Fast Pointer Nullification for Use-After-Free Prevention

Yubo Du
University of Pittsburgh
yubo.du@pitt.edu

Youtao Zhang
University of Pittsburgh
youtao@pitt.edu

Jun Yang
University of Pittsburgh
juy9@pitt.edu

***Abstract*—Low-level programming languages like C and C++ offer dynamic memory management capabilities but are vulnerable to Use-After-Free (UAF) vulnerabilities due to improper deallocation handling. These vulnerabilities, arising from accessing memory through dangling pointers, pose significant risks. While various defense mechanisms have been proposed, existing solutions often face challenges such as high performance overhead, excessive memory usage, or inadequate security guarantees, limiting their practicality. Pointer Nullification (PN) has gained attention as a promising UAF mitigation technique by tracking pointers and nullifying them upon buffer deallocation. However, existing PN techniques incur inefficiencies due to precisely associating each pointer with its target buffer, leading to expensive metadata lookups. Moreover, they overlook spatial locality in pointer storage, resulting in a larger number of registrations than necessary. This paper introduces Fast Pointer Nullification (FPN), a new PN-based defense that organizes metadata at the region level to eliminate costly search operations and uses block-based registration to efficiently capture pointer locality. Experiments on SPEC CPU benchmarks and real-world applications demonstrate that FPN provides strong security guarantees while significantly reducing performance and memory overhead compared to prior PN techniques.**

I. Introduction

Low-level programming languages like C/C++ provide the flexibility of dynamic memory allocation and deallocation at runtime. However, improper use of this feature can introduce Use-After-Free (UAF) vulnerabilities, which have become a critical issue in software security [1]. UAF vulnerabilities occur when the program accesses memory through dangling pointers (pointers referencing a previously deallocated buffer). Such vulnerabilities can lead to unpredictable program behavior, including crashes, or can be exploited for arbitrary code execution, privilege escalation, denial of service, or information leakage [2]. The prevalence of UAF vulnerabilities continues to rise, as demonstrated by the growing number of reported cases in the Common Vulnerabilities and Exposures (CVEs) database.[1] These vulnerabilities affect a broad range of

[1]https://cve.mitre.org/

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240753
www.ndss-symposium.org

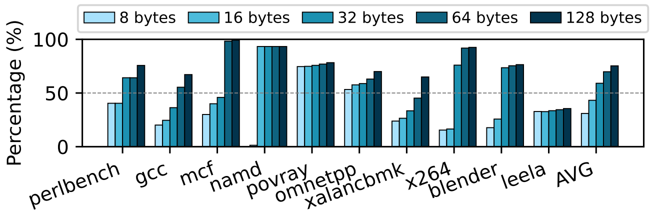


Fig. 1: Percentage of adjacent pointer storage locations registered to the same buffer on SPEC CPU 2017 under different adjacency thresholds (8–128 bytes).

widely used applications, with their impact becoming increasingly severe. For instance, in 2024, 84 UAF vulnerabilities were found in Google Chrome, 58 of which were classified as high-severity vulnerabilities capable of enabling arbitrary code execution or privilege escalation.[2] This marks a significant increase compared to 2019, when only 39 UAF vulnerabilities were reported in Google Chrome, highlighting the growing need for robust and efficient UAF mitigation or defense strategies.

Defending against UAF vulnerabilities is inherently challenging. Dangling pointers can exist anywhere in a program and propagate across functions or modules, making detection and mitigation particularly difficult. Various defense mechanisms have been proposed, each with trade-offs in performance, memory consumption, security, and compatibility. Static analysis methods [3]–[7] trace the execution flows to identify dangling references at compile time. Runtime sanitizers [8]–[20] validate pointers before memory access at runtime. Secure allocators randomize buffer addresses [21]–[23] to reduce the probability of UAF. One-Time Allocators (OTAs) prevent the reuse of freed buffer addresses [24]–[28], limiting each virtual address to be used only once. Garbage collection (GC) [29]–[32] ensures buffers are not deallocated until no pointers reference them, preventing address reuse.

Among these solutions, Pointer Nullification (PN) methods [25], [33]–[37] stand out as a promising approach that effectively prevents UAF vulnerabilities while achieving a favorable balance between security, performance, and compatibility, making them well-suited for real-world deployment. PN

[2]https://cloud.google.com/security-command-center/docs/finding-severity-classifications

methods prevent UAF vulnerabilities by nullifying dangling pointers through precise tracking of the relationship between pointers and their associated heap buffers. When the program stores a pointer, PN methods instrument the code to determine which buffer the pointer references and register the pointer's storage location in the buffer's metadata. During deallocation, PN methods traverse all registered storage locations associated with the freed buffer and invalidate (nullify) the stored pointers. Further memory accesses through dangling pointers will stop the program, effectively detecting or preventing UAF vulnerabilities.

However, tracking the exact points-to relationship between pointers and buffers in prior PN methods is inefficient, often incurring either high performance overhead or substantial memory usage. First, determining the target buffer for a given pointer and locating the corresponding metadata entry typically involves either traversing all active buffer boundaries (i.e., start and end addresses) [25] or executing computationally intensive arithmetic operations [37]. A performance breakdown of CAMP reveals that computing the address of the buffer's metadata entry alone contributes 62.50% performance overhead. This observation motivates the design of a constant-time metadata lookup mechanism that also minimizes computational complexity.

Second, PN methods register pointer storage locations independently, without leveraging the strong spatial locality often exhibited among these locations. Our empirical study on the top 10 SPEC CPU 2017 benchmarks—ranked by the number of heap pointer storage operations—reveals that pointers referencing the same buffer are frequently stored at adjacent memory addresses. As shown in Fig. 1, we define pointer storage locations as adjacent if the distance between them falls below a threshold ranging from 8 to 128 bytes. During each registration, we check whether a newly registered pointer storage location has a previously registered neighbor for the same buffer within the threshold. Results show that 30.96% of newly registered locations are adjacent to a previously registered one at the 8-byte threshold, increasing to 75.31% at 128 bytes (see Section IV-A). Despite this high locality, existing PN methods treat each storage location individually, missing opportunities to reduce memory overhead through locality-aware registration schemes. This leads to substantial memory inefficiency, especially in programs with frequent pointer storage operations. For example, CAMP imposes more than 20× memory overhead on `perlbench` and fails to complete `omnetpp` on a desktop with 32GB RAM due to memory exhaustion. These findings underscore the need for a more efficient PN design that better exploits spatial locality, while preserving strong security guarantees.

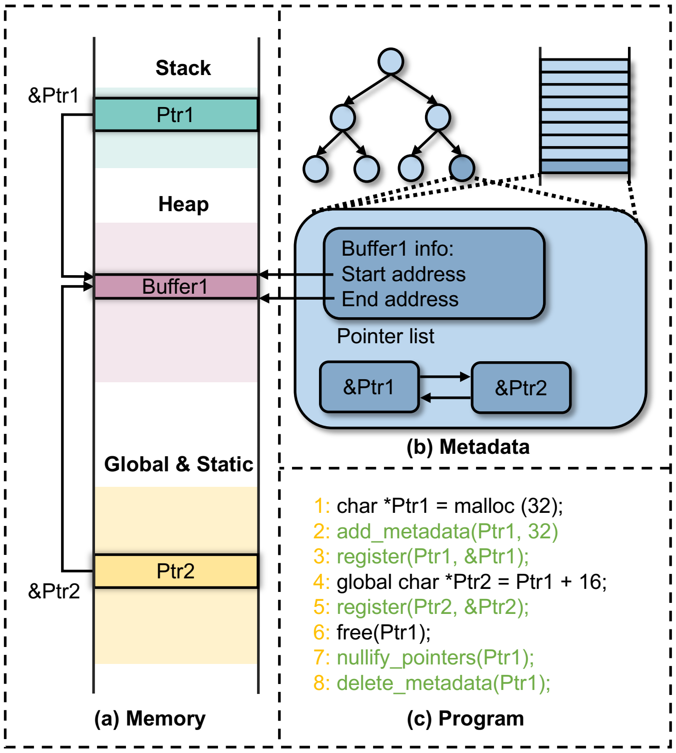
To address these inefficiencies, we propose Fast Pointer Nullification (FPN), which improves prior PN methods from two key perspectives. First, FPN enables constant-time and lightweight metadata address computation by organizing metadata at the granularity of 2^N-byte-aligned memory regions, rather than per buffer. When the program stores a heap pointer, FPN computes the address of the corresponding region's metadata using a single bit-shift and addition operation. This eliminates the need for costly tree traversals or computationally intensive computations to identify the target buffer, significantly reducing the performance overhead of registration. Second, FPN exploits spatial locality in pointer storage locations by registering 2^M-byte-aligned memory blocks instead of individual storage locations. In the remainder of this paper, we use the term *region* to refer to the address range targeted by a pointer and *block* to refer to the memory chunk that stores that pointer. FPN calculates the starting address of the block containing the pointer and looks up the region metadata to identify and eliminate duplicate registrations. By leveraging the clustering of pointer storage locations, this block-based design substantially reduces the number of registrations and associated memory overhead. Although pSweeper [36] also bypasses precise points-to tracking, it does not optimize registration numbers and therefore fails to achieve the same memory efficiency as FPN.

We then evaluate FPN's performance and memory overhead using the SPEC CPU 2017 and SPEC CPU 2006 benchmark suites. On SPEC CPU 2017, FPN incurs a geometric mean performance overhead of 17.78% and memory overhead of 8.34%; on SPEC CPU 2006, the average performance and memory overhead are 15.55% and 9.18%, respectively. Both are significantly lower than baseline PN methods. To validate FPN's applicability to real-world software, we instrument Chrome and Nginx, confirming that FPN is compatible with large-scale, multi-threaded browsers and server applications while maintaining low overhead. We further analyze FPN's internal behavior to illustrate how its block-based registration approach reduces the number of pointer storage location registrations, contributing to lower memory usage. We also present a performance breakdown across FPN's core components. Finally, we conduct a parameter sensitivity study to understand how different design configurations affect FPN's performance and memory efficiency.

Our contributions can be summarized as follows [3]:

- We analyze the inefficiencies of existing PN methods and identify that tracking the exact points-to relationship between pointers and heap buffers introduces significant performance and memory overhead.
- We propose FPN, which instead tracks relationships between blocks and regions. Through lightweight addressing and locality-aware registration, FPN significantly reduces performance overhead and memory consumption.
- We implement FPN and evaluate it across standard benchmarks and real-world applications. FPN achieves lower performance overhead and substantially lower memory overhead than prior PN methods, while preserving strong security guarantees and full compatibility with multi-threaded applications.

[3]FPN is avaliable at https://github.com/duyubo/Fast-Pointer-Nullification-NDSS-2026



```
1: char *Ptr1 = malloc (32);
2: add_metadata(Ptr1, 32)
3: register(Ptr1, &Ptr1);
4: global char *Ptr2 = Ptr1 + 16;
5: register(Ptr2, &Ptr2);
6: free(Ptr1);
7: nullify_pointers(Ptr1);
8: delete_metadata(Ptr1);
```

Fig. 2: The overview of PN.

II. Background

A. Use-After-Free (UAF)

UAF vulnerabilities occur when a program accesses memory through dangling pointers, which are pointers referencing already deallocated buffers. In C/C++, a program can dynamically allocate memory at runtime using the `malloc` function, which reserves space on the heap and returns a pointer to the start of the allocated buffer. The buffer can then be accessed through this pointer or any pointers derived from it. The lifetime of these pointers varies depending on where they are stored; for example, pointers stored in global memory may remain valid for the entire duration of the program. However, once the program frees the buffer, these pointers are no longer valid to be used, even though their storage locations may continue to be accessible. After deallocation, the memory previously occupied by the buffer may either become inaccessible or be reassigned to another buffer for a different purpose. Accessing memory through accessible dangling pointers can therefore lead to system crashes, data corruption, or even the execution of malicious code. This mismatch between pointer lifetime and buffer lifetime is the fundamental cause of UAF vulnerabilities. Unfortunately, C/C++ lacks built-in mechanisms to prevent the misuse of dangling pointers, making UAF vulnerabilities both common and dangerous.

B. Pointer Nullification (PN)

The key idea of PN is to track the associated pointers for each buffer and nullify them when the program frees the buffer. This approach ensures that any memory access using a dangling pointer will reference an invalid address, thereby stopping the program and preventing further issues. PN consists of three parts: (1) setting up metadata when the program allocates a new buffer; (2) searching the associated buffer for a pointer and registering this pointer's storage location for the associated buffer, when the program stores a heap pointer (a pointer points to heap); (3) nullifying all associated pointers of the buffer when the program frees it.

Fig. 2 provides an overview of how PN works. In part (1), PN maintains metadata for each heap buffer, stored either as nodes in a red-black tree [25] or entries in a hash table [33]–[35], [37]. This metadata includes the buffer's boundaries (i.e., start and end addresses) and a dynamically maintained pointer list that records the storage locations of all pointers referencing the buffer. In part (2), when the program stores a pointer, PN identifies the buffer it references by checking which buffer's boundaries cover the pointer value. Once identified, PN appends the pointer's storage location to this buffer's pointer list. In part (3), upon deallocation, PN retrieves the metadata of the freed buffer, iterates through the pointer list, and nullifies the pointers that still refer to this buffer by modifying them to invalid values (e.g., changing the unused upper 16 bits from `0x0000` to `0xffff`). After completing this nullification step, PN deletes the metadata entry associated with the buffer. Fig. 2(c) presents a concrete example. Lines instrumented by PN methods are highlighted in green. At line 1, the program allocates a buffer labeled as `Buffer1`. The malloc function returns the pointer `Ptr1`, which is saved on the stack at `&Ptr1`. Then PN sets up metadata for `Buffer1` at line 2, recording its starting and end addresses. At line 3, PN registers the location `&Ptr1` to `Buffer1`'s pointer list. At line 4, the program adds offset 16 to `Ptr1` and saves this derived pointer to `&Ptr2`, which is on global memory. In response to line 4, at line 5, PN registers `&Ptr2` to the pointer list of `Buffer1`. At line 6, the program frees `Buffer1`. Then, at line 7, PN goes through the pointer list and nullifies the pointers saved at `&Ptr1` and `&Ptr2` to invalid values. In the end, PN deletes the metadata for `Buffer1` at line 8.

III. Threat Model

FPN aims to prevent UAF vulnerabilities by eliminating memory access through dangling pointers, which can reside in the stack, heap, or global/static memory. It follows the same threat model as previous PN methods [25], [33], [34], [37]. FPN operates in user space and assumes access to C/C++ source code for instrumentation at compile time. FPN does not track pointers residing in CPU registers due to their transient nature and limited exposure surface. The model assumes that the program contains only UAF vulnerabilities, with other classes of memory safety bugs (e.g., buffer overflows, type confusion) mitigated through orthogonal mechanisms such as [38]. The adversary is assumed capable of manipulating program inputs and controlling allocation/deallocation patterns to induce UAF vulnerabilities. Their goal may include arbitrary code execution or privilege escalation via exploitation of UAF conditions. We assume a trusted compiler toolchain and runtime environment, and that the adversary cannot access FPN's internal metadata, which can be achieved by mechanisms such as address space layout randomization (ASLR) [39] (used in our implementation) or hardware-based isolation (e.g., ERIM [40]).

IV. Fast Pointer Nullification (FPN)

A. Motivation

Tracking the precise points-to relationship between pointers and buffers introduces significant inefficiencies. Previous PN methods either require time-consuming search operations or do not exploit the spatial locality of pointer storage locations. As a result, they suffer from either high performance overhead or excessive memory usage. First, identifying the target buffer for a pointer and locating its corresponding pointer list typically requires searching across all active buffer metadata, resulting in high performance overhead. For example, DangNULL [25] uses a red-black tree to organize metadata, incurring logarithmic lookup time proportional to the number of active buffers. Because these lookups are performed frequently, they contribute significantly to runtime overhead. CAMP [37] addresses this issue by leveraging a *segregated list allocator*[4], which enables constant-time metadata lookups. However, each lookup still involves computationally expensive arithmetic operations. As shown in Fig. 3(a), when storing a pointer, CAMP first identifies the page to which this pointer points (①). It then conducts a table lookup to retrieve metadata for the corresponding span, which represents a contiguous region allocated for buffers of the same size (②). Later, it calculates the buffer ID using addition and division operations (③). Finally, it uses this ID to access the associated pointer list's address through a table lookup (④). Although this design avoids tree-based lookups, it still incurs measurable performance overhead. Our performance breakdown of CAMP shows that getting the pointer list address alone accounts for 62.50% performance overhead. This observation motivates the need to find a method that enables constant-time lookups while only involving lightweight operations.

Second, previous PN approaches register pointer storage locations without leveraging spatial locality, resulting in high memory overhead. We observe that the pointers to the same buffer are usually stored at adjacent locations. Our quantitative analysis on SPEC CPU 2017 further confirms this observation, as shown in Fig. 1. We define two pointer storage locations as adjacent if their address difference is within a given threshold, varying from 8 to 128 bytes. Before registering a new location, we check whether an adjacent one has already been registered for the same buffer under this definition. At an 8-byte threshold, 30.96% of new registrations are adjacent to existing ones. This rate increases to 43.21%, 59.09%, 69.82%, and 75.31% for thresholds of 16, 32, 64, and 128 bytes, respectively. Despite this clear locality, all previous PN methods register pointer storage locations independently, without

[4]https://google.github.io/tcmalloc/

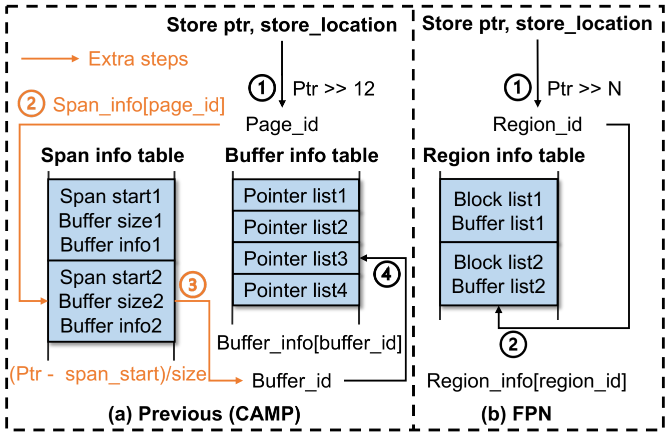


Fig. 3: Comparison of search operations between CAMP and FPN.

considering this property. Techniques like static analysis [33], shadow memory [25], [33], [35], and caching [37] attempt to reduce redundancy but only eliminate exactly the same addresses, not nearby addresses. On the benchmarks with frequent buffer allocations and heap pointer storage operations, such as `omnetpp`, prior methods incur more than 10x memory overhead or even crash systems with 32 GB RAM. An alternative approach by Liu et al. [36] also avoids tracking exact pointer-to-buffer relationships by associating pointers with all heap buffers collectively. While this design offloads nullification to a separate thread to reduce runtime overhead, it does not reduce the number of registrations. This method still incurs high memory overhead—112.50% on average on SPEC CPU 2006, as reported in their paper. This observation motivates us to explore a method that can fully take advantage of the locality in pointer storage locations.

B. Preliminary

The motivation behind FPN is to address two key inefficiencies in prior PN approaches during pointer registration: high performance overhead caused by time-consuming lookup operations, and high memory overhead caused by individually registered pointer storage locations. FPN introduces a constant-time addressing mechanism composed of only one bitwise operation and one table lookup, eliminating the need for complex arithmetic or tree traversal. Concurrently, FPN exploits spatial locality among pointer storage locations by grouping adjacent locations into blocks to reduce memory overhead. FPN's core idea is to track the points-to relationship from 2^M-byte-aligned blocks (each of size 2^M bytes) to 2^N-byte-aligned regions (each of size 2^N bytes), as illustrated in Fig. 4. FPN maintains metadata per region using a shadow memory, rather than maintaining metadata per buffer or address. Each region is aligned to a 2^N-byte starting address, and its corresponding metadata entry is stored at a fixed offset in the shadow memory. When the program stores a heap pointer, FPN relies solely on one bit shift and one table lookup to get the address of the corresponding pointer list, keeping the

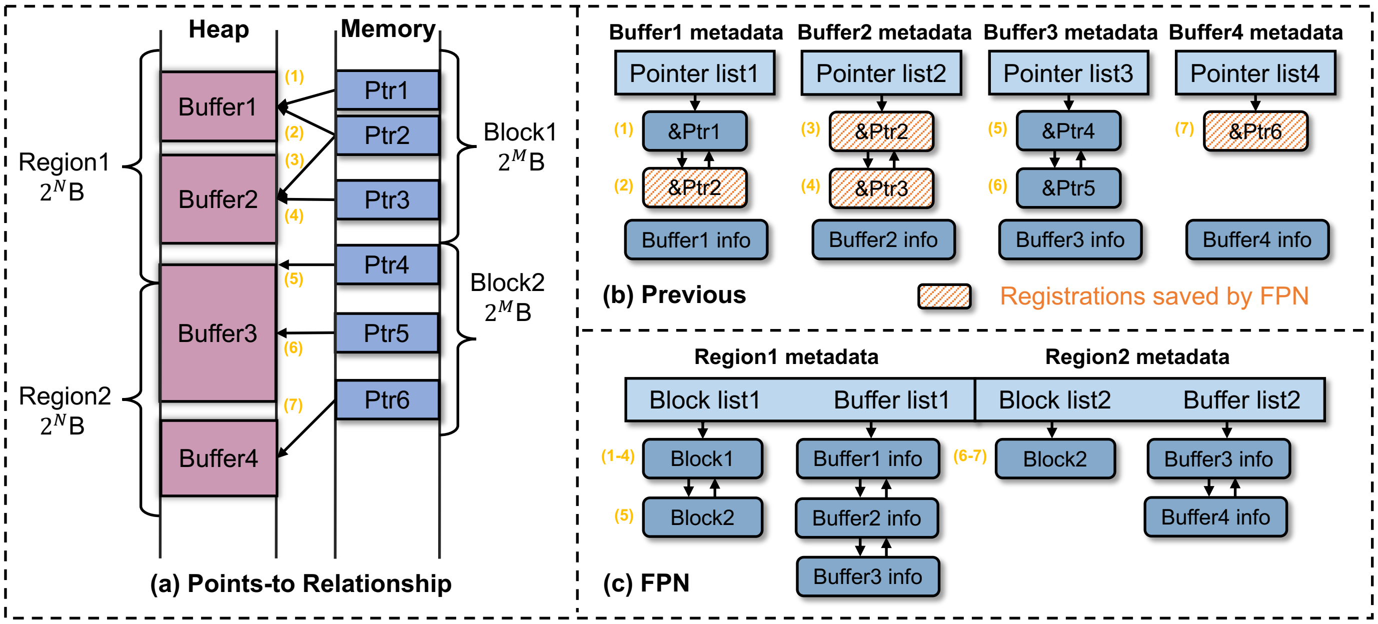


Fig. 4: Diagram of FPN and comparisons of pointer registrations between previous PN methods and FPN. To illustrate the points-to relationships more clearly, (a) presents two conceptual views of memory (overlapping with each other): a zoomed-in view highlighting the heap region that heap pointers reference, and a full-memory view showing where these pointers may be stored.

performance overhead low. FPN then registers the base address of the block that contains the pointer, rather than a precise pointer location. Because pointer storage locations exhibit spatial locality—where nearby locations tend to reference the same region—FPN checks whether the block has already been registered before performing a new registration. This design enables a single block to represent multiple adjacent pointer locations, thereby significantly reducing the memory overhead. Although scanning entire blocks during nullification may introduce additional performance overhead, deallocations occur infrequently in most applications, making the overall performance impact negligible.

C. FPN Design Details

FPN also consists of three core components, similar to other PN methods (introduced in Section II-B). To address potential false positives introduced by block-based registration, FPN incorporates a status bit mechanism to precisely track locations storing heap pointers. The roles of each component are detailed below.

1) Setting Up the Metadata: Upon buffer allocation, FPN initializes metadata entries for all regions overlapped by the buffer. FPN maintains metadata for each region using a shadow memory table starting from `region_info`. Each entry stores two pointers: one to a *buffer list* containing metadata (start and end addresses) for overlapping buffers, and one to a *block list* for registered block addresses. When the program allocates a buffer, FPN first calculates the region range covered by the buffer based on its start and end addresses: `start_region_id = start_address >> N`, `end_region_id = end_address >> N`. FPN then checks whether the metadata entry for each region in this range (`region_info[start_region_id]` to `region_info[end_region_id]`) has already been initialized. If so, FPN appends the buffer information to the region's buffer list. Otherwise, FPN initializes both lists and stores their pointers in the corresponding metadata entry before appending the buffer information.

This region-based metadata design has two key properties: multiple buffers within the same region share a single metadata entry, and each region overlapped by one buffer records metadata for that buffer. As illustrated in Fig. 4, the program allocates four buffers sequentially: `Buffer1`, to `Buffer4`. Since `Buffer1` is the first buffer allocated into `Region1`, FPN initializes both the buffer list and block list, then appends `Buffer1`'s information to `Region1`'s buffer list. When allocating `Buffer2`, FPN detects that `Region1`'s metadata is already initialized and appends `Buffer2`'s metadata to the same buffer list. When allocating `Buffer3`, which spans both `Region1` and `Region2`, FPN creates metadata only for `Region2` and updates both regions' buffer lists with `Buffer3`'s information. When allocating `Buffer4`, FPN only updates `Region2`' buffer list with `Buffer4`'s information.

2) Registering the Block Address: When the program stores a heap pointer (`store ptr, store_location`), FPN registers the starting address of the block containing that pointer to the corresponding block list. As shown in Fig. 3(b), first, FPN determines the region that the pointer references using a bit-shift operation:

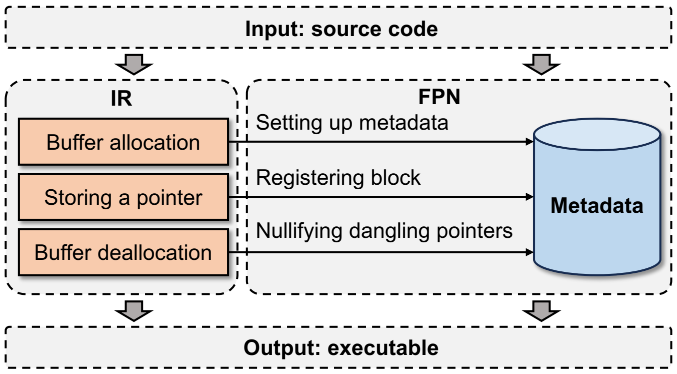


Fig. 5: Workflow of FPN.

`region_id = ptr >> N` (①). Then it gets the address of the corresponding block list: `block_list_addr = region_info[region_id]` (②). Compared with the search operations in CAMP in Fig. 3(a), which requires four steps in total, including multiple table lookups and division operations, FPN only needs one bit shift operation and one table lookup. Next, FPN calculates the starting address of the block where the pointer is stored: `block_address = store_location & ~(M - 1)`. FPN then appends this block address to the end of the region's block list. To reduce redundant registrations, FPN checks whether the new block address is already among the most recent C registered blocks. If it is found, FPN skips the registration. Here, C is a configurable constant, meaning the number of check steps, which can be decided at compile time.

As illustrated in Fig. 4(a) and (c), the program stores pointers `Ptr1` to `Ptr6` sequentially. The order of store operations is labeled (1) through (7). Among these, the program stores `Ptr2` twice to the same location: first pointing to `Buffer1`, then updated to point to `Buffer2`.

(1) `Ptr1`: Points to `Region1` and resides in `Block1`. FPN registers `Block1` to `Region1`'s block list.
(2) First store of `Ptr2`: Also in `Block1` and references `Region1`. FPN detects a duplicate and skips registration.
(3) Second store of `Ptr2`: Although the pointer's target changes, it still resides in `Block1`. FPN again skips registration.
(4) `Ptr3`: Same block and region as previous pointers. No registration needed.
(5) `Ptr4`: Stored in `Block2`, points to `Region1`. FPN registers `Block2` to `Region1`'s block list.
(6) `Ptr5`: Points to `Region2` and also resides in `Block2`. FPN registers `Block2` to `Region2`'s block list.
(7) `Ptr6`: Same block and region as previous pointers. No registration needed.

This example illustrates how FPN leverages spatial locality to minimize the number of registrations. For example, FPN captures pointers such as `Ptr1` to `Ptr3`, which reside within `Block1`, pointing to different buffers in `Region1`, by registering only one block address, labeled as (1-4) in Fig. 4(c). In contrast, as shown in Fig. 4(b), prior PN methods register each pointer individually, resulting in four separate registrations, labeled as (1) to (4). Similarly, for pointers `Ptr5` and `Ptr6`, FPN handles them together through `Block2`, labeled as (6-7) in Fig. 4(c). However, prior PN methods incur two separate registrations, labeled as (6) and (7) in Fig. 4(b). In total, FPN incurs four fewer registrations.

3) Nullifying the Pointers: Upon buffer deallocation, FPN nullifies any pointers referencing the freed buffer. FPN first computes the region number and locates the corresponding metadata entry using the buffer's address, following the same approach as in Section IV-C2. It then traverses the buffer information list to identify the entry whose starting address matches the buffer being freed. Using these boundaries, FPN scans each region that overlaps with the freed buffer. It scans each registered block for the region and treats each pointer-sized word as a candidate for nullification. If a value falls within the memory range of the freed buffer, FPN nullifies it, as described in Section II-B. If the buffer is the only one associated with the region, FPN removes a scanned block from the block list after completing the nullification. If multiple buffers share the region, FPN checks whether the block may contain pointers to other live buffers. If so, it keeps the block in the list. Although scanning entire blocks is more expensive than targeting individual addresses, deallocations are infrequent, keeping the overall performance overhead negligible (Section VI-E2).

4) Ensuring Compatibility via Status Bit: FPN faces two compatibility challenges related to pointer nullification during buffer deallocation. The first issue arises when FPN scans a block located on a freed buffer. In such cases, the operating system may have already reclaimed the corresponding page and marked it as inaccessible. Attempting to access such a block causes the program to crash unexpectedly. The second issue involves false positives: nullifying non-heap data that happens to match a freed buffer's address range. During scanning, FPN conservatively treats every pointer-sized value as a potential heap pointer. If a non-pointer value happens to fall within the address range of the recently freed buffer, FPN may mistakenly overwrite it with an invalid value. Although this nullification does not cause immediate failure, subsequent use of the corrupted data can lead to unpredictable program behavior. FPN addresses both issues by introducing a second shadow memory, `status_table`, which tracks the allocation and pointer status of each pointer-sized word. FPN assigns one status bit per eight-byte word. FPN computes the corresponding status bit for a memory address via a bit shifting and table lookup: `status_table[address >> 3]`. When the program stores a heap pointer to a location, FPN sets the associated bit to one; when non-heap-pointer data is stored, FPN clears the bit to zero. Upon buffer deallocation, FPN clears the status bits for all addresses covered by the freed buffer. During scanning, FPN only examines locations with status bits set to one, thereby avoiding unnecessary or unsafe checks.

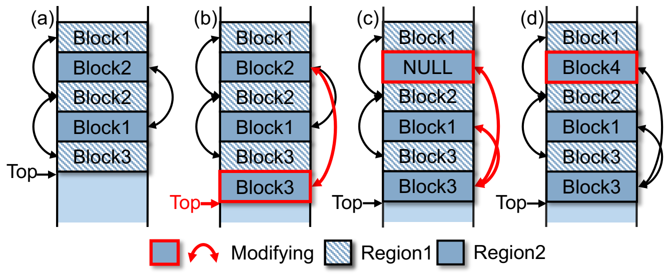


Fig. 6: Maintenance of lists in FPN.

V. Implementation

A. Implementation Pipeline

We implement FPN on top of LLVM 18.1.0, adding approximately 2K lines of C++ code. The overall workflow is illustrated in Fig. 5. LLVM first compiles the C/C++ source code into intermediate representation (IR), and FPN instruments the program at the IR level as follows: During buffer allocation (e.g., `malloc()`, `new()`), FPN initializes metadata entries for all regions the buffer overlaps (Section IV-C1). During pointer storage (e.g., `store i32* %a_ptr, i32** %ptr_to_a_ptr, align 8`), FPN calculates the block address of the pointer's storage location and registers it in the corresponding region's metadata (Section IV-C2). Upon buffer deallocation (e.g., `free()`), FPN scans the registered blocks and nullifies all pointers referencing the freed buffer (Section IV-C3). FPN is agnostic to the underlying allocator and compatible with all standard heap allocators.

B. Maintenance of Lists

FPN manages buffer information lists and block lists using doubly linked lists to support efficient bidirectional traversal and dynamic insertion or deletion. Since both list types follow the same structure, Fig. 6 uses block lists as an example. FPN reserves a contiguous memory region with the `mmap()` system call and stores list elements from low to high addresses. To efficiently manage these elements, FPN maintains a pointer called `Top`, which marks the highest used address in the reserved region where the content of the next inserted element will be stored. When inserting a new element, FPN places it at the current `Top` and increments `Top` afterward. Each element's position in memory depends on insertion order, not the specific list it belongs to. Consequently, elements from different lists may be adjacent in memory, while those from the same list may be separated. Fig. 6(a) shows the initial state, where FPN has already registered `Block1` to `Block3` for `Region1`, and `Block1` and `Block2` for `Region2`. Later, as shown in Fig. 6(b), FPN registers `Block3` for `Region2` at the current `Top`, then updates `Top`. This new element is adjacent to `Block3` of `Region1`, rather than `Block2` of `Region2`. When deregistering a block, FPN clears the block address stored in the corresponding element but does not remove the element from the list. Instead, it updates the list links and moves the invalidated element to the end of its list, as shown in Fig. 6(c). When registering a new block, FPN first checks for available invalidated elements. If one exists, FPN reuses it to store the new block address. Only when no reusable element is found does FPN insert a new element at the `Top` address and increment `Top`. This reuse strategy minimizes the need to increase `Top` or update list connections, which is beneficial because registrations occur more frequently than deregistrations. As shown in Fig. 6(d), when FPN registers `Block4` for `Region2`, it reuses the invalidated element previously stored in `Block2` instead of inserting a new element at `Top`. To further avoid the performance overhead of increasing `Top` and adding a new element to a list, FPN does not request a single element when the associated list is full. Instead, FPN requests E elements at a time, setting up the connections of these elements all at once. Here, E is also a constant number that can be decided at compile time.

CWE	Type	Total	Passed	Rate
Double Free (DF)	Good	820	820	100%
	Bad	820	820	100%
Use After Free (UAF)	Good	394	394	100%
	Bad	394	394	100%
Invalid Free (IF)	Good	288	288	100%
	Bad	288	288	100%

TABLE I: Security evaluation of FPN on Juliet Test Set.

C. Thread Locks

To support real-world multithreaded applications, FPN introduces a lightweight locking mechanism that ensures thread-safe metadata maintenance. FPN categorizes locks into two types: global-level locks and region-level locks. FPN uses global-level locks to protect shared metadata structures that are accessed or modified across all regions. Specifically, it assigns a dedicated global-level lock to guard the highest used address in the memory reserved for block list elements and another for the highest used address in the memory reserved for buffer list elements. These addresses are frequently updated as new elements are inserted, making synchronization necessary to prevent race conditions. FPN also assigns a global-level lock to the table that maintains status bits. When a thread attempts to access or update any of these data structures, it must acquire the corresponding global-level locks to ensure exclusive access. This guarantees consistency while minimizing the lock scope to avoid unnecessary contention. FPN also assigns fine-grained region-level locks. FPN assigns one lock per region, which guards all metadata associated with that region—including the region's buffer list and block list. When a thread needs to modify or read the metadata of a specific region, it must first acquire that region's lock to ensure exclusive access. This design ensures that only one thread can update or read metadata within a region at a time, preventing race conditions, while still allowing other threads to operate on different regions in parallel. This lock structure balances safety and scalability, enabling FPN to remain efficient in multi-threaded environments without introducing bottlenecks during frequent metadata updates.

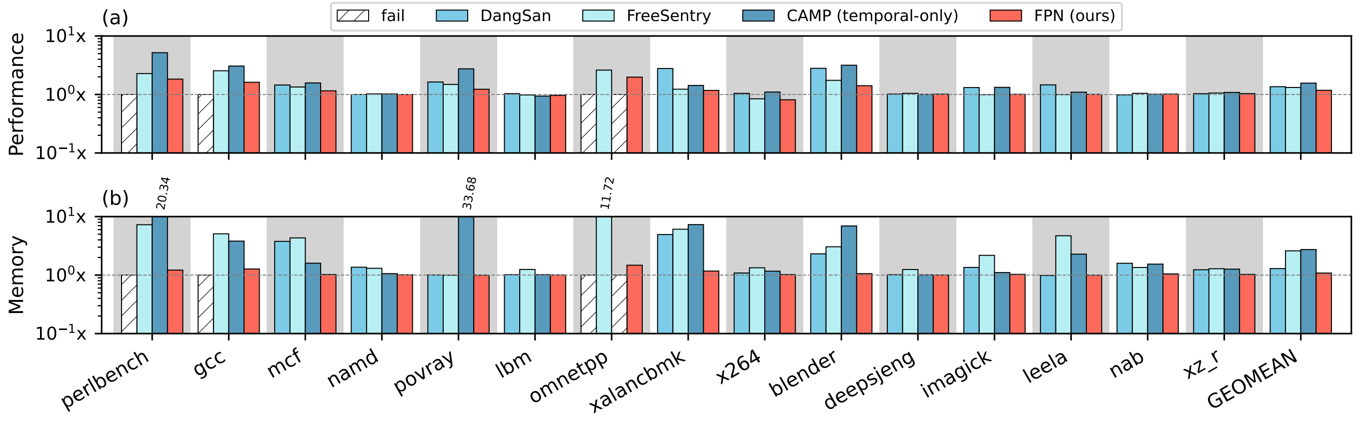


Fig. 7: Evaluation of normalized performance and memory of FPN and previous PN methods on SPEC CPU 2017, under optimization level -O2. Baseline is 1x.

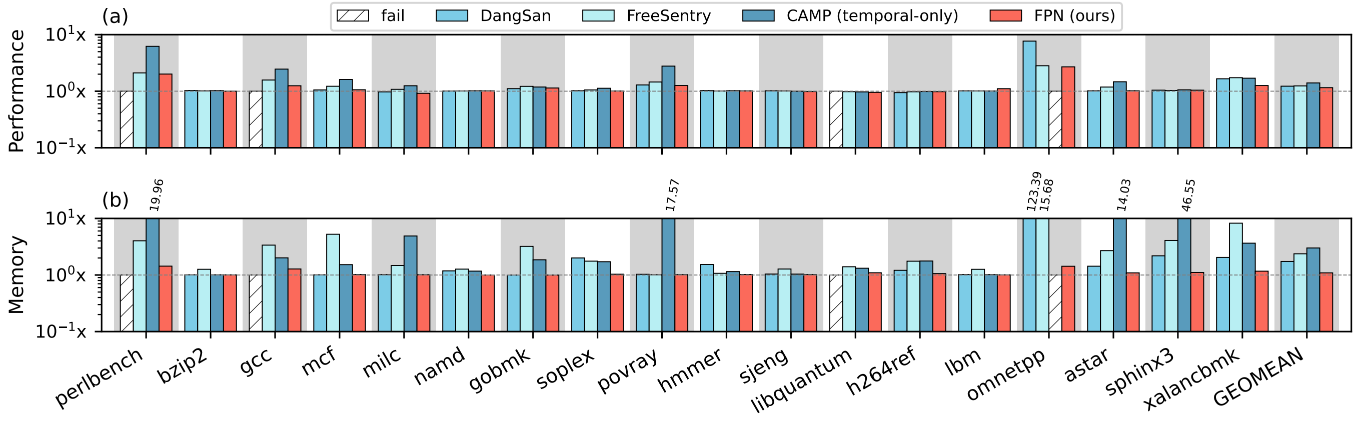


Fig. 8: Evaluation of normalized performance and memory of FPN and previous PN methods on SPEC CPU 2006, under optimization level -O2. Baseline is 1x.

CVE	Freesentry	Dangsan	CAMP	FPN (ours)
CVE-2015-3205	✖	✔	✔	✔
CVE-2015-2787	✔	✔	✔	✔
CVE-2015-6835	✖	✖	✔	✔
CVE-2016-5773	✔	✖	✔	✔
Issue-3515	✔	crash	✔	✔
CVE-2020-6838	✔	crash	✔	✔
CVE-2021-44964	✔	✔	✔	✔
CVE-2020-21688	✔	✔	✔	✔
CVE-2021-33468	✔	✔	✔	✔
CVE-2020-24978	✔	✔	✔	✔
Issue-1325664	✖	✖	✔	✔
CVE-2022-43286	✔	✔	✔	✔
CVE-2019-16165	✔	✔	✔	✔
CVE-2021-4187	✔	✔	✔	✔

TABLE II: Security evaluation of FPN and related work on real-world applications with UAF bugs. ✔ means successfully preventing UAF vulnerabilities. ✖ means failure in preventing UAF vulnerabilities.

D. Detection of Invalid Free

In addition to detecting use-after-free (UAF) vulnerabilities, FPN also identifies Double Free (DF) and Invalid Free (IF) errors during buffer deallocation. When a program calls `free()`, FPN scans the regions that overlap with the given address and traverses the registered blocks to locate the corresponding buffer boundary information (as described in Section IV-C3). If FPN fails to find any matching buffer information, it reports a DF error, indicating the buffer was already deallocated. If FPN finds the buffer information but sees that the freed address differs from the buffer's starting address, it reports an IF error.

VI. Evaluation

A. Methodology

We conduct all experiments on a 16-core 12th Gen Intel(R) Core(TM) i9-12900K CPU with 32GB RAM, operating on Ubuntu 24.04.3 LTS. We compile the programs with optimization level `-O2`. Our evaluation consists of four main parts. In

Benchmark	Metric	FFmalloc	DangZero	FreeGuard	MarkUS	BUDAlloc	SwiftSweeper	xTag	RSan	FPN (ours)
SPEC CPU 2017	Performance	1.04x	1.26x	1.02x	1.24x	1.11x	1.05x	1.04x	1.61x	1.17x
	Memory	1.74x	1.24x	2.11x	1.36x	1.25x	1.34x	2.39x	2.30x	1.08x
SPEC CPU 2006	Performance	1.04x	1.30x	1.05x	1.19x	1.14x	1.07x	1.13x	1.46x	1.16x
	Memory	1.60x	1.22x	2.10x	1.27x	1.33x	1.22x	2.42x	3.18x	1.09x

TABLE III: Comparison of geometric mean normalized performance and memory between FPN and non-PN UAF mitigation methods on SPEC CPU 2017 and SPEC CPU 2006. The detailed results are included in Fig. 15 and Fig. 16

Section VI-B, we first assess FPN's effectiveness in detecting and preventing UAF bugs using a standard vulnerability benchmark and a collection of real-world vulnerabilities. In Section VI-C, we measure FPN's performance and memory overhead on two widely used SPEC CPU benchmark suites. In Section VI-D, we evaluate FPN's compatibility with large-scale, multithreaded web applications. In Section VI-E, we present a hit-rate distribution analysis under varying check step values and a breakdown of FPN's core components to explain its performance and memory efficiency compared to other PN methods. Finally, in Section VI-F, we conduct a parameter sensitivity analysis to study the effects of varying region sizes, block sizes, and the number of check steps on FPN's overhead. For all the experiments, we leave the libraries for which source code is not avaliable uninstrumented. For example, on SPEC CPU 2017 and SPEC CPU 2006, we leave the glibc libraries uninstrumented. And we apply the same criteria to other PN-based methods.

We compare FPN against the following PN-based techniques: DangSan [34], a shadow-memory-based PN approach optimized for scalability; FreeSentry [33], an early PN method based on fine-grained shadow memory; and CAMP [37], the latest PN-based system that employs a *segregated list allocator* to accelerate metadata lookup. FreeSentry and DangSan rely on LLVM versions earlier than 4.0, which are incompatible with Ubuntu 24.04.3 LTS. To ensure compatibility, we update their LLVM passes on LLVM 18.1.0. We also compare FPN with representative non-PN UAF mitigation techniques, including: OTA-based allocators—FFmalloc [41] and BUDAlloc [27]; GC-based approaches—MarkUS [29], DangZero [26], and SwiftSweeper [31]; the secure allocator FreeGuard [22]; and UAF sanitizers xTag [42] and RSan [43]. For the evaluation of performance and memory overhead, we normalize each method to its corresponding baseline; more details are included in TABLE V.

B. Security Evaluation

1) Juliet Test Set: To evaluate the effectiveness of FPN in preventing UAF bugs, we test FPN on Juliet Test Set[5], a comprehensive collection of C/C++ test cases. We select three subsets: UAF (CWE-416), Double Free (DF) (CWE-415), and Invalid Free (IF) (CWE-761), as summarized in Table I. Good test cases are programs without any UAF, DF, or IF errors. They are used to confirm that programs instrumented with FPN do not mistakenly crash the program when no UAF-related vulnerabilities occur. The bad tests use proof-of-concept (POC) to trigger the corresponding UAF-related vulnerabilities, enabling us to verify whether FPN can effectively stop the program when a UAF-related vulnerability happens. Since FPN prevents UAF by stopping the program when a dangling pointer is referenced. To ensure that the program stops specifically due to accessing a dangling pointer and not for other reasons, we utilize a gdb script to debug the accessed address at the point of the program crash, following the methodology outlined in [37]. The script checks if the accessed address has upper bits set to `0x8000`. If this condition is met, we conclude that FPN successfully prevents the UAF. FPN successfully passes all bad and good test cases, achieving zero false negatives and false positives, demonstrating both its precision and reliability.

2) Real-world CVEs: To further assess FPN's effectiveness against real-world UAF vulnerabilities, we evaluate it using the same test set as CAMP [37]. As shown in Table II, the evaluation includes 14 vulnerabilities from 11 widely used applications—spanning language interpreters, libraries, browsers, web servers, and UNIX utilities. FreeSentry fails to detect three vulnerabilities. DangSan misses five, with unexpected crashes on two of them. In contrast, both FPN and CAMP successfully compile, execute, and prevent all tested UAF vulnerabilities. FPN matches CAMP's security guarantees and outperforms FreeSentry and DangSan in coverage, demonstrating its robustness as a PN method. Compared with non-PN UAF mitigation methods summarized in Table VI, FPN provides stronger UAF protection coverage. For example, FreeGuard, a performance-oriented secure allocator, relies on probabilistic defenses and therefore offers limited protection against targeted UAF exploits, whereas FPN deterministically prevents UAF dereferences through pointer nullification. RSan primarily targets spatial memory safety violations and provides limited coverage for temporal errors; in contrast, FPN achieves a substantially higher UAF prevention rate by explicitly eliminating dangling pointers.

C. Performance and Memory Consumption Evaluation

In this section, we evaluate and compare FPN with previous PN methods on the SPEC CPU 2017 and SPEC CPU 2006 benchmark suites. We measure performance using built-in timers provided by the benchmarks themselves. We measure memory consumption using a shell script to track the Resident Set Size (RSS) occupied by the program during execution, recording the maximum value as the memory consumption for the program. Each measurement is repeated 10 times, with the average value taken as the final result. We setup N with the value of 18, M with the value of 30, and C with the value

[5]https://samate.nist.gov/SARD/test-suites/112

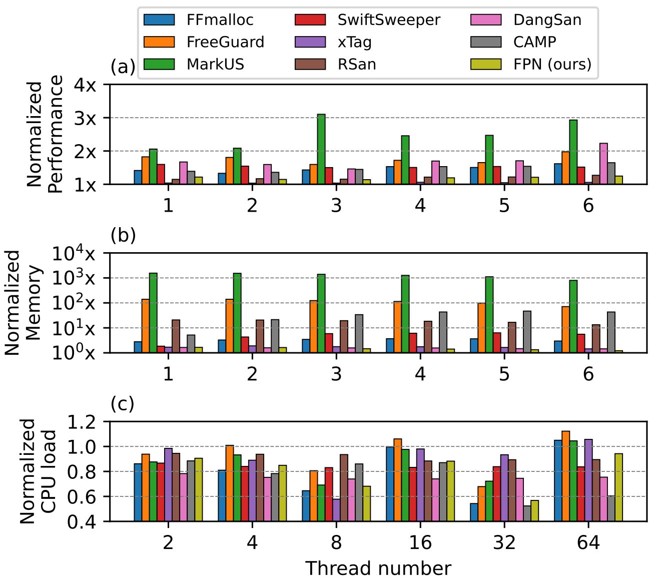


Fig. 9: Performance, memory, and CPU utilization comparison between FPN and other UAF mitigation methods on Nginx under different thread numbers. Baseline is 1x.

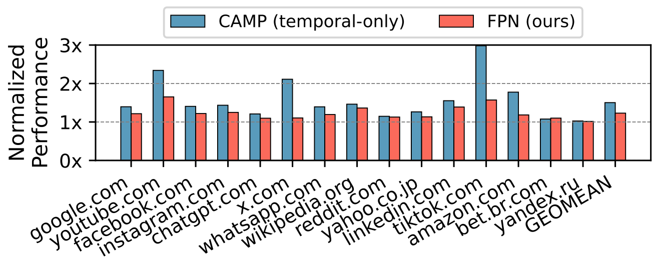


Fig. 10: Normalized performance comparison between CAMP and FPN on Chrome. Baseline is 1x.

3. The measurement of FPN incurs less than 1.00% standard error of performance and less than 3.00% of memory. The details of each measurement's standard error are included in TABLE VII.

SPEC CPU 2017 As shown in Fig. 7, FPN incurs geometric mean performance overhead of 17.78% and a memory overhead of 8.34% across all SPEC CPU 2017 benchmarks, with maximum overheads of 98.40% (performance) and 47.61% (memory). DangSan has 35.87% mean performance and 29.25% memory overhead but fails to execute several benchmarks with intensive buffer operations, including `perlbench`, `gcc`, and `omnetpp`. On the benchmarks where DangSan completes execution, FPN shows lower overhead—5.87% for performance and 3.17% for memory. FreeSentry reports 31.76% performance and 159.78% memory geometric mean overhead. CAMP shows the highest overhead among the evaluated methods, with a geometric mean overhead of 56.37% for performance and 173.44% for memory. CAMP fails to complete `omnetpp` due to memory exhaustion. Across the benchmarks where CAMP completes execution, FPN reports only 13.46% performance and 5.97% memory overhead.

SPEC CPU 2006 On SPEC CPU 2006, as shown in Fig. 8, FPN incurs a geometric mean performance overhead of 15.56% and memory overhead of 9.18%, with peak overheads of 169.71% (performance) and 43.4% (memory). DangSan reports the geometric mean overhead of 22.56% for performance and 73.47% for memory, and fails on `perlbench`, `gcc`, and `libquantum`. On the benchmarks where DangSan runs successfully, FPN achieves significantly lower overhead—12.22% for performance and 6.06% for memory on average. FreeSentry introduces 24.21% performance and 137.34% memory overhead, both higher than FPN's. CAMP imposes 40.15% performance and 100.58% memory overhead and crashes on `omnetpp`. On the benchmarks where CAMP completes execution, FPN maintains lower average overhead—9.93% for performance and 7.48% for memory.

Compared to FreeSentry, DangSan, and CAMP—all of which employ fine-grained shadow memory—FPN organizes metadata at the region level. This design increases the likelihood that a newly registered block address matches an existing entry, reducing the number of redundant registrations and thus lowering memory overhead. Moreover, FPN typically finds an existing registered block within three check steps (see Section VI-E1), ensuring that deduplication introduces minimal performance overhead. As a result, FPN achieves both lower memory and performance overhead than prior PN methods.

We additionally compare FPN with representative non-PN UAF mitigation techniques (Table III). FPN attains the lowest memory overhead among all evaluated systems, demonstrating the efficiency of its metadata design. In terms of performance, FPN outperforms DangZero, MarkUS, and RSan. xTag reports low overhead on SPEC CPU 2017; however, this is largely because it fails to run on several long-running benchmarks (e.g., `perlbench`), which artificially lowers its average. On SPEC CPU 2006—where xTag successfully completes most workloads—its performance overhead is comparable to that of FPN. FreeGuard achieves lower performance overhead but offers weaker security guarantees, as discussed in Section VI-B2. FFmalloc also exhibits relatively low performance overhead; however, it introduces substantial memory overhead and is susceptible to exhausting the virtual address space in long-running applications, limiting its practical deployment. Compared with BUDAlloc and SwiftSweeper, FPN incurs slightly higher performance overhead but delivers significantly lower memory overhead.

D. Real-world Multi-threaded Applications

Nginx We evaluate FPN's performance, memory overhead, and CPU utilization on Nginx v1.22.1 using the wrk v4.2.0 benchmarking tool. Nginx is tested under varying workloads with thread counts 2, 4, 8, 16, 32, 64 and 1000 concurrent connections. Each experiment is repeated 10 times, with a 10-second pause between runs to reduce measurement noise. All

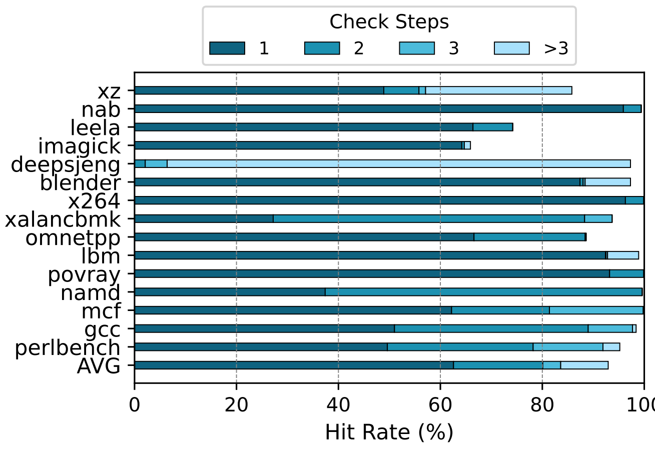


Fig. 11: Percentage of deduplicated blocks at different check steps on SPEC CPU 2017.

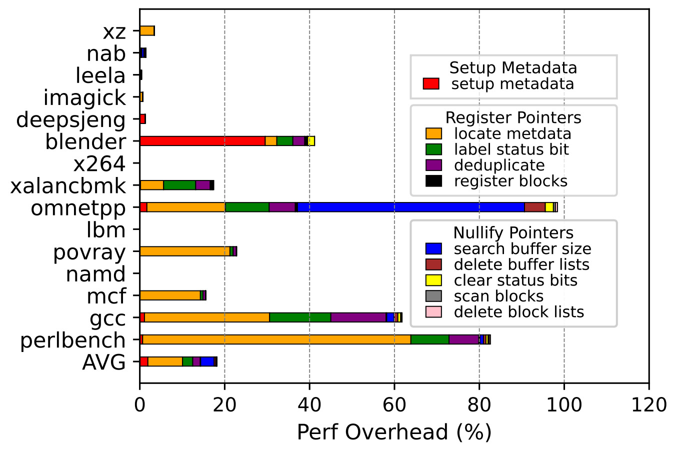


Fig. 12: Performance breakdown analysis of FPN on SPEC CPU 2017.

results are normalized to their respective baselines. As shown in Fig. 9, FPN consistently achieves lower performance and memory overhead than DangSan and CAMP across all configurations. Compared with non-PN UAF mitigations, FPN also incurs substantially lower memory overhead. Regarding CPU utilization, FPN incurs lower CPU utilization compared with its baseline due to the serialization introduced by metadata locks. However, it still achieves higher utilization compared with DangSan and CAMP. These results demonstrate that FPN's hybrid locking mechanism—combining coarse-grained global locks with fine-grained region locks—supports correct and efficient execution in large-scale, multithreaded applications. This design minimizes lock contention and avoids severe serialization bottlenecks during pointer registration and nullification, enabling FPN to maintain consistently lower performance and memory overhead compared with CAMP and DangSan.

Chrome We further evaluate FPN and CAMP on Chrome by measuring the performance overhead of loading the 15 most frequently visited websites.[6] We repeat each measurement 10 times and calculate the performance overhead with the default timer in the Linux system. As shown in Fig. 10, FPN incurs an average overhead of 18.32%, while CAMP incurs 36.97%. FPN's highest overhead is 65.24% on *youtube.com*, whereas CAMP reaches 198.23% on *tiktok.com*.

E. FPN Internal Behavior Analysis

1) Deduplication Percentage Analysis: Figure 11 shows the distribution of deduplicated blocks at different check steps on the SPEC CPU 2017 benchmarks. Here, we do not set up the check steps (C) to a specific value. Instead, when the program stores a pointer, we make FPN traverse the corresponding block list until it either finds a matching block or reaches the list's end. We record the number of steps required to find a match and group them into four categories: found at step 1, step 2, step 3, and step >3. On average, 62.58% of blocks are identified at the first step; 17.51% of blocks are identified at the second step, 3.51% of blocks are identified at the third step, and 9.32% at the other steps. In benchmarks such as `povray`, `lbm`, `x264`, and `nab`, over 90% of deduplicated blocks are found in the first step, demonstrating strong spatial locality in pointer storage and validating FPN's block-based registration strategy. A few benchmarks, such as `deepsjeng`, show a higher proportion of matches in the ">3" category. However, these benchmarks still report less than 5.00% performance overhead, indicating that even longer block list traversals do not significantly impact runtime. Overall, these results confirm that most deduplication lookups require only a small number of traversal steps. This supports our design choice that block-based deduplication introduces minimal performance overhead even under diverse and dynamic memory access patterns. In Section VI-F2, we further evaluate the impact of the check step threshold C on FPN's performance and memory usage.

2) Performance Breakdown: To understand where FPN incurs runtime overhead, we analyze its costs across three components using SPEC CPU 2017 benchmarks: (1) metadata setup during buffer allocation, (2) block registration when a pointer is stored, and (3) pointer nullification during buffer deallocation. The breakdown results are shown in Fig. 12. Across all benchmarks, block registration is the dominant source of overhead, accounting for 12.44% performance overhead. This cost stems from four main operations: computing the pointer list address, setting the corresponding status bit, deduplicating to avoid redundant block registrations, and updating the doubly linked block list. Among these, computing the pointer list address is the most expensive, contributing 9.18% on average, while the remaining operations together account for 3.26%. In contrast, our breakdown of CAMP shows that computing the pointer list address alone contributes over 60.0% performance overhead. This highlights the efficiency of FPN's region-based metadata management in reducing address computation overhead. Pointer nullification contributes 3.85% overhead on average, with some variation

[6]https://www.similarweb.com/top-websites/

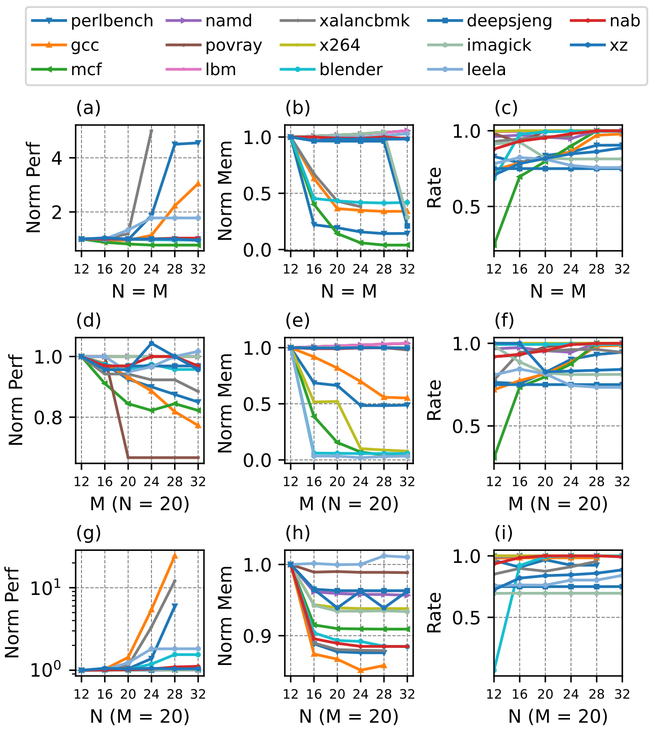


Fig. 13: Explorations on how the block and region sizes affect FPN's performance and memory consumption: (a)-(c) N and M vary together from 12 to 32; (d)-(f) N is fixed and M varies from 12 to 32; (g)-(i) M is fixed and N varies from 12 to 32.

across benchmarks. In workloads such as `omnetpp` and `gcc`, which involve frequent deallocations, this component incurs a higher performance cost due to increased buffer list traversal and updates. Metadata setup introduces the lowest overhead—just 1.89% on average. This is attributed to FPN's region-level metadata organization, which avoids per-buffer initialization. When buffers fall within an existing region, the new buffer's information is appended to a list with minimal cost.

F. Explorations on FPN

1) Exploration on Region Sizes and Block Sizes: To evaluate how region size (N) and block size (M) affect FPN's performance, memory consumption, and deduplication rate, we conduct a series of experiments using the SPEC CPU 2017 benchmark suite. In Fig. 13(a–c), we vary both N and M together (i.e., $N = M$) from 12 to 32 in steps of 4. In Fig. 13(d–f), we fix $N = 20$ and vary M from 12 to 32. In Fig. 13(g–i), we fix $M = 20$ and vary N over the same range. For each setting, performance and memory are normalized to a baseline: ($N = M = 12$) for Fig. 13(a–c), ($N = 20$, $M = 12$) for Fig. 13(d–f), and ($N = 12$, $M = 20$) for Fig. 13(g–i). As shown in Fig. 13(a), increasing both N and M initially reduces normalized performance, reaching its minimum when $N = M = 20$. Beyond that, larger values

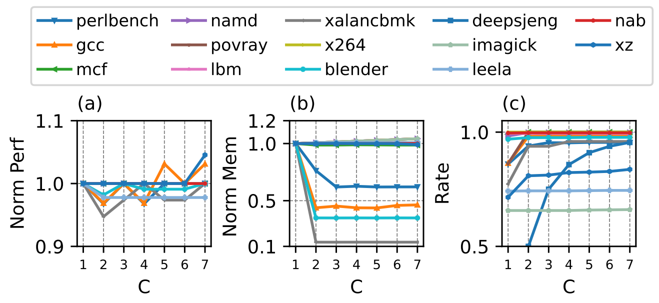


Fig. 14: Explorations on how the checked block number C affects FPN's performance and memory consumption on SPEC CPU 2017

introduce more overhead due to increased scanning and lookup costs during deallocation. At smaller N and M, FPN registers more blocks, which raises overhead due to frequent insertions. In contrast, larger values improve deduplication by grouping more pointers, but they also enlarge block and region sizes, requiring more bytes to be scanned at free time. Fig. 13(b) shows that the normalized memory decreases consistently as N and M grow, due to the reduced number of registered blocks and metadata entries. Fig. 13(c) reports deduplication hit rates, which improve up to $M = 20$ but degrade afterward—indicating diminishing spatial locality benefits at larger block sizes. In Fig. 13(d–f), where $N = 20$ and M varies, increasing M enhances deduplication and reduces both performance and memory (Fig. 13(d–e)). This trend is confirmed by Fig. 13(f), which shows rising deduplication rates as M increases. Conversely, in Fig. 13(g–i), where $M = 20$ and N varies, normalized performance increases significantly with larger N (Fig. 13(g)), mainly due to longer buffer scanning ranges. Meanwhile, Fig. 13(h) shows normalized memory decreases with larger N, since more buffers share the same metadata entry. Fig. 13(i) shows a slight improvement in deduplication rate with growing N. Overall, these results suggest that balancing M and N is essential to optimizing FPN. A configuration with larger blocks (e.g., $M = 32$) and moderate region sizes (e.g., $N = 20$) strikes an effective tradeoff between performance and memory efficiency.

2) Exploration on the Number of Checked Registered Blocks: To evaluate the impact of the deduplication step limit C on FPN's performance and memory efficiency, we conduct experiments using the SPEC CPU 2017 benchmark suite. Recall that C determines the maximum number of previously registered blocks FPN checks when attempting to deduplicate a new registration. Performance and memory are normalized to the baseline configuration with $C = 1$. As shown in Fig. 14(a), increasing C results in a modest increase in normalized performance, due to the additional comparisons required during block list traversal. Fig. 14(b) shows a substantial drop in normalized memory when C increases from 1 to 2, as more redundant registrations are eliminated. Beyond $C = 2$, the memory savings level off.

Fig. 14(c) further confirms that deduplication rates improve as C increases. These results suggest that a small C value (e.g., $C = 2$) strikes an effective balance—achieving most of the memory savings with minimal impact on performance.

VII. Discussion

A. Supporting Third Party/Uninstrumented Libraries

When source code is available, third-party libraries are instrumented using the same FPN compilation pipeline. For precompiled libraries, we recompile them from source or substitute FPN-instrumented versions when possible. If a library cannot be instrumented (e.g., closed-source binaries or inline assembly), FPN remains compatible: uninstrumented modules continue operating correctly, although pointer operations occurring entirely inside such modules cannot be tracked.

B. Integration with Other Mitigation Systems

FPN is designed to be compatible with existing memory safety mechanisms. Its modular design and compile-time instrumentation make it compatible with a wide range of defenses that target different classes of vulnerabilities.

C. Unaligned Pointers

C/C++ programs may store pointers inside packed structures, resulting in unaligned (non–8-byte) pointer locations. To support this scenario, FPN can extend its status table from 1 bit per 8 bytes to 1 bit per byte. With this extension, FPN incurs a 19.17% geometric-mean performance overhead and 12.91% memory overhead—still lower than prior PN-based techniques (details are included in TABLE IV). In our evaluation of SPEC CPU benchmarks and real-world CVEs, we do not observe unaligned pointer storage; all pointers are stored with 8-byte boundaries.

D. Implications of a Large Block Size

As block size increases, FPN scans larger memory ranges during buffer deallocation. In the extreme case, an oversized block makes FPN resemble a garbage collector, which considers broad memory regions during pointer identification. However, unlike GCs, FPN uses status bits to restrict scanning only to locations known to store pointers. Thus, even with large blocks, FPN avoids full-region scans and maintains significantly lower deallocation overhead than GC-based approaches.

VIII. Related Work

A. UAF Sanitizers

UAF sanitizers [8]–[13], [17], [44], [45] are tools or techniques designed to detect and mitigate UAF vulnerabilities at runtime. They aim to ensure memory safety by validating the pointer before each memory access and preventing dangling references.

1) Fat and Non-Fat Pointers: Fat and Non-Fat Pointers [8]–[17], [17]–[20], [43], [45] modify traditional pointer structures to include additional metadata, such as the lifetime info of a pointer. When the program frees a buffer, the lifetime info of this buffer is set to be invalid, preventing access to the memory through the associated pointers. However, these methods require compiler modifications to pointer dereference operations, increase pointer size, and introduce high memory and performance overhead.

2) Lock-and-Key Checks: Lock-and-Key techniques [14]–[20], [43], [46] assign a unique key to each allocated buffer and embed a corresponding lock in all associated pointers. Upon deallocation, the buffer's key is invalidated, and every pointer dereference triggers a lock–key consistency check; any mismatch is treated as a UAF violation. While highly accurate, these schemes incur substantial performance and memory overhead due to frequent key updates and metadata maintenance.

3) Hardware-based Sanitizers: Hardware-assisted UAF defenses [47]–[53] accelerate validity checks through dedicated hardware units. Buffer-bound checks [54], [55] and hardware implementations of Lock-and-Key schemes [56], [57] further reduce software overhead. Pointer-tagging approaches [45], [58], [59] encode metadata in upper pointer bits, while cryptographic methods [60] use hardware engines to protect buffer lifetimes. Although these designs achieve very low runtime overhead, they face significant deployment barriers due to specialized hardware requirements.

B. Secure Allocators

Unlike traditional memory allocators that prioritize performance, secure allocators aim to mitigate UAF vulnerabilities by modifying allocation and deallocation behavior. Some approaches [14], [21]–[23], [61] randomize allocation to reduce the likelihood of a malicious buffer reusing the same address as a deallocated victim buffer. However, this only limits, rather than eliminates, an attacker's control over memory layout. Other methods [38], [62] delay virtual address reuse, but UAF exploits remain possible once buffers are eventually released. More restrictive techniques, One-Time-Allocator (OTA) [24]–[27] prevent freed buffers from ever reusing virtual addresses, blocking UAF exploits via address reuse. However, long-running programs risk exhausting virtual address space.

C. Garbage Collectors (GCs)

GC systems [29], [32], [63], [64] mitigate UAF by periodically scanning memory to identify live pointers and reclaim buffers no longer referenced. This automated reclamation prevents dangling-pointer dereferences but introduces notable costs: periodic scanning and bookkeeping incur runtime overhead, delayed reclamation reduces reuse efficiency, and background GC threads can interfere with multi-threaded execution due to CPU contention and synchronization.

D. Reference Counters

Reference counters [32], [65], [66] track the number of active references to a memory object and reclaim the object when

Ratio	Normalized	perlbench	gcc	mcf	namd	povray	lbm	omnetpp	xalancbmk	x264	blender	deepsjeng	imagick	leela	nab	xz	GEOMEAN
Per 1-byte	Performance	1.92x	1.56x	1.21x	0.96x	1.27x	0.99x	2.03x	1.20x	0.84x	1.45x	1.03x	1.01x	0.98x	1.01x	1.02x	1.19x
	Memory	1.34x	1.17x	1.14x	1.05x	1.01x	1.01x	1.58x	1.29x	1.04x	1.17x	1.00x	1.18x	0.99x	1.16x	0.97x	1.13x
Per 8-byte	Performance	1.83x	1.62x	1.16x	1.00x	1.23x	0.97x	1.98x	1.17x	0.81x	1.41x	1.01x	1.01x	1.00x	1.01x	1.03x	1.18x
	Memory	1.22x	1.27x	1.02x	1.01x	1.00x	1.00x	1.48x	1.17x	1.02x	1.06x	1.00x	1.03x	1.00x	1.05x	1.03x	1.08x

TABLE IV: FPN's normalized performance and memory under different status bit ratios on SPEC CPU 2017. Baseline is 1x.

the count reaches zero. While effective in preventing premature deallocation, these methods introduce runtime overhead due to frequent counter updates and thread synchronization.

E. Static Analysis Approaches

Static analysis approaches [3]–[7], [67] detect UAF vulnerabilities at compile time by analyzing source code or binaries without executing the program. They impose no runtime overhead and are useful early in development. However, real-world software—featuring dynamic memory usage, indirect calls, and multithreading—often exceeds the precision and scalability of static analysis, leading to false positives and false negatives. Consequently, static analysis is typically combined with dynamic or runtime techniques for comprehensive UAF protection.

IX. Conclusion

FPN improves pointer nullification by introducing region-based metadata and block-based registration, reducing both metadata lookup cost and registration volume. Our evaluation shows that FPN significantly lowers performance and memory overhead compared with prior PN methods while preserving strong temporal memory safety. FPN is compatible with large-scale, multithreaded software, demonstrating its practicality. Overall, FPN makes pointer nullification more scalable and efficient without compromising security.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback and suggestions. This work is supported by NSF grants #2334628 and #2154973.

References

[1] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 414–425.

[2] M. Miller, "A snapshot of vulnerability root cause trends for micrsoft remote code execution (rce) cves, 2006 through 2017," 2018.

[3] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, "Coccinelle: tool support for automated cert c secure coding standard certification," *Science of Computer Programming*, vol. 91, pp. 141–160, 2014.

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.

[5] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[6] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 327–337.

[7] ——, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 42–54.

[8] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: a safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[9] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 128–139.

[10] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 232–244, 2003.

[11] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[12] G. J. Duck and R. H. 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*, 2018, pp. 181–195.

[13] J. Zhou, J. Criswell, and M. Hicks, "Fat pointers for temporal memory safety of c," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 316–347, 2023.

[14] 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*, 2010, pp. 31–40.

[15] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 381–392.

[16] H. Cho, J. Park, A. Oest, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Vik: practical mitigation of temporal memory safety violations through object id inspection," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 271–284.

[17] Y. Du, Y. Guo, Y. Zhang, and J. Yang, "Rtt-uaf: Reuse time tracking for use-after-free detection," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 376–387.

[18] B. Gui, W. Song, and J. Huang, "Uafsan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 309–321.

[19] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1037–1054.

[20] M. Momeu, S. Schnückel, K. Angnis, M. Polychronakis, and V. P. Kemerlis, "Safeslab: Mitigating use-after-free vulnerabilities via memory protection keys," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1345–1359.

[21] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," *Acm sigplan notices*, vol. 41, no. 6, pp. 158–168, 2006.

[22] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC*

Conference on Computer and Communications Security, 2017, pp. 2389–2403.

[23] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, "Guarder: A tunable secure allocator," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 117–133.

[24] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical Page-Permissions-Based scheme for thwarting dangling pointers," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 815–832.

[25] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[26] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1307–1322.

[27] J. Ahn, J. Lee, K. Lee, W. Gwak, M. Hwang, and Y. Kwon, "BUDAlloc: Defeating Use-After-Free bugs by decoupling virtual address management from kernel," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 181–197.

[28] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "PUMM: Preventing Use-After-Free using execution unit partitioning," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 823–840.

[29] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 578–591.

[30] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 212–225.

[31] J. Ahn, K. Lee, C. Park, H. Moon, and Y. Kwon, "Swiftsweeper: Defeating use-after-free bugs using memory sweeper without stop-the-world," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 793–809.

[32] J. Shin, D. Kwon, J. Seo, Y. Cho, and Y. Paek, "Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++." in *NDSS*, 2019.

[33] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.

[34] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[35] Z. Shen and B. Dolan-Gavitt, "Heapexpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 454–465.

[36] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1635–1648.

[37] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4015–4032.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[39] B. Spengler, "Pax: The guaranteed end of arbitrary code execution," *G-Con2: Mexico City, Mexico*, 2003.

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

[41] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing Use-After-Free attacks with fast forward allocation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2453–2470.

[42] L. Bernhard, M. Rodler, T. Holz, and L. Davit, "xtag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on intel x86-64," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 502–519.

[43] F. Gorter and C. Giuffrida, "Rangesanitizer: Detecting memory errors with efficient range checks," in *USENIX Security*, 2025.

[44] E. Q. Vintila, P. Zieris, and J. Horsch, "Evaluating the effectiveness of memory safety sanitizers," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 774–792.

[45] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1901–1915.

[46] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient heap memory protection through advanced metadata management and customized compiler optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7177–7193.

[47] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "Zerø: Zero-overhead resilient operation under pointer integrity attacks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 999–1012.

[48] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 545–557.

[49] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 762–775.

[50] Z. Liu, Y. Rong, C. Li, W. Tan, Y. Li, X. Han, S. Yang, and C. Zhang, "Cctag: Configurable and combinable tagged architecture." in *NDSS*, 2025.

[51] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, "Cornucopia: Temporal safety for cheri heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 608–625.

[52] N. W. Filardo, B. F. Gutstein, J. Woodruff, J. Clarke, P. Rugg, B. Davis, M. Johnston, R. Norton, D. Chisnall, S. W. Moore *et al.*, "Cornucopia reloaded: Load barriers for cheri heap temporal safety," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 251–268.

[53] B. Gutstein, "Memory safety with cheri capabilities: security analysis, language interpreters, and heap temporal safety," University of Cambridge, Computer Laboratory, Tech. Rep., 2022.

[54] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 631–644.

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

[56] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 189–200, 2012.

[57] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 175–184.

[58] K. Sinha and S. Sethumadhavan, "Practical memory safety with rest," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 600–611.

[59] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 916–929.

[60] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal *et al.*, "Cryptographic capability computing," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 253–267.

[61] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Top of the heap: Efficient memory error protection of safe heap objects," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1330–1344.

[62] C. Park and H. Moon, "Efficient use-after-free prevention with opportunistic page-level sweeping," in *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[63] H.-J. Boehm, "Space efficient conservative garbage collection," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 197–206, 1993.

[64] ——, "Bounding space usage of conservative garbage collectors," *Acm Sigplan Notices*, vol. 37, no. 1, pp. 93–100, 2002.

[65] D. Anderson, G. E. Blelloch, and Y. Wei, "Concurrent deferred reference counting with constant-time overhead," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 526–541.

[66] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang, "FreeWill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2497–2512.

[67] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover cross-entry use-after-free vulnerabilities in the linux kernel," in *Network and Distributed System Security (NDSS) Symposium*, 2025.

Appendix

A. Baseline of Each UAF Mitigation Methods

Method	Baseline
FPN	Clang 18.1.0
DangSan	Clang 18.1.0 with tcmalloc
FreeSentry	Clang 18.1.0
CAMP	Clang 12.0.1 with tcmalloc
FFmalloc	Clang 18.1.0
DangZero	gcc9
FreeGuard	Clang 18.1.0
MarkUS	Clang 18.1.0
BUDAlloc	gcc 13.3.0
SwiftSweeper	gcc 13.3.0
xTag	Clang 10.0.1 with mimalloc
RSan	Clang 16.0.6 with tcmalloc

TABLE V: Baseline of each UAF mitigation method

CVE	FFmalloc	DangZero	Freeguard	MarkUS	BUDAlloc	SwiftSweeper	xTag	RSan
CVE-2015-3205	✔	✔	✔	✔	✔	✔	✔	✔
CVE-2015-2787	✔	✔	✔	✖	✔	✔	✔	✖
CVE-2015-6835	✔	✔	✔	✖	✔	✔	✔	✔
CVE-2016-5773	✔	✔	✔	✖	✔	✔	✔	✖
Issue-3515	✔	✔	✔	✖	✔	✔	✔	✖
CVE-2020-6838	✔	✔	✖	✖	✔	✔	✔	✖
CVE-2021-44964	✔	✔	✔	✔	✔	✔	✔	✖
CVE-2020-21688	✔	✔	✔	✔	✔	✔	✖	✔
CVE-2021-33468	✔	✔	✖	✔	✔	✔	✖	✔
CVE-2020-24978	✔	✔	✔	✔	✔	✔	✔	✔
Issue-1325664	✔	✖	✔	✖	✖	✔	✖	✖
CVE-2022-43286	✔	✖	✔	✔	✔	✔	✔	✖
CVE-2019-16165	✔	✔	✔	✔	✔	✔	✔	✖
CVE-2021-4187	✔	✔	✔	✔	✔	✔	✔	✖

TABLE VI: Security evaluation of FPN and related work on real-world applications with UAF bugs. ✔ means successfully detecting or preventing UAF vulnerabilities. ✖ means failure in detecting or preventing UAF vulnerabilities.

		SPEC CPU 2017		SPEC CPU 2006	
Methods	Standard errors (%)	Max	Min	Max	Min
FFmalloc	Performance	0.33	0.00	0.49	0.00
	Memory	2.15	0.00	1.78	0.00
FFmalloc baseline	Performance	1.89	0.00	0.45	0.00
	Memory	1.94	0.00	1.13	0.00
DangZero	Performance	1.64	0.10	3.51	0.82
	Memory	0.75	0.00	0.59	0.00
DangZero baseline	Performance	2.71	0.02	1.25	0.11
	Memory	0.98	0.00	0.55	0.00
FreeGuard	Performance	0.96	0.00	0.44	0.00
	Memory	2.16	0.00	1.22	0.00
FreeGuard baseline	Performance	1.89	0.00	0.45	0.00
	Memory	1.94	0.00	1.13	0.00
MarkUS	Performance	2.33	0.00	2.35	0.00
	Memory	0.93	0.00	1.94	0.01
MarkUS baseline	Performance	0.33	0.00	0.45	0.00
	Memory	1.94	0.00	1.13	0.00
BUDAlloc	Performance	2.52	0.03	1.66	0.13
	Memory	0.49	0.00	1.82	0.00
BUDAlloc baseline	Performance	0.51	0.02	1.99	0.78
	Memory	0.71	0.00	0.60	0.00
SwiftSweeper	Performance	1.06	0.05	1.16	0.05
	Memory	0.13	0.00	0.64	0.00
SwiftSweeper baseline	Performance	2.35	0.02	1.49	0.05
	Memory	0.63	0.00	0.69	0.00
xTag	Performance	1.26	0.00	0.76	0.00
	Memory	1.39	0.00	1.33	0.00
xTag baseline	Performance	0.79	0.09	0.74	0.00
	Memory	1.17	0.00	1.01	0.00
RSan	Performance	0.37	0.00	1.31	0.00
	Memory	2.25	0.00	3.64	0.00
RSan baseline	Performance	1.81	0.00	1.98	0.13
	Memory	0.91	0.02	0.98	0.00
FreeSentry	Performance	2.37	0.00	1.79	0.00
	Memory	2.37	0.00	3.62	0.02
FreeSentry baseline	Performance	1.89	0.00	0.45	0.00
	Memory	1.94	0.00	1.13	0.00
DangSan	Performance	0.46	0.00	0.58	0.00
	Memory	0.49	0.00	1.02	0.00
DangSan baseline	Performance	1.17	0.00	0.45	0.00
	Memory	1.94	0.00	1.04	0.00
CAMP	Performance	1.56	0.00	0.26	0.00
	Memory	3.09	0.00	1.21	0.00
CAMP baseline	Performance	1.27	0.24	0.85	0.00
	Memory	0.34	0.00	1.19	0.00
FPN	Performance	0.94	0.00	1.07	0.00
	Memory	0.69	0.00	2.63	0.08
FPN baseline	Performance	0.33	0.00	0.45	0.00
	Memory	1.94	0.00	1.13	0.00

TABLE VII: Maximum and minimum standard errors of each UAF mitigation method and corresponding baseline on SPEC CPU 2017 and SPEC CPU 2006. Each method and corresponding baseline are run 10 times to calculate standard errors.

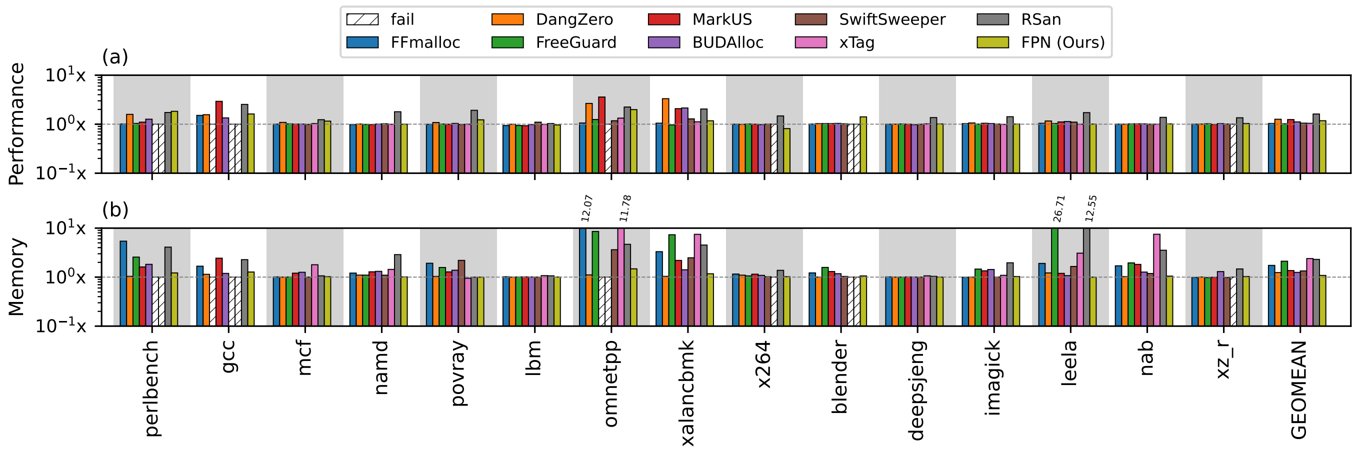


Fig. 15: Evaluation of normalized performance and memory of FPN and previous non-PN methods on SPEC CPU 2017, under optimization level -O2. Baseline is 1x.

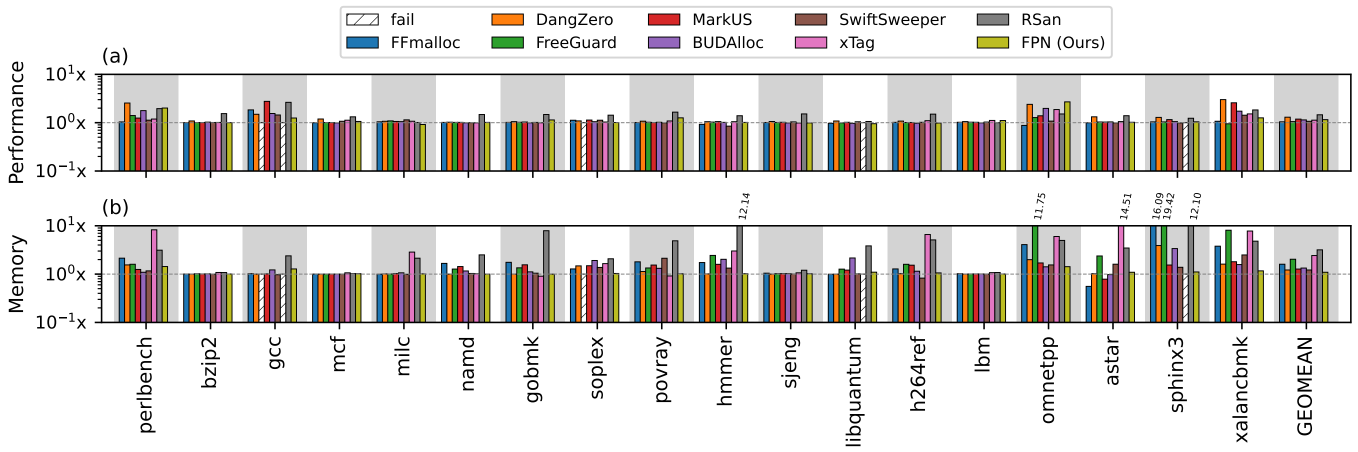


Fig. 16: Evaluation of normalized performance and memory of FPN and previous non-PN methods on SPEC CPU 2006, under optimization level -O2. Baseline is 1x.