longevity, plus direct byte-addressability, makes it more vulnerable as attacks to a PMO could span across executions of the same or different applications. We anticipate further growth of memory corruptions and disclosures targeting PMOs, and thus emphasize the need for practical primitives that eliminate or reduce such threats. Despite decades of research, unauthorized memory reads and writes are still among the most common security attacks [1], [4], [5]. A recently work combines isolation and re-randomization to provide better protection and efficiency for PM [6].

In our HPCA 2022 paper*, it develops temporal exposure reduction protection (TERP) as a framework for enforcing memory safety. Aiming to minimize the time when a PM region is accessible, TERP offers a complementary dimension of memory protection. The paper gives a formal definition of TERP, explores the semantics space of TERP constructs, and the relations with security and composability in both sequential and parallel executions. It proposes programming system and architecture solutions for the key challenges for the adoption of TERP, which draws on novel supports in both compilers and hardware to efficiently meet the exposure time target. Experiments validate the efficacy of the proposed support of TERP, in both efficiency and exposure time minimization.

II. Programming Support

As a class of protections in a dimension complementary to existing memory protections, rather than seeking to protect against specific vulnerabilities, TERP makes unauthorized reads or writes to data in PMOs difficult by applying the principle of least privileges.

Temporal protection: If a memory attack requires a memory region to be stationary (i.e. location unchanged) and accessible for at least t time to succeed, the attack can be prevented as long as the exposure window of the memory region is smaller than t, and locations of the region changed before t elapses.

EW-Conscious Semantics: We propose exposure window (EW) conscious semantics or *EW-conscious semantics* in short. It also requires non-overlapping attach-detach pairs, but only within a thread. It considers thread-level permissions and process-level address mappings together, allows implicit lowering of TERP constructs in a TERP poset, and supports both function and thread composability as well as automatic deployment.

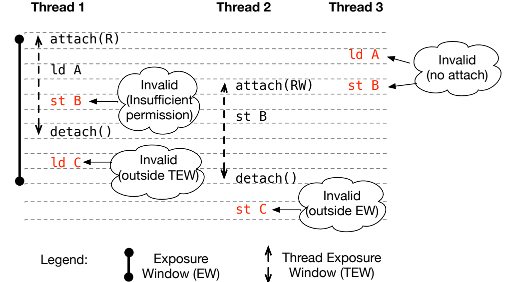

Fig. 1. An example of EW-conscious semantics.

An example of the semantics is shown in Figure 1. Suppose that addresses A, B and C reside in PMO1. First, Thread 1 attaches PMO1 with read permission. Since PMO1 was unmapped, the attach is performed to map PMO1 to the address space. The subsequent ld A is permitted but st B is denied due to insufficient thread permission. For Thread 2, the attach adds intended read and write thread permission and hence the subsequent st B is permitted. After that, the detach call from Thread 1 removes its permission, but does not detach the PMO as thread 2 can still access it. The subsequent ld C is denied access. The detach call from Thread 2 removes its permission and detaches PMO1 from the process address space. The subsequent st C is denied and generates a segmentation fault since the PMO is no longer mapped to address space. For Thread 3, all accesses are denied because the thread never makes an attach call prior to the accesses.

III. Design

The second key challenge for TERP adoption is how to make it easy to use, while simultaneously offering the desired protection without imposing any substantial performance overhead?

Automatic Constructs Insertion: The EW-conscious TERP semantics requires non-overlapping and matching *attach* and *detach* calls on every path (in a thread). Requiring the programmer to manually insert such calls would be a burden, hence we propose to automate it. The compiler's goal is to insert *attach* and *detach* completely and correctly, while meeting the target EW length. We develop a region-based code analysis to do it efficiently.

The possible execution paths is exponentially large due to the number of branches. we make an observation that inserting a detach call changes the PMO attachment state back to the initial state (PMO is not attached) regardless of paths leading to the detach point. If at a confluence point, the state is known to be detached, the control flow graph can be split into two regions: one with paths preceding the confluence point and one following it. These regions can then be considered independently, hence reducing the analysis complexity. For each region, attach calls

*Full Paper: Yuanchao Xu, et al., "Temporal Exposure Reduction Protection for Persistent Memory" in HPCA 2022 [7].

can be inserted to each path so that attach and detach pairs are maintained. The detailed algorithm is in Section-V-A of the original paper.

Architecture Design: Our architecture support seeks to reduce overhead. First, we observe that static analysis often detaches too soon, only to attach again soon afterward. This presents an opportunity to combine two closely spaced EWs into one. Second, we note that an attach and detach call may be executed in one of two ways: (1) performed fully to map or unmap into/from address space, (2) performed partially to grant or revoke thread access permission. While both ways can be implemented as system call handling code, the latter can be accelerated. We refer to the former opportunity as *window combining* and the latter as *conditional attach/detach*.

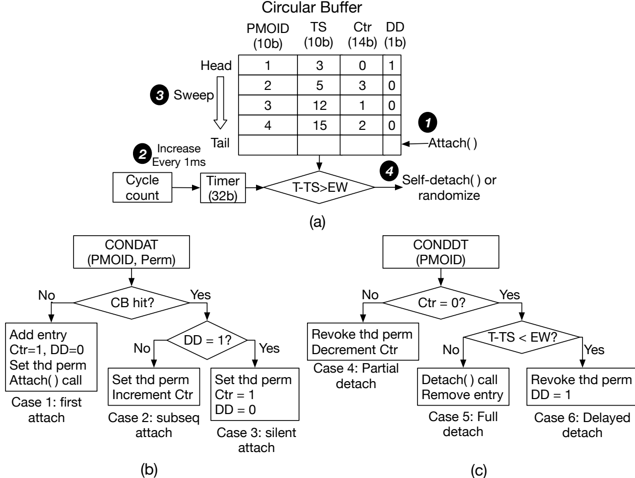

Fig. 2. Circular buffer (a), and the logic for conditional attach (b) and detach (c).

For *window combining*, we use a circular buffer shown in Figure 2(a). Each entry keeps the *PMO ID*, a *timestamp* (TS) that records the time of last PMO attach, a *counter* (Ctr) which tracks the number of threads that have made an attach call, and a *delayed detach* (DD) status which indicates if a detach has been delayed. A newly attached PMO is added at the tail 1. A timer is incremented at a coarse granularity, such as every 1us 2. Periodically, we sweep the buffer from head to tail 3, to identify PMOs for which a detach call has been made but they have not been detached yet, as indicated by $DD = 0$. Then, the counter is checked 4. If the counter is zero, the detach() system call is made to fully detach the PMO. Otherwise, some threads still access the PMO, hence the PMO is randomized. Randomization requires all threads to be suspended and appropriate structures invalidated or updated (e.g., TLB shootdowns and page table update).

To support *conditional attach/detach call*, we add two user-space instructions, conditional attach (`CONDAT`) and conditional detach (`CONDDT`). `CONDAT`'s two source operands include a PMO ID and a permission request (R or RW). `CONDDT` only takes PMO ID as its source operand. Our compiler algorithm performs static analysis and inserts `CONDAT` and `CONDAT` into code in place of actual attach or detach system calls. Figures 2(b) and (c) show how the two instructions are executed.

IV. Evaluation

Methodology: Our simulator is built on Sniper, a cycle-accurate X86 simulator. We implement an LLVM pass [3] that uses the region-based analysis in LLVM to insert conditional instructions (magic instructions in Sniper), producing protected programs.

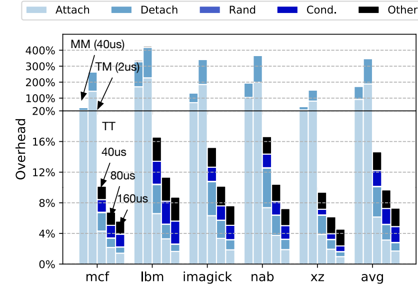

Fig. 3. Single-thread multi-PMO execution time overheads for SPEC.

Results: The execution time overheads for 40us, 80us, and 160us EWs are shown in Figure 3. The values are averaged over all PMOs. SPEC benchmarks have multiple PMOs and PMO accesses make up for a large fraction of total accesses. Hence, the figure shows more than 300% overheads when all attach and detach are performed through system calls in the TM. Our optimizations turn 96.8% attach and detach silent, hence reducing the overheads to only 14.8% for 40us EW, and 7.6% for 160us EW, representing more than an order of magnitude reduction. Compared with MERR (MM), our whole design is able to merge closely-spaced attach/detach sessions and introduce TEW to further augment security while lowering the overheads from 156.3% to 14.8%.

V. Impacts

We highlight two key contributions of TERP. First, TERP presents the first programming support for intra-process isolation. The semantic explorations, compiler support, and architecture support are applicable to all intra-process (e.g, MPK) or intra-enclave isolation. Second, TERP demonstrate the performance, programmability, and security benefits of region-based memory management. This encourages researchers to rethink region-based memory allocation for security.

References

[1] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[2] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[3] LLVM, "Writing an llvm pass." https://llvm.org/docs/WritingAnLLVMPass.html, online; accessed August, 2020.

[4] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[5] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.

[6] Y. Xu, Y. Solihin, and X. Shen, "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, pp. 987–1000.

[7] Y. Xu, C. Ye, X. Shen, and Y. Solihin, "Temporal exposure reduction protection for persistent memory," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 908–924.