SafePM: Memory Safety for Persistent Memory

Paper #206

Abstract

Memory safety violation is a major root cause of reliability and security issues in software systems. Byte-addressable persistent memory (PM), just like its volatile counterpart, is also susceptible to memory safety violations, e.g., object overflows and use-after-free bugs. While there is a couple of decades of work in ensuring memory safety for programs based on volatile memory, there exists no memory safety mechanism for PM—the existing approaches are incompatible since the PM programming model introduces a persistent pointer representation for persistent memory objects and allocators, where it is imperative to design a crash consistent safety mechanism.

We introduce SafePM, a memory safety mechanism that *transparently* and *comprehensively* detects both spatial and temporal memory safety violations for PM-based applications. SafePM's design builds on a shadow memory approach, and augments it with crash consistent data structures and system operations to ensure memory safety even across system reboots and crashes. We implement SafePM based on the AddressSanitizer compiler pass, and integrate it with the PM development kit (PMDK) runtime library. We evaluate SafePM across three dimensions: overheads, effectiveness, and crash consistency. SafePM overall incurs reasonable overheads while providing comprehensive memory safety, and has uncovered real-world bugs in the widely-used PMDK library.

1 Introduction

Performance-critical software systems are prominently written in low-level languages, such as C/C++. While these languages allow the programmer to explicitly control the application's memory, they can unfortunately lead to memory safety bugs, i.e., *illegal accesses to unintended memory regions* [24, 33, 77, 78, 85]. More specifically, memory safety violations are categorized as *spatial* and *temporal* errors. Spatial violations occur when an operation accesses a memory region outside its assigned boundaries (e.g., buffer overflows). Temporal violations are accesses to a memory object before its creation or after its deletion (e.g., dangling pointers).

This fundamental trade-off between performance and memory safety in the context of low-level unsafe languages manifests in the form of numerous dependability issues in software systems. For instance, among the well-known memory safety violations [1, 6, 39], three major projects: Chromium [8], Android [5] and Windows [4] report that 70 − 75% of their discovered issues are memory safety bugs. These safety violations are wide-spread—three out of ten most critical software weaknesses are memory safety issues [7]. More importantly, Szekeres et al. [77] illustrate that memory safety is the root cause of security issues in software systems.

Byte-addressable persistent memory (PM), similar to volatile memory, is also susceptible to memory safety violations. In particular, PM is a non-volatile storage medium, accessible at a byte granularity via `ld/st` instructions with access latencies close to DRAM [52]. PM content is directly mapped into an application's address space and is manipulated at a byte granularity through native pointers making PM programming prone to memory safety errors, especially in the context of memory unsafe languages (e.g., C, C++).

Over the last two decades, a range of hardware- and software-based memory safety approaches have been proposed to tackle the problem of memory safety for volatile memory (§ 7). These approaches are prominently designed based on *deterministic dynamic bounds-checking*, which relies on compile-time code instrumentation and enhances the original application's memory layout with memory safety metadata which is checked during the run-time upon each memory access.

While these memory safety approaches are extensively used in commercial software eco-systems for volatile memory-based applications, either during development [12, 14, 74] or production phases [3, 11, 21]—there exists a distinct research gap when we consider memory safety issues in the context of the emerging persistent memory technology. That is, there exists *no memory safety mechanism designed for PM-based applications written in unsafe languages*.

Importantly, the state-of-the-art memory safety approaches for volatile memory are insufficient for PM. Unlike volatile memory, which uses native volatile pointers, the PM programming model introduces a persistent pointer representation for its objects, and the persistent memory heap is handled using memory allocators designed for PM, which rely on persistent heap metadata [27, 47]. In addition, memory safety needs to be ensured for the recovery code paths designated to be executed after a crash. Therefore, we need to design a memory safety mechanism that simultaneously ensures the correctness and crash consistency of both the application's data and the memory safety metadata.

To address this problem, we propose SafePM, a memory safety mechanism for PM-based applications. More precisely, SafePM provides *comprehensive memory safety* by detecting both spatial and temporal memory safety bugs. It is *transparent* to the application, requiring no source code modifications. Lastly, SafePM ensures the *crash consistency* property while incurring reasonable performance overheads.

At a high-level, SafePM's design is based on a shadow memory approach (§ 2.2). It reserves a PM region (*persistent shadow memory*) where it places its memory safety metadata. This PM part stores the state — accessible or not — of each 8-byte chunk of PM and is mapped over a specific, calculated location of the virtual address space. We call this operation *overmap*. Thus, SafePM checks this overmapped PM region on every memory access during runtime. These checks are injected in the application through ASan's compiler pass. Further, SafePM ensures that each PM object is surrounded by guard regions (*red zones*) that are marked as inaccessible in the persistent shadow memory. Lastly, SafePM incorporates a runtime library that instruments the PM management functions to reflect their modifications in the persistent shadow memory region in a crash consistent manner.

We implement SafePM based on Intel's PMDK and AddressSanitizer (ASan) [74]. SafePM's persistent shadow memory follows the same binary format as that of ASan, and is stored as part of the persistent pool created by SafePM. SafePM keeps PMDK's memory layout intact, such that a persistent pool created by SafePM is a valid PMDK pool. When a persistent pool is opened, SafePM maps its persistent shadow memory over the relevant portion of the ASan's shadow memory. Further, SafePM's run-time library augments the functionality of PMDK's libpmemobj [47] PM management routines to incorporate the handling of the persistent memory safety metadata. Thus, the PM heap operations that modify the state of a PM range, transparently update the respective part of the persistent shadow memory. These operations allow SafePM to keep the ASan's compiler pass and optimizations intact and use them for memory checks, while supporting the programming interface and semantics of PMDK, without requiring modifications to an application's source code.

We evaluate SafePM along three dimensions: *(i)* performance and space overheads using PMDK's benchmark framework, *pmembench* [50] and a persistent KV store, *pmemkv* [46], *(ii)* effectiveness using the *RIPE* framework [81] and *(iii)* crash consistency, which we validate for *pmembench* using the established *pmemcheck* tool [20]. Our evaluation shows that SafePM offers the same memory safety guarantees for persistent memory as ASan provides for volatile memory with reasonable performance overheads, e.g., $1.20 - 2.62\times$ slowdown for the KV store, while ensuring crash consistency. Through SafePM we have also identified two memory safety bugs in the widely-used PMDK library.

Overall, our paper makes the following contributions.

- We present the design (§ 4) of SafePM, the first solution for comprehensive spatial and temporal memory safety for PMDK-based PM applications.
- Our design leverages a shadow memory approach transparently supporting the established PMDK programming model, thus allowing seamless integration into existing testing workflows and tool chains.
- We prototype (§ 5) and extensively evaluate SafePM across three metrics: overheads (§ 6.2, § 6.3, § 6.6), effectiveness (§ 6.4, § 6.7) and crash consistency (§ 6.5).

SafePM will be released as an open-source project along with the reported bugs in the PMDK library.

2 Background

2.1 Persistent Memory

PM is a byte-addressable, non-volatile memory type with low access latency. It is connected to the CPU via the memory bus and is accessible with `ld/st` instructions [69]. PM can be used in two distinct modes [23, 51]; *(i) memory* and *(ii) app-direct* mode. In the former, the PM device acts as a volatile memory extension, while in the latter, data persists in PM even when the system is powered off. In the app-direct mode, PM interfaces with the OS via PM-aware direct access file systems (DAX) [41, 56]. DAX eliminates the page cache from the I/O path and allows mmap(2) to establish direct mappings to PM [45]. Thus, PM content can be directly accessed as memory mapped files in an application's address space.

Persistent Memory Development Kit (PMDK). Intel has developed a collection of libraries in Persistent Memory Development Kit (PMDK) [16] to support and facilitate application development for PM. The included libraries cover a wide range of applications and provide different ways to manage PM, ranging from low-level primitives [13] to a persistent transactional object store [47] and a persistent key-value store [46].

libpmemobj. `libpmemobj` [47] is a core component of PMDK that implements software based transactions to provide support for atomic updates to PM data leveraging redo and undo logging. It exposes intuitive non-transactional as well as transactional APIs [42, 43] for PM memory management. `libpmemobj` organises PM in files, called PM *pools*, which are mapped into contiguous regions in the application's address space. PM pools contain a pool header, a section dedicated to the redo and undo logs required for transactions support, called *lanes*, and the persistent heap, which hosts the PM objects allocated by the application as well as heap metadata.

The kernel can map PM pools to different regions of the address space in different runs. To maintain consistent object references across restarts, `libpmemobj` introduces the concept of *persistent pointers*. This is a fat-pointer scheme, where each object is described by a 16B structure, called *PMEMoid*, containing a *pool_id* and an *offset* relative to the start of the pool. `libpmemobj` exposes the `pmemobj_direct()` function to construct the native pointer for an object, based on its offset within the pool and the virtual address where the pool is mapped.

2.2 Memory Safety

Memory safety bugs have been prominent for low-level unsafe languages with poor built-in memory protection capabilities, like C/C++. Memory safety bugs are classified as *spatial* and *temporal*. The former defines accesses beyond allocated memory regions, while the latter accesses to regions that have not yet been allocated or have been freed. These bugs constitute one of the main targets for attackers to get access to unintended memory regions and, thus, be able to hijack the control-flow or leak sensitive data [63, 78]. To prevent such vulnerabilities, memory accesses beyond allocated memory regions must be prevented. This requires a suitable instrumentation of an application to perform bound checks on memory accesses. Towards this direction, several memory safety techniques have been proposed based on software implementations [22, 31, 37, 70, 74] or on hardware modifications [32, 71, 82].

More specifically, the proposed approaches can be classified into three broad categories [65]:

(a) Trip-wire or shadow memory based approach. This approach uses a part of the available memory to store whether or not each fixed-size chunk of memory is addressable. This memory part is called *shadow memory*. Allocated memory regions are surrounded with guards, marked as inaccessible, allowing the detection of spatial violations [37, 38, 74].

(b) Object-based approach. Such approaches check all the pointer manipulations to ensure that the resulting pointer is not out-of-bounds with respect to the object it points to [22, 28, 29, 31, 32, 34, 73]. They track metadata on a per-object level.

(c) Pointer-based approach. Approaches of this category maintain the upper and lower bounds that a pointer is allowed to access and perform the bound checks on every memory access [53, 64, 66, 68, 76]. Metadata is kept on a pointer level.

2.3 ASan: a Shadow Memory-based Approach

Among these approaches, the shadow memory based approach, as adopted by AddressSanitizer (ASan) [74], is the most popular memory safety technique [77]. It is widely used at Google and other organizations to detect memory safety violations. ASan supports both GCC [10] and Clang/LLVM [9].

In particular, ASan reserves a part of the address space for *shadow memory*, where it keeps metadata indicating the state of the memory regions of an application, including the stack, heap and global variables. ASan updates the shadow memory whenever an object is created, freed, or moved. It surrounds objects with memory regions, called *red zones*, which are marked as inaccessible (poisoned) in the shadow memory. On each memory access, ASan checks if the requested address is addressable. These checks are added via ASan's compiler instrumentation module. Any access to an unallocated region or red zone is detected, usually resulting in the crash of the program before its state can be corrupted or sensitive information leaks. Additionally, to ensure temporal memory safety, ASan implements a quarantine zone for recently-freed objects which prevents their regions from being allocated for some time.

3 Overview

SafePM is an efficient tool that detects memory safety violations in PM applications developed with PMDK. SafePM leverages ASan to detect spatial and temporal memory safety violations within a PM pool without requiring any source code modifications. Our key insight is to benefit from the optimizations and the efficiency of ASan. Therefore, SafePM keeps ASan intact and adds metadata to PM pools to integrate with ASan. An overview of the modified PM pool design is shown in Figure 1. `libpmemobj` maps PM pools to the virtual address space. SafePM reserves a part of the PM pool heap for the *persistent shadow memory* (PSM) which maintains the memory safety metadata of the pool. The part of the pool corresponding to PSM is mapped over the ASan's shadow memory. We name this core operation of SafePM as *overmap*. Further, SafePM's memory allocator inserts and poisons the appropriate *red zones* in the same way ASan does for volatile memory while preserving the programming model of PMDK and the crash consistency of the data and metadata.

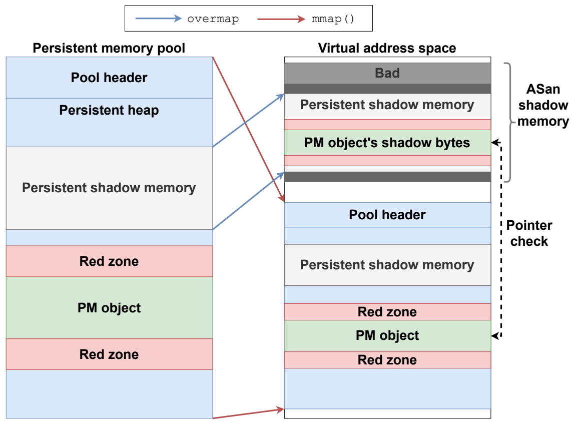

Figure 1. Overview of SafePM.

Design goals. SafePM achieves the following goals:

- *Memory safety:* SafePM provides memory safety for a wide range of potential PM access violations including both spatial (e.g., PM object-overflows) and temporal (e.g., use-after-free) in a similar manner as ASan does for volatile memory.
- *Transparency & compatibility:* We design SafePM based on PMDK and ASan which allows for seamless integration in existing PM applications without code modifications.
- *High code coverage:* SafePM should be able to detect memory safety violations in different code paths including recovery paths for abrupt shutdowns. This is achieved by ensuring the crash consistency for both PM (meta)data and SafePM's metadata leveraging the transactional interfaces and logging mechanisms provided by PMDK.
- *Performance:* SafePM keeps the compiler pass of ASan intact and leverages its optimizations to limit the introduced overheads due to the additional memory checks, making SafePM suitable for performance critical environments.

3.1 System Model

Fault model. SafePM aims to ensure memory safety for PMDK-based applications. It is capable of detecting and reporting both spatial (e.g., object overflows) and temporal (e.g., use-after-free) memory safety bugs in PM.

SafePM must also preserve the crash consistency property, as it targets PM-enabled applications. Crashes or unexpected system shutdowns can lead to data inconsistencies.

This means that SafePM has to enforce mechanisms to ensure the recovery to a consistent state not only for the PM (meta)data but also for the memory safety related metadata. Further, SafePM extends the scope to *also* provide memory safety guarantees for the recovery path.

Usage model. SafePM is a generic testing tool to prevent memory safety bugs in PMDK programs during the development phase. It provides the same memory safety guarantees as ASan. Further, it also offers "partial safety coverage" to manually select the code parts where the checks will be applied to limit the performance overhead.

Programming model. SafePM is based on PMDK which is entirely written in the unsafe C/C++ languages. SafePM preserves the programming model and semantics of PMDK.

3.2 Design Challenges

#1: Transparency. An effective memory safety testing tool should require no source code modifications. State-of-the-art approaches, like ASan [74] and memcheck [14], already cover this need with the instrumentation of the volatile memory management functions. SafePM should provide the same level of transparency for the PM pool heap management APIs.
Approach: SafePM preserves PMDK's programming model and instruments the PM management functions via carefully designed wrappers. Precisely, SafePM adapts the functions that manage the pool (create/open/close) and the PM objects (alloc/realloc/free) so that their changes are reflected in the PSM. SafePM supports PMDK's PM management APIs, requiring no source code modifications.

#2: Compatibility with ASan. ASan is one of the most prominent tools to detect memory safety violations in the volatile memory regions of an application. SafePM needs to extend ASan checks to objects residing in PM, which are managed by `libpmemobj`'s memory management functions.
Approach: SafePM reserves a shadow memory region inside the PM pools created with `libpmemobj`. This region corresponds to the respective shadow memory used by ASan. To comply with ASan's design, SafePM also augments the PMDK allocator to surround the PM objects with poisoned red zones.

#3: Crash consistency. PM applications have to ensure crash consistency for the PM data in case of an unexpected shutdown. In SafePM, the scope of crash consistency is extended to include memory safety metadata as well.
Approach: SafePM leverages PMDK's software transactions to ensure crash consistency of its memory safety metadata along with PM pool's (meta)data. Each PM object modification (alloc/realloc/free) needs to be reflected in the shadow memory representation. Since SafePM's metadata is part of the pool, its modifications are performed inside transactions along with the respective PM management operations, guaranteeing crash consistency via PMDK's logging mechanism.

```
1  struct my_obj { int src; int dest; } // object structure
2  ...
3  PMEMobjpool *pop = pmemobj_open(path); // open the PM Pool
4  init_shadow_memory(pop);
5  overmap_pool(pop);
6  ...
7  PMEMoid  obj_oid[N]; // declare the object handles
8  size_t size = sizeof(struct my_obj);
9  for (int i=0; i<N; i++) {
10     TX_BEGIN(pop){
11         pmemobj_alloc(pop,&obj_oid[i],size+2*RZ,...);
12         snapshot_and_set_shadow_memory();
13     } TX_END
14 }
15 ...
16 int val;
17 for (int i=0; i<M; i++) {
18     sh_src = get_shadow(&D_RO(object_oid[i])->src);
19     if (*sh_src != 0 && ...)
20         error(sh_src);
21     val = D_RO(object_oid[i])->src; //load from PM
22     sh_dest = get_shadow(&D_RW(object_oid[i])->dest);
23     if (*sh_dest != 0 && ...)
24         error(sh_dest);
25     D_RW(object_oid[i])->dest = val; //store to PM
26 }
27 ...
28 pmemobj_close(pop); //close the PM pool
29 unmap_shadow_mem();
```

Listing 1. SafePM code transformation: lines in blue are injected by SafePM's wrappers and lines in red by ASan.

#4: Coverage of recovery paths. PM applications are designed to be able to recover from abrupt crashes. This requires special code paths that restore the PM to a consistent state. SafePM has to ensure memory safety for these recovery paths.
Approach: SafePM maintains the memory safety metadata as part of the PM pool. This memory safety metadata remains consistent and can be retrieved across reboots or failures, unlike with ASan. Thus, SafePM leverages the durability of its metadata along with its crash consistency and can enforce memory safety on PM objects even on the recovery code paths.

4 SafePM Design

Figure 2 demonstrates an overview of SafePM's components. SafePM reserves part of the PM pool for the PSM and *overmaps* it to the location expected by ASan in the program's virtual address space. Moreover, SafePM augments the memory management operations of PMDK to surround allocated objects with poisoned *red zones* and update the PSM accordingly.

Listing 1 illustrates an application using SafePM, which opens a PM pool, allocates and accesses PM objects. The highlighted lines of code indicate the additional operations and checks inserted by SafePM (in blue) and ASan (in red). On line 3, the application opens an existing PM pool. SafePM transparently initializes the pool's PSM, if needed (line 4) and overmaps it on the relevant section of the ASan's shadow memory (line 5). Then, the application allocates *N* objects

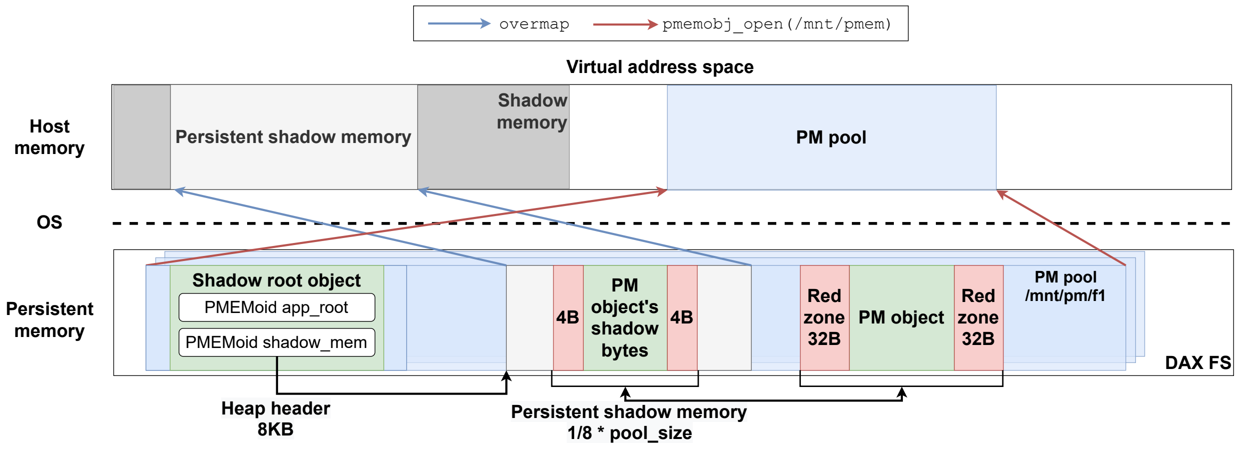

Figure 2. Detailed architecture of SafePM.

(line 11). Note that SafePM transparently converts each allocation to a transaction in order to ensure the crash consistency of its memory safety metadata (lines 10-13). The application then accesses the PM objects (lines 21 and 25) and ASan introduces the appropriate shadow memory checks (lines 18-20, 22-24). These checks get redirected to the overmapped PSM and leverage the memory safety metadata to ensure that the requested PM addresses are addressable. In case that any of these tests fails, an error is reported (e.g., if $M > N$). Finally, the PM pool is closed and SafePM unmaps the PSM (line 29).

4.1 Persistent Memory Safety Metadata

SafePM constructs persistent data structures to store the required memory safety metadata: *(i)* persistent shadow memory, *(ii)* persistent red zones and *(iii)* the shadow root object.

The persistent shadow memory. The central data structure of SafePM's design is the persistent shadow memory (PSM), which stores information about which PM pool regions are addressable. To be compatible with ASan's compiler pass, we use the same format for the PSM as the one used by ASan, which requires allocating one byte of PSM for every 8 bytes of a PM pool. Following the format of ASan for the (volatile) shadow memory, the PSM is an array of bytes, where each byte stores the number of accessible bytes for its 8 corresponding bytes, or 0 to mark them all accessible. For non-accessible 8-byte blocks, it can store why they are non-accessible, for example that they were freed or are part of a red zone. ASan's runtime library reserves $1/8th$ of the virtual address space for shadow memory. SafePM maps the PSM over ASan's shadow memory at the corresponding location that ASan uses for the mapped pool's virtual address range. Importantly, this *overmap* operation allows SafePM to leverage ASan's shadow memory checks without modifying its run-time library. By reserving a fixed region in the lower part of the virtual address space, the corresponding shadow memory address can be easily found with a simple translation, where offset is platform and OS-dependent:

```
#define GET_SM(addr) (void *)((long long)addr >> 3 + offset)
```

SafePM needs to persist the PSM data and ensure its crash consistency. To this end, SafePM creates PSM as a persistent object during the creation of a new persistent pool. The size of the PSM is at least $1/8th$ of the pool size requested by the application. Further, SafePM initializes the PSM as inaccessible. This ensures that the application code cannot manipulate any PM regions which are not explicitly allocated by the application, including PM pool's metadata and the PSM.

The persistent red zones. Similar to ASan, SafePM places red zones around PM objects. A red zone is a region of memory marked inaccessible (poisoned) in the shadow memory, which prevents user code from accessing it. This enables the checks inserted by ASan's compiler pass to detect out-of-bounds accesses. Persistent red zones are allocated on object (re)allocation. Upon an object deallocation, the red zones are removed along with the object, and are marked inaccessible in the PSM, providing temporal violation detection capabilities.

The red zone size constitutes a trade-off between safety and space efficiency. Large red zones waste space, while small red zones might fall short in detecting non-contiguous memory violations. For instance, in case that two objects are separated by a 16B red zone, SafePM will not detect under-/overflows of more than 16B as the problematic memory access might fall within another object's boundaries.

Shadow root object. PM pools contain a root object that is used as the reference point by the application to reach the other pool's objects. SafePM creates a shadow root object during the pool creation. It contains persistent pointers to the PSM and the user root object, as well as the size of the user root object. From `libpmemobj`'s perspective, the shadow root object is the root object of the PM pool. SafePM wrappers masquerade the shadow root object by returning the `app_root` field to the application when requested.

```
struct shadow_root {
 PMEMoid psm; //PMEMoid of the PSM
 PMEMoid app_root; //PMEMoid of the app's user root object
 uint64_t app_root_size; // size of the user root object
};
```

4.2 System Operations

In this section we describe the different operations of SafePM that manipulate the PSM and the red zones to transparently ensure both memory safety on PM and crash consistency.

PM pool creation. When a program calls the function `pmemobj_create`, SafePM's wrapper creates the PM pool, allocates and initializes the shadow root object as well as the PSM in a crash-consistent way. If the operation is torn after the pool is created but before the initialisation of the shadow root object and the PSM completes, the `pmemobj_open` wrapper will recover the persistent pool using the transaction capabilities of `libpmemobj` and recreate the PSM. During the creation of the pool, the PSM is initialized so that no region of PM is user-accessible, guaranteeing that an application cannot modify the pool's metadata, or access non-allocated PM regions.

```
PMEMobjpool pmemobj_create(path, size) {
   //create a pool with extra 1/8th of size for the PSM
   PMEMobjpool
     * pool = pmemobj_create_orig(path, size+size/8);
   //transactionally create the PSM , set to inaccessible
   PMEMoid sm_root = init_psm(pool);
   //mmap the PSM  to its designated region
   overmap_psm(sm_root);
   return pool;
}
```

Note that the PSM of a memory pool is stored alongside the data of the pool. The PSM can be located at an arbitrary position within the mapped PM pool. Thus, ASan's compiler pass won't be able to correctly map virtual addresses within the PM pool to the PSM. This would require changes to ASan's compiler pass, hampering the transparency property. As a workaround, after a PM pool is mapped to the virtual address space during `pmemobj_create`, SafePM overmaps the PSM region of the pool to the position determined by the virtual-address-to-shadow-address formula used by ASan. Figure 2 shows a schematic that visualizes this operation.

PM pool opening. `pmemobj_open` is called to open an existing PM pool . SafePM's wrappers check the pool's root object to determine if the shadow root object and the PSM are set up correctly. If it is the case, the creation of the PM pool was completed, and the PSM gets overmapped to the respective location in the virtual address space according to the memory location returned by PMDK's original `pmemobj_open`. Otherwise, the creation of the pool must have been torn and the transaction for creating and initializing the shadow root object and the PSM is executed again before the overmap operation.

```
PMEMobjpool pmemobj_open(path) {
   PMEMobjpool* pool = pmemobj_open_orig(path);
   // ensure the shadow root and PSM  are set up correctly
   recover(pool);
   overmap_psm(pool);
   return pool;
}
```

Memory management operations. PMDK supports different memory management operations to (re/de)allocate memory regions within a PM pool. We classify these operations into two distinct types: transactional operations (e.g., `pmemobj_tx_alloc`) that operate within a programmer-defined transaction and non-transactional operations (e.g., `pmemobj_alloc`) that do not require a transaction, but leverage atomic operations to ensure crash consistency.

Transactional PM management operations. The transactional memory management operations are executed within a transaction and use redo/undo logs to ensure crash consistency. SafePM builds on this and uses the memory snapshotting capabilities of `libpmemobj` to guarantee that the changes made to the PSM during a PM operation are atomic with respect to the changes made to the PM heap state, even in case of a torn operation or the abortion of a transaction. SafePM snapshots the respective PSM region, thereby adding it to the undo log of the current transaction, before performing any in-place updates to the PSM to demarcate the user-accessible regions.

The *transactional allocation* takes into account the size of the red zones and adds it to the object size requested by the user. Then, the relevant region of the PSM is snapshotted and the user-accessible region is marked as such, while the adjacent red zones are marked inaccessible. Note that the PMEMoid returned by PMDK's allocator indicates the offset at the left red zone. Therefore a simple translation is performed to return to the application the offset of the object's actual payload.

```
PMEMoid pmemobj_tx_alloc(size) {
   //allocate an object with extra space for the red zones
   PMEMoid *oid = pmemobj_tx_alloc_orig(size+2*RZ_SIZE);
   //get the corresponding address within PSM
   void *oid_psm = get_psm_address(oid);
   //add the existing PSM  region to the undo log
   snapshot(oid_psm, sm_size);
   //update the PSM  region with the correct values
   mark_addressable(oid_psm, size);
   //set the pointer returned to the start of the payload
   oid.off += RZ_SIZE;
   return oid;
}
```

The *transactional reallocation* might cause an existing object to be moved. In this case, SafePM marks the old location of the object as inaccessible. The corresponding PSM region to its new location is modified to reflect the new user-specified size of the object. Still, all changes to the PSM are crash consistent, thanks to the transactional support of `libpmemobj`.

```
PMEMoid pmemobj_tx_realloc(oid, new_size) {
   oid.off -= RZ_SIZE;
   //reallocate the object to the new size
```

```
    PMEMoid *new_oid
      = pmemobj_tx_realloc_orig(oid, new_size+2*RZ_SIZE);
    void *oid_psm = get_psm_address(oid);
    if (oid != new_oid) { //object has been moved
       //add the old corresponding PSM  region to undo log
       snapshot(oid_psm, sm_size);
       //mark the old PSM  region as freed
       mark_inaddressable(oid_psm, size);
    }
    void *new_oid_psm = get_psm_address(new_oid);
    snapshot(new_oid_psm, new_psm_size);
    mark_addressable(new_oid_psm, new_size);
    new_oid.off += RZ_SIZE;
    return new_oid;
}
```

The *transactional deallocation* uses the relevant original PMDK routine to free the specified object (`pmemobj_tx_free`). SafePM's respective wrapper marks the memory region inaccessible. Note that `libpmemobj` includes built-in protection against double-frees, but it is based on the state of the persistent heap, the modification of which is delayed until the transaction is committed. Thus, double-frees that happen within a single transaction escape detection. To detect such cases, the wrapper explicitly verifies that the region the application is attempting to free is accessible. Unlike ASan, SafePM has no explicit quarantine for freed memory regions, but based on our experience, `libpmemobj` delays reallocating a deallocated region of PM.

```
void pmemobj_tx_free(oid) {
    oid.off -= RZ_SIZE;
    //verify object's validity
    void *oid_psm = get_psm_address(oid);
    if (!is_addressable(oid_psm))
       error();
    //free the requested object
    pmemobj_tx_free_orig(oid);
    snapshot(oid_psm, sm_size);
    mark_inaddressable(oid_psm, size);
}
```

Non-transactional PM management operations. SafePM transparently replaces the non-transactional memory management operations with their transactional counterparts. This is functionally correct, but forgoes the performance advantage of non-transactional operations. Unfortunately, it is inevitable, because each memory management operation causes modifications to the shadow memory which cannot be performed with a single atomic operation in conjunction with the actual PM heap metadata modification.

4.3 Additional Design Details

Crash consistency. SafePM ensures crash consistency for the memory safety metadata stored on PM: if an application crashes, both the application data and the SafePM metadata will be able to recover to a consistent state. To this end, we leverage the transactional interface of PMDK. All PM management operations, whether transactional or atomic, are executed using their transactional counterpart. A pool's PSM object is allocated and its shadow root object is initialized in a single transaction during the creation of that pool and the modification of the PSM happens within the transaction that modifies that state of an object. The shadow memory is modified after its state is snapshotted using the undo log, hence guaranteeing that the modifications are crash consistent.

System recovery. If an application crashes abruptly during the execution of a transaction, SafePM's metadata may be left in an incorrect state. However, whenever a pool is opened, PMDK checks if there exists any valid redo or undo logs. A transaction, that is interrupted before atomically validating its redo log, will apply its valid undo log to revert the PM pool's data to a consistent state. Otherwise, if a transaction is interrupted after persisting its redo log, its redo log entries will be applied. Neither case affects the correctness of the state as PSM modifications are performed in place and its initial content is tracked in the transaction's undo log. One unique case is when an application fails after the pool was created but before the transaction that allocates and initializes the PSM and the shadow root object persisted its redo log. To handle such cases, after a pool is opened successfully, SafePM checks if the pointer to the PSM object within the shadow root object is `null`. In this case, the PSM will be reinitialized and the shadow root object will be set accordingly.

Temporal safety. Similar to ASan, SafePM provides probabilistic temporal safety capabilities. When a PM object is freed or moved, the corresponding shadow memory is marked as freed. Any subsequent access to this region will be detected by the shadow memory checks inserted by ASan. Further, the native PM allocator does not reuse freed memory regions immediately, thus allowing for the detection of violations such as use-after-free or double frees that occur before the PM region is allocated again, hence the probabilistic capability.

Multi-threading support. PMDK transactions do not provide any level of isolation and it is the programmer's responsibility to ensure the application is free of race conditions. Because different persistent objects have disjoint corresponding regions in the PSM, unless the application is racy, the modifications on the PSM will be thread-safe. Further, PMDK reserves a space within the pool that is divided into *lanes*. Lanes are thread-specific and are used to store the logs of each thread's transaction. Consequently, SafePM's transactional operations are also thread-safe.

Metadata protection. SafePM initializes the PSM and marks all the PM pool as inaccessible, including the heap metadata of PMDK. As the heap metadata region is never allocated via the `libpmemobj` API, its corresponding shadow memory is never set to accessible. Accordingly, any access to a metadata region by the application's code will be detected by SafePM, thus providing metadata protection without the need for any changes to PMDK, unlike state of the art approaches [27].

Partial safety coverage. ASan is mostly used in offline testing phases due to its prohibitive instrumentation costs for every memory access. Therefore, ASan provides the option to disable the instrumentation for specific global variables and functions with the `(no_sanitize("address"))` attribute. This option deducts all ASan checks from the annotated function. It is designed for cases where the programmer trusts specific functions and wants to avoid the performance overhead.

SafePM also supports this functionality. It allows users to denote functions that will not be instrumented. For such code parts, SafePM exposes a series of 'unsafe'-prefixed wrappers which internally call the PMDK's PM management functions without performing any PSM and red zones management. However, SafePM imposes one limitation: objects allocated with unsafe wrapper functions should only be accessed in uninstrumented functions. Accessing them in instrumented code causes SafePM to report an error, as their corresponding bytes in PSM remain marked as inaccessible.

Limitations. SafePM follows the same design for the shadow memory as ASan and relies on the ASan's compiler pass for detecting memory safety violations. Hence, it inherits the same limitations. SafePM is incapable of detecting intra-object overflows as the red zones are inserted at the object level to avoid changing the objects' memory layouts. SafePM also misses out-of-bounds access that fall within the boundaries of another object.

5 Implementation

SafePM consists of *(i)* a runtime library based on PMDK and *(ii)* the ASan's compiler pass, for the instrumentation of the application code.

Runtime library. We implement the run-time library of SafePM as a fork of PMDK v1.9. In SafePM, we develop wrappers around the exposed PM management functions which require modifications to the PSM. These wrappers augment the PMDK functions with their respective shadow memory operations while ensuring crash consistency for both the PM pool (meta)data and the memory safety metadata, as explained in Section 4.2. Table 1 enumerates the functions that were wrapped by SafePM.

To ensure *transparency* and *compatibility* with existing PMDK-based applications, SafePM's wrappers are named after their PMDK equivalent function. Thus, the function calls from an unmodified PMDK application, which is linked against SafePM, get redirected to their respective wrapped version to include the memory safety metadata management.

The PSM is created as an object within a PM pool. To be able to perform the *overmap* operation, the size of the PSM must be a multiple of the page size (4KB for x86/AMD64). `mmap` also requires that the in-file offset of the portion to be memory-mapped is a multiple of the page size. The in-pool offset of the persistent shadow memory must satisfy this condition. Furthermore, the PSM must be mapped to a starting address that is page-aligned, which requires that the persistent pool is mapped to a starting address that is aligned to eight times the page size. Finally, the entire pool needs to be padded to a multiple of eight times the page size, since each shadow byte corresponds to eight application bytes. SafePM's wrapper for `pmemobj_create` is responsible for enforcing all of these padding and alignment constraints.

Pool management	
`pmemobj_create`	creates a PM pool
`pmemobj_open`	opens an existing memory pool
`pmemobj_close`	closes a memory pool
Memory management	
`pmemobj_tx_alloc`	transactional allocation
`pmemobj_tx_realloc`	transactional reallocation
`pmemobj_tx_free`	transactional deallocation
`pmemobj_alloc`	atomic allocation
Other operations	
`pmemobj_alloc_usable_size`	returns allocated size
`pmemobj_type_num`	returns object's type number
`pmemobj_first`	returns first pool object
`pmemobj_next`	iterates over the objects

Table 1. List of PMDK API modified to support SafePM

ASan prevents the application code from modifying the shadow memory by using `mprotect` to set the shadow memory's protection level to `PROT_NONE`. Unlike ASan, SafePM relies on the PSM being initialized to inaccessible to ensure these guarantees. Thanks to `mprotect`, ASan avoids allocating physical memory to protect the shadow memory itself. However, since the PSM is physically allocated during pool creation, SafePM's approach does not add extra memory overheads.

Compiler pass. SafePM leverages ASan's compiler pass without any modifications. When an application is compiled with the `-fsanitize=address` flag enabled, the compiler runs the ASan compiler pass, as shown in Listing 1. ASan's compiler pass runs after all other compiler optimizations so that only memory accesses that remain after the optimizations are instrumented. In SafePM we assume that PMDK is correct and has no memory safety violations. Therefore, we do not compile the PMDK internal functions with ASan. Note that this is necessary as these functions manipulate both the PMDK metadata and the PSM.

6 Evaluation

Our evaluation is structured around three dimensions.

Space & performance overheads. We evaluate the performance (§ 6.2) and space (§ 6.3) overheads of SafePM using PMDK's micro-benchmarks as well as pmemkv [46], a persistent KV store. We further evaluate the efficiency of the partial safety coverage approach (§ 6.6).

Effectiveness. We evaluate the effectiveness of SafePM (§ 6.4) using the RIPE framework [81] to test the exploitability of a wide range of memory vulnerabilities. We also report

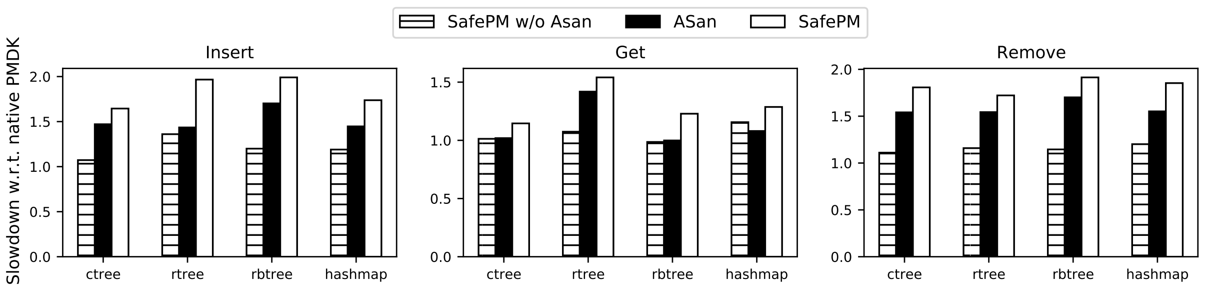

Figure 3. Performance overheads of persistent indices for PMDK w/ ASan, SafePM w/o ASan and SafePM versions.

some memory safety bugs and programming anomalies discovered during our experiments (§ 6.7).

Crash consistency. Lastly, we validate the crash-consistency property (§ 6.5) for both the application data and SafePM's metadata using the pmemcheck tool [20].

6.1 Experimental Setup

Testbed. We conduct our experiments on a server machine, equipped with Intel(R) Xeon(R) Gold 6212U CPU with 24 cores, 192 GB (6 channels × 32 GB/DIMM) DRAM and 768 GB (6 channels × 128 GB/DIMM) Intel Optane DC DIMMs running Ubuntu 20.04.02 with Linux kernel version 5.4.0.

Variants. We conduct the experiments with the variants described in Table 2. Native refers to the application being linked against the native PMDK without ASan instrumentation, while ASan indicates that the application was compiled with gcc's ASan extension and linked against native PMDK. These two variants serve as our baselines as they represent unhardened applications and applications hardened only with ASan, respectively. SafePM w/o ASan uses the SafePM's wrappers without compiling the application with the ASan extension. The goal of this variant is to demonstrate the overheads incurred by our wrappers without ASan's compiler pass instrumentation. Finally, SafePM denotes our complete tool; applications are linked against our runtime library and are compiled with the ASan instrumentation enabled.

6.2 Performance Overheads

We evaluate the performance overheads of SafePM using four different persistent indices (ctree, rbtree, rtree and hashmap) and a persistent KV store (pmemkv [46]). We further measure the performance of SafePM for the atomic and transactional PM memory management operations (alloc, realloc and free), as well as creating and opening a PM pool.

All experiments are conducted with a red zone size of 16 bytes. Each object is surrounded by two red zones. The reported values are the average of at least 3 runs.

Persistent indices. We evaluate the performance of SafePM for four persistent indices over the four variants shown in Table 2. We use pmembench [50], shipped with PMDK, and perform one million insert, get and remove operations on

Variant	Compile w/ ASan	SafePM wrappers
Native	No	Disabled
ASan	Yes	Disabled
SafePM w/o ASan	No	Enabled
SafePM	Yes	Enabled

Table 2. Benchmarking variants

each data structure. The keys are 8 bytes and the operations choose keys at random following a uniform distribution.

Figure 3 illustrates the slowdown for ASan, SafePM w/o ASan and SafePM versions normalised to the native PMDK execution. In general, SafePM is 1.68-2.00×, 1.16-1.50× and 1.68-1.87× slower than the native PMDK for the insert, get and remove operations, respectively. Figure 3 further indicates that the usage of SafePM's wrappers w/o ASan does not significantly affect the performance of the indices except for the case of rtree insert where it incurs a of 1.34× slowdown. In the other cases the respective overhead w.r.t. PMDK remains below 20%. This demonstrates the ability of SafePM to achieve its performance goal by keeping its overheads very close to those of the highly optimized ASan. The only exception is the case of the hashmap get where inserting red zones changes the objects' alignment leading to additional cache line accesses.

Persistent KV store. We evaluate SafePM's performance using pmemkv [46], a persistent KV store designed for PM, with its default cmap backend storage engine. We use the pmemkv-bench [44] benchmark suite with different workloads: *(i)* update intensive (50% reads-50% writes), *(ii)* read intensive (95% reads-5%writes), *(iii)* random reads and *(iv)* sequential reads. The KV store is populated with 1M entries at the beginning of each run. Each workload consists of 10M operations where keys and values are set to 16B and 1024B, respectively.

Figure 4 reports the slowdown of the throughput of PMDK w/ ASan and SafePM w.r.t. to native PMDK while varying the number of threads. Enabling ASan with PMDK slows down the queries' execution by 1.14-2.36× depending on the workload. The respective values for SafePM are 1.20-2.62×. The additional performance overhead for SafePM stems from the extra operations that SafePM needs to perform in order to ensure the crash consistency of memory safety metadata. The marginally higher overheads of SafePM compared to ASan further demonstrate the ability of SafePM to achieve

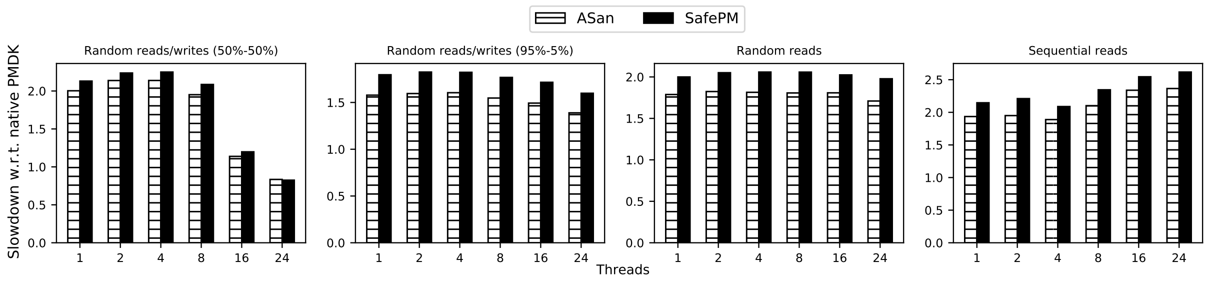

Figure 4. Performance overheads w.r.t. to native PDMK of pmemkv under different workloads and thread count.

its performance goal in real-life workloads. Furthermore, SafePM does not affect the scalability of the KV-store as its behaviour is similar to PMDK and PMDK w/ ASan even for increasing number of threads. We can notice, though, a significant drop in the overheads beyond 8 threads in the update-intensive workloads. This is attributed to the native application suffering from increasing level of contention while the instrumenation decreases this stress.

Atomic and transactional PM operations. We next measure the performance of the basic atomic and transactional PM management operations (`alloc`, `realloc` and `free`) for SafePM. We design a microbenchmark based on `pmembench` where we execute 100K operations per experiment with varying object sizes. The reported results are the average of 10 runs.

Figure 5 shows the throughput slowdown of SafePM for several PM operations normalized w.r.t. native PMDK. For object allocation, we observe that the overhead decreases for both atomic and transactional allocation as the object size grows (1.83-3.01×). The reallocation operation maintains a relatively constant overhead for all the tested data sizes (1.23-1.85×). SafePM incurs a higher performance overhead for the free operation (2.17-4.09×) compared to alloc and realloc for every object size. The aforementioned overheads are caused by *(i)* runtime checks introduced by ASan instrumentation and *(ii)* the added operations of SafePM to ensure crash consistency for PM safety metadata. Lastly, as it was expected, SafePM poses a higher overhead for the atomic versions of the operations, as it transparently converts them into their transactional counterpart in order to atomically update the appropriate PSM region along with heap state in a crash consistent manner.

PM pool create/open. Figure 6 shows the average time of the PM pool *create* and *open* operations for the variants listed in Table 2. We created a microbenchmark using the `pmembench` framework where we create/open PM pools of various sizes ranging from 256MB to 64GB. We observe that opening a pool with SafePM takes ~40ms instead of 15ms with native PMDK, a slowdown of up to 2.5×. The slowdown appears to be largely caused by the introduced ASan checks because the performance of the `SafePM w/o Asan` variant is close to that of the native PMDK. Further, during pool creation, SafePM incurs significant slowdown which increases with the pool size, causing the *create* operation to take a few seconds to complete. This overhead stems from the need to overmap and initialise the PSM object, which grows with the size of the pool. It is worth noting, though, that pool creation is an one-time operation, hence, the high overhead is largely irrelevant to application performance.

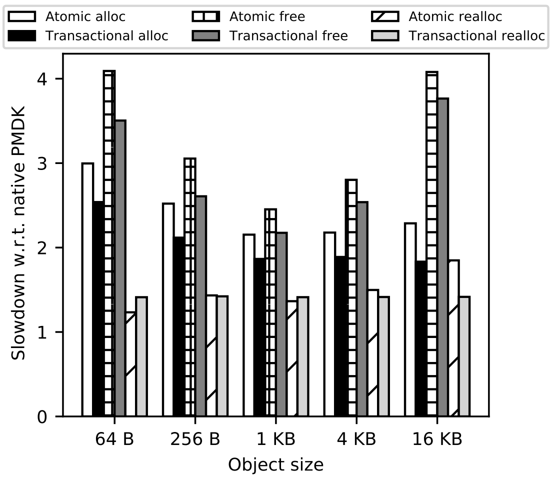

Figure 5. Performance overhead of SafePM's wrappers for selected memory operations across different object sizes.

6.3 Space Overhead

We measure the extra PM space that SafePM requires: the persistent shadow memory and the object red zones. This section ignores the shadow root object, as it represents a small, fixed overhead, independent of the pool size or allocated objects. We conduct experiments on the same four persistent indices performing `insert`, `get` and `remove` operations, as discussed in 6.2. We report the peak space overhead when applications are linked against SafePM.

Table 3 summarizes the PM space overheads of SafePM expressed in percentage of the total pool size. The persistent shadow memory always occupies one eighth of the pool which corresponds to an overhead of 12.5%. For the `ctree`, `rbtree` and `hashmap_tx`, we observe that this is the only considerable space overhead as the persistent object red zones occupy space which is wasted to padding by the native PMDK allocator. For the `rtree` index the object red zones increase

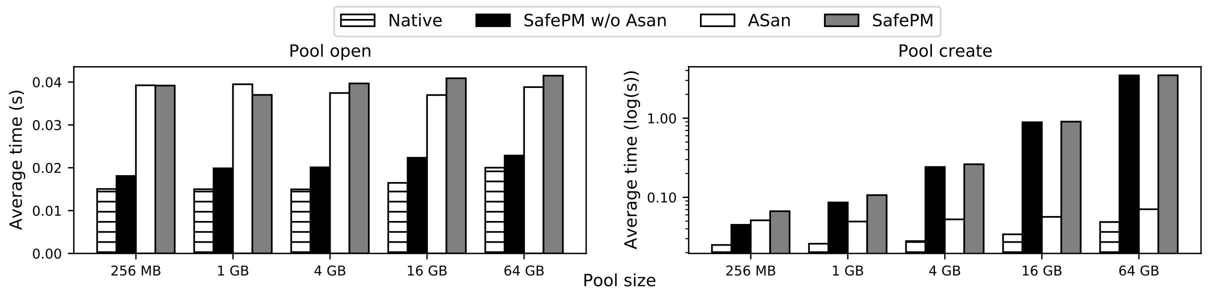

Figure 6. Performance overheads for creating and opening pool of various sizes.

Data structure	insert	remove	get
ctree	12.5%	12.5%	12.5%
rtree	14.25%	13.8%	13.8%
rbtree	12.5%	12.5%	12.5%
hashmap_tx	12.5%	12.5%	12.5%

Table 3. SafePM space overhead

persistent memory usage leading to slightly higher space overheads. However, even in this case, the main contributor to the increase in PM space usage is the persistent shadow memory.

6.4 Effectiveness

We evaluate the effectiveness of SafePM using the RIPE framework [81], a comprehensive suite of memory vulnerability exploits. We modified the 64-bit port of the RIPE benchmark [2] to compare the effectiveness of the following variants: *(i)* Intact, where the victim application is unmodified, *(ii)* ASan w/ system heap, where the application is compiled with ASan, *(iii)* ASan, where the application uses the persistent heap and is compiled with ASan, which protects only the volatile heap and not the PM heap, *(iv)* SafePM, which extends ASan's memory safety to the PM heap, and *(v)* memcheck [19], the current state-of-the-art for detecting memory violations in persistent memory. All variants use gcc 9.3.0 and are compiled with the default GCC stack protections enabled.

Table 4 reports the number of exploits that either *always* (or with high probability), *sometimes* or *never* succeed. We observe that when the victim application uses the volatile (system) heap, ASan is able to prevent most attacks. When the victim application is modified to use the PM heap, the number of exploitable vulnerabilities increases, as the layout of the persistent heap is not available to ASan. However, linking the application against SafePM restores ASan's protection capabilities even for memory violations occurring in the persistent heap, reducing the number of exploitable vulnerabilities back to the levels observed with the system heap. Finally, we observe that SafePM is able to detect and prevent a higher number of memory vulnerabilities compared to the state-of-the-art memcheck [19].

6.5 Crash Consistency

We validate the crash-consistency property for both the application data and SafePM metadata using existing tools, `pmemcheck` [20] and `memcheck` [70]. As SafePM wraps PMDK routines instead of modifying them or the pool layout, these tools work without modifications. We run the persistent indices and PM operations benchmarks described in § 6.2 with `pmemcheck` and `memcheck` enabled. Note that ASan is disabled due to its incompatibility with Valgrind. The number of operations for each index is limited to 10000 to keep the runtime reasonable despite the slowdown caused by Valgrind. We observe that for the tested indices, neither `pmemcheck` nor `memcheck` report any error. For the PM operations benchmark, `pmemcheck` again reports no error, while `memcheck` does not report any error beyond the ones also reported for the case of unmodified PMDK.

RIPE variant	Always	Sometimes	Never
Intact	306	14	1014
ASan w/ system heap	27	1	1306
ASan w/ PM pool heap	119	12	1203
SafePM	27	1	1306
memcheck	62	0	1272

Table 4. Number of RIPE attacks that always, sometimes or never succeed with different protection mechanisms.

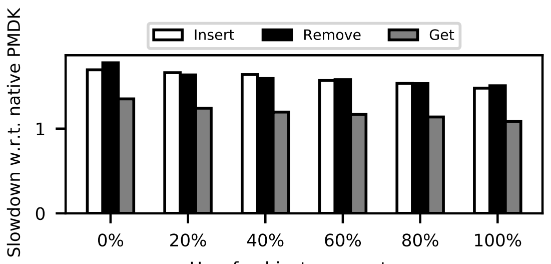

Figure 7. Performance overheads of persistent hashmap index with varying percentage of unsafe PM objects.

6.6 Partial Safety Coverage

We evaluate the efficiency of our proposed partial safety coverage approach. In this experiment we use the persistent `hashmap` in a similar fashion to 6.2. We vary the proportion of operations that are performed using memory safe (instrumented) and memory unsafe (uninstrumented) objects. Figure 7 illustrates the performance slowdown for each operation as the proportion of used unsafe objects increases, normalised w.r.t. the native PMDK execution. We observe that for all three operations the relative overhead decreases as more objects are excluded from the ASan instrumentation and runtime checks. However, there is still an inevitable overhead that stems from ASan intercepting the volatile heap management functions, which are used by PMDK internally. Note that with get operation there is no overhead as there are no intercepted `malloc`/`free` calls.

6.7 Discovered Bugs and Anomalies

Our experiments led to discovering and reporting[1] the following bugs: *(i)* in the btree example of PMDK version 1.9.2, a call to *memmove* on line 378 of *btree_map.c* causes an off-by-one overflow on PM residing data objects and *(ii)* in the transactional operations benchmark, shipped as part of `pmembench`, a configuration file lacks the configuration setting `nestings`. This causes the transaction to not be aborted, which triggers invalid frees at line 295 in *pmemobj_tx.cpp*, that is detected by SafePM, when the benchmark attempts the cleanup.

7 Related work

Persistent-memory based systems. Several well-known, high performant data management systems, such as RocksDB [18] and Redis [17], have already been adapted to incorporate persistent memory in their system stack [15, 17, 18, 46]. Apart from that, there exists proposed filesystems specially designed to benefit from PM as a storage medium [26, 55, 79, 83]. Additionally, accessing PM remotely is an active area of research in order to enable PM usage in distributed settings [48, 49, 54, 62, 75, 84]. While these systems mainly target to ensure crash consistency and high performance with the use of the innovative PM technology, SafePM focuses on the important aspect of memory safety in PM programming.

SW-based memory safety. Protecting against DRAM memory safety bugs with software based approaches has been the target of several works [22, 57, 58, 74]. They leverage different techniques such as compiler pass instrumentation accompanied with run-time libraries and compact representation of upper and lower pointer bounds, needed to perform the appropriate runtime checks. They aim to minimise the performance and memory overheads while maintaining compatibility and efficiency. Another alternative for ensuring memory safety is to use a memory safe language. Corundum [40] is a generic library for persistent memory management written in Rust, which statically enforces language based memory safety for PM. SafePM, on the contrary to Corundum, does not require programmers to use specific libraries or languages, but targets applications developed using PMDK, the de facto library for PM, while requiring no source code modifications. The `memcheck` tool [19] uses Valgrind and instrumentation built into PMDK to achieve memory safety similar to SafePM. Unlike ASan, it does not require compiler support, as it uses run-time translation. Further, unlike SafePM, it has no persistent memory overhead. The trade-offs are that it incurs a much larger performance overhead, and its spatial violation detection capabilities are not as precise as SafePM's.

HW-based memory safety. There exists a large body of work that enforces memory safety for volatile memory using hardware extensions [3, 32, 71, 82]. Lowfat pointers [32] enforce spatial safety by associating the pointer with its bounds. It contains gate-level implementations of the logic for updating and validating the compact fat pointers. Cheri [82] ensures memory safety bug detection with the support of hardware capabilities. Intel MPX [71] provides ISA extensions of Intel x86-64 architecture for memory protection. Arm MTE [3] is an ARM extension that enables hardware-assisted memory tagging to detect both temporal and spatial memory safety bugs. Besides being designed for volatile memory only, these solutions require specialized hardware, whereas SafePM can be deployed in commodity servers to ensure memory safety for PM.

PM debuggers, allocators and libraries. Several frameworks have been developed to test the correctness of PM software [20, 30, 35, 36, 59–61, 67], an orthogonal problem to memory safety. Many projects strive to manage persistent memory efficiently while also ensuring crash consistency [25, 27, 47, 80]. Mnemosyne [80] exposes a simple interface for PM programming with respect to crash consistency and persistence. NVHeaps [25] is a lightweight, high-performance persistent object system with transaction support and persistency semantics. Poseidon [27] is another allocator designed for PM that relies on Intel MPK [72] to avoid the corruption of the persistent metadata by memory bugs.

8 Conclusion

In this paper, we present SafePM, a framework that ensures the memory safety of PMDK-based PM applications by detecting spatial and temporal memory safety violations. To this end, SafePM leverages the compiler pass and reporting mechanism of the popular ASan. During the creation of a memory pool, SafePM creates a persistent shadow memory object that is mapped when the pool is opened to the corresponding location in ASan's shadow memory. SafePM manipulates the persistent shadow memory along with the persistent heap in a transparent and crash consistent manner. Consequently, any PMDK-based application can make use of SafePM without source code modifications to test for memory violations at runtime, including the recovery process. Our extensive evaluation shows that SafePM provides the same level of memory safety guarantees for PM applications as ASan provides for volatile memory at a cost of marginal overheads.

Artifact evaluation. We will release SafePM along with the evaluation setup and reported bugs.

[1] Citation is omitted for double blind purposes.

References

[1] 2015. Poject Zero - Stagefrightened? https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html. Accessed 27-09-2021.

[2] 2019. A 64-bit port of the RIPE benchmark. https://github.com/hrosier/ripe64.git. Accessed 27-09-2021.

[3] 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety. Accessed 27-09-2021.

[4] 2019. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. Accessed 27-09-2021.

[5] 2019. Queue the Hardening Enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html. Accessed: 2021-02-27.

[6] 2020. The Heartbleed Bug. https://heartbleed.com/. Accessed 27-09-2021.

[7] 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. Accessed: 31-08-2021.

[8] 2021. The Chromium Projects - Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety. Accessed 27-09-2021.

[9] 2021. Clang 13 Documentaion - AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html Accessed 27-09-2021.

[10] 2021. GCC - Program Instrumentation Options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html Accessed 27-09-2021.

[11] 2021. Hardware-Assisted Checking Using Silicon Secured Memory (SSM). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html. Accessed 27-09-2021.

[12] 2021. Intel® Inspector: Deliver reliable applications. Locate and debug threading, memory, and persistent memory errors early in the design cycle to avoid costly errors later. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html#gs.cdp2vf. Accessed 27-09-2021.

[13] 2021. The libpmem webpage. https://pmem.io/pmdk/libpmem/. Accessed: 2021-02-27.

[14] 2021. Memcheck: a memory error detector. https://valgrind.org/docs/manual/mc-manual.html. Accessed 27-09-2021.

[15] 2021. Memhive: Scale applications with persistent memory! https://www.memhive.io/. Accessed 27-09-2021.

[16] 2021. The PMDK webpage. https://pmem.io/pmdk/. Accessed: 2021-02-27.

[17] 2021. Pmem-Redis. https://github.com/pmem/pmem-redis. Accessed 27-09-2021.

[18] 2021. pmem-rocksdb. https://github.com/pmem/pmem-rocksdb. Accessed 27-09-2021.

[19] 2021. pmem-valgrind. https://github.com/pmem/valgrind. Accessed 27-09-2021.

[20] 2021. Pmemcheck: persistent memory analyzer. https://pmem.io/valgrind/generated/pmc-manual.html. Accessed 27-09-2021.

[21] 2021. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html#hardware-tag-based-kasan. Accessed 27-09-2021.

[22] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *18th USENIX Security Symposium (USENIX Security 09)*. USENIX Association, Montreal, Quebec. https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/baggy-bounds-checking-efficient-and

[23] Jan Willem Aldershoff. 2021. Intel reveals App Direct mode and Memory mode for Optane DC Persistent Memory. https://www.myce.com/news/intel-reveals-app-direct-mode-and-memory-mode-for-optane-dc-persistent-memory-85373/ Accessed 27-09-2021.

[24] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[25] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 105–118. https://doi.org/10.1145/1961295.1950380

[26] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[27] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/3423211.3425671

[28] Dinakar Dhurjati and Vikram Adve. 2006. *Backwards-Compatible Array Bounds Checking for C with Very Low Overhead.* Association for Computing Machinery, New York, NY, USA, 162–171. https://doi.org/10.1145/1134285.1134309

[29] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (Ottawa, Ontario, Canada). ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

[30] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/3445814.3446744

[31] Gregory J. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium (NDSS)*.

[32] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[33] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 181–195. https://doi.org/10.1145/3192366.3192388

[34] Frank Eigler. 2003. Mudflap: Pointer use checking for C/C++. In *GCC Developers Summit*.

[35] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles* (Virtual) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 1–15.

[36] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support*

for Programming Languages and Operating Systems (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.org/10.1145/3445814.3446735

[37] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[38] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. 125–138.

[39] Red Hat. 2015. GHOST: glibc vulnerability (CVE-2015-0235). https://access.redhat.com/articles/1332213. Accessed 27-09-2021.

[40] Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/3445814.3446710

[41] indradead. 2021. Direct Access for files. https://www.infradead.org/~mchehab/kernel_docs/filesystems/dax.html Accessed 27-09-2021.

[42] Intel. 2021. An introduction to pmemobj (part 4) - transactional dynamic memory allocation. https://pmem.io/2015/06/17/tx-alloc.html Accessed 27-09-2021.

[43] Intel. 2021. An introduction to pmemobj (part 5) - atomic dynamic memory allocation. https://pmem.io/2015/06/18/ntx-alloc.html Accessed 27-09-2021.

[44] Intel. 2021. Benchmarking tools for pmemkv. https://github.com/pmem/pmemkv-bench Accessed 27-09-2021.

[45] Intel. 2021. Documentation for ndctl and daxctl. https://pmem.io/ndctl/ndctl-create-namespace.html Accessed 27-09-2021.

[46] Intel. 2021. Persistent Memory Development Kit : pmemkv. https://pmem.io/pmemkv/ Accessed 27-09-2021.

[47] Intel. 2021. Persistent Memory Development Kit : The libpmemobj library. https://pmem.io/pmdk/libpmemobj/ Accessed 27-09-2021.

[48] Intel. 2021. Persistent Memory Development Kit : The librpma library. https://pmem.io/rpma/ Accessed 27-09-2021.

[49] Intel. 2021. Persistent Memory Development Kit : The librpmem library. https://pmem.io/pmdk/librpmem/ Accessed 27-09-2021.

[50] Intel. 2021. pmembench: PMDK benchmark framework. https://github.com/pmem/pmdk/blob/master/src/benchmarks/pmembench.cpp. https://github.com/pmem/pmdk/blob/master/src/benchmarks/pmembench.cpp Accessed 27-09-2021.

[51] Intel. 2021. The Challenge of Keeping Up with Data. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html Accessed 27-09-2021.

[52] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[53] Trevor Jim, J. Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. *Proc. of the 2002 USENIX Annual Technical Conference*, 275–288.

[54] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 105–119. https://doi.org/10.1145/3419111.3421294

[55] Chandan Kalita, Gautam Barua, and Priya Sehgal. 2018. DurableFS: A File System for Persistent Memory. *CoRR* abs/1811.00757 (2018). arXiv:1811.00757 http://arxiv.org/abs/1811.00757

[56] The Linux kernel archives. 2021. DAX - Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt Accessed 27-09-2021.

[57] Taddeus Kroes, Koen Koning, Erik Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: buffer overflow checks without the checks. 1–14. https://doi.org/10.1145/3190508.3190553

[58] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[59] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3445814.3446691

[60] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[61] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[62] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[63] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[64] Santosh Nagarakatte, Aa Bb, Milo Martin, and S. Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *Sigplan Notices - SIGPLAN* 44, 245–258. https://doi.org/10.1145/1542476.1542504

[65] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

[66] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[67] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: Healing Persistent Memory Bugs without Doing Any Harm. In *Proceedings*

of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/3445814.3446694

[68] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[69] Netapp. 2021. What is persistent memory? https://www.netapp.com/knowledge-center/what-is-persistent-memory/ Accessed 27-09-2021.

[70] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[71] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[72] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[73] Olatunji Ruwase and M. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Network and Distributed System Security Symposium (NDSS)*.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[75] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. 2020. TH-DPMS: Design and Implementation of an RDMA-Enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage* 16, 4, Article 24 (Oct. 2020), 31 pages. https://doi.org/10.1145/3412852

[76] Matthew S. Simpson and Rajeev K. Barua. 2010. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 199–208. https://doi.org/10.1109/SCAM.2010.15

[77] László Szekeres, M. Payer, Tao Wei, and D. Song. 2013. SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy* (2013), 48–62.

[78] Victor van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future, Vol. 7462. https://doi.org/10.1007/978-3-642-33338-5_5

[79] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

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

[81] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. *ACM International Conference Proceeding Series*, 41–50. https://doi.org/10.1145/2076732.2076739

[82] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 457–468. https://doi.org/10.1145/2678373.2665740

[83] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[84] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 111–125. https://www.usenix.org/conference/nsdi20/presentation/yang

[85] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs. *ACM Computing Surveys - CSUR* 44 (06 2012), 1–28. https://doi.org/10.1145/2187671.2187679