2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00071

Temporal Exposure Reduction Protection for Persistent Memory

Yuanchao Xu*, Chencheng Ye†, Xipeng Shen*, Yan Solihin‡
*North Carolina State University
{yxu47, xshen5}@ncsu.edu
†Huazhong University of Science and Technology
yecc@hust.edu.cn
‡University of Central Florida
Yan.Solihin@ucf.edu

***Abstract*—The long-living nature and byte-addressability of persistent memory (PM) amplifies the importance of strong memory protections. This paper 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.**

***Keywords*-Memory Security, Persistent Memory, Memory Exposure Reduction, Hardware-Software Co-Design**

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 [1]–[4]. To enable PM to host persistent data in memory, an abstraction of *persistent memory objects* (PMO) [5] has been proposed to allow one or more data structures to be kept in memory without file backing. This removes the need to serialize or deserialize data and avoids the high overheads of interfacing with the file system. PMOs can be managed by the OS similar to files (in terms of namespace and permission) but can be accessed directly through load/store instructions and contain pointers. They are *attached* (i.e., mapped) to a process address space for access, and later *detached* (i.e., unmapped). The attach and detach constructs are typically provided as system calls.

While good for performance, allowing direct memory access to a PMO poses serious security challenges. First, persistent data in 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 [6]–[9]. There are many schemes already proposed for improving memory safety in general, including adding memory safety features to C and C++ [10]–[12], Data Flow Integrity (DFI) [13], data region isolation techniques [14]–[16], data re-randomization [17]–[19], etc. While these techniques can improve memory safety, only recently their principles have been applied to the security of PMO [5], a technique that combines isolation and re-randomization to provide better protection and efficiency for PM.

MERR [5] provides temporal protection of PMO data by bookending a group of accesses with a pair of attach and detach, along with randomizing the location PMO maps in the address space at each attach. Protection is provided in two ways. First, by keeping each *exposure window or EW* (i.e. the length of a single attached session) short and randomizing PMO locations for each EW, any access outside EW generates protection fault while data location learned by the attacker in one EW is no longer valid in the next EW due to the randomization. However, MERR suffers from four major problems: semantics, security, composability, and performance overheads.

First, MERR relies on manual insertion of attach-detach and EWs do not overlap. However, this is problematic in real life. First, the programmers must ensure that there is a corresponding detach in each execution/branch path following the attach, which could be challenging in a program with complex control flow. A missing attach leads to program crashes and a missing detach may lead to zero security improvement due to large EWs. Second, the error-prone manual insertion likely leads to many incorrect usages of MERR, hence reducing or removing MERR security. Third, MERR is not composable. It defines attach detach as process-wide in scope, and safe multi-threaded usage is not guaranteed. Threads execute concurrently, hence EWs from

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00071

several threads may overlap and produce undefined behavior. Furthermore, a nesting of attach-detach may occur (due to function/library calls, recursion, etc.), and MERR does not guarantee correct behavior or security under nesting. Finally, the cost of attach/detach is still prohibitively high for frequent use as it still involves a system call, TLB shootdown, etc.

In this work, we propose a framework for temporal exposure reduction protection (TERP) that solves the aforementioned challenges. First, TERP provides rigorously-defined semantics of attach and detach, allowing it to deal with complex control flow. TERP introduces *TERP poset* (partially ordered set) as a way to organize the set of mechanisms for access protection of different strengths. We explore the semantics definition space and investigate each semantics affects security protection and composability.

Our choice of semantics provides high thread composability, which allows safe execution of multi-threaded programs. Multi-thread safety allows TERP to introduce a new concept of *thread exposure window* (TEW), which adds thread-specific attach-detach semantics to MERR's process-wide semantics. The new concept allows TERP to substantially decrease PMO exposure compared to MERR. Finally, our solution co-designs novel compiler and architecture support based on formal definition of TERP. The architecture support ensures the protection goal (i.e. size of the maximum exposure windows) by transparently silencing (or lowering on a TERP poset) unnecessary invocations of TERP operations. Through novel instruction support, nearly 90% of system calls can be avoided and their overheads substantially reduced. The support also monitors exposure windows and improves security by ensuring that the target exposure window is met.

While TERP is potentially applicable to all region-based memory, is a particularly good fit for addressing the vulnerability of the long-living byte-addressable PMOs. Hence, our discussion in this paper focuses on PMO protection. It is worth mentioning that TERP design is potentially applicable to other isolation techniques [14]–[16], [20], [21], as these techniques also require programmers to manually insert special instructions. Overall, this paper makes the following contributions:

1) A formal TERP poset framework, which rigorously defines the semantics of the attach and detach constructs.
2) Exploration of the semantics space of TERP, and the choice of TERP semantics that achieves both composability and security for multi-threaded programs and allows nesting.
3) Novel architecture and compiler mechanisms to support efficient programming systems for TERP.
4) Empirical validation of TERP showing huge performance overheads reduction (6% and 15% for WHISPER and SPEC benchmarks vs. 20% and 156% with MERR).
5) A quantitative case study of security analysis against data-only attack, showing 20× reduction in attack surface vs. MERR.

II. Background

Persistent Memory Programming Support: There are at least two paradigms for using PM. One may use 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 file backing. 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) [5]. 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 upon software crashes or system power failures, *system naming and permission* allows a PMO to be located and reused across runs, *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. PMOs may be implemented as pools [22]–[25] and given a unique identifier to each PMO.

Table I
Pool APIs described in prior work [5], [23], [24]

Function	Description
PMO* PMO_create (size, mode)	Create a PMO with the specified size. The running process is the owner.
PMO* PMO_open (name, mode)	Reopen a PMO using name that was previously created.
PMO_close(PMO* p)	Close a PMO p.
OID pmalloc (PMO* p, size)	Allocate a chunk of persistent data with the given size on PMO p and return the PMOID of the first byte.
pfree(OID)	Free persistent data pointed by the ObjectID.
void* oid_direct(oid)	Translate an ObjectID to a virtual address.
void* attach(PMO* p, permission)	Memory map a opened PMO p into process address space with requested permission.
void detach(PMO* p)	Memory unmap an attached PMO p from process address space.

To support relocatability, each pointer (64-bit) used in a data structure consists of a pool ID (ObjectID) and an offset within the PMO. PMDK [23] and other prior works [5], [22], [24] have described an interface for manipulating pools and objects (Table I), which we adopt. We assume the following assumptions of PM programs: (1) attach() returns an immutable handler that records the current virtual address of this PMO. The programs access the data of this PMO through this handler. (2) No inter-PMO pointers, which means objects pointing to each other reside on the same PMO. (3) The program has at least one matched attach() and detach() pairs.

Fast PMO Attach and Detach: A PMO attach system call attaches (memory maps) a PMO to a process address space by initializing page table entries (PTEs) that point to this PMO; the detach system call detaches (memory unmaps) this PMO from the process address space by invalidating these page table entries. The overhead of an attach() or a detach() call grows linearly with PMO size, as the number of modified PTEs grows linearly with PMO size.

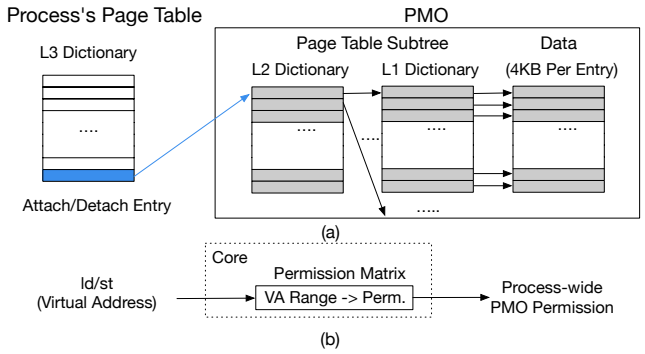

Figure 1. (a) Fast attach and detach a PMO with the embedded page table subtree. (b) Permission Matrix.

A recent work called MERR [5] proposes to embed a page table subtree into the PMO as persistent metadata [5], as shown in Figure 1 (a). With it, upon an attach() call, the process page table only needs to initialize one PTE to point to the PMO page table subtree. When the detach() is invoked, the process page table removes this PTE and a TLB invalidation happens, and the process loses access to the PMO. This mechanism reduces the overhead of attach() and detach() to constant, regardless of the PMO size. This basic idea, embedding a page table subtree into a PMO, however, cannot discern the different permissions for different processes. MERR [5] hence expands the design with a process-specific permission matrix hardware support, as shown in Figure 1 (b). *attach(PMO ID, Permission)* adds one entry to the permission matrix with the requested permission and virtual address range of this PMO, and *detach(PMO ID)* removes the entry from the permission matrix. A ld/st instruction, checks permission against both the permission matrix and TLB. The fast attach and detach mechanisms are combined with address randomization to improve memory safety. TERP leverages it as one of the underlying mechanisms for temporal protections, and builds up a complete conceptual framework TERP along with the needed programming/architectural support.

III. TERP Concepts, Constructs, and Relations with Security

This section presents formal definitions of several key concepts of TERP and its core constructs, along with the connections with security. The conceptual framework offers the necessary clarity and foundation for the deployment of TERP and the needed programming system support. It then examines the programming challenges of TERP, which motivates the discussions on the semantics of TERP constructs, driven by which, it proposes three alternative semantic definitions and discusses their pros and cons, in terms of both security and composability.

A. Definitions

We start with several auxiliary terms.

Definition 1: **Permission set:** A permission set is a set of binary values that define the access permissions to a set of data objects (say S), represented as $\{a(O_i) = b \text{ — } a \in \{read, write, execute\}, O_i \in S, b = 0|1\ \}$.

Definition 2: **Permission group G(P):** A permission group $G(P)$ is a set of entities G (e.g., threads, processes, users) that share a permission set P, that is, $P \subseteq \bigcap_{g \in G} p(g)$, where, $p(g)$ is the permission set of agent g.

Definition 3: **TERP:** A mechanism offers a TERP protection to a memory region against a permission group G if the mechanism reduces the time when the memory region is accessible by G.

TERP protections can be in various kinds with different *strengths* and *scope*. Two examples are as follows. (i) Insertions of *detach()* and *attach()* calls on a PMO periodically into a program, as done in MERR [5], which reduce the time when a process can access a PMO. (ii) Insertion into a program the instructions to grant or revoke thread access permission for a PMO, which reduces the time when the thread can access the PMO. The two kinds of protections differ in the targeted permission groups: the former against all threads in a process, the latter against one specific thread only.

Definition 4: **TERP Poset:** A set of TERP protection mechanisms form a TERP poset if they are in a partial order based on certain attributes.

The formulation makes existing mathematical tools in order theory directly applicable to TERP. The Hasse Diagram, for instance, can be used to visually represent the elements of a TERP poset and the relations of their partial ordering, as illustrated in Figure 2. Section IV-C will show some important benefits a multiple-level TERP poset can bring.

A TERP protection mechanism usually consists of two types of basic constructs: one grants accessibility, the other revokes it. We call them **granting** and **depriving** constructs respectively. Different TERP protections have their own implementations of the two basic constructs.

A primary metric used for measuring the effect of deployed TERP protections is the length of an *exposure window*.

Definition 5: **Exposure Window:** For a permission group G, an exposure window of a PMO is a contiguous time window in which the PMO is accessible by G.

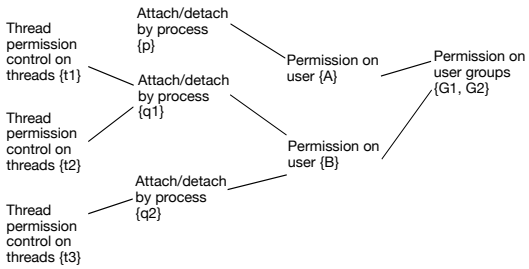

Figure 2. A Hasse Diagram that represents a TERP poset in terms of the partial order of permission groups, regarding their accessibility to a PMO.

Efficiently reducing the length of exposure windows is a central mean of TERP for enhancing memory safety. We next elaborate the connections of this temporal protection with memory security.

B. Connections with Security

By offering protection at the temporal dimension, TERP complements other security measures. We discuss it in the context of the following threat models on PMO.

Threat Model: The focus is on data-only attacks [9], [16], [26]. Side-channel and microachitectural leaks are not the focus. In the threat models, data structures in PMO may contain buffers and pointers. Code that accesses PMO may contain vulnerabilities via which malicious reads or writes are possible to PMOs (e.g., buffer overflow, format string). While a PMO is attached, the attacker can probe, read or write the data in the PMO. The attacker may compromise a thread of the same process and try to exploit the memory vulnerabilities. Control-flow attacks are mitigated through state-of-the-art control-flow attacks mitigations [27]–[32].

We assume trusted system software (e.g., OS), which manages address space isolation between processes. With that, reading or writing data to the memory region that is not mapped in the page table will not be permitted and will generate segmentation fault exceptions. 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. We also assume a randomization mechanism (e.g. random number generator) that is trustworthy.

User-level instructions can only be inserted by the programmer or compiler. It is assumed that attackers are prevented from injecting or reusing these instructions (e.g. through ROP); call gates and binary inspection and rewriting techniques like ERIM can be used to ensure it [33].

Connections to Security: 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: Unauthorized reads or writes are the fundamental schemes many types of memory attacks rely on. The following theorem is the fundamental connection of TERP with memory security:

Theorem 6: **Temporal protection theorem:** 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.

The theorem is intuitive; its implication is however profound. It provides the fundamental rationale for the benefits of TERP, and also offers the basis for setting the objective of TERP protections in preventing certain kinds of memory attacks. Moreover, it suggests the powerful synergy between TERP and other security measures. For instance, TERP may combine with memory space layout randomization such that every time a PMO gets (re)attached to the memory space, it is put to a random location. The randomization makes memory probing hard to continue across exposure windows. As a result, as long as TERP makes each exposure window smaller than the minimum time the probing requires to find the base address, the attacks can be prevented.

TERP mechanisms form a poset due to the mechanisms having different levels of permission and isolation. A higher level isolation usually provides higher protection but incurs higher overhead. For example, a process-wide attach/detach provides stronger security protection when PMOs are detached. For example, even Spectre attacks or attacks exploiting errors in implementing the MMU cannot succeed when PMO is detached since it is not present in the process address space. Furthermore, attaching the PMO requires a system call through which the OS may perform additional security checks. In comparison, setting thread access permission control relies on intra-process isolation techniques such as Intel MPK. It provides weaker protection because the permission control setting relies on (PKRU) registers that can be changed at the user level. Therefore, the higher level protection should be used at coarser grain triggers or intervals, while lower level protection should be used at finer grain triggers or intervals.

Finally, in addition to temporal protection, TERP constructs are applied separately for separate PMOs, hence also providing *spatial* protection. Our empirical study in Section VII reveals that even though an application may use multiple PMOs, the application only uses 1 or 2 PMOs at any given time, which provides opportunities for TERP to keep unused PMOs inaccessible to the application. Prior work [19] found that the fewer data is mapped in a process, the lower the possibility to identify a target page table entry (PTE).

IV. Adoption Challenge I: Semantics

We identify two key challenges for the adoption of TERP. One is on defining the semantics (i.e., intended behaviors) of TERP core constructs, the other is on inserting into a

program the calls of TERP constructs appropriately. We will now discuss these *semantics* challenges and *deployment* challenges. We note that our discussion will focus on a single PMO, but the semantics extend trivially to multiple PMOs as we assume that no inter-PMO pointers.

A. Why isn't the basic semantics sufficient?

As Section III mentions, there are two basic constructs in TERP, *granting* and *depriving* constructs. At the very high-level, their semantics are that the *granting* construct adds accessibility of a PMO, while the *depriving* construct removes accessibility of a PMO. Such a level semantics definition however is insufficient as it does not lend to the necessary security guarantee and practical deployment. To illustrate the complex consideration in defining the semantics, we will draw on *attach()* and *detach()* system calls as example TERP granting and depriving constructs. MERR [5] did not define clear semantics and ignore important issues such as security guarantee, composability, and concurrency.

We start by defining three semantics choices for attach() and detach(): Basic, Outermost, and first-come first-serve (FCFS), illustrated in Figure 3. The lessons we learned from the three semantics lead to our fourth and chosen semantics EW-Conscious.

Basic Semantics: Each attach() must be followed by a detach(), and every detach() must follow an attach(). Any other attach or detach is considered invalid. All valid attach and detach calls are fully executed and applies to the whole process.

Example: Figure 3 shows an example code, with a pair of attach/detach, followed by nested pairs of attach/detach that from a function call. Accesses to data `a` within the PMO are interleaved. With the basic semantics, the first attach/detach pairs (lines 1 and 3) are valid and define the first exposure window (EW). An access to the PMO within the EW (line 2) is valid while outside the EW (line 4) is invalid. The second attach() (line 5) is valid, but the third attach() (line 7) returns an error. The basic semantics requires that an attach() is followed by a detach, not by another attach(). Further access and detach (lines 8-9) are undefined due to the prior attach call error.

Security and Composability: Let us analyze Basic on *security* and *composability*. Security is about reaching the security objective in terms of the maximum exposure window size. Composability has two aspects. For the function or module aspect, it means that when well-formed functions or modules with TERP constructs are invoked in a program, they would introduce no extra errors nor jeopardize the exposure window guarantees for the program. For the thread/concurrency aspect, it means that multiple well-formed threads with TERP constructs can work well together without compromising the exposure window guarantees or introducing errors.

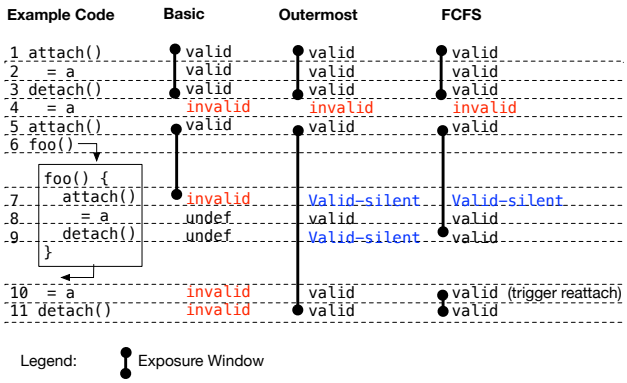

Figure 3. Illustrating the differences of the attach/detach semantics and their implications on security and composability.

For strong memory security, one may need to insert many pair-matched *attach()* and *detach()* into a program to ensure small exposure windows. However, composability concerns may arise. For each attach() call inserted, the programmer is burdened with inserting a matching detach call following every execution path from the attach call. Similarly, for each detach call inserted, there must be a matching preceding attach call leading to the execution path where the detach call is located. This places a substantial burden on the programmer. Missing an attach() may crash the programs (for detaching an unmapped PMO), while missing a detach can even void security improvements (for creating a large exposure window).

Another serious composability concern arises with a multi-threaded program. Suppose that each thread is well-formed with the right attach and detach calls added to provide intended exposure windows in the thread. But when multiple threads are combined to run concurrently, the attach-detach windows in different threads may overlap in time and violate the semantics. If an attach() is made by one thread and then by another, according to the basic semantics, the second attach is invalid. Likewise, double detaching also leads to execution errors.

There is a parallel between the attach/detach semantics for PMOs with fopen/fclose and map/munmap constructs for file systems. Ignoring that the interface applies to different system objects, it is noteworthy that fopen/fclose and mmap/munmap appear to approximately follow the basic semantics, and therefore suffer from similar strengths (simplicity) and weaknesses (complexity from pair-matching on diverging execution paths, and function and thread composability problems).

It is worth noting that the basic semantics in Section IV-A is similar to the semantics of traditional fopen/fclose and mmap/munmap constructs in file systems. It is however not preferable in this security context. Unlike the previous

contexts where usually only a few matching calls of the constructs are needed in a program, in our context, with that semantics, many matching attach/detach pairs would need to be put into a program by a programmer, causing programming challenges and compatibility problems in multithreaded programs.

B. Other Considered Alternatives

Figure 3 illustrates two other semantics. One is *Outermost semantics*: Attach-detach pairs must form perfect nesting relations if they overlap; only the outermost attach or detach is performed and inner attaches and detaches are all made silent. This semantics cannot offer needed temporal protections as the actual attached time can be arbitrarily long.

The other alternative is *FCFS semantics*: An outermost attach is valid and performed, whereas inner attach calls are silent. The first detach encountered after an attach() is performed, other detaches are silent. Any access prior to the outermost detach triggers an automatic PMO reattach, and the first detach following it is performed. The issue with this semantics is that within the outermost attach and detach, it is hard to distinguish a benign access (that should result in reattaching the PMO) from an invalid access (that may be triggered by the attacker).

Both Outermost and FCFS also suffer from multi-thread composability concerns.

C. EW-Conscious Semantics

Based on lessons learned from the previous three semantics, we propose exposure window-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.

Semantic Description: Within a thread, no overlap of attach-detach pairs is allowed. Attach-detach pairs across different threads[1] may overlap. At an *attach* call, real attach (address mapping) happens if and only if the PMO is not yet attached. Otherwise, the call lowers to a thread-level TERP operation which opens the access permission of the calling thread to the PMO. At a *detach* call, real detach happens if and only if (i) the time span from the most recent real attach (by whatever thread) exceeds a predefined constant L (e.g., a value near the target exposure window size) and at the same time (ii) no other threads can access this PMO. Otherwise, the detach lowers to a thread-level TERP operation which closes the access permission of the calling thread to the PMO.

[1]Unless stated otherwise, all threads in discussion are in the same process.

Example: An example of the semantics is shown in Figure 4. 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. The example also illustrates that the semantics creates a distinction between *process-wide exposure window* (EW) versus *thread-level exposure window* (TEW).

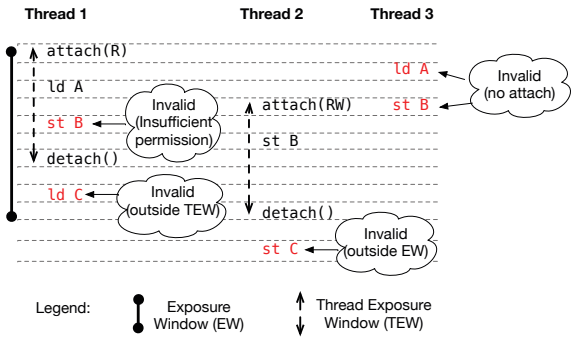

Figure 4. An example of EW-conscious semantics.

Composability and Security: The EW-conscious semantics achieves high thread composability by allowing each thread to focus on its own attach and detach calls regardless of what other threads do. It benefits from the implicit lowering to thread permission controls. Its function composability is good for the non-overlapping requirement within a thread. Thread-level permission controls ensure that the thread exposure window size meets the requirement as long as the attach-detach pairs are inserted appropriately within each thread. The time span-based real detaches add another layer of process-level protection.

The benefits of the temporal protection show up when combined with other protections. For instance, a PMO will not reside on the same address for a time longer than an exposure window if it combines with address layout randomization [2]. The randomization just needs to augment the TERP protection by (1) mapping a PMO to a random address at every real attach, and (2) remapping the PMO

[2]It is assumed that accesses to a PMO are through relocatable PMO APIs [23], [24].

to a different address at the detaches where condition (i) in the semantics description holds but condition (ii) does not. The latter ensures that even if the PMO is never completely detached due to window overlapping across threads, it still gets randomized periodically.

V. Adoption Challenge II: Ease and Efficiency

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

A. 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. After the automatic insertion, we employ architecture support to reduce the overheads at runtime, which is described later in the paper.

Figure 5(a) illustrates our static analysis approach. Each node in the *control flow graph* (CFG) represents a *basic block* (BB) and each edge represents a control flow edge. Gray nodes represent BBs that contain PMO accesses. From the starting point to the end point, the naive path-sensitive insertion needs to consider all 32 paths for inserting attach() and detach() calls appropriately. The reason why so many paths need to be analyzed is that the PMO states may be different at a confluence point for different paths.

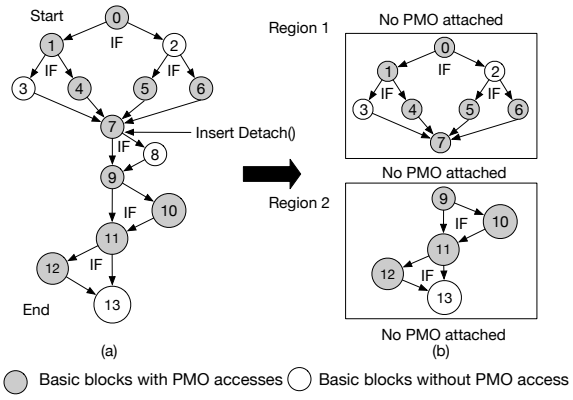

Figure 5. Control flow graph (CFG) illustrating basic path-sensitive insertion (a) vs. our *localized path-sensitive insertion* (b).

In order to reduce the static analysis complexity, 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 CFG 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 can be inserted to each path so that attach and detach pairs are maintained. We refer to this approach as *localized path-sensitive insertion*. In Figure 5(b), the algorithm first inserts a detach call to the exit of BB7, and then independently works on each of the two regions separated by BB7 (BB8 is ignored as it does not have PMO accesses).

The algorithm works on a data structure called PMO window flow graph (PMO-WFG), which is a set of subgraphs of the program CFG, covering all BBs with PMO accesses. Specifically, each subgraph is a code region (R), which meet the following three conditions: 1) There is a header in R that dominates all BBs in it; 2) A BB post-dominates all nodes in R; 3) The longest execution time (LET) in all paths of R is less than a threshold (preset based on the target maximum exposure window size). If the total iterations of a loop cannot be decided in static, we follow the common practice in static analysis to assume it to be a large number (e.g., 1k) to estimate LET. Sec VII will show that the assumption does not cause oversized exposure windows thanks to a timer-based measure in hardware (next section) and the fact that a loop always forms a code region with *attach* added at the confluence point. With the PMO-WFG, we can independently perform path-sensitive insertion in each region in the PMO-WFG.

Algorithm 1 PMO-WFG construction and attach()/detach() insertion.

```
1: Build CFG and the hierarchy of regions
2: Compute longest execution time (LET) of each region
3: Identify BBs with PMO accesses, set them as unvisited.
4: for each unvisited basic block B with PMO accesses do
5:     create an empty graph g and add B into it
6:     while LET of the next-level region NLR < threshold do
7:         if NLR covers unvisited BBs then
8:             add NLR into g
9:     put g into PMO-WFG
10:    Change the status of all blocks visited by g
11: for each graph in PMO-WFG do
12:    if thread_exposure_window ≠ 0 then
13:        Perform path-sensitive insertion in this graph
14:    else
15:        Insert attach() and detach() at the entrance and the exit in this graph
```

The algorithm of PMO-WFG construction and attach/detach insertion is outlined in Algorithm 1. Initially, we construct the CFG of the program and build the hierarchy of regions by the classic code region analysis. With a conservative cycles per instruction, we estimate the longest execution time (LET) of all paths in each region from lower-level regions to higher-level regions, and the higher-level region computations can reuse LETs of the lower-level regions. Pointer analysis is used to identify BBs with PMO accesses and pointer aliases. An unvisited array of these BBs is initialized for the next step. Thread exposure window is a

configurable value to control the frequency to insert attach() and detach().

B. Efficiency via Architecture Support

A main challenge of using TERP is the overhead by frequent attach and detach calls inserted by the compiler conservatively to limit the EW and TEW in a region. A call, if fully performed as a system call, involves overhead similar to a context switch, including saving and restoring process state, switches between user and kernel modes, pipeline flushes, and other microarchitecture costs. Our architecture support seeks to reduce these overhead.

There are two opportunities for performance improvement that we explore. 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*.

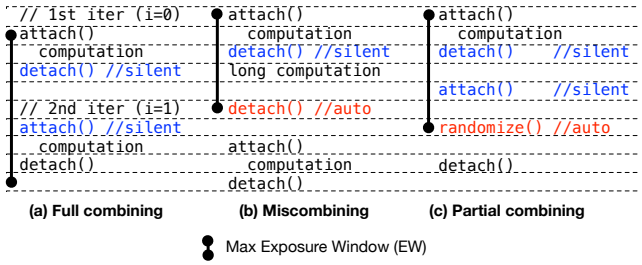

Figure 6. Illustrating three window combining cases.

The window combining may encounter three cases illustrated in Figure 6. In part (a), the first detach() is encountered long before reaching the maximum EW, hence it is delayed (*silent*). Later, the second attach() is encountered and because the PMO was not detached, the attach() is also converted to silent. The result is having a larger window that combines two (or more) smaller EWs, without exceeding the target maximum EW. In part (b), due to long computation after the first silent detach(), the maximum EW is met before the next EW, hence the system automatically triggers detach() to close the current EW. In part (c), two EWs are combined into one, but the maximum EW is encountered before the EW ends. In this case, the system automatically triggers PMO randomization in the mean time before the upcoming detach() is performed. In this way, even though the combined EWs may slightly exceed the maximum EW, the PMO stays in one virtual address location not any longer than the maximum EW. This partial combining case is crucial for multi-threaded scenarios where the combination of EWs from multiple threads may create a combined EW that exceeds the maximum EW, even though the compiler has ensured that each thread's EW is short.

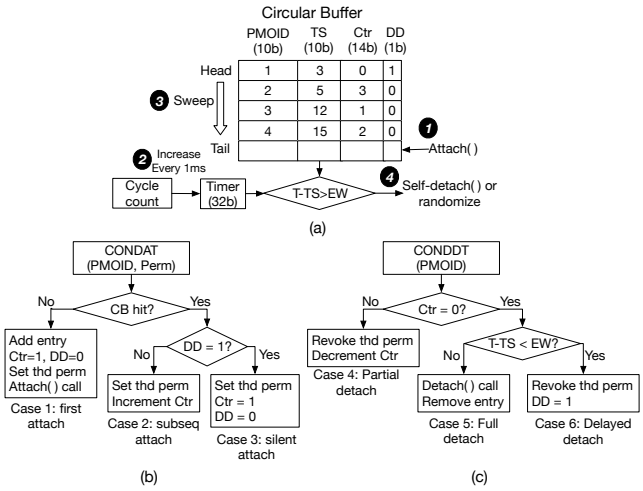

Figure 7. Circular buffer (a), and the logic for conditional attach (b) and conditional detach (c).

To support window combining, our architecture first assumes that each attached PMO is assigned its own protection domain using support such as Intel MPK [20], which allows per-thread access control. We also assume fast attach and detach using embedded PMO page table and randomization using PMO space layout randomization technique from [5]. Next, we describe the support we propose.

For *window combining*, we use a circular buffer shown in Figure 7(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 ❶. A timer is incremented at a coarse granularity, such as every 1us ❷. Periodically, we sweep the buffer from head to tail ❸, 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 ❹. If the counter is zero (indicating no thread still works on the PMO), 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). In the example in Figure 7(a), if current time is 15 and maximum EW is 10, PMO1 will be detached ($Ctr = 0, DD = 1$) but PMO2 will be randomized ($Ctr > 0$). PMO3 and PMO4 are left alone as maximum EW has not been reached.

To support *conditional attach/detach call*, there are several possible mechanisms. One design is to register the PC addresses of attach and detach system calls in special registers. When the program counter points to any of them, the hardware intercepts it and directs the instruction fetch only if a certain condition is met. An alternative design is to introduce conditional attach and detach instructions that if they lead to a certain condition at execution, an attach or detach system call is made. The first design avoids modifications to the ISA but adds special watch registers. Either design is equally possible; however for simpler illustration, we assume the second design. 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 7(b) and (c) show how the two instructions are executed. For `CONDAT`, if the PMO is not found in the circular buffer (CB), we allocate an entry, initialize its Ctr to 1 and reset the DD. The thread permission is also initialized as requested. An attach() system call is made to map the PMO to address space since this is the first attach attempt (Case 1). If the PMO is found in the CB, we check the DD. DD=0 indicates subsequent attach attempts by other threads (Case 2). In this case, we set the thread permission appropriately and increment Ctr; system call is skipped. Finally, if DD is set (Case 3), the PMO was in a delayed detach state due to window combining. Since the PMO was not actually detached yet, we simply reset DD, set Ctr to 1, and set the thread permission. In this case, we have elided a pair of detach and attach system calls, resulting in substantial performance savings.

For `CONDDT`, we check whether Ctr is zero. If not, this is not the last thread to detach it, hence all we need to do is to revoke thread permission and decrement Ctr (Case 4). Otherwise, we check if EW has been met or exceeded. If exceeded (Case 5), detach system call is invoked. Otherwise (Case 6), we do not need to detach the PMO at the moment, and we can simply set the delayed detach status (DD bit) and revoke thread permission. The PMO will later be automatically detached through sweeping when EW is met/exceeded, or if an attach call is made before then, we turn the attach() silent and combine the EWs.

The hardware cost is mainly due to the addition of the circular buffer. It contains 32 entries and each entry is 34-bit in size. We use Cacti [34] to evaluate the die area needs with the 45nm Nehalem processor [35]. The total on-chip space introduced is 140 bytes and consumes a tiny 0.006% of the die area.

VI. Evaluation Methodology

Workloads: To measure TERP performance on real-world persistent applications, we use WHISPER benchmarks [36]. WHISPER includes key-value stores Echo and Redis, a database YCSB, a transaction processing benchmark TPCC, and two data structures, hashmap and ctree. All these benchmarks use a single PMO. We execute 100K transactions or operations over a 1GB PMO with a single thread.

To measure multi-threaded performance on multiple PMOs, we use all SPEC2017 [37] applications that are written in C/C++ parallelized using OpenMP directives. We allocate each heap object larger than 128KB as a PMO, to evaluate a multi-PMO scenario. These parallel benchmarks run on the simulated multi-core system with test input size.

PMO only stores heap data in these benchmarks. All other data is allocated in DRAM, including all stack variables, non-persistent data in WHISPER, and small objects in SPEC2017.

Simulation: Our simulator is built on Sniper [38], a cycle-accurate X86 simulator. We implement an LLVM pass [39] that uses the region-based analysis in LLVM [40] to insert conditional instructions (magic instructions in Sniper), producing protected programs. (Section V-B).

Table II
Simulation parameters.

Processor	4-core, each 2.2 GHz, 4-way OoO, 128-entry ROB, x86-64 ISA, Pentium M branch predictor
Cache	private L1D cache, 8-way, 32KB, 1 cycle access time; Shared L2 cache: 16-way, 1MB, 8 cycles access time
Memory	DRAM latency: 120 cycles; NVM latency: 360 cycles; 64 GB/s Bandwidth;
TLB	L1 data TLB: 4KB pages, 4-way, 64 entries, 1 cycle; L2 TLB: 6-way, 1536 entries, 4 cycle; 30 cycles TLB miss penalty
Others	Permission matrix check/update: 1 cycle; Silent conditional attach/detach: 27 cycles Attach(): 4422 cycles, Detach(): 3058 cycles Randomization: 3718 cycles, TLB invalidation: 550 cycles

Table II shows our simulation parameters. PMO-related latencies are obtained from microbenchmarking on a real system. In the setting, permission matrix check adds one cycle after TLB lookup [5]. For silent attach/detach latencies, we use the average time to set Intel MPK permission, which includes memory fence overheads. For system call overheads, we implement them and microbenchmark them on a real machine. Then, we use the average overheads of these system calls in the simulation.

Configurations: We evaluate the following schemes: (1) MERR insertion and MERR architecture (MM), where programs execute with manually-inserted attach and detach that are executed fully as system calls with EW target of 40 us on the MERR architecture. (2) TERP insertion on MERR architecture (TM), where programs execute with automatic compiler-inserted attach and detach with EW target of 40us and additionally TEW target of 2us, on

MERR architecture. With TM, conditional attach/detach are fully executed as system calls. (3) TERP insertion and TERP architecture (TT), where programs execute with automatic compiler-inserted attach and detach with EW targets of 40us/80us/160us and TEW target of 2us, with TERP architecture support that includes window combining and conditional attach/detach. The rationale for the choice of EW and TEW is discussed in Section VII-A.

VII. Experiment Results

A. Exposure Window Size Selection

To select an appropriate thread exposure window (TEW) target for both security and performance considerations, we perform an empirical study while assuming a data-only attack pattern from a previous work [26]. In that threat model, one way to cause persistent corruption in a single object is to corrupt the content after the last write by the victim program, because the corruption persists until object deallocation. We call the time window *object dead time*. If earlier writes were selected instead, the corruption would be less effective as the object would be overwritten by the victim program later. Hence, the object dead time represents an important attack surface.

We measured eight SPEC 2017 benchmarks and five Heap Layer benchmarks [41], a benchmark suite with frequent allocations and deallocations. We report the distribution of dead times in Figure 8. From the figure, we can see that in 95% of the cases, the dead time is 2us or larger. So if we choose a target TEW of 2us, the attack surface would be reduced by 95%.

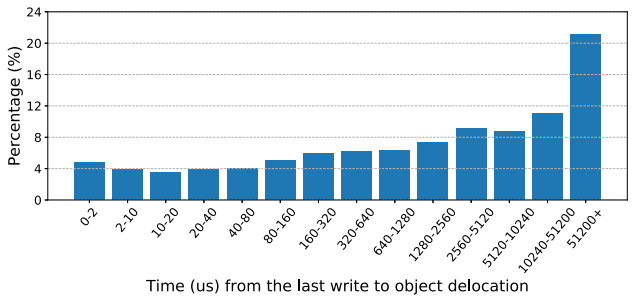

Figure 8. Distribution of attack surface for all heap objects, representing the time from last write to an object until the object deallocation.

The selection of an exposure window (EW) size is related with the time needed by memory attacks to discover the target addresses. Our experiments consider EW of 40us, 80us, 160us, as these sizes produce lower than 0.01% probabilities for the state-of-the-art ASLR breaking techniques [42], [43] to find a target address for a 1 GB PMO.

B. Single-PMO and Single-Thread Results on WHISPER

Results for WHISPER benchmarks with 40us EW are shown in Table III; TEW is set to 2us. For the MERR insertion on MERR architecture (MM) scheme, the table shows the average and maximum EWs, as well as the PMO exposure rate (ER) defined as the total exposed time divided by the total time. For the TERP insertion on TERP architecture (TT), the table additionally shows the number of conditional attach/detach per second (Cond. freq.), the percentage of conditional attaches that are lowered to thread permission changes (Silent), average TEW size, and the thread exposure rate (TER). ER and TER are calculated by dividing the sum of all EWs and TEWs by the total execution time.

As Table III shows, MM's EWs vary widely; the gap between average and maximum EWs is large, indicating that MERR is unable to achieve stable EW values. In contrast, TERP's automatic attach/detach insertion and window combining achieve stable EW values close to the target of 40us. Furthermore, thanks to the new TEW concept introduced by TERP, the average TEW ranges from 0.7 to 1.5us, below the 2us target. The result is much stronger security protection in TERP, where protection is provided by TEW of less than 2us instead of the much higher EWs in MERR. Overall, TERP reduces exposure window size by 92% (14.5us to 1.2us) and exposure rate by 86% (24.5% to 3.4%).

Figure 9 shows the execution time overheads of MM, TM, and TT over the unprotected executions. The overhead is broken down into those from attach and detach system calls, re-randomization (Rand), execution of conditional attach/detach instructions (Cond) and others (e.g., permission matrix). The figure shows that for the 40us EW target, TERP's stronger security protection does not result in a performance penalty; in fact TERP reduces execution time overheads by 70% over MERR (20% to 6%). Table III shows why TERP is so effective: the Silent column shows that nearly 9 out of 10 system calls are eliminated due to the conditional attach/detach. The overheads of TM is actually 50% higher than MM, indicating that without TERP architecture, managing thread exposure windows adds substantial overheads.

Table III
WHISPER results with target EW as 40us

(**EW:** exposure window size; **ER:** Time(exposed)/Time(all); **TEW & TER:** thread-level EW & ER)

Prog.	MERR (MM)		TERP (TT)				
	EW (us) avg/max	ER (%)	Silent (%)	EW (us) avg/max	ER (%)	TEW (us)	TER (%)
Echo	17.3/33.5	14.1	90.4	39.2/40.0	25.1	1.5	3.3
YCSB	13.1/38.1	28.1	87.3	39.4/40.0	66.8	0.9	5.6
TPCC	11.2/32.5	31.1	92.5	39.7/40.0	77.2	0.7	2.3
ctree	16.3/39.4	22.2	80.1	38.9/40.0	33.9	1.8	4.2
hash	19.7/37.2	19.2	91.2	39.5/40.0	34.7	0.9	5.2
Redis	8.1/25.1	32.5	91.1	39.5/40.0	81.7	1.1	3.3
Avg.	14.5/34.3	24.5	88.8	39.4/40.0	53.2	1.2	3.4

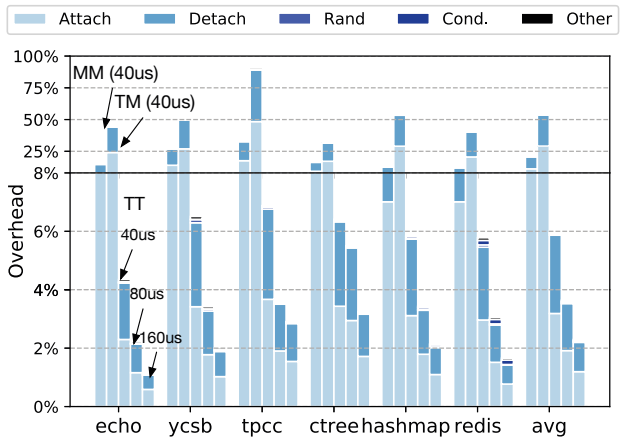

Figure 9. Overhead on WHISPER with different EWs and 2us TEW

Table IV
SPEC RESULTS ON 40US EW (AVG OVER ALL PMOS).

Prog.	# of PMOs	MERR (MM) EW(us) avg/max	MERR (MM) ER (%)	TERP (TT) Silent (%)	TERP (TT) EW(us) avg/max	TERP (TT) ER (%)	TERP (TT) TEW (us)	TERP (TT) TER (%)
mcf	4	4.5/25.1	12.9	97.2	39.6/40.0	27.5	0.7	4.9
lbm	2	1.1/17.1	49.6	98.7	39.7/40.0	67.5	0.5	14.9
imag.	3	3.4/28.6	28.4	96.7	39.7/40.0	33.5	0.9	11.2
nab	3	2.4/18.9	37.1	96.2	39.9/40.0	45.3	1.1	17.3
xz	6	10.4/37.5	8.1	95.2	39.8/40.0	16.7	1.9	1.8
Avg.	3.6	4.4/25.4	27.2	96.8	39.7/40.0	38.1	1.02	10.0

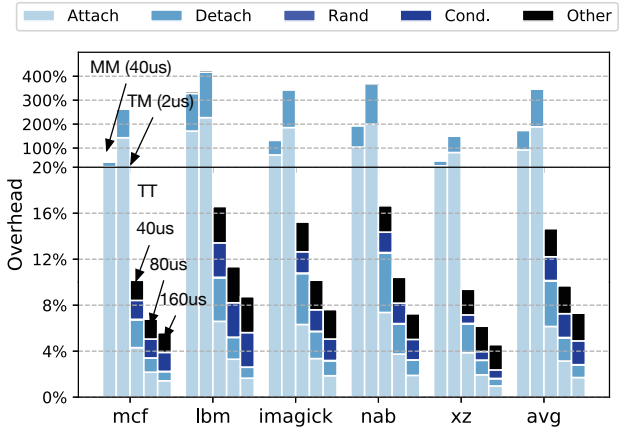

Figure 10. Single-thread multi-PMO execution time overheads for SPEC.

C. Multi-PMO, Single and Multi-Thread Results on SPEC

Multi-PMO single-thread results for SPEC on 40us EW are shown in Table IV, and wall-clock execution time overheads for 40us, 80us, and 160us EWs are shown in Figure 10. The values are averaged over all PMOs. Compared to WHISPER, SPEC benchmarks have multiple PMOs (last column) and PMO accesses make up for a large fraction of total accesses (vs. a small fraction in WHISPER). 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%. Due to the much higher fraction of PMO accesses, the overheads from the execution of conditional instructions and permission matrix checking are higher with SPEC. However, for most benchmarks, performance overheads are not much affected by the number of PMOs, because typically only 1 or 2 PMOs are actively used at a given time. 619.lbm is an exception as it actively uses two PMOs during most of its execution, leading to the highest overheads among the benchmarks. In terms of security, the higher the PMO count, the lower the exposure rates, since a program uses different PMOs in different computation stages. For example, 657.xz, which has the highest PMO count (6), enjoys the lowest exposure rate (16.7%).

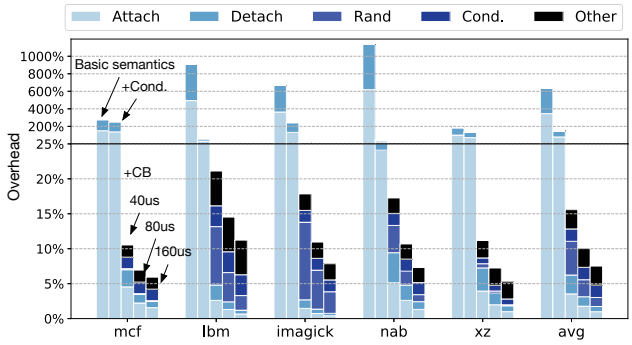

Figure 11. 4-thread SPEC results over different EWs.

Benefits Breakdown. The multi-PMO and multi-threaded SPEC (4 threads) results for TERP and each optimization are shown in Figure 11. The "basic semantic" bars show the overhead of TERP when it uses the basic semantics in MERR; "+Cond" add the conditional instruction optimization only, and "+CB" adds the circular buffer optimization. The basic semantic incurs very high overheads, because with it, at most one thread can attach a PMO. If other threads execute the attach() on an already-attached PMO, they need to wait until this PMO is detached. Using conditional instruction supports EW-Conscious semantics to allow multiple threads to use a PMO simultaneously. Circular buffer further performs window combining to reduce overheads. Both optimizations reduce the overhead significantly. Compared

Table V
QUANTITATIVE COMPARISON

	MERR (40usEW)			TERP (40usEW,2usTEW)		
	Success Probability (%)					
Each Attack Time	x us	1us	0.1us	x us	1us	0.1us
Stack Buffer Overflow [47]	0.015/x	0.015	0.15	0.0005/x	0.0005	0.005
Heap Overflow [48], [49]						
Format String [50], [51]						
Integer Overflow [52]						

to the single-thread case, the overhead of randomization is somewhat higher due to the overhead from suspending all threads when TLBs are invalidated.

D. Security Analysis and Case Study

TERP introduces three possible PMO data states for a thread: 1) detached, 2) attached without thread access permission, and 3) attached with thread permission. In the detached state, even attacks that exploit flaws in virtual memory implementation cannot succeed (e.g. Spectre [44], [45], Meltdown [46]) due to non-existent mapping. In the second and the third states, through short TEWs and randomization, TERP provides probabilistic security protection against memory disclosure and corruption on PMOs.

To provide quantitative analysis of TERP and MERR, we follow the effectiveness analysis [8], [43] of randomization and each attack time. We define the success of this attack as corrupting one value in the PMO either directly (e.g., through format string) or through corrupting local pointers that point to the PMO (e.g., through stack buffer overflow). Each attack time is represented as x, which equals 1us [42], [43] or 0.1us in our experiments.

The quantitative comparison between MERR and TERP for 1GB PMO is shown in Table V. Assuming each attack time is x us, during a 40us EW, the attacker can probe 40/x position out of 18-bit (1GB PMO) entropy. The probability of success is therefore $(0.015/x)\%$ in MERR. In contrast, with thread-level protection, during an EW, the malicious thread only has the permission to access this PMO during a small fraction (3.4% in WHISPER) of an EW. The probability of success is therefore $(0.0005/x)\%$ in TERP, which is $30\times$ smaller than MERR. Meanwhile, each attack time must be smaller than the TEW (about 2us in TERP) as it needs the permission to the PMO during the attack.

To provide deeper insights into TERP security strength, we discuss it in the context of an important class of attacks: *data-only attacks*. The rationale for this is that while many other attacks focus on overwriting pointers, data-only attacks work by overwriting some local variables to reuse the victim program to perform malicious computation with respect to the control-flow graphs. Figure 12 shows an example of data-only attack from [9]: part (a) shows a vulnerable code section from an FTP server while part (b) shows an example attack goal. The code in part (a) processes network requests based on message types, truncates the "STREAM" (line 6), maintains total received bytes (line 10) and throttles user requests to a maximum limit (line 3). Let us assume there is a buffer overflow vulnerability in function readData (line 4) which fails to check the bounds of buf. The buffer overflow vulnerability allows the attackers to control the values of all local variables.

```
1   struct server{int *cur_max, total, typ;}*srv; int *size, *type;
2   int cnt=MAX; char buf[MAXLEN]; size = &buf[8]; type = &buf[12];
3   while (cnt--) {
4     readData(socked, but);              //buffer overflow
5     if(*type == NONE) break;
6     if(*type == "STREAM")               //condition
7         *size = *(srv->cur_max);        //dereference
8     else {
9         srv->typ = *type;               //assignment
10        srv->total += *size;            //addition
11    }
12  }
```

(a) Vulnerable FTP server code

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  for(; list != NULL; list=list->next)
3     list->prop += value;
```

(b) The Attack goal: Increase the target linked list value by a given value

	Overflow	Executed Instr.	Simulated Instr.

Odd Rounds	type<-&list
	size<-&value
	srv<-&srv - 8
		if(*type==NONE) break;
		srv->typ = *type;
		srv->total += *size;
			if(list==NULL) break;
			srv=list;
			list->prop+= value;

Even Rounds	type<- &STREAM
	size<-&list
	size<-&list
		if(*type==NONE) break;
		if(*type == STREAM)
		*size = *(srv->cur_max);
			if(list==NULL) break;
			if(list==STREAM)
			list=list->next;

(c) Simulate the attack goal by corrupting local variables

Figure 12. Data-only Attack Example [9]

Suppose that the attack goal is to execute the code in Figure 12 (b) which increases a target linked list content by a given value. The underlined code in part (a) achieves this goal by corrupting local variables. Consider the following operations in the program. In line 9 is an assignment operation on two memory locations by two local variables (srv and type), which are under the control of attackers. Similarly, line 7 has a controllable dereference operation while line 10 has a controllable addition operation. These individual operations in the program are *data-only gadgets*. By creating attacker-controlled inputs and chaining them in a sequence, a meaningful computation can be executed. The loop in line 3 allows chaining and dispatching gadgets in an infinite sequence, since the loop condition variable is also controllable. Such loops are called *gadget dispatchers*.

Figure 12 (c) shows that by corrupting local per round (loop iteration), the code performs linked list node addition operation in odd rounds and performs pointer moving forward operation in even rounds. The attacker obtains the fine-grained control; the attack goal is achieved.

Through TEW and timely randomization, TERP mitigates data disclosure and corruptions on PMOs from this kind of attacks. As in the example, the vulnerable code needs to have access permission to the sensitive data list and obtain the address of this sensitive data to perform attacks. Isolation techniques allow programmers to only open permission before accessing sensitive data and close permission

after accessing, reducing the attack surface. PMO layout randomization hinders the attackers by requiring extra time in inferring the target memory address. Data-only attacks also need (or at least prefer) the target address to remain the same in all rounds. Otherwise, deployed corruption of this round cannot perform computation or link the previous round and the next round.

Table VI
SECURITY ANALYSIS OF DIFFERENT ATTACK SCENARIOS.

Capability of attacks	Relationship between gadgets and attach-detach pairs		
	No overlap	Gadgets within an attach-detach pair	Gadgets include an attach-detach pair
One arbitrary read or write	Prevented by the permission	Hindered by EW and address randomization	Same as the left
An infinite loop that includes several arbitrary reads or writes	Prevent 96.6% gadgets in WHISPER Prevent 89.98% gadgets in SPEC	1) Interactive data-only attacks are impossible. 2) Non-interactive data-only attacks need complicated mechanisms. 3) State-of-art address probing: 0.01% chance of finding the address.	Hindered by EW and address randomization 1) Accumulated probability. 2) Each attack session is limited to EW size.

Table VI analyzes several kinds of attack scenarios. We categorize the relationship between gadgets and attach-detach pairs into three cases. To be more specific, in Figure 12 in Section VII-D, the three cases correspond to the three relationships between the vulnerable code and the attach-detach pairs of the sensitive data `list`. The partial overlap case correspond to the scenario, "gadgets within an attach-detach pair". In this scenario, there are two subcases depending on whether this arbitrary read/write gadget can happen in an infinite loop corrupted by attackers. For both subcases, our protection can disarm about 96.6% gadgets in WHISPER and 89.98% gadgets in SPEC by limiting gadgets capability. To see that, first note that only code regions with thread permission are able to access a PMO. Gadgets outside those regions cannot be used to perform read/write to any PMO. For the remaining gadgets within attach-detach pairs, they have the permission to access one PMO. However, data-only attacks require finding the target PMO virtual address within a 40us window, which is difficult to achieve. For interactive data-only attacks, network latencies (ms level) are much larger than EW (40us). So, a probing result would become useless when attackers obtain it, since the victim program would have already gotten out of the gadget and the PMO address re-randomized. Non-interactive data-only attacks need complicated steps to handle the probing results. If attackers use the state-of-art address probing method [42], they have only 0.01% probability to find the PMO address during an EW. For gadgets that include an attach-detach pair used by an infinite loop, attackers may accumulate the chances to successfully launch the attack. However, the attackers still need complicated steps to handle probing results and subsequent attack steps within an EW window. Therefore, the success probability of these attacks is very low. Comparatively, MERR keeps 24.5% gadgets in WHISPER and 27.2% gadgets in SPEC and gives less stable randomization frequencies, incurring much larger overhead than TERP does.

VIII. RELATED WORK

The possibility of data-only attacks had been identified before [53], [54]. Hu *et al*. [9] introduced a tool to chain or stitch together data-flows to generate data-oriented attacks on Linux and Windows binaries. They further showed the Turing-complete property of such attacks and introduced Data-Oriented Programming (DOP). Ispoglou *et al.* [26] extended DOP to Block-Oriented Programming (BOP), which would automatically locate dispatching basic blocks to construct a successful attack. Restricting access reduces risk to DOP/BOP, which is the principle that TERP uses to protect PMOs.

Our work is the first one to address the vulnerabilities of PMOs to data corruption through a novel TERP framework, and define semantics of protection constructs. For traditional volatile memory, many security mechanisms were proposed to improve code and data security, including software-fault isolation (SFI) [55], [56], ISboxing [57], heterogeneous isolated execution [58], probe step detection [59], data Shield [60], virtualization to keep sensitive information in disjoint memory [14]–[16], [61].

Randomization techniques have long been used to obfuscate address space layout. These include ASLR [62], Enhanced ASLR [43], TASR [30] and Runtime ASLR [63]. Shuffler [64] and Morpheus [65] augment ASLR by adding encryption. There is broad recognition that re-randomization is required because memory disclosure can be used to defeat randomization over time. Such works are orthogonal to TERP and can be used in combination with TERP to improve general security.

Several studies proposed to maintain crash consistency and improve the performance of encryption in PM [66]–[68]. Some work proposed faster Merkle tree mechanisms to verify the integrity of PM [69]–[72]. These studies assume a different threat model where system software is not trustable and physical attacks are possible, but the programs are trusted. In contrast, we are more concerned about attacks that arise from memory safety due to vulnerabilities in programs. Hardware-based memory protection key (MPK) virtualization [21] has the same threat model as our work and proposes to use intra-process isolation to improve PM security. This paper only focuses on the hardware design part but does not address non-trivial programming challenges to use intra-process isolation in applications. Our work could guide future MPK programming support to ease the programming burden.

IX. Conclusion

In this paper, we have shown that the PMO model suffers from persistent corruption vulnerabilities, and requires a protection framework that can systematically reduce PMO exposure to attacks relying on memory corruption. We discussed our proposal called TERP, which provides semantics of attach/detach that is secure and composable in terms of function and multi-threaded aspects. We then presented a novel compiler-architecture design that addressed key TERP challenges. The compiler allows the automatic insertion of TERP constructs into code, while the architecture support reduces performance overheads associated with frequent attach and detach calls. Our experiments on both sequential and parallel programs demonstrate TERP's effectiveness in achieving security at low overheads.

Acknowledgment

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. 1900724, 2106629, 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 or ONR.

References

[1] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[2] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro *et al.*, "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, 2007, pp. 480–617.

[3] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[4] 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.

[5] 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.

[6] 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.

[7] H. Shacham *et al.*, "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,, 2007, pp. 552–561.

[8] 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.

[9] 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.

[10] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[11] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1153–1166.

[12] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[13] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.

[14] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1607–1619.

[15] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 437–452.

[16] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 584–598.

[17] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 69–89.

[18] Y. Wang, Q. Li, Z. Chen, P. Zhang, and G. Zhang, "Shapeshifter: Intelligence-driven data plane randomization resilient to data-oriented programming attacks," *Computers & Security*, vol. 89, p. 101679, 2020.

[19] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables." in *NDSS*, 2017.

[20] Intel, "Intel 64 and ia-32 architectures software developer's manual." https://software.intel.com/en-us/articles/intel-sdm, online; accessed August, 2020.

[21] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 680–692.

[22] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[23] A. R. Intel, "Persistent memory programming," http://pmem.io/.

[24] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 800–812.

[25] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, "Supporting legacy libraries on non-volatile memory: a user-transparent approach," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 443–455.

[26] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1868–1882.

[27] M. Abadi and M. Budiu, "lfar erlingsson, and j. ligatti. control-flow integrity," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2005.

[28] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[29] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 147–163.

[30] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 268–279.

[31] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 763–780.

[32] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[33] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "{ERIM}: Secure, efficient in-process isolation with protection keys ({MPK})," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1221–1238.

[34] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[35] "Nehalem," https://en.wikichip.org/wiki/intel/microarchitectures/nehalem_(client).

[36] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 135–148.

[37] Intel, "Spec cpu 2017." https://www.spec.org/cpu2017/, online; accessed August, 2020.

[38] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.

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

[40] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[41] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing high-performance memory allocators," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 114–124, 2001.

[42] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 380–392.

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

[44] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[45] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[46] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[47] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[48] C.-. 2019, "Cwe-122: Heap-based buffer overflow," https://cwe.mitre.org/data/definitions/122.html.

[49] Y. Huang, "Heap overflows and double-free attacks," *Retrieved April*, vol. 6, p. 2018, 2016.

[50] O. 2015., "Format string attack." https://www.owasp.org/index.php/Format_string_attack.

[51] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: practice and experience*, vol. 33, no. 5, pp. 423–460, 2003.

[52] D. Ahmad, "The rising threat of vulnerabilities due to integer errors," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 77–82, 2003.

[53] W. D. Young and J. McHugh, "Coding for a believable specification to implementation mapping," in *1987 IEEE Symposium on Security and Privacy*. IEEE, 1987, pp. 140–140.

[54] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX Security Symposium*, vol. 5, 2005.

[55] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

[56] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[57] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 555–566.

[58] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 455–468.

[59] K. Bhat, E. Van Der Kouwe, H. Bos, and C. Giuffrida, "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, 2019, pp. 545–558.

[60] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 193–204.

[61] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "Seimi: Efficient and secure smap-enabled intra-process memory isolation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 592–607.

[62] P. T. 2003, "Pax address space layout randomization (aslr)." http://pax.grsecurity.net/docs/aslr.txt.

[63] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make aslr win the clone wars: Runtime re-randomization." in *NDSS*, 2016.

[64] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 367–382.

[65] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco *et al.*, "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, 2019, pp. 469–484.

[66] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

[67] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 33–44, 2015.

[68] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 479–492.

[69] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 104–115.

[70] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 157–168.

[71] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Streamlining integrity tree updates for secure persistent non-volatile memory," *arXiv preprint arXiv:2003.04693*, 2020.

[72] A. Freij, H. Zhou, and Y. Solihin, "Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1227–1240.