DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Memory Safety for Persistent Memory: Safe Persistent Pointers

Alexandrina Panfil

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Memory Safety for Persistent Memory: Safe Persistent Pointers

Speichersicherheit für Persistenter Speicher: Sichere Persistente Zeiger

Author:	Alexandrina Panfil
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Dr. Myoung Jin Nam, Dimitrios Stavrakakis
Submission Date:	March 15th, 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, March 15th, 2022 Alexandrina Panfil

Abstract

One of the biggest vulnerabilities in computer security is violation of memory safety. System software written in low-level languages such as C/C++ produces memory corruption bugs that enable attackers to take control of it and alter its behaviour.

Even though most defense mechanisms are designed and implemented for volatile memory, non-volatile memory is just as prone to have memory safety bugs. Recent years saw a considerable increase in Persistent Memory's (PM) usage in computer systems, which further emphasizes the need for memory safety mechanisms for PM.

We propose Safe Persistent Pointers (SPP), a memory defense mechanism for PM that leverages state-of-the-art approaches on memory safety.

SPP is mainly based on the tagged-pointer approach that was first proposed in the Delta Pointers Project. It detects out-of-bounds accesses on 64 bit pointers by maintaining a *delta tag* on the upper bits of the pointer and performs update. Evaluation results show a very high effectiveness against buffer-overflow attacks.

Contents

1 Introduction

Programming languages such as C/C++ are frequently employed for writing system software because of their high performance. However, they implement pointers as direct memory accesses and allow the use of arbitrary pointer arithmetic with no provision for bounds-checking, which makes the programs written in them more prone to have security vulnerabilities.

In the past decades numerous memory safety mechanisms have been developed and adopted as industry standards. Basic protection settings include Stack Canaries, Address Space Layout Randomization, Data Execution Prevention, etc. [SPWS13] However, several attack vectors are still effective, and the current effective mechanisms induce a big overhead in either performance or memory usage.

Despite the large body of work that was done towards ensuring memory safety, memory corruption bugs remain one of the biggest vulnerabilities in system software. Fine-grained memory safety (against both spatial and temporal errors) is extremely expensive to guarantee, more so for system languages such as C/C++.

Recent years have seen a rapid increase in the usage of the Persistent Memory (PM) technology, which has the characteristics of both memory and storage. Its main property is crash consistency, while it also provides performance close to DRAM.

Just like volatile memory, PM is vulnerable to memory corruption bugs, but in comparison to its volatile counterpart, there are no defense mechanisms that were designed for PM or are fully compatible with PM. One reason for making the existing technologies incompatible with PM is that PM uses a persistent pointer representation for the objects that it handles, while volatile memory uses native volatile pointers. Additionally, persistent memory allocators can pose more safety threats than the ones of volatile memory due to its persistent heap metadata [DKK+20]. Therefore, the design of protection mechanisms for PM must ensure the protection of the objects managed by the application as well as the persistent heap metadata.

One promising approach to counter buffer overflows in volatile memory is Delta Pointers, which is a tagged-pointer methodology where metadata is stored as a *delta tag* in the upper bits of the pointer and it is used to implicitly manage the out-of-bounds state of the pointer. Therefore, when the overflow bit is set the pointer cannot be used for memory access. Delta pointers has no space overhead, while the performance overhead is much better than competing approaches (close to 35%).

1.1 Contributions

We propose SPP, which leverages the Delta Pointers mechanism, to ensure protection against buffer overflows for Persistent Memory (PM). SPP aims to enforce memory safety based on low-level inline reference monitors, preventing memory errors by tracking live pointers to persistent objects and inserting bounds-checks.

Just like Delta Pointers, SPP instruments the application with instructions to keep the *delta tag* consistent and perform accurate bounds-checking on pointers at memory accesses. That is, our contributions are as follows:

- We propose a memory defense mechanism for PM that is based on a state-of-the-art approach
- We present and discuss the effectiveness of pointer tagging for PM applications written in C/C++
- A prototype of our design that is composed of a series of LLVM compiler passes and supporting changes in PM's Persistent Memory Development Kit (PMDK)

2 Background

2.1 Persistent Memory

PM is a relatively new technology that converges features that are characteristic to both memory and storage: it provides much larger capacities (128 GiB, 256 GiB, and 512 GiB) than DRAM while providing byte-addressability and an access latency similar to DRAM. PM is also offering flexibility through its two main operational modes: *(i)* memory mode and *(ii)* app-direct mode [Int21l]. In the first mode, PM is volatile and is extending the capacity of addressable main memory. Furthermore, in this mode the memory controller of the CPU is using PM as main memory and DRAM as a cache. In the app-direct mode, PM is non-volatile, i.e. the data stays persistent after the machine is powered off. Furthermore, the OS connects with PM via PM-aware direct access file systems (DAX) [arc21; ind21]. DAX eliminates the page cache from the I/O path and allows mmap(2) to establish direct mappings to PM [Int21a]. Thus, PM content can be directly accessed as memory mapped files in an application's address space.

2.2 Persistent Memory Development Kit (PMDK)

In order to ease the development, testing and evaluation of PM applications, Intel has introduced a series of libraries and tools in Persistent Memory Development Kit (PMDK) [Pmd]. The libraries contained in PMDK were designed to provide support for a wide range of applications by managing PM as a persistent transactional object store [Int21e], a KV store [Int21b] or an extension of the volatile memory [Int21i; Int21j]. It also exposes API to create PM resident logs [Int21d] or to treat PM as an array of persistent, crash-consistent blocks [Int21c]. Further, PMDK provides options for remote PM access [Int21g; Int21h] and other tools for PM pool management [Int21k; Int21f].

2.3 libpmemobj

`libpmemobj` [Int21e] is a library contained in PMDK that provides an API for managing persistent objects, transactions. Similar to malloc/free in volatile memory, the API of libpmemobj ensures crash consistency . It further implements software based transactions to support atomic updates of PM data that exceed the size of 8 leveraging redo and undo logging.

2.3.1 Transactions

In order to help the programmer keep the application consistent, PMDK introduces transactional operations that can be distinguished as allocation, free and set. They can be managed through the usage of functions that have a `pmemobj_tx_*` prefix. For instance, *pmemobj_tx_add_range()* and *pmemobj_tx_add_range_direct()*

2.4 Persistent memory pool

PM content is organised in files, called *persistent memory pools*. These files are mapped into a contiguous region in the application's virtual address space during runtime. PM pools managed by `libpmemobj` place a pool header at their start, containing metadata for the pool's attributes. The following pool section is dedicated to the pool logs. It is split into smaller parts, called *lanes*, each of which stores its own set of logs. Lanes are assigned to distinct threads to support concurrent transactions. The remaining part of the pool is the persistent heap, where the PM objects are allocated.

PMDK interacts with a persistent memory pool through a structure called Pool Object Pointer (POP) that contains metadata about the pool. It resides in volatile memory and is created at every program invocation.[Int] A more detailed overview of a persistent memory pool is illustrated in Figure 2.1.

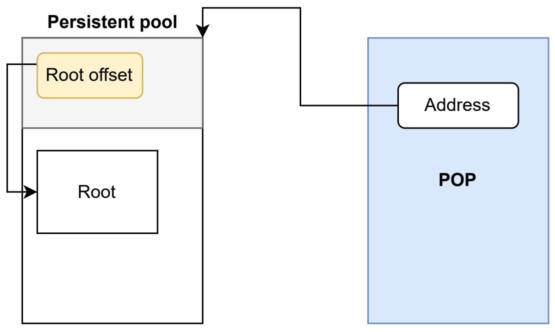

Figure 2.1: Overview of a persistent memory pool.[Int]

2.5 Persistent pointers

PM pools can be mapped in different regions of the virtual address space on each run. To maintain consistent object references across restarts, `libpmemobj` introduces the concept of *persistent pointers*. It relies on a fat-pointer scheme. Each object is described by a 16-byte structure, called `PMEMoid`, which contains the `pool_id` of the pool it belongs to and its *offset* relative to the beginning of the PM pool file. `libpmemobj` exposes a function (`pmemobj_direct()`) to construct the correct native pointer for an object, based on its persistent pointer representation. The computing of each persistent pointer is similar to the one in Figure 2.1.

2.6 Persistent memory management

`libpmemobj` provides an allocation mechanism to manage the PM heap. It focuses on efficiency while minimizing fragmentation and avoiding PM leaks. `libpmemobj` splits the PM heap into smaller parts, called *chunks* and *runs*, and maintains heap-related data structures in volatile memory to improve its performance. It supports all the common memory operations (e.g. alloc, realloc, free). [DKK+20]

2.7 LLVM

LLVM is a collection of compiler and toolchain technologies which can be used to develop a front-end for any programming language or a back-end for any instruction set architecture using its Intermediate Representation (IR). [Llva]

2.7.1 LLVM Intermediate Representation

LLVM IR is a low-level language that is similar to Assembly. It is a Static Single Assignment (SSA) representation that is able to represent any programming language.

Getelementptr Instruction

getelementptr (GEP) instruction is used to obtain the address of a subelement of an aggregate data structure (arrays, structs, etc.), thus it is the underlying IR instruction behind pointer arithmetic operations.[Llva]

```
%0 = getelementptr %struct.ST, %struct.ST* %s, i32 1
```

Listing 2.1: Example of getelementptr instruction, accessing the second index of a structure

Load/Store Instructions

Memory can be accessed through the load/store instructions. Load is used for reading from a specific address, while store is used for writing to a memory address. [Llva]

```
%ptr = alloca i32
store i32 8, i32* %ptr
%val = load i32, i32* %ptr
```

Listing 2.2: Example of load/store instructions in LLVM IR that are performed on a newly allocated pointer

2.8 Memory safety vulnerabilities

Years of research in the field of system security [SPWS13] show that memory bugs are the primary reason for safety and reliability issues in software systems written in low-level unsafe languages as C and C++. It is characteristic for these languages to provide facilities for low-level memory manipulation and have poor built-in memory protection capabilities. Specifically, they allow arbitrary pointer arithmetic with pointers implemented as direct memory addresses and with no provision for bounds checking. There are two types of memory bugs: *(i)* spatial and *(ii)* temporal.

2.8.1 Spatial memory safety bugs

Spatial memory bugs describe accesses to prohibited memory regions such as buffer under- or overflows or out-of-bounds errors.

Buffer overflows

The most common type of *spatial* memory safety bugs are *buffer overflows*. Out-of-bounds memory accesses (read/write) can corrupt from adjacent objects of the targets, either by leaking data or manipulating the return addresses to change the application's behaviour. It is not uncommon that *buffer overflow* attacks can help intruders bypass ASLR.

The following code presents a simple *buffer overflow* error, where the adjacent memory of a buffer gets overwritten with a string manipulation function:

```
#include<stdlib.h>
...
{
    char buffer[10];
    strcpy(buffer, "This payload will overflow the buffer");
}
```

The most efficient way to insure complete *spatial memory safety* is to monitor pointer bounds - the upper and lower address of the object it points to.

Numerous approaches have been developed to enforce *spatial memory safety*[NBMZ09; KKK+18; ACCH09; KOA+17; SBPV12] in C/C++ applications. Unfortunately, almost all mechanisms incur performance and/or space overhead.

2.8.2 Temporal memory safety bugs

Temporal memory bugs describe accesses to memory regions either before or after they have been allocated or deallocated, respectively, such as *use-after-free* or *double free* errors.

Dangling pointers are encountered at dereferencing pointers right after the object they pointed to has been deallocated and the portion of memory was returned to the memory management system. The following code presents a typical case where `ptr` becomes a dangling pointer after the memory has been deallocated:

```
#include<stdlib.h>
{
    void *ptr = malloc(sizeof(int));
    ...
    free(ptr);
}
```

Approaches that can ensure temporal memory safety [NZMZ10] can classified in two broad categories:

- avoiding temporal memory bugs in the code itself
- avoiding the exploitation of temporal memory errors.

2.9 Memory safety approaches

The described bugs are some of the main vulnerabilities for attackers to exploit in order to access unprotected memory regions and, subsequently, be able to change the program behaviour or leak data [VDSCB12; MML+16]. To prevent malicious attacks, out-of-bounds memory accesses have to be prevented. This requires a suitable instrumentation (hardening) of an application to perform bound checks on memory accesses. Numerous memory safety approaches have been developed based on high-level [SBPV12; HMS12; ACCH09; DYC17; NS07] or on low-level [DY16; WWC+14; OKB+18] implementations.

More specifically, the proposed approaches can be classified into three categories [NMZ15]: trip-wire, object-based and pointer-based.

2.9.1 Trip-wire or shadow memory based approach

This approach utilizes guard memory regions called redzones that are marked as inaccessible (poisoned) and are placed around every memory object. With this approach, any memory access will change values in these regions and consequently be detected [SBPV12; HMS12; HJ91].

2.9.2 Object-based approach

Such approaches check all the pointer manipulations to ensure that the resulting pointer has a valid referent (object that it points to) [ACCH09; DKA06; DY16; DYC17; Eig03; DA06; RL04]. In this class, metadata is tracked on a per-object level. This class's main idea is enforcing the intended referent, i.e., making sure that pointer manipulations do not change the pointer's referent object

2.9.3 Pointer-based approach

Approaches of this category consider the upper and lower bounds that a pointer is allowed to access and keep track of whether the pointer is not used to access memory t the appropriate bound checks on every memory load or store [NBMZ09; SB10; NCH+05; NZMZ10; JMG+02]. Metadata is kept on a pointer-based level.

Pointer-tracking approaches generally provide near-complete spatial memory safety. A pointer's metadata, which contains status information about its allocation/dereference along with the object's upper- and lower bounds, is stored in order to instrument pointer dereferences and memory accesses.

2.9.4 Delta Pointers

Delta Pointers [KKK+18] is a pointer-based memory safety approach that stores a *delta tag* in the upper bits of pointers. The *delta tag* is the two's complement negative distance to the end of the object, and pointer arithmetic instructions are instrumented to mirror the operation on the delta tag. In case of an overflow, the distance will become negative, causing top-most bit to be set, which renders the pointer undereferencable, until a later operation brings it back in bounds. The delta tag bits need to be masked before dereferencing pointers, but the top-most bit is left as is to let the memory protection hardware detect overflown pointers.

Delta pointers has negligible space overhead, as it required no padding, aligning or data structure. Some major drawbacks are the lack of protection against underflows, as well as temporal violations. Moreover, the size of the delta tag constitutes a trade-off between the amount of virtual address space available to the application and the maximum object size. In the paper, the size of the delta tag is set to 31 bits, which leaves 32 bits to the application.

3 Related Work

There is a large body of work that was done towards ensuring safety for volatile memory. This section presents prior approaches and discusses their trade-offs between detection coverage and performance.

3.1 Software Memory Safety Approaches

These works can be classified into *trip-wire* and *bounds metadata-based* approaches. In *trip-wire* methods, the objects are surrounded with specialized memory regions, called *red zones*, and every access is checked for validity. Subsequently, these approaches introduce significant memory overhead. For example, AddressSantizer [SBPV12] (ASan) allocates an extra 1/8th of the virtual address space in addition to the per-object red zones to support such checks. *Bounds metadata-based* approaches, on the other hand, use a technique in which the upper- and lower-bounds of the objects are stored as metadata, either as part of the object [KOA+17; NBMZ09] or associated with the object itself [DKA06]. Typically, this metadata is then extracted to perform out-of-bounds checks at memory accesses. In respect to *trip-wire* methods, these approaches reduce memory consumption significantly, however, they can arise performance, programmability, and safety issues at runtime.

In addition to the memory and performance trade-offs, there is a preeminent difference between these two classes: *trip-wire* supports temporal safety and is able to catch bugs such as *use-after-free* and null pointer dereferences, while *bounds metadata-based* have a fine-grained intra-object under/overflow detection but non deterministic coverage for temporal safety bugs. Therefore, the approaches used to harden a certain program should be chosen based on the application's characteristics and behaviour.

3.2 Baggy Bounds

Baggy bounds [ACCH09] registers a low performance overhead for certain scenarios. It instruments only pointer operations and skips dereferences. All the objects are padded and aligned to the nearest power of two, and then the base-2 logarithm of the size is stored in the upper bits of pointers. Then, the base of a given pointer can be computed by rounding it down to the nearest multiple of its size.

Baggy Bounds incurs a high space overhead due to its power-of-two aligning and padding. Additionally, this solution does not offer temporal memory safety.

3.3 Low-Fat Pointers

In the Low-fat pointers [DY16] approach, the size information is stored within the pointer representation by dividing the virtual address space into chunks of size 4 GB each, and objects of a predefined size are placed on each chunk. To insure this type of mapping, memory allocation functions are intercepted and modified by a memory allocator such that all objects are size-aligned. This size-alignment enables the base of the pointer to be computed by rounding the pointer down to the nearest multiple of its size.

Like other pointer-based approaches, pointer dereferences have to be instrumented with bounds-checking operations. The implementation discussed in the paper provides no protection against memory violations that happen within the stack, largely because information on the size of stack objects cannot be stored within the pointer representation, as the stack is fixed to a region. This approach only offers protection against spatial violations.

3.4 Address Sanitizer

ASan reserves a portion of memory (shadow memory) where it marks different portions with special tags, for instance memory that has been assigned to an object is tagged as *addressable*, whereas freed memory is marked as *freed*. The stored metadata covers all the volatile memory regions, including heap, stack and globals, and keeps . All memory accesses that occur inside an application are instrumented by a series of compiler passes, and there are also mechanisms to keep the shadow memory updated accordingly to heap region, i.e. to library function calls that manipulate the heap memory. Overflow- and underflow- accesses are detected by the so-called *redzones*, which are inaccessible. However, the *redzone* approach does not always detect the accesses that skip over the *redzone* and access another access. In order to detect temporal bugs, Asan also implements a FIFO queue called *quarantine*, which is meant to delay memory reuse.

From the presented approaches, the shadow memory based approach, as adopted by AddressSanitizer (ASan) [SBPV12], is the most popular memory safety technique. It is widely to detect both spatial and temporal memory safety violations. However, as

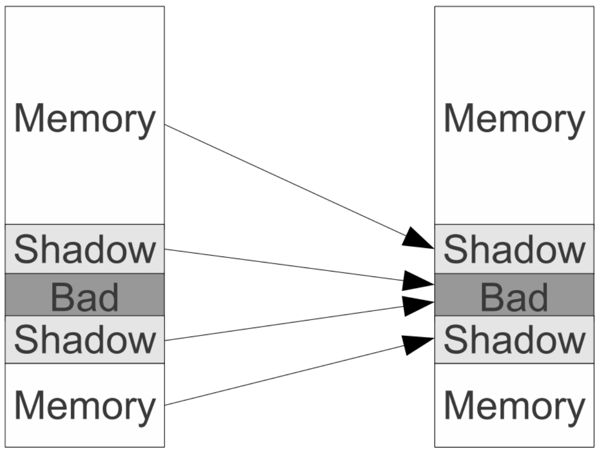

Figure 3.1: AddressSanitizer memory mapping. Reprinted from 'AddressSanitizer: A Fast Address Sanity Checker'[SBPV12]

priorly discussed, it incurs a large space overhead, thus it is frequently used only in the testing phase of an application.

Several pointer-based approaches [NBMZ09; OKB+18] choose memory layout compatibility over the speed with high locality. Using *disjoint metadata* achieves compatibility by decoupling metadata from a pointer representation and storing metadata in a remote memory region. SoftBound [NBMZ09] implements both a hash table or shadow memory space to map pointers to the metadata. Unfortunately, the performance overhead of SoftBound is comparably high, 79% on average [NBMZ09].

3.5 SGXBounds

SGXBounds is a primary example of the *tagged pointers* [**sgxbounds**; KKK+18] approach, which leverages the fact that some portion of the pointer's bits are unused e.g. top 16 bits in a 64-bit pointer. A specific insight about the tagged pointers approaches is that they encounter a trade-off between *address space* and almost-complete spatial memory safety. SGXBounds encode the metadata in the object's footer as shown in Fig. 3.2, and reserves the upper 32 bits of a pointer to store the address of the footer. This way, the upper bound is placed on the upper bits while the rest of the object size metadata is placed on the footer's location.

Storing the absolute address of bounds frees SGXBounds from violation of intended referents that challenge many object-tracking approaches. However, the tagged-pointer

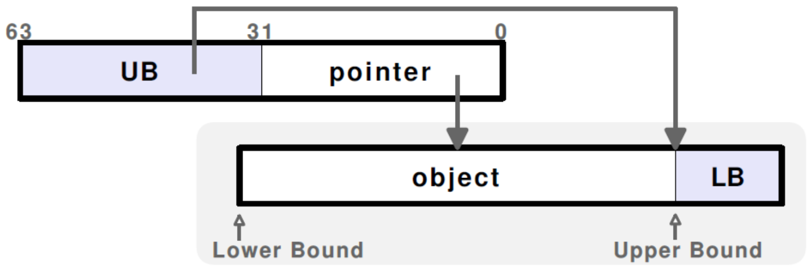

Figure 3.2: Pointer Representation in SGXBounds. Reprinted from 'SGXBOUNDS: Memory Safety for Shielded Execution'[KOA+17]

approach works in specific instances when there is a lower priority for a large address space and/or there is a portion of the pointer that is unused. The latter is the case with SGX enclaves, which use only 36 bit pointers.

3.6 PM based systems

The research community developed several tools [NRS+20; LMRK21] that can help at testing the correctness of PM applications. But these frameworks do not add protection against memory corruption bugs.

3.7 PM safety

The memory defense techniques that were discussed in this section were all designed for *volatile memory* and therefore they do not take into consideration neither persistence nor crash-consistency of the persistent heap metadata. Thus, they cannot be applied to PM applications.

3.7.1 Poseidon

Poseidon [**Poseidon**] proposed a novel persistent memory allocator that is based on Intel Memory Protection Keys (MPK) features to prevent corruption of the persistent metadata through memory safety bugs. Poseidon leverages MPK's page protection granularity to separate objects from their metadata. The major drawback of Poseidon is that it provides protection against memory corruption only for the persistent metadata, therefore application memory remains unhardened against memory safety errors.

3.7.2 Corundum

Corundum [HS21] is a Rust library for PM applications that enforces memory safety by blocking the usage of unsafe programming constructs, which proves to be very unflexible in terms of programmability as users have to depend on Corundum to make use of its memory defense mechanism.

In contrast to Poseidon and Corundum, SPP does not require its users to change the design/implementation of their applications, but rather targets the software that uses PMDK's functionalities.

4 Safe Persistent Pointers Design

4.1 Overview

SPP leverages the Delta Pointers encoding method in which the pointer is stored along with a *delta tag* and an overflow bit for storing the out-of-bounds state of the pointer. In order to be able to encode the object's size into the pointer, we modify *libpmemobj*'s persistent object handle known as *PMEMoid* struct to store the object size along with the existing metadata. Our SPP prototype was tested and evaluated for pointers that have a 24-bit delta tag, but it can be configured to support tags of any length.

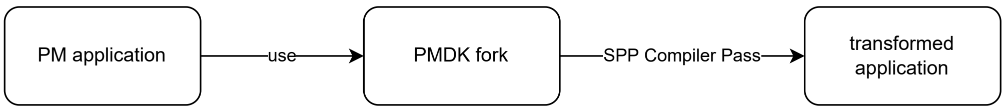

Figure 4.1: Interplay between PMDK and SPP-Pass

4.1.1 Flow of usage

The flow of using SPP is as follows: the programmer uses our modified *libpmemobj* to allocate persistent objects, after which the PM application should be compiled with the SPP compiler passes enabled. The transformation passes instrument the PM application with code that updates the so-called *delta tags* of pointers of persistent objects. The interaction between the SPP components and the PM application is presented in Figure 4.1.

4.2 Pointer Encoding

The pointer encoding in SPP does not differ from the one used in Delta Pointers. We keep the encoding of the object size (distance to the end of the object) into a delta tag along with an overflow bit. Upon calling the `pmemobj_direct()` function, we extract the size of the object and use it to obtain the initial delta tag, after which we place it along with the overflow bit on the upper 25 bits of the pointer. The initial delta

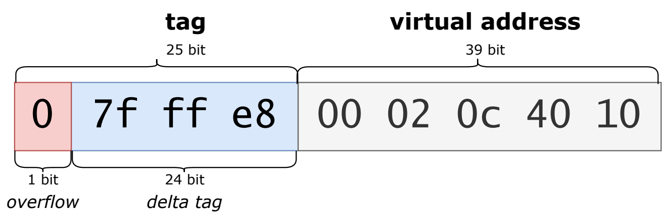

Figure 4.2: Encoding of the Pointer

tag represents the negated size of the object ($-object\_size$), and it is kept consistently updated at each pointer arithmetic operation.

4.3 Runtime Operations

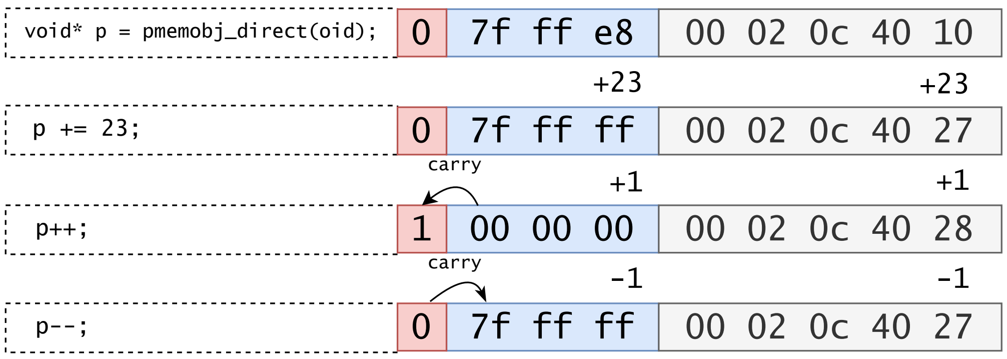

Figure 4.3: Instance of how the metadata bits get updated in parallel to the virtual address bits at every pointer arithmetic operation. The pointer is to an object of size 23, thus the initial delta tag is -23 and the pointer gets invalidated after it is incremented by 24.

The delta tag of the persistent pointer gets updated throughout pointer arithmetic operations, as exemplified in Figure 4.3. `pmemobj_direct()` returns an encoded pointer, where, as described previously, the last 39 bits of the pointer correspond to the virtual address of the beginning of the persistent object. The first 25 bits represent the metadata that gets consistently updated at each pointer arithmetic operation.

The main idea of delta pointers' approach is that the overflow bit is managed implicitly at arithmetic operations, therefore, at bounds-checking we only need to check the overflow bit instead of performing expensive comparison operations.[KKK+18]

Whenever the program tries to access memory with a pointer, we check if the overflow bit is set on and stop the program accordingly. If the overflow bit is not set, we apply another bitwise AND operation to mask away the encoded delta tag as shown in 4.4. By "cleaning" the upper bits we ensure that the CPU operates on a regular pointer that will not cause a segmentation fault or induce unwanted behaviour to the program.

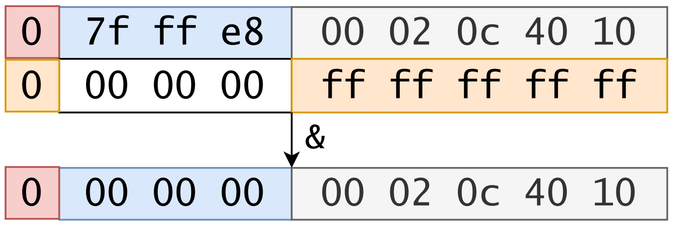

Figure 4.4: Tag cleaning i.e. masking away the metadata from the pointer. The overflow bit is kept in order to prevent out-of-bounds data access.

4.4 Discussion and Limitations

4.4.1 Thread Safety

Just like Delta Pointers, SPP are thread-safe on the grounds that the instrumentation that is added by the compiler passes are arithmetic instructions that operate on registers and no additional memory accesses are introduced.

4.4.2 Pointers as integers

The optimizations that are implemented into the LLVM infrastructure can produce arbitrary typecasts from pointers to integers, which makes the compiler pass miss the instrumentation to update or mask the tags and therefore leads to unwanted behavior of the program. In order to solve this issue, the pass needs to perform static analysis and determine if these integers are pointers. This can be achieved by two distinct approaches.

4.4.3 Performance Overhead

Redundant instructions from instrumenting untagged pointers would incur a significant performance overhead. We did not implement any related optimizations as part of the set of transformation passes, thus this would be one of our future work directions.

4.4.4 Space Overhead

Delta Pointers project incurs insignificant space overhead [KKK+18] because the additional metadata is stored in the upper pointer bits and there are no additional memory allocations. SPP, on the other hand, allocates 8 bytes of memory for storing the size of the persistent object. This would induce 50% space overhead on data structures that store persistent indices.

5 Implementation

This section presents the current implementation of SPP which is built using the PMDK library and LLVM infrastructure. Additionally, we describe how the project components are interconnected and how we ensure compatibility between instrumented modules and regular pointer representation.

System overview. There are two main parts of our implementation:

PMDK fork and

LLVM-assisted program instrumentation

5.1 PMDK fork

5.1.1 Managing persistent objects

PMDK's *libpmemobj* library stores metadata about each persistent object in its `PMEMoid` handle, which is subsequently used to compute the pointer to the object or to extract the necessary information at object manipulations. In addition to the existing pool id and object's offset inside the pool, we make `PMEMoid` store the object's size which would allow us to compute the *delta tag*.

5.1.2 Memory allocation

In contrast to the Delta Pointers project, the pointers are not encoded by the compiler pass, but at the high-level of PMDK's `pmemobj_direct()` function which computes the pointer of a persistent object. `pmemobj_direct()`'s original version adds the object's offset to the address of the pool it is located in. We modify it to additionally place the *delta tag* and the overflow bit on the upper bits of the pointer. Then we modify the memory allocation functions (`pmemobj_alloc()`, `pmemobj_realloc()`, `pmemobj_free()`) to ensure crash consistency for the newly introduced field that stores the object size. It is important to note that we do not track pointers that are allocated as global variables.

5.1.3 Address space layout

Reserving a portion of the pointer for the *delta tag* and the overflow bit leaves fewer bits for the actual address, which in turn reduces the size of addressable virtual address space. For this reason, we configure our fork of PMDK to map the persistent memory pools in the lower part of virtual address space. PMDK uses 64 bit pointers with maximum object size of 2 GB [Int21e]. We configure our PMDK fork to limit the maximum object allocation size to 1 $<<$ *tag_size* bits and the maximum persistent memory pool size to 1 « 63 - *tag_size* bits.

5.1.4 Interface address

The library is implemented as a fork of the PMDK. There are no substantial changes to the library API.

5.2 LLVM-assisted Program Instrumentation

SPP's instrumentation consists of two parts: LLVM transformation passes and runtime library of hook functions.

The flexible structure of LLVM framework facilitated the implementation using function interposition and IR traversals. One of our main transformation passes is implemented as an LLVM Link Time optimization (LTO) pass for whole program analysis, and runs as an LTO pass on gold linker [Llvb]. This aims at avoiding segmentation faults that can be caused by the *delta tag* bits.

5.2.1 Hook functions and wrappers

The target source code and hook functions in C are first compiled to LLVM intermediate representation (IR). Our first transformation pass runs on each translation unit, and inserts callsites to our hook functions at pointer arithmetic (for tag update) and memory access (for tag cleaning) in the target IR code. After per-module instrumentation, the second transformation pass, implemented as an (LTO) pass, instruments external function calls/jumps for the whole program. The hooks, which are pre-compiled to IR, are linked to target IR codes as an LLVM IR object file, instead of as static libraries. This aims at enabling a function inliner and taking full control over LLVM pass pipeline for the customized optimization. We have the inserted callsites later inline to target codes by placing our transformation pass before the LLVM-builtin function inliner during the link time optimization (LTO), thus avoid jumps to the functions, which slowdowns program execution.

Table 5.1: SPP inserts code, highlighted in gray, for creating a tagged pointer, updating metadata and detecting out-of-bounds accesses. Codes in line 2, 5, and 8 in the first column are transformed to codes in the second column.

	Original C	Instrumented C
1	`struct my_root {size_t len, char buf[20];};`	`struct my_root {size_t len, char buf[20];};`
2	`PMEMoid root = POBJ_ROOT(pop, struct my_root);`	`PMEMoid root = POBJ_ROOT(pop, struct my_root);`
3	`void* ptr = pmemobj_direct();`	`void* ptr = pmemobj_direct();`
4		`void* ptr1 = clean_tag(ptr);`
5	`printf("%lu ", ptr);`	`printf("%lu ", ptr1);`
6	`ptr++;`	`ptr++;`
7		`ptr2 = update_tag(ptr, 1);`

Several C library functions (`free` routine and string functions) can get overriden to LLVM intrinsic ones at optimization steps, which prevents the pass from performing bounds-checking at memory accesses. To ensure that those functions are memory safe and tag-free, we introduce wrappers around them to perform preventive bounds-checking. The following is a code example of a C library function wrapper that adds preventive checks and cleans the tag before passing the pointers to the original function:

```
int
    __wrap_strcmp(char *str1, char *str2)
    {
        return __real_strcmp(check_and_clean(str1, strlen(str1) + 1),
                            check_and_clean(str2, strlen(str2) + 1));
    }
```

Another solution to counter this problem would be to implement customized functions instead of wrapping, which would also reduce the overheads created by calling subroutines. We also introduce similar wrappers for several `libpmemobj` transaction functions that can be exploited for memory corruption (`pmemobj_tx_add_range(), etc.`).

5.2.2 Pointer Arithmetics

Generally, our compiler pass instruments pointer arithmetic operations by inserting a callsite to our runtime library functions. After every getelementptr (GEP) instruction, we add the offset to the *delta tag* and replace the subsequent pointer uses with the newly updated pointer. Table 5.1 showcases a simple example in which a pointer gets incremented and the pass instruments the code with a callsite to `update_tag()` hook function. Rough example of how our hook function updates the tag:

```
void*
```

```
update_tag(void *ptr, int64_t off) {

    if (!extract_x86_tagval(ptr))
    {
        return ptr;
    }

    tag = extract_tagval(ptr);
    tag = tag + off;

    ptrval = clean_tag(ptr) | (tag << (64 - TAG_BITS));

    return ptrval;
}
```

5.2.3 Memory Accesses

SPP's first transformation pass traverses each translation unit and inserts callsites to our bounds checking function right before each `load/store` instruction, so that each pointer is examined and its tag is stripped-off before being dereferenced. The hook extracts the *delta tag* from the pointer; gets the header location; performs the check using metadata in the header; and then returns an untagged pointer after cleaning the tag. The transformation pass replaces a tagged pointer operand of `store/load` with an untagged one to avoid segmentation fault caused by dereferencing it.

Our tag cleaning hook function takes into consideration that persistent memory has two kinds of pointers with different address spaces: pointers to persistent memory utilise only lower 39 bits in a 64-bit pointer, while volatile pointers use full 48 bits. These two cases are easily handled, since a pointer tagged in the top 16 bits is surely pointing to persistent memory. If a pointer is not tagged in upper 16 spare bits, it is either a volatile one, which does not need tag cleaning, or a persistent pointer that has already been masked by a previous tag cleaning operation. Therefore, we mask the upper 25 bits if the top 16 bits are tagged; otherwise the hook returns the pointer itself. The following code presents the previously described behaviour:

```
void*
clean_tag (void *ptr)
{
    if (!extract_x86_tagval(ptr))
    {
        return ptr;
    }
    return (void*)((ptr << TAG_BITS) >> TAG_BITS);
}
```

Bounds checking and masking are also performed on `memcpy`, `memmove` and `memset` in similar way. (Note that LLVM overrides the C library functions to their intrinsic ones). `memmove` and `memcpy` has two pointer operands, so we instrument each argument separately.

As for string functions, these are interposed at link time. Wrapper functions perform preventive checks on their arguments; call the `wrapped` functions with pointers masked from tags; and then return the pointer with its original *delta tag*.

5.2.4 Interoperability

SPP ensures compatibility between the instrumented modules and pre-compiled non-instrumented libraries with regular pointer representation by masking off tagged pointers before passing them to non-instrumented functions.

There may exist several cases of false positives, where an external function uses tagged pointers to read from memory, and our instrumentation did not mask every tag as follows:

```
 struct Node {   // Linked list node
  int data;
  struct Node* next;
};

int length(struct Node * head){
  int result = 0;
  struct Node* current = head;
  while (current != NULL) {
    result++;
    current = current->next;
  }
  return result;
}
```

At the `length()` function call, SPP's pass does not clean all instrumented nodes except the `head` pointer that has been passed as the function argument. One way to address this is to completely track the allocated memory – performing a whole program analysis, and instrument only objects whose pointers stay inside the instrumented modules. A heavy static analysis will, however, skip too many objects from tracking and leave room for memory exploitation. To ensure that the *delta tags* are masked at every memory access, one could use hardware support or, with a performance overhead, configure a segmentation fault handler.

6 Evaluation

The evaluation of SPP is structured around two important dimensions.

- **Performance overheads.** We evaluate the performance (§ 6.2) of SPP using PMDK's micro-benchmarks.
- **Effectiveness.** The effectiveness of SPP (§ 6.3) is evaluated using the RIPE framework [WNY+11] that focuses on testing the exploitability of buffer overflows.

6.1 Experimental Setup

6.1.1 Testbed

All the experiments were conducted on a server machine, equipped with AMD EPYC 7713P CPU with 64 cores, 540 GB RAM running NixOS 21.11 ("Porcupine") with x86_64 Linux kernel version 5.10.103.

6.1.2 Variants

The experiments are conducted with 2 variants of PMDK: native PMDK, with baseline configuration (no instrumentation), and PMDK with SPP instrumentation. The native PMDK was compiled with gcc.

In the native version, we link the application against the unchanged PMDK version without any instrumentation. This will serve as our benchmark. The variant with SPP stands for our complete tool. Here applications are linked against our changed PMDK fork and are compiled with clang with our transformation passes and wrappers enabled.

6.2 Performance Overheads

It is important to note that only the SPP variant provides protection from persistent memory errors, and we test the other variant in order to evaluate the overhead of our instrumentation and wrappers.

We evaluate the performance of SPP using four different persistent indices (`ctree`, `btree`, `rbtree`, `rtree` and `hashmap`) that are part of the example section of PMDK.

6.2.1 Persistent indices

We evaluate the performance of SPP using four persistent indices over the mentioned variants. We use `pmembench` [**pmembench**], deployed as part of PMDK, and perform 5×10^5 insert, get and remove operations on each data structure. Each key is 8 bytes and the operations chooses the keys randomly.

Figures 6.1, 6.2 and 6.3 illustrate the throughput of SPP version normalised to the native PMDK execution. In general, SPP is 1.7-4.25×, 1.84-14.34× and 3.01-23.96× slower than the native PMDK for the insert, remove and get operations, respectively. We observe that the biggest performance overhead is incurred at get operations. This is explained by the large number of memory accesses that our compiler pass had to instrument with bounds-checking.

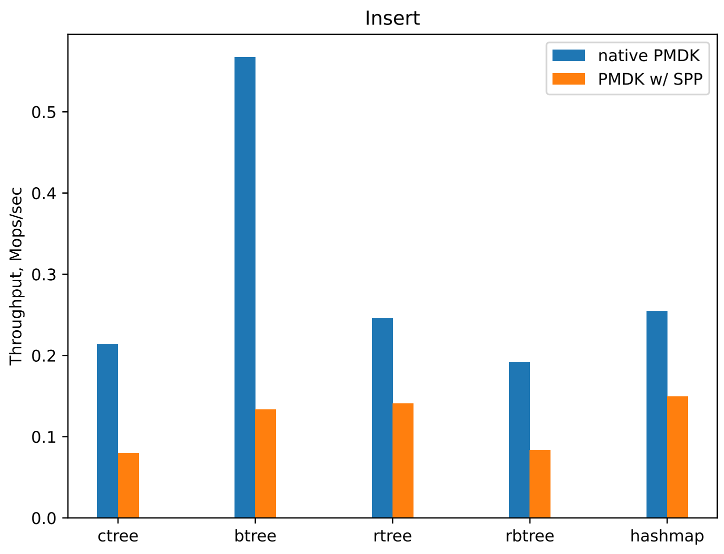

Figure 6.1: Performance overheads for inserting values into data structures.

6.3 Effectiveness

The effectiveness of SPP is evaluated using the RIPE framework [WNY+11], a comprehensive suite of memory vulnerability exploits. To ensure compatibility with SPPwe

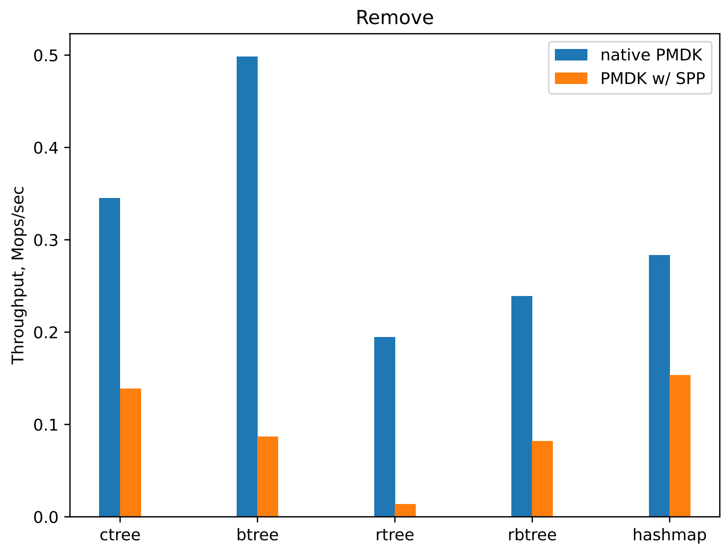

Figure 6.2: Performance overheads for removing values from data structures.

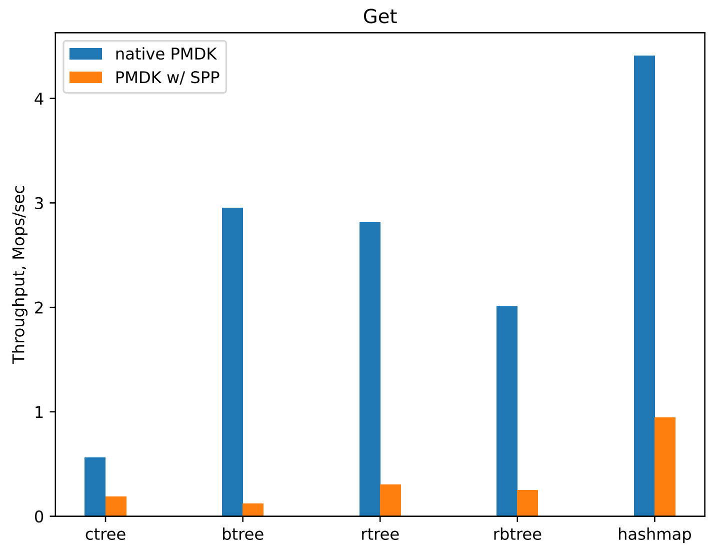

Figure 6.3: Performance overheads for extracting values from data structures.

use a modified version of the 64-bit port of the RIPE benchmark [Rip] to compare the effectiveness of the following variants: *(i)* Intact with PM heap, *(ii)* SPPwith PM heap.

RIPE variant	Always	Sometimes	Never
Intact	83	0	140
SPP	4	0	219

Table 6.1: Number of RIPE attacks that always, sometimes or never succeed .

We perform each exploit from the RIPE benchmark *3* times. An exploit is marked as *always* if it succeeds in all attempts. If it succeeds only in some trials or in none of the attempts, it is marked as *sometimes* or *never*, respectively. We have conducted the experiment several times to ensure that the results are stable. Note that out of all the attacks provided by the RIPE benchmark, only 223 could be applicable given our experimental setup.

Table 6.1 reports the number of RIPE buffer-overflow attacks that were proven to succeed *always*, *sometimes* or *never*. When the application uses the volatile heap, *37%* of the attacks are *always* successful. However, when instrumenting the application with SPP passes and using our wrappers for C library functions, the number of successful attacks reduces to *1.7%*. This proves that SPP achieves a greater memory safety effectiveness for the PM heap. In the 4 cases of attacks that succeeded, the injected payload didn't have a size greater than the size of the persistent buffer, thus SPP could not detect an overflow in these instances.

7 Future Work

7.1 Performance Optimization

Our first step towards reducing the performance overhead of our tool is to introduce optimizations that would skip instrumenting untagged pointers.

7.2 Failure-oblivious Computing

In order to avoid sudden program crashes at runtime, we would like to implement a failure-oblivious approach similar to the one presented in the SGXBounds project [KOA+17]. The idea is that out-of-bounds memory accesses get redirected to an overlay address in an LRU cache as illustrated in Figure 7.1. The return values from the cache memory are zero values.

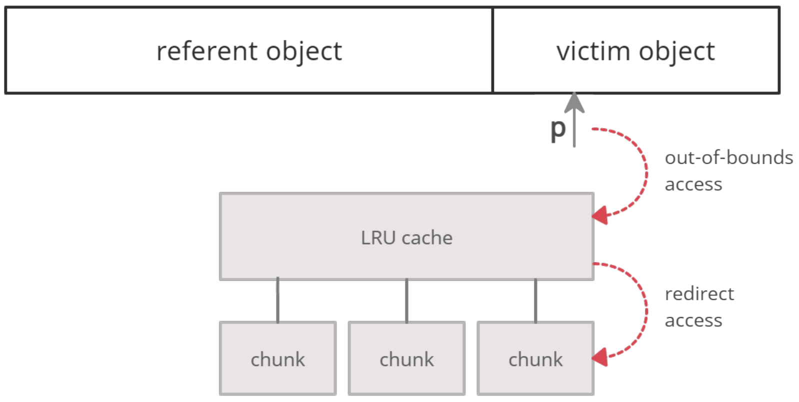

Figure 7.1: Failure-oblivious computing is enforced by returning an address that is located on an LRU cache when the program tries to access an out-of-bounds address.[KOA+17]

8 Conclusion

In summary, our work applies a highly practical memory protection mechanism to be compatible with PM. We demonstrate that Delta Pointers approach proves its efficiency in defending applications against buffer overflows. We also discuss current limitations and future steps towards improving our tool.

List of Figures

List of Tables

Bibliography

[ACCH09] P. Akritidis, M. Costa, M. Castro, and S. Hand. “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.” In: Jan. 2009, pp. 51–66.

[arc21] T. L. kernel archives. *DAX - Direct access for files*. 2021. URL: `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[DA06] D. Dhurjati and V. Adve. “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead.” In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2006, 162–171. ISBN: 1595933751. URL: `https://doi.org/10.1145/1134285.1134309`.

[DKA06] D. Dhurjati, S. Kowshik, and V. 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*. Ottawa, Ontario, Canada: ACM, 2006, pp. 144–157. ISBN: 1-59593-320-4. DOI: `http://doi.acm.org/10.1145/1133981.1133999`.

[DKK+20] A. Demeri, W.-H. Kim, R. M. Krishnan, J. Kim, M. Ismail, and C. Min. “Poseidon: Safe, Fast and Scalable Persistent Memory Allocator.” In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, 207–220. ISBN: 9781450381536. DOI: `10.1145/3423211.3425671`. URL: `https://doi.org/10.1145/3423211.3425671`.

[DY16] G. Duck and R. Yap. “Heap bounds protection with low fat pointers.” In: Mar. 2016, pp. 132–142. DOI: `10.1145/2892208.2892212`.

[DYC17] G. J. Duck, R. Yap, and L. Cavallaro. “Stack Bounds Protection with Low Fat Pointers.” In: *NDSS*. 2017.

[Eig03] F. Eigler. “Mudflap: Pointer use checking for C/C++.” In: Jan. 2003.

[HJ91] R. Hastings and B. Joyce. “Purify: Fast detection of memory leaks and access errors.” In: *In Proc. of the Winter 1992 USENIX Conference*. 1991, pp. 125–138.

[HMS12] N. Hasabnis, A. Misra, and R. Sekar. "Light-Weight Bounds Checking." In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: Association for Computing Machinery, 2012, 135–144. ISBN: 9781450312066. DOI: `10.1145/2259016.2259034`. URL: `https://doi.org/10.1145/2259016.2259034`.

[HS21] M. Hoseinzadeh and S. 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. Virtual, USA: Association for Computing Machinery, 2021, 429–442. ISBN: 9781450383172. DOI: `10.1145/3445814.3446710`. URL: `https://doi.org/10.1145/3445814.3446710`.

[ind21] indradead. *Direct Access for files*. 2021. URL: `https://www.infradead.org/~mchehab/kernel_docs/filesystems/dax.html`.

[Int] *Intel's webpage on Optane PMem Solutions*. `https://web.archive.org/web/20210406170804/https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`. Accessed: 2021-04-06.

[Int21a] Intel. *Documentation for ndctl and daxctl*. 2021. URL: `https://pmem.io/ndctl/ndctl-create-namespace.html`.

[Int21b] Intel. *Persistent Memory Development Kit : pmemkv*. 2021. URL: `https://pmem.io/pmemkv/`.

[Int21c] Intel. *Persistent Memory Development Kit : The libpmemblk library*. 2021. URL: `https://pmem.io/pmdk/libpmemblk/`.

[Int21d] Intel. *Persistent Memory Development Kit : The libpmemlog library*. 2021. URL: `https://pmem.io/pmdk/libpmemlog/`.

[Int21e] Intel. *Persistent Memory Development Kit : The libpmemobj library*. 2021. URL: `https://pmem.io/pmdk/libpmemobj/`.

[Int21f] Intel. *Persistent Memory Development Kit : The libpmempool library*. 2021. URL: `https://pmem.io/pmdk/libpmempool/`.

[Int21g] Intel. *Persistent Memory Development Kit : The librpma library*. 2021. URL: `https://pmem.io/rpma/`.

[Int21h] Intel. *Persistent Memory Development Kit : The librpmem library*. 2021. URL: `https://pmem.io/pmdk/librpmem/`.

[Int21i] Intel. *Persistent Memory Development Kit : The libvmem library*. 2021. URL: `https://pmem.io/vmem/libvmem/`.

[Int21j] Intel. *Persistent Memory Development Kit : The libvmmalloc library*. 2021. URL: https://pmem.io/vmem/libvmmalloc/.

[Int21k] Intel. *Persistent Memory Development Kit : The pmempool utility*. 2021. URL: https://pmem.io/pmdk/pmempool/.

[Int21l] Intel. *The Challenge of Keeping Up with Data*. 2021. URL: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[JMG+02] T. Jim, J. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. "Cyclone: A safe dialect of C." In: Jan. 2002, pp. 275–288.

[KKK+18] T. Kroes, K. Koning, E. Kouwe, H. Bos, and C. Giuffrida. "Delta pointers: buffer overflow checks without the checks." In: Apr. 2018, pp. 1–14. DOI: 10.1145/3190508.3190553.

[KOA+17] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. "SGXBOUNDS: Memory Safety for Shielded Execution." In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, 205–221. ISBN: 9781450349383. DOI: 10.1145/3064176.3064192. URL: https://doi.org/10.1145/3064176.3064192.

[Llva] *LLVM Project Documentation*. URL: https://llvm.org/doxygen.

[Llvb] *The LLVM Gold Plugin*. URL: https://www.llvm.org/docs/GoldPlugin.html.

[LMRK21] S. Liu, S. Mahar, B. Ray, and S. 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. Virtual, USA: Association for Computing Machinery, 2021, 487–502. ISBN: 9781450383172. DOI: 10.1145/3445814.3446691. URL: https://doi.org/10.1145/3445814.3446691.

[MML+16] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. 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. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, 1–15. ISBN: 9781450342612. DOI: 10.1145/2908080.2908081. URL: https://doi.org/10.1145/2908080.2908081.

[NBMZ09] S. Nagarakatte, A. Bb, M. Martin, and S. Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C.” In: vol. 44. June 2009, pp. 245–258. DOI: `10.1145/1542476.1542504`.

[NCH+05] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. “CCured: Type-Safe Retrofitting of Legacy Software.” In: *ACM Trans. Program. Lang. Syst.* 27.3 (May 2005), 477–526. ISSN: 0164-0925. DOI: `10.1145/1065887.1065892`. URL: `https://doi.org/10.1145/1065887.1065892`.

[NMZ15] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. “Everything You Want to Know About Pointer-Based Checking.” In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 190–208. ISBN: 978-3-939897-80-4. DOI: `10.4230/LIPIcs.SNAPL.2015.190`. URL: `http://drops.dagstuhl.de/opus/volltexte/2015/5026`.

[NRS+20] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci. “AGAMOTTO: How Persistent is your Persistent Memory Application?” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1047–1064. ISBN: 978-1-939133-19-9. URL: `https://www.usenix.org/conference/osdi20/presentation/neal`.

[NS07] N. Nethercote and J. Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.” In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 89–100. ISBN: 9781595936332. DOI: `10.1145/1250734.1250746`. URL: `https://doi.org/10.1145/1250734.1250746`.

[NZMZ10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. “CETS: Compiler Enforced Temporal Safety for C.” In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM ’10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, 31–40. ISBN: 9781450300544. DOI: `10.1145/1806651.1806657`. URL: `https://doi.org/10.1145/1806651.1806657`.

[OKB+18] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. “Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack.” In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[Pmd] *The PMDK webpage*. `https://web.archive.org/web/20210227154002/https://pmem.io/pmdk/`. Accessed: 2021-02-27.

[Rip] *RIPE64: a 64bit port of the Runtime Intrusion Prevention Evaluator*. 2019. URL: `https://github.com/hrosier/ripe64`.

[RL04] O. Ruwase and M. Lam. "A Practical Dynamic Buffer Overflow Detector." In: *NDSS*. 2004.

[SB10] M. S. Simpson and R. 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*. 2010, pp. 199–208. DOI: `10.1109/SCAM.2010.15`.

[SBPV12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: a fast address sanity checker." In: June 2012, pp. 28–28.

[SPWS13] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: `10.1109/SP.2013.13`.

[VDSCB12] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. "Memory Errors: The Past, the Present, and the Future." In: vol. 7462. Sept. 2012. ISBN: 978-3-642-33337-8. DOI: `10.1007/978-3-642-33338-5_5`.

[WNY+11] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. "RIPE: Runtime Intrusion Prevention Evaluator." In: Dec. 2011, pp. 41–50. DOI: `10.1145/2076732.2076739`.

[WWC+14] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk." In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), 457–468. ISSN: 0163-5964. DOI: `10.1145/2678373.2665740`. URL: `https://doi.org/10.1145/2678373.2665740`.