SPP: Safe Persistent Pointers for Memory Safety

Dimitrios Stavrakakis
TU Munich &
University of Edinburgh

Alexandrina Panfil
TU Munich

MJin Nam
TU Munich

Pramod Bhatotia
TU Munich

***Abstract*—Memory safety violations, such as buffer overflows, are the primary cause of security and reliability issues in software systems. Like the volatile main memory, byte-addressable persistent memory (PM) storage devices are also prone to memory safety exploits because PM devices are directly mapped to the address space and accessed via the load/store interface using pointers. However, the PM pointer representation is *persistent*, i.e., its offset and the associated object are persistent across system reboots. Therefore, the current memory safety mechanisms for the volatile main memory are *inadequate* for ensuring the safety of persistent pointers.**

To this end, we propose Safe Persistent Pointers (SPP), a practical memory safety approach against buffer overflows for PM applications. SPP augments persistent pointers with memory safety properties. SPP is based on a simple combination of tagged pointers, efficient persistent memory layout, and transactional updates to the memory safety metadata for ensuring crash consistency. SPP's efficient pointer representation does not require additional memory lookup operations at run-time while incurring minimal space overheads for storing the memory safety metadata.

We implement SPP based on the LLVM compiler infrastructure accompanied by a runtime library and an adapted version of PM development kit (PMDK). Our evaluation demonstrates that SPP incurs low runtime and space overheads while preserving the crash-consistency property and maintaining the PMDK API intact, i.e., requiring no source code modifications.

I. Introduction

Low-level unsafe languages, such as C/C++, provide developers with control over the system's memory. While this is crucial performance-wise [101], it can lead to, potentially harmful, memory safety bugs [31], [46], [108], [110], [120]. These bugs are broadly separated into two categories; *spatial*, e.g. buffer overflows, and *temporal*, e.g., dangling pointers.

Memory safety bugs cause many critical security and reliability issues [2], [7], [8], [54]. The severity of memory safety violations is also confirmed by the reports of major software projects, such as Windows [5], Android [6] and Chromium [10], where $70-75\%$ of the detected issues stem from memory safety bugs. According to Szekeres et al. [108], the majority of security attacks in software systems occur through exploiting memory safety vulnerabilities.

Designing efficient approaches to enforce memory safety is an active area of research for *the volatile main memory*, including software and hardware-based memory safety solutions (§VII). At a high level, these approaches implement *deterministic dynamic bounds checking* [89], [108], which utilizes runtime metadata (bounds information) [75], rather than relying on probabilistic heuristics [31], [83]. These approaches instrument the code during compilation and inject run-time metadata management into an application that allows deterministic run-time checks for validating memory accesses.

Unfortunately, existing memory safety approaches are *restricted to the volatile main memory devices and are inadequate for byte-addressable persistent memory (PM) devices*. In particular, the emergence of the Compute Express Link (CXL) technology [37] is leading to byte-addressable PM storage devices [51], [105]. These PM devices are either attached to the memory bus [95] or the PCIe bus [37] and can *also* be accessed over the network (e.g., via RDMA) [65]. PM applications memory map (`mmap()`) these devices directly to their address space. Using pointers, their mapped content is accessed at a byte granularity via the `ld`/`st` interface.

However, the PM pointer representation is persistent, i.e., its offset and the associated PM object are durable. Therefore, addressing memory safety issues for PM is challenging, especially due to the idiosyncrasies of the PM programming model [15]. More specifically, PM applications rely on persistent pointers [9] and use specialized, crash-consistent memory allocators [39], [64]. This entails two challenges: (a) how do we preserve crash consistency for memory safety metadata?, and (b) how do we ensure memory safety on the recovery paths after a system crash or reboot? Unfortunately, current memory safety mechanisms for PM, with the most prominent being SafePM [33], a shadow memory-based memory safety solution built on AddressSanitizer [104], are deemed as *impractical* since they either require adopting a new programming model/language [55] or are restricted to the offline testing phase [20], [33] due to prohibitive performance costs.

To this end, we propose Safe Persistent Pointers (SPP), a practical memory safety approach for applications accessing byte-addressable PM storage devices via PM pointers. SPP provides *PM buffer overflow protection*. Its design is based on DeltaPointers [75], a memory safety approach for volatile memory. SPP essentially extends DeltaPointers to PM. SPP is built on the prevalent PM programming model and employs *tagged pointers*, as well as an *efficient PM layout*, in combination with transactional updates to the memory safety metadata.

Our SPP prototype consists of an adapted PMDK [15] version, the state-of-the-art PM programming toolchain, and an instrumentation using LLVM [77]. The evaluation of SPP is structured around three dimensions: performance and space overhead, effectiveness, and crash consistency. We measure the performance and space overheads of SPP using PMDK micro-benchmarks and a persistent KV store [59]. We evaluate the effectiveness of SPP with the RIPE framework [114] that contains a set of memory safety exploits. Lastly, we validate the crash-consistency of SPP's metadata using the `pmemcheck` [20] tool. SPP incurs low performance overheads and requires no source code modifications, while pre-

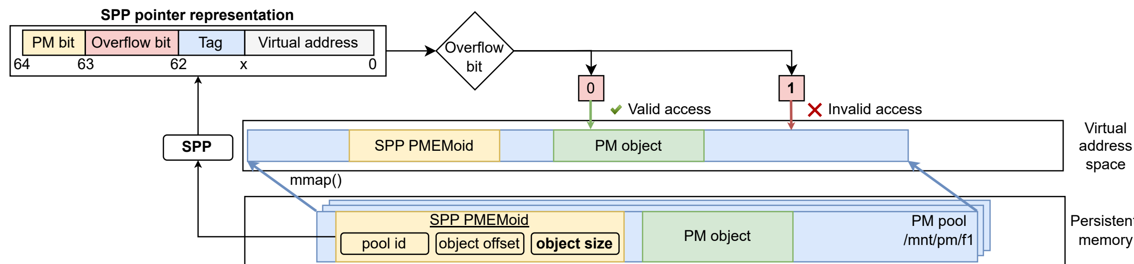

Fig. 1. **SPP pointer representation:** *The* SPP *PM pointer representation (*SPP *PMEMoid) is used to derive the tagged pointer to the PM object (*SPP *pointer representation). On a PM access, the PM bit and the pointer tag get masked while the overflow bit is preserved. If the access is valid (in green), the overflow bit is 0 and the access succeeds. In case of a PM buffer overflow (in red), the overflow bit is 1 which makes the address invalid.*

serving the crash consistency property.

Altogether, SPP makes the following contributions:

- SPP introduces *safe persistent pointers* (§IV), a spatial memory safety solution against PM buffer overflows for PMDK applications. They are the first tagged pointer scheme specifically designed for PM. They consist of an enhanced durable representation of PMDK's persistent pointers and a native-pointer tagging scheme.
- SPP offers a configurable pointer encoding scheme, inspired by DeltaPointers [75], to fit the PM management requirements of every PM application. The number of bits in the pointer tag is adjustable and and can be easily tuned without breaking the compatibility with pre-compiled, uninstrumented libraries.
- We implement the SPP prototype and design our compiler optimizations based on LLVM (§V). Our evaluation (§VI) indicates that SPP is a practical approach that prevents PM buffer overflow exploits, while incurring low performance and negligible space overheads.

II. Background and Motivation

A. Byte-Addressable Persistent Memory

Byte-addressable PM storage devices reside on the memory bus [95], providing access latencies similar to DRAM. However, the recently-emerged and evolving Compute Express Link (CXL) technology [51] allows these devices to be attached on the PCIe bus or even be exposed over the network (e.g., through RDMA), enabling the creation of large pools of byte-addressable storage.

Precisely, PM devices are accessed with `ld`/`st` instructions. They are directly mapped to an application's address space. Applications use pointers to access PM, which must be reconstructible and consistent across reboots or crashes, leading to a deviation from the established programming model.

B. Byte-addressable PM Programming Model

Persistent pointers. *Persistent pointer* is the core programming abstraction to access byte-addressable PM. Based on this abstraction, a PM memory pool can be mapped in different regions of an application's address space across different runs and, thus, the application needs a consistent way to identify the stored, persistent objects. In contrast to volatile pointers, persistent pointers are durable, crash-consistent data structures that are used to reconstruct native pointers to PM objects across application restarts or crashes.

PM Development Kit (PMDK). To facilitate the application development for PM devices, Intel introduced PMDK [15]. PMDK includes a variety of generic libraries and high-level tools that allow for flexible PM management [59]–[61], [64] while also exposing a low-level PM support API [12].

libpmemobj. The `libpmemobj` [64] library handles PM files as object stores and exposes an intuitive API for PM management, similar to the conventional malloc/free API. Further, it employs a fat-pointer scheme [9] for the PM pointers. Each object is identified by a *PMEMoid* that contains a *pool_id* (8 B) and an *offset* (8 B) relative to the beginning of the pool. `libpmemobj` translates *PMEMoids* to native pointers via the `pmemobj_direct()` function. Lastly, `libpmemobj` introduces SW transactions to ensure the crash-consistency of PM data updates exceeding the atomicity boundary of 8 B.

C. Memory safety

Low-level unsafe languages (e.g., C/C++), allow applications to interact with the system's memory. While this is powerful for optimizing performance, it can lead to memory safety violations with dire consequences when exploited by malicious attackers [86], [110]. Memory safety violations fall into two categories. *Spatial* memory safety errors refer to accesses beyond the intended boundaries on memory (e.g., buffer overflows) while *temporal* memory safety bugs occur when a memory region is accessed before its allocation or after its release (e.g., dangling pointers). Due to their severity, such illegal memory accesses must be prevented. To this end, various approaches have been proposed, which can be categorized as *(i)* pointer-based [68], [75], [90]–[92], [94], [107], *(ii)* shadow memory-based [38], [50], [52], [53], [104] and *(iii)* object-based [30], [41], [42], [44], [45], [69], [103].

D. Memory safety for Byte-Addressable Persistent Memory

Byte-addressable PM is susceptible to memory safety issues since it is accessed through pointers. However, existing solutions for volatile memory cannot be directly applied to PM. First, the durability of PM pointers representation implies that their associated memory safety metadata needs not only to be persistent but also consistent across application restarts or crashes. Precisely, consider a crash during the update of memory safety metadata (e.g., object re-allocation). The derived PM pointer on the next run can contain stale or erroneous bounds information due to partially updated or inconsistent memory safety metadata which can lead to bugs or allow for

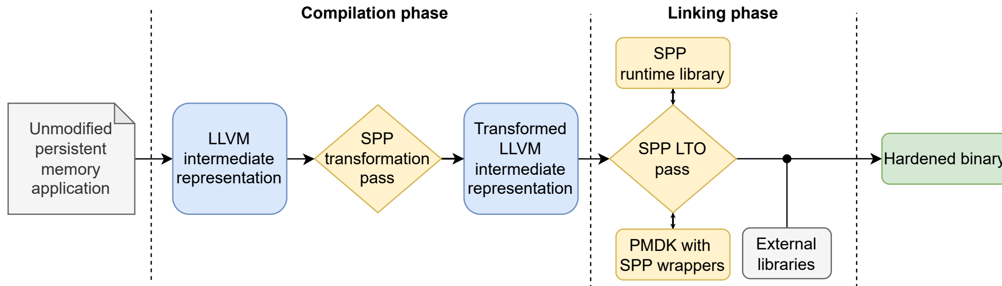

Fig. 2. **SPP overview (yellow colored boxes denote the SPP components):** *The unmodified PM application is converted to its LLVM IR, where the* SPP *transformation pass transforms the runtime function calls for the pointer tag management. At the link phase,* SPP *applies its optimizations via its LTO pass and the transformed application is linked against the* SPP *runtime library, adapted PMDK version and external libraries to produce the final binary.*

exploits from a malicious attacker. To tackle this problem, a PM memory safety solution must update its metadata in a fail-safe manner via atomic operations or transactions.

Additionally, metadata fetching during runtime can cause a significant performance degradation due to the slower PM accesses rendering an approach impractical. Therefore, minimizing the stored memory safety metadata and optimizing their placement is imperative.

Lastly, PM comes with its own programming model and specialized PM management APIs that need to be carefully handled and instrumented to preserve their persistency and crash-consistency semantics.

SafePM [33] is the current state-of-the-art solution for PM memory safety. It is a shadow-memory [38], [50], [52], [53] approach based on ASan [104] and PMDK [15]. SafePM targets mainly debugging environments due to its performance and PM space overheads. It reserves a part of the PM pool for its PM safety metadata which is mapped on ASan's shadow memory during runtime. Thus, SafePM leverages ASan's instrumentation and detects PM safety bugs across runs as it also ensures crash consistency for PM safety metadata.

III. Overview

SPP is a system that provides spatial memory safety for PMDK-based applications. It requires no source code modifications and introduces minimal performance and space overheads. Thus, SPP is a practical PM safety solution, in contrast to the state of the art approaches that incur significant space and runtime overheads [13], [33] or demand the usage of specific memory safe languages [55].

Figure 1 captures the SPP PM pointer representation (SPP PMEMoid), the tagged pointer structure and how SPP handles PM accesses. Precisely, SPP enhances the SPP PMEMoid with a field containing the object size. This persistent data structure is used to generate the tagged pointer [75] to a PM object. Importantly, SPP sets and updates the SPP PMEMoid in a crash consistent manner either by wrapping its content inside transactions or through atomic operations. In case of a PM access, SPP preserves only the virtual address and the overflow bit of the tagged pointer. If the access lies within the bounds of the PM object, the overflow bit is clear and the access proceeds normally using the virtual address. However, if the pointer is beyond the objects boundaries, the overflow bit is set through the SPP pointer tag operations rendering the address invalid. Thus, the upcoming access triggers a fault.

An overview of SPP's workflow is shown in Figure 2. An unmodified PM application is initially instrumented with SPP's transformation pass that inserts the PM pointer tag operations and the runtime checks. The instrumented code is then linked against SPP's runtime library and the modified PMDK that includes the enhanced PM pointer representation and the adapted PM management functions. Note that the programming model and the APIs of PMDK remain intact. During the linking process, SPP's link time optimization (LTO) pass scans the application for external function calls and masks away the tag from the PM pointers passed as arguments to preserve compatibility. Thus, SPP can be seamlessly integrated in existing workflows and deployments, providing complete control of the memory safety-critical code parts.

A. System Model

Fault model. SPP protects against spatial memory safety bugs on PM. It detects PM buffer overflows in PMDK-based applications, while preserving crash consistency for both application data and metadata. SPP correctly reconstructs tagged pointers across crashes and provides complete code coverage, including the application's recovery code paths.
Usage model. SPP aims to be integrated in production deployments of PMDK-based systems. It can be tuned to fit multiple use cases. SPP also provides complete control to the developers to define the memory safety-critical code files to further reduce the instrumentation and run-time overheads.
Programming model. SPP supports the native PM programming model and the PMDK APIs. It provides spatial memory safety against PM buffer overflows for PMDK applications.

B. Design Goals and Key Ideas

#1: Transparency. For practical memory safety, SPP should transparently provide memory safety using the native PMDK API, similarly to prior approaches [13], [20], [33]. This is essential to ease the integration in existing toolchains.
Approach: SPP adapts the PMDK functions for PM object management and transactional logging to consider the additional *size* field of the PM pointer representation without altering the APIs (§V). It further adapts PMDK to construct PM pointers with the SPP's encoding transparently.
#2 Performance and compatibility. SPP aims to be deployed in performance critical environments. To this end, SPP must

(i) keep runtime and storage overheads at the bare minimum levels, while offering high code coverage, and *(ii)* be compatible with existing, uninstrumented, external libraries.
Approach: To minimize the performance and storage overheads, SPP includes optimizations (e.g., pointer tracking) and limits its metadata to the *size* field (8 B), added to the *PMEMoid*. Further, to preserve compatibility, SPP identifies the external library calls and removes the tag from their pointer arguments. Thus, linking against shared libraries is supported without recompilation. Note that SPP cannot provide any memory safety for the code paths of the external functions.

#3: Heterogenous memory systems. Modern applications are designed to operate on heterogeneous systems that combine PM and volatile memory [16], [17], [59], [84]. A program accesses both PM and volatile memory via native 64-bit pointers. Therefore, SPP should distinguish between the instrumented PM pointers and the uninstrumented volatile memory pointers.
Approach: To identify pointers to PM, SPP *sets* their most significant bit. In that way, SPP tags and instruments exclusively the PM pointers. Additionally, SPP preserves the volatile memory management of an application intact, since in current systems, pointers utilize only 48-57 bits [21].

#4: Crash consistency. PM applications are designed to recover from crashes and maintain the PM data consistent. This process requires designated recovery code paths which, inevitably, include PM accesses. Therefore, SPP should be able to reconstruct tagged PM pointers correctly to provide memory safety across restarts and cover the recovery paths.
Approach: SPP enhances the PM pointer representation with a field holding the *size* of the PM object. This representation is updated in the adapted PM management operations using atomic operations or PMDK software transactions. Thus, SPP is able to recreate the PM pointer tags across reboots/crashes since the PM object size information is durable and valid.

IV. Design

SPP enforces a *tagged pointer* (§IV-A) approach to detect PM buffer overflows in PMDK applications. SPP consists of *(i)* an adapted PMDK version (§IV-B), *(ii)* static analysis compiler passes (§IV-C), and *(iii)* a runtime library (§IV-D).

A. SPP *Pointer Representation*

SPP introduces the first tagged pointer scheme for PM. SPP encodes memory safety metadata in the upper bits of each PM pointer [21], as performed in prior tagged pointer approaches [75], [76], [92], [117]. More specifically, a native 64-bit PM pointer is split into four distinct parts (Figure 1). Its most significant bit (MSB) is set to 1 to indicate that it points to a PM address. The following bits contain an *overflow bit* and the *tag*, which has a configurable size. The lower bits maintain the actual virtual address of the pointer in the mapped PM file. SPP's pointer *tag* specifies the current distance from the upper bound of the PM object. SPP initializes it as the negated allocated object size, similarly to Delta pointers [75], and updates it on pointer arithmetic operations.

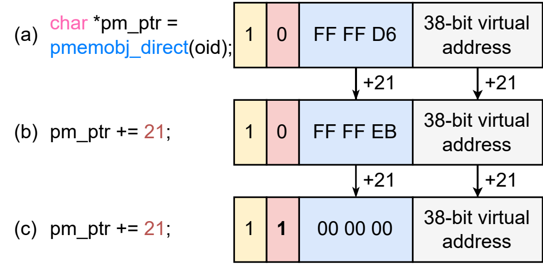

Fig. 3. **SPP pointer management:** *On each pointer arithmetic operation, the respective action is applied to the tag of the pointer (b). When a pointer surpasses the object's upper bound, the overflow bit gets implicitly set (c).*

Figure 3 presents an example of an SPP pointer with 24 tag bits for a 42 B object. Initially, the `pmemobj_direct` function receives the enhanced *PMEMoid* of the object and returns the tagged pointer (Figure 3a). On every pointer arithmetic operation, the virtual address modification is also applied to the tag (Figure 3b). Once the pointer surpasses its upper bound, the overflow bit gets set as shown in Figure 3c. Thus, on an out-of-bounds access, an error is triggered since the pointer is implicitly invalidated due to the overflow bit. This means that SPP requires no explicit, actively-performed runtime bound checks. However, if subsequent pointer arithmetic operations bring the pointer back within its assigned boundaries, the overflow bit gets unset and the pointer becomes valid again.

SPP's tag encoding is designed to detect PM buffer overflows while aiming to reduce the performance and space overheads. To incorporate protection against additional memory safety bug types, different encoding schemes that maintain the location of memory safety metadata (e.g., lower bound) [76], [117] or fat-pointer approaches [115] can be adapted for PM. In every case the system needs to consider the crash consistency of the additional metadata as well.

Further, if the usable pointer bits were not limited, SPP could include a second part in the tag for the lower bound. However, this approach would significantly limit the buffer size, making the approach non-practical. It would also require further manipulation of the tag on pointer operations, which would introduce additional overheads.

B. *PMDK Modifications*

SPP adapts PMDK to correctly construct the pointer tags even across restarts or crashes. SPP enhances the PM pointer representation (*PMEMoid*) with an additional field maintaining the size of the PM object, incurring minimal space overheads. This choice also improves the access locality and reduces cache pollution, as SPP does not need to search in disjoint memory areas to fetch the metadata.

```
struct PMEMoid {
 uint64_t pool_uuid_lo; // pool id of the pool
 uint64_t off; // offset of the PM object
 uint64_t size; // size of the PM object
};
```

To ensure fault-tolerance, PMDK performs the PM object allocation and management either with atomic operations or software transactions. A PMEMoid is considered valid only after its offset field is set. Therefore, SPP adapts the PM management functions (e.g. alloc, realloc) to set the size field,

```
PMEMoid obj_id;
pmemobj_alloc(pool, &obj_oid, size, ...);//alloc a PM Object
void* pm_ptr = pmemobj_direct(obj_id);//get tagged ptr
pm_ptr += 42;//apply ptr arithmetics
__spp_updatetag(pm_ptr, /*off*/ 42);//update the tag
...
__spp_checkbound(pm_ptr, sizeof(int));//impl. bounds check
int x = (int)*pm_ptr;
...
clean_pm_ptr = __spp_cleantag(pm_ptr);//tag masking
uint64_t ptrtoint = (uintptr_t)clean_pm_ptr;
...
internal_foo(pm_ptr);//internal function call
clean_pm_ptr = __spp_cleantag(pm_ptr);//tag masking
external_foo(clean_pm_ptr);//external function call
...
__wrap_memcpy(src_pm_ptr, dst_pm_ptr, 42);//memcpy call
__wrap_strcpy(src_pm_str, dst_pm_str);//strcpy call
```

Listing 1. **SPP code transformation**: *modifications are highlighted in blue.*

given as an argument to each PM management function call, right before the offset field, thus preserving the semantics of PMDK. In a similar fashion, when a PMEMoid is modified inside a PMDK transaction using the dedicated PMDK API, SPP's size field is logged to preserve crash-consistency.

PMDK uses PMEMoids to locate objects across the runs of an application. It exposes the `pmemobj_direct` function that converts a PMEMoid to a 64-bit pointer to the PM object. SPP modifies this function to consider the size field of PMEMoid and return a tagged pointer (§IV-A). The additional operations to construct the PM pointer are presented below:

```
#define ADDRESS_BITS (PTR_SIZE - TAG - OVERFLOW - PM_BIT)
#define PM_PTR_BIT ((uint64_t)1 << (PTR_SIZE - 1))
#define OVERFLOW_BIT (~((uint64_t)1 << (PTR_SIZE - 2)))
...
void* pmemobj_direct(PMEMoid oid) {
 //calculate untagged PM pointer
 ...
 //Take the two's complement of the size
 uint64_t tag = (~oid.size + 1) << ADDRESS_BITS;
 return (void*)(pm_ptr | tag & OVERFLOW_BIT | PM_PTR_BIT);
}
```

Despite these changes, SPP does not alter the semantics of the PMDK programming model. It leaves both the atomic and the transactional APIs [24] intact, supports the type-safety macros [26] and provides multi-threading support with the same thread-safety guarantees with PMDK. Note that SPP's approach is not bound to PMDK but can be adapted for PM programming frameworks following similar principles.

C++ support. PMDK exposes a C API. However, the imposed limitations by the C semantics led Intel to develop C++ bindings in `libpmemobj-cpp` [60]. This library enriches `libpmemobj` with C++ features such as containers and smart pointers. To provide complete support for applications developed in C++, SPP adapts the base class for PM pointers to transparently use the modified `pmemobj_direct` function and consider the additional *size* field of the PMEMoid.

C. Compiler Passes

Transformation pass. The transformation pass of SPP instruments the target application by injecting the appropriate function calls to update the PM pointer tag, propagate its value and perform its masking. It identifies the instructions that involve pointer arithmetic operations and updates the tag accordingly (Listing 1 Line 5). It further cleans up the tag prior to *ld*/*st* instructions to perform the implicit bound check on the upcoming PM access (Listing 1 Lines 7-8).

However, in LLVM intermediate representation (IR) there is no distinction between the pointers to volatile memory and those to PM. SPP addresses this by performing static analysis on the produced IR: it tracks the pointer origins and skips inserting runtime checks to operations for pointers that are statically identified to point to volatile memory. Thus, SPP can remove the instrumentation for pointers to volatile memory. Similarly, for pointers that are guaranteed to point to PM, SPP can directly perform the tag cleaning, without checking the PM bit. For pointers, whose type cannot be deduced by SPP on compilation, SPP preserves their instrumentation and the runtime operations are performed based on their PM bit.

Additionally, pointers can be converted to integers and be used as operands in mathematical or comparison operations. The insertion of the *tag* can affect the correctness of these calculations. Therefore, SPP masks the tag of the pointer prior the pointer-to-integer conversion (Listing 1 Line 10).

LTO pass. SPP's link-time-optimization (LTO) pass ensures compatibility with non-instrumented shared libraries [92]. It scans through the application code and locates the external function calls with pointer arguments. Right before these calls, the LTO pass masks the pointer tag and passes the untagged pointers to the external function (Listing 1 Lines 14-15).

Further, SPP interposes the memory management and string functions (e.g., memcpy, strcpy) with wrapper functions (Listing 1 Lines 17-18). The wrappers are verifying the validity of the accessed address ranges based on the function parameters and perform the respective operation if no violation occurs.

D. Runtime Library

SPP's runtime library contains the implementation of the functions that are injected through the compiler instrumentation. These functions operate on SPP pointers to clean and update the tag after they verify that the pointer points to PM.

Precisely, the `__spp_cleantag` function returns the PM pointer after masking out its tag and PM bit as shown below:

```
/* PTR_BITS denote the bits for the virtual address */
#define PTR_MASK (1ULL << 62) | ((1ULL << PTR_BITS) - 1)
...
void* __spp_cleantag(void *ptr) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    /* keep the overflow and the virtual address bits */
    return ptr & (PTR_MASK);
}
```

Thus, the application gets the actual virtual address which can be normally accessed through *ld*/*st* instructions. The overflow bit is preserved so that any subsequent memory access through an overflown pointer is detected.

Further, the `__spp_updatetag` function is invoked when a tag needs to be updated due to a pointer arithmetic operation. The provided offset to this function is determined via the static analysis compiler pass. The tag of the given pointer is extracted and incremented by the offset value. If a PM pointer overflows, this operation implicitly sets the overflow bit. After

the tag update, the new value is merged into the PM pointer which is returned back to the application, as presented here:

```
void* __spp_updatetag(void *ptr, int64_t off) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    /* extract and update the tag */
    int64_t tag = (int64_t)__spp_extract_tag(ptr);
    tag = tag + off;
    /* return the updated tagged pointer */
    return (void*)__spp_insert_tag(ptr, tag);
}
```

Additionally, the runtime library implements the `__spp_checkbound` function, shown below. It is called prior to every PM access. It updates the tag based on the size of the dereferenced pointer type since this indicates the upper bound of the memory access. After the update, the PM pointer gets masked and is returned to the application to perform the intended access. If the overflow bit is set, this access will trigger a segmentation fault or a bus error.

```
void* __spp_checkbound(void *ptr, size_t deref_size) {
    /* check if ptr points to PM */
    if (!__spp_is_pm_ptr(ptr))
        return ptr;
    void* upd_ptr = __spp_updatetag(ptr, deref_size - 1);
    return __spp_cleantag(upd_ptr);
}
```

Lastly, the SPP runtime library includes the wrapper functions for memory intrinsic (e.g., memcpy, memmove) and string management functions (e.g., strcpy, strcmp). These functions perform memory accesses to specified address ranges. Therefore, SPP calculates the maximum addresses they intend to access for each PM pointer argument and updates the tag(s) before the actual function call. If any of the addresses lies outside the defined PM objects' boundaries, the respective pointer's overflow bit is set. Then, SPP masks out the tags and the PM bit and executes the built-in function. This execution will raise an error if any masked pointer is invalid due to the overflow bit, preserving SPP's memory safety properties.

E. Optimizations

SPP aims to provide spatial memory safety for PM with low overheads. To this end, the instrumentation includes optimizations to reduce the SPP function calls (e.g., for pointers to volatile memory) and merge or omit instrumentation steps, whenever possible (e.g., constant pointer increments in a loop).

Pointer tracking. SPP's compiler passes perform pointer origin tracking and divide the pointers into three categories based on the memory type they point to, namely, *volatile*, *persistent* and *unknown*. The category of each pointer is decided based on the API that generates it. More specifically, in SPP, we refer to the pointers returned by the traditional volatile memory management APIs (e.g., malloc, realloc, new) as *volatile*. Pointers referring to C++ Vtables and error handling are also known to be *volatile*. Equivalently, pointers that were constructed by specific PMDK functions (e.g., pmemobj_direct) are characterized as *persistent*. The remaining pointers (e.g., pointers loaded from memory) are considered *unknown*. SPP also tracks the derived pointers (e.g., via pointer arithmetics) and adds them in the category of their predecessor, if specified.

SPP's LTO pass proceeds one step further and analyzes the function pointer arguments. It scans the calling sites of each function and records the type of the pointer arguments passed by the caller. With this method, SPP can determine the category of a function pointer argument, provided that *all* the callers use pointers falling into a single category.

The benefit from the pointer classification is twofold. First, SPP can omit the instrumentation for the *volatile* pointers. avoiding multiple redundant function calls injection. Second, SPP can skip the PM bit check in its hook functions when operating on known *persistent* pointers. For pointers whose category cannot be determined (*unknown*), SPP keeps the instrumentation including the runtime pointer type checks.

Currently, the pointer tracking is designed for PMDK APIs. However, it can be adapted and incorporated in different PM frameworks that can benefit from or require the characterization of whether a pointer points to PM or volatile memory.

Bound checks preemption. SPP leverages the static analysis to identify basic code blocks and simple loops that include consecutive updates on the same pointer with constant offsets or following a known pattern during compile time. In this case, SPP's transformation pass calculates the maximum pointer offset and performs a single tag update followed by a dummy memory access on the updated pointer (blue highlight). This memory access acts as a bound check. It silently verifies the validity of the upcoming memory accesses related to this pointer, and, thus, SPP can omit the associated tag updates and bound checks in the specified code block (red highlight).

```
void* pm_ptr = pmemobj_direct(obj_id); // get tagged ptr
__spp_updatetag(pm_ptr, /*total_off*/ 16);
__spp_checkbound(pm_ptr, sizeof(int));
pm_ptr += 8;
__spp_updatetag(pm_ptr, /*current_off*/ 8);
__spp_checkbound(pm_ptr, sizeof(int));
int x = (int)*pm_ptr;
pm_ptr += 8;
__spp_updatetag(pm_ptr, /*current_off*/ 8);
__spp_checkbound(pm_ptr, sizeof(int));
int y = (int)*pm_ptr;
```

F. Additional Design Details

Crash consistency. SPP preserves the crash consistency property for the PM data. SPP adapts PMDK internally, so that the added *PMEMoid* field is set and updated in a fail-safe manner when the application uses the dedicated PMDK APIs.

Precisely, in PM allocations, SPP atomically sets the *size* field of the *PMEMoid* before PMDK validates the object allocation by assigning it with its *offset*. This is achieved through writing the *size* object in the redo log, which ensures that the setting of this field precedes the setting of the *offset*.

For the case of reallocation of a PM object, the entire *PMEMoid* structure is captured in a log. Since the amount of logged bytes is determined by the size of *PMEMoid* object, SPP does not have to interfere with this operation, further than simply setting the new *size* of the reallocated object.

Lastly, for the general case that a PM object containing a *PMEMoid* needs to be snapshotted in a transaction, the additional 8 B, that SPP introduces, are implicitly added in the transactional undo log. This is achieved with the help of

the type system that accounts for these bytes when calculating the *PMEMoid* size, e.g., with the `sizeof()` function.

However, if a *PMEMoid* is updated manually, it must be wrapped in a transaction and be snapshotted in the undo log by the developer. Thus, in case of an unexpected crash, the recovery process of PMDK will restore the logged value.

Address space layout. Reserving a part of the pointer for the PM bit and the tag reduces the number of available bits leading to virtual address space limitation. This limitation only affects the regions where a PM pool is mapped. Therefore, we configure our PMDK version to map the PM pools in the lower part of the virtual address space. The exact address space limit depends on the configurable tag size so that every PM object can be addressed using $(64 - tag_bits - 2)$ bits. Volatile memory management can utilize 63 out of the 64 bits (excluding the PM bit) which are sufficient for current systems [21]. In this case the address space layout randomization (ASLR) is disabled for PM mappings. While ASLR has a broader memory safety spectrum than SPP (e.g., use-after-free), its guarantees are probabilistic. Instead, SPP offers deterministic PM buffer overflow protection. Overall, a more sustainable future solution would be to use fat-pointers (e.g., 128 bits) where the first 64 bits contain the safety metadata. This approach comes with higher performance overheads as it requires additional memory accesses for metadata fetching.

G. Limitations

SPP detects PM buffer overflows in PMDK applications, provided that PM is managed with the PMDK APIs. SPP comes with inherent limitations due to *(i)* limited pointer bits, *(ii)* potential arbitrary pointer operations and *(iii)* the requirement for compatibility with pre-compiled shared libraries.

PM object & PM pool size. In SPP design, we face the limitations imposed by the 64 bit native pointers. The PM and overflow bits decrease the number of available bits to 62, which should enclose both the tag and the virtual address of the PM objects. The number of tag bits limits the PM object size while the virtual address space bits limit the maximum size of a PM pool. Therefore, we configure our PMDK version to set the maximum PM object size to *1<<tag_bits* bits and the maximum PM pool size to *1<<62-tag_bits* bits. However, SPP allows for a configurable amount of *tag_bits* which can be tuned by the developers. We apply SPP on sample applications shipped with PMDK and on a key-value store [59]. We observe that SPP provides complete coverage for these applications.

Pointer operations. Typically, the pointer subtraction is performed after converting the pointers to integers, based on the LLVM standard. Such operations on tagged pointers can lead to incorrect results. Therefore, SPP masks the pointer before the subtraction to provide the expected operation outcome.

Similarly, in cases of pointer comparisons, SPP also masks the pointers to ensure correctness. This process does not affect SPP's guarantees, since the converted values are only used for the comparison and are never dereferenced.

Additionally, when an application performs a pointer to integer operation, SPP preserves only the virtual address bits. Thus, the application receives the expected value. However, when an integer is converted to a pointer, SPP cannot provide its memory safety guarantees since the integer does not contain a tag, even if it was derived from a previously tagged pointer. The latter case could be addressed by tracking the origin and type of such pointers (e.g., with the use-def chain of LLVM [1]) and maintaining the tag in a data structure to restore it in an upcoming integer-to-pointer conversion.

In general, arbitrary pointer operations can result in an out-of-bounds pointer by an offset that resets the overflow bit hindering SPP from reporting the memory safety violation. A typical example is when the offset of a pointer exceeds the representation range of the address bits. Currently, this case is not handled explicitly but SPP can be enhanced to either emit an error or manually set the overflow bit. The former would be a better approach to prevent any further pointer misuse, since such actions mostly originate from malicious activities.

Lastly, SPP does not protect against arbitrarily generated pointers that might end up landing on PM, as it can be neither predicted nor prevented. Similar limitations apply in most memory safety approaches.

Shared libraries. SPP masks the pointers passed to shared libraries to preserve correct functionality. For the pointers returned from shared libraries, SPP cannot provide any memory safety guarantees as they are not guaranteed to be tagged and the way they are originated within the shared library is unknown. Therefore, SPP cannot assign them with a tag. SPP provides memory safety guarantees for PM pointers generated and preserved in the compilation units it has access to.

Additionally, SPP is not able currently to identify whether a shared library is instrumented, because SPP treats each compile module on its own. Therefore, the functions of shared libraries that an instrumented application is linked against, are seen as external. SPP precedes calls to these functions with a tag cleaning operation for their pointer arguments. Thus, despite the libraries being potentially instrumented, the tag is not propagated to them by the application and memory safety guarantees cannot be ensured in such cases.

V. Implementation

SPP is built based on PMDK v.1.9 and LLVM v.12.0. It consists of *(i)* a static analysis transformation pass, *(ii)* a link-time-optimization pass and *(iii)* a runtime library.

A. Compiler Support

Transformation pass. The SPP static analysis transformation pass scans the application and inserts the appropriate SPP runtime function calls. The pass operates on the LLVM Intermediate Representation (IR) of every translation unit.

Initially, the SPP transformation pass tracks the pointer variables of the IR. Global pointers and pointers to volatile heap allocated objects are *volatile*, while pointers derived through the `pmemobj_direct` function are *persistent*. The rest are classified as *unknown*. Following, each instruction of the module is examined. Based on the instruction type, the

SPP's transformation determines where to insert the appropriate $callsites$ to SPP's functions in the target module's IR.

More precisely, when the pass locates a pointer arithmetic operation, or a $GetElementPtrInst$ (GEP) in LLVM terminology, it calculates the offset of the operation and inserts a call to the __spp_updatetag to update the tag after the GEP, as shown in Figure 3. Similarly, on *ld/st* instructions, SPP updates the pointer tag based on the pointer type size and masks the pointer for the actual dereference by inserting a call to __spp_checkbound, as described in §IV-D. Further, calls to __spp_cleantag are injected before the pointer-to-integer conversions ($PtrToIntInst$) to preserve correct code behaviour. Lastly, SPP's transformation pass masks the pointer for function arguments passed by value ($Attr::ByVal$) since they implicitly perform an object copy.

LTO pass. The link-time-optimization (LTO) pass of SPP performs an analysis and instrumentation of the whole program during the linking phase for further optimizations. In our implementation we use the *gold* linker [22], [25].

We place our pass before the LLVM inliner in the pipeline. SPP compiles its runtime functions into object files. These files are linked against the target application's IR. In this way, SPP allows LLVM to apply its effective optimizations and perform the inlining of SPP's functions whenever possible. SPP hints the compiler to always_inline its functions and prevents them being optimized out with the used attribute.

Further, the LTO pass performs a more exhaustive pointer tracking, since it iterates over all the compile units of the application. Thus, SPP classifies further *volatile* and *persistent* pointer arguments by examining each function's calling sites. Using this information, it omits the pointer type check in the hook functions for the identified *persistent* pointers. Apart from that, this tracking allows SPP to prune injected calls when they have a *volatile* pointer as argument, whose category could not be determined via the transformation pass.

B. Runtime Library

PMDK wrappers. SPP includes wrappers for the core PMDK operations (i.e., PM heap management). The exposed PMDK API remains intact. The only deviation is that the environment variable PMEM_MMAP_HINT is set to 0 so that the PM pool is mapped to the lower part of the virtual address space.

SPP's wrappers are responsible to set and update the introduced 64-bit *size* field of a *PMEMoid* without violating the crash consistency property. Both the atomic and the transactional operations that affect the object's size (e.g., alloc, realloc) are considered, covering all the PM heap allocations. Precisely, for the persistence of the pointers, SPP provides the same atomicity guarantees with PMDK. In case of an object (re)allocation outside a PMDK transaction, SPP leverages the PMDK redo logging and performs an atomic operation that validates the (re)allocation after setting the *size* field in the *PMEMoid*. When the object management is performed within a PMDK transaction, SPP intercepts the functions that perform the snapshotting to ensure that the additional *size* field is included in the undo log, so that it can be restored in case of a crash during the transaction. SPP also performs a bounds check to prevent overflows on the snapshotted objects that could lead to information leakage through the transaction logs.

Lastly, SPP adapts the pmemobj_direct() function to construct a tagged pointer from a *PMEMoid* (§IV-B). It leverages the *size* field of the *PMEMoid* to create the tag and returns the tagged pointer to the caller.

Hook functions. SPP's runtime library contains a set of hook functions that are injected in the LLVM IR of the instrumented code. These functions update and mask the tag of PM pointers, as explained in §IV-D. Apart from the described hook functions, SPP implements equivalent functions with a _direct suffix that omit the pointer type check. They are only used when a pointer is determined to point to PM.

SPP handles separately the memory management functions, e.g., memcpy, memset and memmove. For each pointer operand, SPP injects a call to its __spp_memintr_check function. This function updates the tag based on the maximum address that the function operates on and masks the pointer. Then, the masked pointer is passed to the original memory management function. If the tag update sets the overflow bit, a fault will be triggered due to the invalid pointer during the function execution, showcasing the overflow.

Similarly, SPP interposes the common string manipulation functions (e.g., strcpy, strcat) at link time. The runtime library includes wrapper functions that perform the tag update and tag masking based on the arguments of each string function and then call the original function.

Lastly, SPP ensures *compatibility* between instrumented code and pre-compiled, uninstrumented libraries via its __spp_cleantag_external function. It is injected before the calls to external functions and removes the tag and the PM bit from the pointer arguments promoting interoperability. However, SPP has a caveat; a tagged PM pointer can be mistakenly passed to an external function as part of a struct, since SPP currently does not perform any intra-object analysis.

C. Optimizations

Pointer tracking. SPP compiler passes perform pointer tracking to differentiate between *volatile* and *persistent* pointers. This is to avoid redundant attempts to perform bounds checking and tag cleaning on tag-free volatile pointers.They iterate over each translation unit in the LLVM IR and categorise each pointer based on the way it is derived. More specifically, if a pointer is obtained via the pmemobj_direct of PMDK (or its equivalent get() function in C++), it is considered *persistent*. Similarly, pointers created via volatile allocation functions (e.g., malloc), pointers to C++ VTables (e.g., vfn or vtable prefixed) or pointers used by common functions that are known to point to the volatile heap (e.g., pthread_create) are classified as *volatile*. Further, pointers returned by external functions are accounted as *volatile*, to avoid the instrumentation, since they are not tagged. These pointer categories are also propagated via the GEP and BitCast LLVM instructions. The remaining pointers are characterized as *unknown*. This classification enables

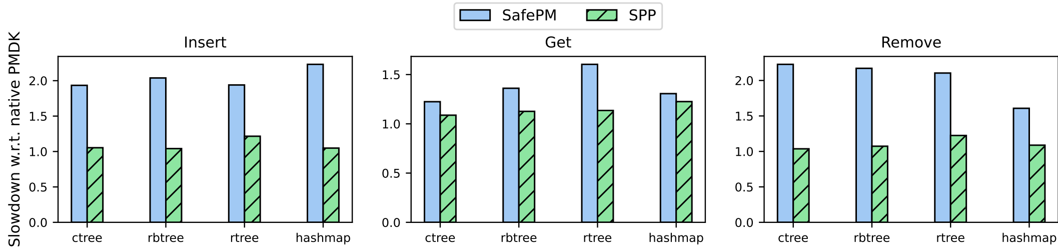

Fig. 4. Performance overheads (throughput) of persistent indices for SPP and SAFEPM w.r.t. the native PMDK execution.

SPP to remove useless function calls that are injected for *volatile* pointers and, equivalently, to omit the pointer type check for the *persistent* ones by using the `_direct` suffixed version of the hook functions. For the pointers with *unknown* type, SPP preserves the instrumentation and checks the PM bit to determine their type and perform the appropriate action.

Bound checks preemption. During development, we observed that many pointers are consecutively updated and dereferenced in a single LLVM $BasicBlock$. Therefore, instead of performing a costly tag update and masking for each `GEP` and `ld`/`st`, SPP calculates the maximum offset that is added to the pointer and updates the tag only once before the first `GEP`. Then, it places a dummy `ld` to implicitly perform the bound check and replaces the uses of the pointer with the masked one, which gets exempt from further instrumentation. The injected `ld` is tagged *v*olatile to avoid being optimized out by the compiler.

To further reduce the SPP's overheads, SPP hoists bound checks out of loops whenever possible. SPP checks every monotonic loop for existing *loop-invariant* expressions referring to pointers using LLVM's *scalar evolution*. If a pointer can be hoisted, SPP calculates its max offset that is used for dereference in the loop and places a tag update and a dummy `ld` in the loop *pre-header*. In this optimization, the `ld` is also tagged *v*olatile. Thus, SPP performs the pointer instrumentation only once rather than at every loop iteration.

However, due to the bound check preemption, SPP might indicate a false code location for an overflow, pointing to the injected dummy `ld`. To address this issue, bound check preemption optimizations can be optional and the programmer can choose to enable them depending on the target environment.

VI. EVALUATION

We evaluate SPP on the following three aspects:

- **Performance & space overheads:** We measure the performance (§VI-B) and space (§VI-C) overheads of SPP using PMDK's microbenchmarks, a persistent key-value (KV) store, designed and optimized for PM, namely pmemkv [59] and a port of Phoenix 2.0 [102] benchmark suite to PM.
- **Effectiveness:** We evaluate the capability of SPP in detecting PM buffer overflows (§VI-D). We use the RIPE framework [114] where we focus on buffer overflow exploits. We also detect memory safety bugs in the PMDK examples.
- **Crash consistency:** We verify the crash-consistency property (§VI-E) with Valgrind's `pmemcheck` tool [20].

TABLE I
BENCHMARKING VARIANTS

Variant	Description
PMDK [15]	PM application using unmodified PMDK
SAFEPM [33]	PM application instrumented with SAFEPM
SPP	PM application instrumented with SPP

A. Experimental Setup

Testbed. We conduct our experiments on a two-socket server machine, equipped with Intel(R) Xeon(R) Gold 6326 CPU (16 cores), 64 GB (4 channels $\times$ 16 GB/DIMM) DRAM and 1 TB (4 channels $\times$ 256 GB/DIMM) Intel Optane DC DIMMs per socket. PM is configured in App-Direct mode [66]. The machine is running NixOS 22.05 with kernel version 5.15.49.

Variants. We perform our experiments with the variants of Table I. As our baselines, we consider the application compiled with *(i)* native PMDK, and *(ii)* SAFEPM sanitizer enabled. We set the optimization level to O2 and use 26 *tag* bits. The results present the average of 3 runs, unless otherwise specified.

B. Performance Overheads

Persistent indices. To measure the performance overhead, we use `pmembench` [56]. For each variant of Table I we execute experiments on the PM indices of PMDK, namely *ctree*, *rbtree*, *rtree* and *hashmap*, with a single query type (*insert*, *get*, *remove*) per run. Each workload consists of one million queries. The keys are 8 B and follow a uniform distribution.

Figure 4 reports the normalized performance overhead for SAFEPM and SPP having the native PMDK as a baseline. Overall, SPP achieves 9.25%, 13.75% and 10.5% lower average throughput compared to PMDK for each query type. The respective values for SAFEPM are 101%, 37.75% and 101.75%. The large overhead difference comes from SPP's compiler optimisations and tagged pointer utilisation – SPP's LLVM pass removes redundant runtime checks on volatile pointers and unlike SAFEPM, SPP does not access remote memory regions for bounds information at every `ld`/`st`. Further, `pmembench` has limited external function calls. This allows SPP to perform better pointer tracing and reduce the tag cleaning operations for external functions. Additionally, compared to DRAM memory safety approaches, SPP introduces lower relative overheads since the performance impact of tag updating and cleaning operations in SPP is proportionally lower due to the slower PM access. For certain experiments, SPP approaches the native PMDK performance having an overhead of around 6%. This indicates the practicality of SPP.

Persistent KV store. We measure the performance impact of SPP on *pmemkv* [59] using its non-experimental, concurrent, persistent engine [23]. For our benchmarking, we use

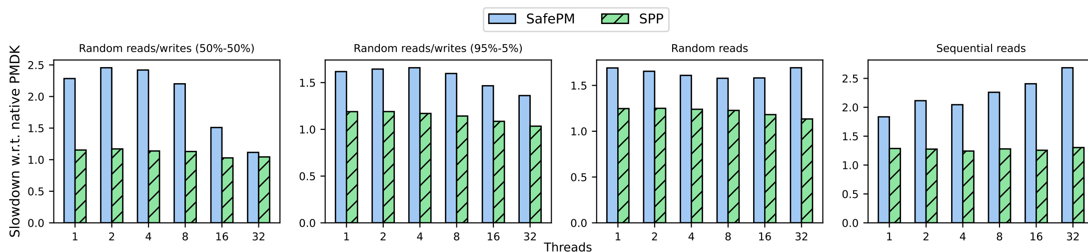

Fig. 5. Performance overheads (throughput) of SPP and SAFEPM w.r.t. the native PMDK execution for `pmemkv`.

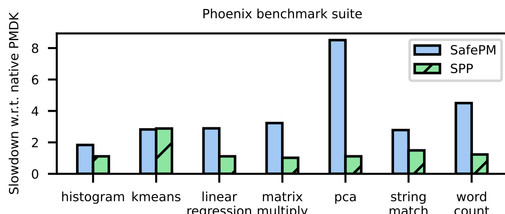

Fig. 6. Performance overheads of SPP and SAFEPM w.r.t. the native PMDK execution for the `Phoenix` benchmark suite.

`pmemkv-bench` [57], which is based on the *db_bench*. We consider four workload types: *(i)* update intensive (50%R-50%W), *(ii)* read intensive (95% R-5%W), *(iii)* random reads and *(iv)* sequential reads. We perform 10M operations for each workload. The key size is set to 16 B and the value size to 1024 B. Prior to each run, we insert 1M keys to the KV store.

Figure 5 illustrates the performance overhead of SPP and SAFEPM normalized to the PMDK. SPP causes an average 18.3% throughput decrease across the workloads while the respective value for SAFEPM is 84.4%. The overhead of SPP mostly stems from redundant checks for volatile pointers, that SPP cannot identify at compile time. Regarding the scalability, we observe that SPP follows a similar pattern to the PMDK, indicating the minimal effect of SPP on the parallel execution.

Phoenix benchmark suite. We evaluate the performance impact of SPP in CPU intensive scenarios. We port all 7 applications of the Phoenix benchmark [102] to use PM objects via the PMDK API. For each application, we use 8 threads and its largest provided dataset as input. To accommodate larger allocation sizes (e.g., the input files), we set the tag bits to 31 for SPP. The presented results are the average of 20 runs.

Figure 6 presents the slowdown of SPP and SAFEPM having PMDK as the baseline. SPP causes a slowdown of 2-23% for Phoenix benchmark applications, except for the *kmeans* where it incurs 180%. The respective values for SAFEPM range from 83% to 750%. The significantly reduced overheads of SPP can be reasoned by the effective pointer tracking since SPP has all the source code of the applications available for its analysis. The unique case of the *kmeans* benchmark can be justified as this application iterates constantly over its working set, leading to a higher performance impact of the SPP's instrumentation in its execution. Note that the Phoenix port is not optimized for PM. It uses plain memory intrinsic functions (e.g., `memcpy`) which do not allow SPP to avoid some pointer type checks. This implies that in a more sophisticated, PM-oriented port, the overheads of SPP can be further diminished.

Atomic and transactional PM operations. We evaluate the effect of SPP on PMDK's atomic and transactional PM management functions. We use `pmembench` [56], where we configure each experiment to perform 100K operations while varying the object size. We present the average of 10 runs.

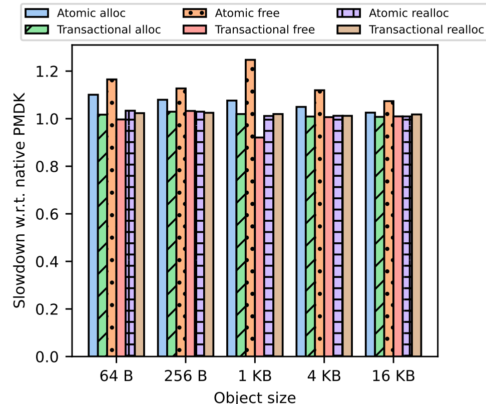

Fig. 7. Performance overhead of SPP for PM management operations.

TABLE II
RECOVERY TIME IN MILLISECONDS (*ms*).

Variant	Snapshotted *PMEMoids*				
	100	1000	10K	100K	1M
PMDK	17.62	17.78	18.82	28.52	119.77
SPP	17.77	17.86	18.87	28.66	120.00

Figure 7 reports the normalized slowdown of SPP for each PM management operation. For almost all the operations, the performance of SPP is close to the PMDK for the various object sizes (1-8% slowdown). It can be justified, as this microbenchmark only allocates PM objects and performs no PM access after the respective operation. Therefore, the overhead comes from redundant checks for the pointer type that SPP's static analysis cannot optimise away. The only operation with high overheads(7-17%) is the *atomic free*. This is due to SPP's required runtime checks, compared to a single atomic operation that PMDK requires to *free* the PM object.

Recovery time. We measure the recovery time of an application and compare SPP with PMDK. We develop a microbenchmark that allocates PM objects in a pool. We present a worst-case scenario for SPP where an application snapshots exclusively *PMEMoid*s in a transaction, resulting in larger logs for SPP. The number of objects per experiment is shown in Table II. After the snapshotting, we inject a crash and trigger a recovery. Our results indicate the average of 100 runs.

The slightly increased recovery time is caused mainly by the need for restoration of the additional *size* field of the *PMEMoid*. Note that SPP does not interfere with the internal recovery process of PMDK. However, user-defined recovery functions could pose higher overhead, since they are also

subject to SPP's instrumentation as part of the application, where SPP must provide its spatial memory safety guarantees.

C. Space Overhead

We measure the space overhead introduced by SPP compared to the native PMDK. SPP's space overhead is caused by the *size* field, added to the *PMEMoid* of PMDK. We reuse the four PM indices with the *insert* and *get* workloads and 1M keys, as explained in §VI-B. The reported values indicate the space overhead after the execution of the application.

The PM space overheads of SPP are presented in Table III. We conclude that SPP wastes minimal PM space to store its memory safety metadata (0-0.43%) for all persistent indices, except the rtree. In the extreme case of rtree, SPP consumes 39.7% more PM space compared to PMDK. It occurs, as each rtree node contains 256 *PMEMoids* and SPP's space overhead is proportional to the number of *PMEMoids* an application stores in PM. Note that this is not a common pattern in PM applications based on our observations. A future design can completely eliminate the PM space overhead if the object size gets encoded along with the object offset in the *PMEMoid* structure, thus requiring no extra PM space. This feature is not included in current SPP version due to time constraints.

TABLE III
SPP SPACE OVERHEAD

Data structure	Insert (MB)	Insert (%)	Get (MB)	Get (%)
ctree	0	0%	0	0%
rtree	2127	39.7%	2127	39.7%
rbtree	0	0%	0	0%
hashmap_tx	5	0.43%	5	0.43%

TABLE IV
RIPE ATTACKS USING DIFFERENT PROTECTION MECHANISMS.

RIPE variant	Successful	Prevented
Volatile heap	83	140
PM pool heap	83	140
SafePM	6	217
SPP	4	219
memcheck	20	203

D. Effectiveness

In this experiment, we examine the effectiveness of SPP. We use the RIPE benchmark framework [114]. It contains a set of different memory vulnerability exploits. We focus on buffer overflows. We use the 64-bit version of RIPE [3] that allocates objects on PM via PMDK [33]. We consider the following variants: *(i)* Volatile heap, where RIPE uses volatile memory, *(ii)* PM pool heap, where it uses the PM heap, *(iii)* SafePM, where it uses the PM heap with SAFEPM sanitizer enabled, *(iv)* SPP, where the application uses the PM heap and is instrumented with SPP and *(v)* memcheck [18], a valgrind tool for memory bugs in PM. Each memory exploit is executed 3 times in each run. We perform the RIPE experiments several times to ascertain the stability of our reported results.

Table IV shows the exploits that are successful or prevented throughout our runs. We observe that porting RIPE to use PM preserves the number of potential buffer overflow exploits (83). Out of these attacks, SPP is able to prevent 79 while SAFEPM detected 77. The memcheck [18] identified 63 attacks. We further examine the non-detected attacks by SPP and realise that the constructed PM buffer is only directly accessed in-bounds. Overall, SPP is capable of detecting almost every PM buffer overflow with notably lower performance overhead, compared to its state-of-the-art counterparts.

Reproducing bugs. To further verify the effectiveness of SPP, we reproduce and detect a reported PM buffer overflow bug in PMDK's btree index [11]. More specifically, on line 378 of *btree_map.c* file, the *memmove* call leads to a buffer overflow on a PM data object, which SPP is able to identify and report.

Additionally, we test various examples shipped with PMDK [63]. We apply SPP on implementations of an array, a queue, a FIFO list, a solution of Buffon's Needle problem, a program for the π calculation and a slab allocator. Using SPP, we identify three PM buffer overflows in the array example. Precisely, when an array *realloc* is requested, its return value is not checked. In case of a failed realloaction to a larger size, the application attempts to fill the newly, supposedly allocated, array which results in an overflow, as the original array is not resized. This bug occurs in lines 215, 235 and 257 of the array example [62]. The remaining examples do not report any error throughout their execution with arbitrary inputs.

Further, we identify an off-by-one buffer overflow in the `string_match` benchmark of Phoenix, when the `read` function is used for the input file. This bug is detected when we execute the ported version with SPP. It occurs when trying to access a character beyond the input buffer [74]. We verify our finding using ASan [104] on the volatile memory version. We report the bug [28] and its respective fix [29].

On top of that, we develop sample examples with various kinds of PM buffer overflows (e.g., overflows during snapshotting, built-in memory functions overflows, etc.) for testing. SPP identifies and reports all of the above cases.

E. Crash Consistency

We verify that SPP preserves the crash consistency for the PM data despite the addition of the *size* field in the PM pointer. We use `pmemcheck` [20] and `memcheck` [96]. `pmemcheck` is a Valgrind plugin that allows for exploring and verifying the data consistency in PM applications [33], [58], [67], [79]. Its output is passed to `pmreorder` [19] to explore the state space. We perform the same experiments as in §VI-B. Due to Valgrind's overheads, we set the number of operations to 10000 to shorten the execution time.

`pmemcheck` and `pmreorder` do not report any error. `memcheck` also has an empty error log for the persistent indices. For the PM operations, `memcheck` provides the same error output with the case of unmodified PMDK which is not indicating any crash consistency violation.

VII. RELATED WORK

Persistent memory systems. There exists a large body of work on PM filesystems that aims to reap the performance benefits of persistent memory as a storage medium [36], [72],

[111], [116], [123]. Further, persistent memory has already been incorporated in the design of many high performant data management systems [14], [16], [17], [59], [73]. On top of that, several proposed distributed systems leverage the capability of accessing PM remotely via RDMA to improve their efficiency [65], [71], [82], [106], [118], [122]. Unlike these systems that focus on high performance and correctness, SPP efficiently tackles the problem of memory safety on PM.

Further, recent works [32], [85], [113] leverage accelerators (e.g., FPGAs) where they offload PM operations (e.g., cacheline flushes, logging). Such systems can improve the PMDK performance. We expect that the performance boost will be similar for SPP, as it uses PMDK underneath for these operations, does not hamper the cache locality and has a minimal contribution to the amount of logged data.

SW memory safety approaches. Various software based approaches have been proposed to deal with the memory safety bugs for *volatile* memory [27], [30], [75], [76], [78], [90], [92], [104], [119]. The main target of these approaches is to preserve compatibility and high efficiency while incurring low performance and memory overheads. They apply different techniques such as pointer tagging [75] or the shadow-memory concept [104] and are often accompanied with compiler instrumentation [90], [119] and runtime libraries. Differently from such approaches, SPP focuses on memory safety for PM and achieves low runtime overheads for PM applications due to its conservative, yet effective, distinction of pointer types through the introduced PM bit as well as its selective instrumentation of PM pointers that eliminates redundant runtime function calls for volatile memory pointers. However, for complete memory safety, SPP can be combined with other SW-based memory safety approaches targeting volatile memory, since it is practically only affecting the PM pointer representation. Note that at the cost of additional performance overhead, SPP could be generalised and include instrumentation and checks for volatile memory pointers, similarly to prior work [75].

Targeting memory safety for PM, SafePM [33] and the valgrind-based `memcheck` [18] tool have been proposed. SafePM is built on google's AddressSanitizer [104] while `memcheck` leverages Valgrind and PMDK's internal code annotations to provide memory safety. Both these approaches incur considerable performance and space overheads and are destined for debugging purposes. On the contrary, SPP intends to be an efficient, low-overhead memory safety solution.

Lastly, Corundum [55] is a PM management library in Rust that enforces language-based memory safety. In contrast, SPP can be deployed in existing PMDK applications without any source code modifications and does not require re-developing applications with certain libraries or languages.

HW memory safety approaches. Striving for lower performance overheads, several works introduce HW extensions to provide memory safety for volatile memory [4], [40], [45], [48], [87], [88], [97], [98], [109], [115], [121]. Cheri [115] employs hardware capabilities while lowfat pointers [45] offer spatial memory safety using compact fat pointers that contain the encoded object bounds. They propose a hardware-level implementation for faster decoding and pointer validation. Intel MPX [97], [98] and Arm MTE [4] provide ISA extensions to prevent memory safety bugs for Intel x86-64 and Arm architecture respectively. Hardbound [40], SafeProc [48] and WatchdogLite [88] propose further ISA extensions that work collaboratively with a compiler instrumentation aiming towards better performance. In contrast to these approaches that rely on specialized HW or require ISA modifications and are dedicated for volatile memory safety, SPP can be used in commodity HW and detect memory safety bugs on PM.

PM allocators and libraries. PM allocation and management is an active area of research [34], [35], [39], [64], [99], [112]. Mnemosyne [112], NVHeaps [35] and PMDK [64] provide APIs for PM management and implement transactions to guarantee crash consistency. They distinguish between pointers to volatile memory and PM to avoid ephemeral or stale references being reused across restarts, if the developer uses their APIs correctly. Notably, NVHeaps [35] and Mnemosyne [112] are evaluated using simulated PM while PMDK is optimized to leverage the HW features of actual PM devices. These approaches, unlike SPP, do not provide any memory safety but only ways to manage PM in a correct manner, which SPP also performs as it is based on PMDK by design.

Poseidon [39] is a PM allocator that prevents metadata corruption using Intel MPK [100]. GPM [99] exposes an API to access PM directly from the GPU with respect to crash consistency and persistence. However, ensuring crash consistency is a non-trivial task. Therefore, multiple frameworks have been proposed to verify, or potentially ensure, this property for PM applications [32], [43], [47], [49], [70], [79]–[81], [93].

VIII. Conclusion

In this paper, we present SPP, the first tagged pointer-based mechanism designed to provide practical memory safety for PM applications. The PM pointer tag indicates the pointer distance from the end of the PM object and gets implicitly invalidated when it surpasses this boundary. SPP consists of a compiler instrumentation based on LLVM, a runtime library and an adapted PMDK version. It enhances the persistent pointer representation of PMDK with memory safety metadata which is set and updated in a crash-consistent manner. Its runtime functions ensure the correct tagged pointer management and provide compatibility with pre-compiled external libraries. SPP maintains the PMDK API intact. Consequently, SPP requires no source code modifications and can be seamlessly integrated into existing PM software. Our thorough evaluation shows that SPP effectively detects PM buffer overflows with low performance costs and negligible space overheads.

Software artifact. SPP is publicly available with the entire setup (https://doi.org/10.5281/zenodo.10211561).

Acknowledgments

We thank our shepherd, Prof. Jian Huang, and the anonymous reviewers for their helpful comments. This work was supported by a Schwerpunktprogramm (SPP) (ID: 2377) from Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] Iterating over def-use & use-def chains. https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains. May 23, 2024.

[2] Poject zero - stagefrightened? https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html, 2015. May 23, 2024.

[3] A 64-bit port of the RIPE benchmark. https://github.com/hrosier/ripe64.git, 2019. May 23, 2024.

[4] Memory tagging extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety, 2019. May 23, 2024.

[5] A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, 2019. May 23, 2024.

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

[7] The heartbleed bug. https://heartbleed.com/, 2020. May 23, 2024.

[8] 2021 cwe top 25 most dangerous software weaknesses. http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021. Accessed: 31-08-2021.

[9] AN INTRODUCTION TO PMEMOBJ (PART 3) - TYPES. https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-3-types/, 2021. Accessed 22-07-2022.

[10] The chromium projects - memory safety. https://www.chromium.org/Home/chromium-security/memory-safety, 2021. May 23, 2024.

[11] GitHub issue on the btree overflow. https://github.com/pmem/pmdk/issues/5333, 2021. Accessed 05-10-2021.

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

[13] Memcheck: a memory error detector. https://valgrind.org/docs/manual/mc-manual.html, 2021. May 23, 2024.

[14] Memhive: Scale applications with persistent memory! https://www.memhive.io/, 2021. May 23, 2024.

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

[16] Pmem-Redis. https://github.com/pmem/pmem-redis, 2021. May 23, 2024.

[17] pmem-rocksdb. https://github.com/pmem/pmem-rocksdb, 2021. May 23, 2024.

[18] pmem-valgrind. https://github.com/pmem/valgrind, 2021. May 23, 2024.

[19] The pmreorder utility. https://pmem.io/pmdk/pmreorder/, 2021. May 23, 2024.

[20] Valgrind: an enhanced version for pmem. https://github.com/efeslab/pmemcheck, 2021. May 23, 2024.

[21] Five-level page tables. https://lwn.net/Articles/717293/, 2022. Accessed 04-07-2022.

[22] GNU Binutils. https://sourceware.org/binutils/, 2022. Accessed 20-07-2022.

[23] pmemkv engines. https://github.com/pmem/pmemkv/blob/master/doc/libpmemkv.7.md#cmap, 2022. Accessed 04-07-2022.

[24] pmemobj API. https://pmem.io/pmdk/manpages/linux/v1.1/libpmemobj.3/, 2022. Accessed 04-07-2022.

[25] The LLVM gold plugin. https://llvm.org/docs/GoldPlugin.html, 2022. Accessed 20-07-2022.

[26] Type safety MACROS in libpmemobj. https://pmem.io/blog/2015/06/type-safety-macros-in-libpmemobj/, 2022. Accessed 04-07-2022.

[27] CAMP: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.

[28] Heap overflow bug in the string_match benchmark of the phoenix benchmark suite. https://github.com/kozyraki/phoenix/issues/9, 2024. May 23, 2024.

[29] Suggested fix for the heap overflow bug in the string_match benchmark of the phoenix benchmark suite. https://github.com/kozyraki/phoenix/pull/10, 2024. May 23, 2024.

[30] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium (USENIX Security 09)*, Montreal, Quebec, August 2009. USENIX Association.

[31] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 158–168, New York, NY, USA, 2006. Association for Computing Machinery.

[32] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 37–44, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. Safepm: A sanitizer for persistent memory. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 506–524, New York, NY, USA, 2022. Association for Computing Machinery.

[34] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, March 2011.

[36] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[37] CXL™ Consortium. Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, May 23, 2024.

[38] Thurston Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 04 2015.

[39] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 207–220, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 103–114, New York, NY, USA, 2008. Association for Computing Machinery.

[41] Dinakar Dhurjati and Vikram Adve. *Backwards-Compatible Array Bounds Checking for C with Very Low Overhead*, page 162–171. Association for Computing Machinery, New York, NY, USA, 2006.

[42] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM.

[43] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 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*, ASPLOS 2021, page 503–516, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Gregory J. Duck, R. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[45] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 132–142, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Gregory J. Duck and Roland H. C. Yap. 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*, PLDI 2018, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP '21, page 1–15, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory violations. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 652–657, 2009.

[49] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 415–428, New York, NY, USA, 2021. ACM.

[50] Istvan Haller, Erik Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software security hardening. pages 1–6, 04 2016.

[51] Jim Handy. SUnderstand how the CXL SSD can aid performance. May 23, 2024.

[52] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 135–144, New York, NY, USA, 2012. Association for Computing Machinery.

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

[54] Red Hat. Ghost: glibc vulnerability (cve-2015-0235). https://access.redhat.com/articles/1332213, 2015. May 23, 2024.

[55] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Intel. pmembench: PMDK benchmark framework, 2021. May 23, 2024.

[57] Intel. Benchmarking tools for pmemkv, 2022. May 23, 2024.

[58] Intel. Discover Persistent Memory Programming Errors with Pmemcheck, 2022.

[59] "Intel". "persistent memory development kit : pmemkv", 2022. May 23, 2024.

[60] Intel. Persistent Memory Development Kit : The C++ bindings to libpmemobj, 2022. May 23, 2024.

[61] Intel. Persistent Memory Development Kit : The libpmemlog library, 2022. May 23, 2024.

[62] Intel. Persistent Memory Development Kit : the libpmemobj array example, 2022. May 23, 2024.

[63] Intel. Persistent Memory Development Kit : The libpmemobj examples, 2022. May 23, 2024.

[64] Intel. Persistent memory development kit : The libpmemobj library, 2022. May 23, 2024.

[65] "Intel". Persistent Memory Development Kit : The librpma library, 2022. May 23, 2024.

[66] Intel. The Challenge of Keeping Up with Data, 2022. May 23, 2024.

[67] Louis Jenkins and Michael L. Scott. Persistent Memory Analysis Tool (PMAT), 2022.

[68] Trevor Jim, J. Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. pages 275–288, 01 2002.

[69] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. 05 2002.

[70] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

[71] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 105–119, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Chandan Kalita, Gautam Barua, and Priya Sehgal. Durablefs: A file system for persistent memory. *CoRR*, abs/1811.00757, 2018.

[73] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[74] Christos Kozyrakis. Phoenix benchmark overflow bug location. https://github.com/kozyraki/phoenix/blob/1276c8d8f3b82050071d0a7a4b8a352a05d1faab/phoenix-2.0/tests/string_match/string_match.c#L158. May 23, 2024.

[75] Taddeus Kroes, Koen Koning, Erik Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. pages 1–14, 04 2018.

[76] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 205–221, New York, NY, USA, 2017. Association for Computing Machinery.

[77] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[78] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. Giantsan: Efficient memory sanitization with segment folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, 2024.

[79] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 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*, ASPLOS 2021, page 487–502, New York, NY, USA, 2021. Association for Computing Machinery.

[80] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 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*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[81] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 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*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[83] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. *SIGOPS Oper. Syst. Rev.*, 42(2):115–124, mar 2008.

[84] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[85] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[86] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of c: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery.

[87] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 189–200, 2012.

[88] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 175–184, New York, NY, USA, 2014. Association for Computing Machinery.

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

[90] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve 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*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery.

[91] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.

[92] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 612–626, New York, NY, USA, 2019. Association for Computing Machinery.

[93] Ian Neal, Andrew Quinn, and Baris Kasikci. 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*, ASPLOS 2021, page 401–414, New York, NY, USA, 2021. Association for Computing Machinery.

[94] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[95] Netapp. What is persistent memory?, 2022. May 23, 2024.

[96] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[97] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017.

[98] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 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.

[99] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. *GPM: Leveraging Persistent Memory from a GPU*, page 142–156. Association for Computing Machinery, New York, NY, USA, 2022.

[100] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[101] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[103] Olatunji Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, NDSS, 2004.

[104] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[105] Anton Shilov. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. May 23, 2024.

[106] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), October 2020.

[107] Matthew S. Simpson and Rajeev K. Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 199–208, 2010.

[108] László Szekeres, M. Payer, Tao Wei, and D. Song. Sok: Eternal war in memory. *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[109] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. Cryptographically enforced memory safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 889–903, New York, NY, USA, 2023. Association for Computing Machinery.

[110] Victor van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. volume 7462, 09 2012.

[111] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[112] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[113] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez, editors, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 800–812. ACM, 2017.

[114] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. Ripe: Runtime intrusion prevention evaluator. pages 41–50, 12 2011.

[115] 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. The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News*, 42(3):457–468, June 2014.

[116] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[117] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.

[118] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 111–125, Santa Clara, CA, February 2020. USENIX Association.

[119] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 88–99, 2014.

[120] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Computing Surveys - CSUR*, 44:1–28, 06 2012.

[121] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. Capstone: A capability-based foundation for trustless secure memory access. In *32nd USENIX Security Symposium*

(USENIX Security 23), pages 787–804, Anaheim, CA, August 2023. USENIX Association.

[122] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.

[123] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.