Received 21 October 2023, accepted 15 November 2023, date of publication 18 December 2023, date of current version 12 January 2024.

*Digital Object Identifier 10.1109/ACCESS.2023.3343777*

APPLIED RESEARCH

# Enhancing a Lock-and-Key Scheme With MTE to Mitigate Use-After-Frees

**INYOUNG BANG [1], MARTIN KAYONDO [1], JUNSEUNG YOU [1], DONGHYUN KWON [2], YEONGPIL CHO [3], AND YUNHEUNG PAEK [1], (Member, IEEE)**

[1]Department of Electrical and Computer Engineering, Inter-University Semiconductor Research Center, Seoul National University, Seoul 08826, South Korea
[2]School of Computer Science and Engineering, Pusan National University, Busan 46241, South Korea
[3]Department of Computer Science, Hanyang University, Seoul 04763, South Korea

Corresponding authors: Yeongpil Cho (ypcho@hanyang.ac.kr) and Yunheung Paek (ypaek@snu.ac.kr)

This work was supported in part by the BK21 FOUR Program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2023; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) (RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response) under Grant 2021-0-00724; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) (Analysis on Technique of Accessing and Acquiring User Data in Smartphone) under Grant 2020-0-01840; and in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant RS-2023-00277326 and Grant NRF-2022R1A4A1032361.

**ABSTRACT** Preventing Use-After-Free (UAF) bugs is crucial to ensure temporal memory safety. Against UAF attacks, much research has adopted a well-known approach, *lock-and-key*, in which unique, disposable locks and keys are first assigned respectively to objects and pointers, and then on every memory access, checked for a match. Attention has been drawn again to this approach by recent work that capitalizes on a vast abundance of virtual address (VA) space in the lock assignment, thus being able to prevent UAFs in stripped binary. However, as this VA-based lock-and-key scheme tends to rapidly consume virtual space, it is likely to suffer from high performance overhead. In this paper, we propose a new scheme, called the *VA tagging*, whose goal is to tackle this performance problem with the support of the *Memory Tagging Architecture* (MTA) introduced in several commodity processors. In our scheme, the original VA-based locks are augmented with tags of MTA. As a VA-based lock can be assigned to multiple objects with different tags, the same VA is reused for many objects without compromising temporal safety. We have observed in our experiments that this tagging scheme lowers the VA consumption rate drastically by one order of magnitude. We implement a light-weight memory allocator, *Vatalloc*, by modifying existing allocators, dlmalloc and jemalloc, to employ the VA tagging scheme for efficient prevention of UAFs. Our evaluation shows that Vatalloc with allocator modifications only incurs 1.70 % (on dlmalloc) and 3.05 % (on jemalloc) of runtime overhead without considering performance degradation of MTE. As a result of simulating the tagging architecture assuming the worst-case, postulating MTE precise trapping mode incurs performance overhead of 30.9 % based on dlmalloc, and 25.5 % based on jemalloc. If imprecise mode is assumed, the slowdown is measured 16.9 % for dlmalloc and 12.0 % for jemalloc respectively. Vatalloc only incurs 19.0 % and 3.0 % memory overhead for dlmalloc and jemalloc respectively.

**INDEX TERMS** Memory safety, temporal safety, hardware, security, memory management, tagging architecture.

## I. INTRODUCTION

Dangling pointers refer to the pointers whose referent objects have been freed. These pointers are common in C/C++, and they themselves do not pose a threat to a program. However, dereferencing dangling pointers introduces susceptibilities to *Use-After-Free* (UAF) bugs, which may lead to arbitrary memory access and control flow hijacking. UAFs are prevalent across applications, as demonstrated in the statistical report of the MITRE where it ranks among the top 25 most dangerous software errors [1]. To date, a lot of techniques [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] have been invented to stymie UAF attacks in question.

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

Conventional techniques for preventing UAF bugs can be broadly classified into two approaches: *lock-and-key* and *pointer nullification*. The former has long been proven effective by many studies [12], [13], [15]. In this approach, UAF bugs are prevented by validating every memory access. To be specific, *locks* and *keys* are first assigned to objects and pointers respectively. Every pointer dereference is validated by checking that the key held by a pointer matches the lock held by its referent object. To prevent UAF attacks that exploit dangling pointers, locks are invalidated as soon as the corresponding objects are freed, rendering the keys of associated dangling pointers outdated. In contrast, the latter relies on compiler instrumentation [2], [3], [5], [11] or garbage collection like solutions [8], [9], [10], [14]. Compiler based solutions require source code, which hampers its general adoption in practice. Some of GC-based techniques necessitate hardware modifications [8], [9] or entail runtime overhead, despite being disguised by concurrency and generous provisioning of compute and memory resources [18].

Earlier studies [12] assigned unique integers (typically 64-bit) to represent locks and keys. Unfortunately, the integer-based representation has a critical downside that it requires heavy data structures to manage the locks and keys associated with objects and pointers. Techniques that use this representation tend to suffer from high runtime overhead, as they have to perform frequent load/store and matching operations for each lock and key assigned during program execution. As explained in Section VIII, due in part to the high computational cost associated with locks and keys, the research community has largely shifted towards the pointer nullification approach stated above [3].

Most recently, however, Oscar [15], Dangzero [17] and FFmalloc [16] proposed an innovative idea that can reduce the computing cost of the lock-and-key approach, rendering it more attractive to use. Whereas traditional lock-and-key-based techniques have been plagued by excessive runtime overhead resulting from managing and computing integer locks/keys, the more recent approaches have moved towards using *virtual addresses* (VAs) of objects/pointers to represent locks/keys. In this scheme, unique, disposable locks are implicitly generated throughout program execution by assigning different VAs to all objects created in the program in their VA-based representation. When an object is freed, the VAs assigned to it are invalidated. In so doing, dereferencing the dangling pointers that hold the invalidated VAs of the freed object is trapped by a memory fault, leading to the prevention of UAFs. However, the VA-based scheme comes with an inherent drawback. The scheme requires that each VA is allocated to objects only once to ensure the uniqueness of locks, which negatively impacts the system and VAs will be run out rapidly. As VAs should be discarded after a single use, caching mechanisms of memory allocators to efficiently reuse freed memory for near future allocation requests becomes inapplicable. Also, it intensifies memory fragmentation over time, ultimately causing performance and memory overheads.

In this paper, we propose a lock-and-key scheme, called the *VA tagging*, which effectively resolves scalability issues associated with the current VA-based prevention scheme. Our scheme significantly reduces VA exhaustion while maintaining efficient performance and minimal memory overhead. The scheme leverages the *Memory Tagging Extension* (MTE), a hardware feature that has been announced in the latest line of ARM processors, ARMv8.5-A [19]. VA tagging scheme augments the original VA-based locks with *tags*, and memory accesses are allowed only when their tags match with the aid of MTE. Specifically, during the allocation of a new object, the object is assigned a VA along with a tag number, and the associated pointer is also assigned the same tag. When an object is freed, we do not simply invalidate its VAs as in the original VA-based schemes; rather, we modify its tag numbers. This alteration prevents UAF attacks in subsequent dereferences of associated dangling pointers, due to the disparity between the tag numbers assigned to the freed objects and the dangling pointers. Unlike the conventional VA based techniques, each VA can be assigned to objects as many times as the number of available MTE tags, without losing the prevention capability against UAF attacks. By doing so, our scheme effectively alleviates the aforementioned issues caused by single-use only VAs.

To evaluate the feasibility and effectiveness of our VA tagging scheme, we have implemented a technique, called *Vatalloc*, that provides light-weight prevention of UAF attacks. It is noteworthy that our VA tagging scheme can be seamlessly incorportated into any allocators. We have demonstrated this by implementing Vatalloc in two popular allocators, namely *dlmalloc* and *jemalloc*, which have distinct design philosophies. With only 0.9K and 0.2K additional LoC respectively, the VA tagging scheme is successfully applied to both allocators.

The empirical results with SPEC2006 [20] and PARSEC [21] benchmark suites clearly demonstrate the Vatalloc can mitigate UAF attacks with high efficiency in terms of time and space. In short, Vatalloc with allocator modifications only incurs 1.70 % (on dlmalloc) and 3.05 % (on jemalloc) of runtime overhead, and 19.0 % (on dlmalloc) and 3.0 % (on jemalloc) of memory overhead. To measure an accurate estimation of worst case performance of MTE, we simulated both tag update and tag matching. Vatalloc on dlmalloc and jemalloc resulted in 16.9 % and 12.0 % slowdown, with 19.0 % and 3.0 % memory overhead respectively.

In summary, our contributions are:

- We propose a VA tagging scheme with MTE enables light-weight and scalable prevention of UAF attacks without source code of a target program.
- We have experimentally showed that Vatalloc can be implemented on existing memory allocators with minimal modification.
- We performed reasonable proxy measurements to evaluate worst-case performance of Vatalloc despite the absence of real hardware or cycle-accurate simulator. We found that most of the overhead incurred is due only to

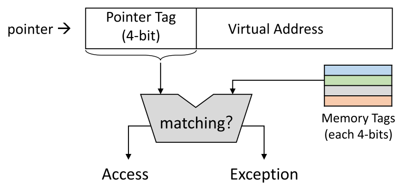


**FIGURE 1.** **Operation of MTE.**

tag matching on memory accesses, and Vatalloc induces insignificant degradation on performance and memory.

## II. BACKGROUND

In this section, we provide background information on MTE, the core hardware feature leveraged by Vatalloc, and two major dynamic memory allocators, dlmalloc and jemalloc, to which we applied Vatalloc for prototype implementation.

### A. MEMORY TAGGING EXTENSION (MTE)

MTE follows the design of tagged memory architecture [22], which aims to efficiently improve the security of the system. This hardware feature has been introduced on the recent ARMv8.5-A [23] architecture. As described in Figure 1, MTE deploys two types of tags: *pointer tags* and *memory tags* that are assigned to each pointer and each 16 B (or 32 B) memory block. Pointer tags are attached to the top bits of pointers and *memory tags* that are stored in a separate memory storage. The pointer tag is implicitly propagated through pointer arithmetic to other pointers that equally refer to the object, thus all the associated pointers can be used to access the object. Each tag is 4 bits large, therefore, its value ranges from 0 to 15. When enabled, MTE checks every memory access by comparing the tag numbers of the pointer and target memory. If a mismatch occurs, MTE raises an exception synchronously or asynchronously by configuration. Pointer tags can be easily extracted from the value of pointers, and similarly memory tags can be loaded with minimal latency by preparing a dedicated cache memory for them or placing them inside ECC bits beside the associated memory blocks.

MTE provides two operation modes: precise and imprecise. The former introduces performance overhead, but supports synchronous exception providing the faulting address, which means that a tag mismatch is identified instantly upon an illegal memory access. The latter operates fast, but exception is raised asynchronously, meaning a tag mismatch is reported later after an illegal memory access. In regards to this imprecise mode, ARM features an option to release the delayed notification upon entering the kernel so that the kernel can recognize the occurrence of any tag mismatch. To enable programs to be informed of the tag mismatch exception from MTE, a recent Linux kernel [24] with a MTE support generates a `SIGSEGV` signal when a tag mismatch occurs. The signal is passed to programs along with a code, `SEGV_MTESERR` (i.e., synchronous error) at the precise mode or `SEGV_MTEAERR` (i.e., asynchronous error) at the imprecise mode. Therefore, it is possible that a program deal with tag mismatch of MTE by writing its own signal handler.

### B. DLMALLOC

dlmalloc [25] is a former primary dynamic memory allocator of Linux and Android, and the base of ptmalloc [26] currently used in Linux by default. Upon receipt of a memory allocation request, dlmalloc internally captures a memory block, called a *chunk*, and returns its address. Chunks are allocated from a pool, called a *segment*, allocated by the kernel through `srbk` and `mmap` system calls. Chunks are classified by size into *small* and *large* groups, based on a defined threshold (by default, 256 KB). Such a classification reflects the common usage pattern for objects that varies by size: generally, small objects are frequently allocated and have a short life-time, but large objects are rarely allocated and have a long life-time. Therefore, for a proper control of allocation overheads, dlmalloc manages these two types of chunks in different methods. Allocation/deallocation requests for large chunks are page-aligned and directly handled by *mmap* and *munmap*, which implies that dlmalloc does not reuse large chunks as its memory pool. On the other hand, the requests for small chunks are managed for effective reuse. On allocation, a request size is aligned to a multiple of 16 (or 32) bytes on 64-bit architecture. Dlmalloc then searches with a best-fit strategy a *free list (or tree* in which freed small chunks have been collected. Absence of a fitting free chunk results in dlmalloc splitting the top chunk that is the topmost free chunk in each segment as the final resort. On deallocation, dlmalloc consolidates (or coalesces) the just freed chunk with adjacent freed chunks to suppress external fragmentation. To facilitate such management, dlmalloc augments each chunk with a 16-byte sized *header* that consists of several fields as described in Figure 3. In addition, dlmalloc inserts node metadata into freed chunks to maintain the free list without consuming additional memory.

### C. JEMALLOC

Jemalloc [27] is being used in Mozilla Firefox for the Windows, Mac OS X and Linux platforms, and was adopted as the default system allocator on the FreeBSD and NetBSD operating systems. For better multithreading support, Jemalloc relies on the concept of *arenas*, where a specific arena is associated with particular execution threads to overcome lock contention issues between threads. The arena serves the required concurrency of the program and its number is proportional to the number of available CPU cores. Jemalloc categorizes the size of malloc requests into small, large and huge. Requests less than 4KB are classified as *small*, those less than 2MB as *large*, and the rest as *huge*. For small and large allocations, bins are used to organize size classes of chunks, and, in our case, for example, 35 size classes are initialized a priori for each small and large allocations. Each bin determines the size and the capacity of chunks in the associated *runs* consisting of one or multiple pages. The fundamental purpose of a run is to keep track of the allocation state of chunks. It holds an indexed bitmap as a part of its metadata, specifying whether the chunk is freed or in-use.

When the malloc request for small or large is placed, a chunk in a run corresponding to the request size is returned to user. In contrast to dlmalloc, jemalloc does not release memory even for huge chunks after deallocation. Instead, it reserves it for later use. For this purpose, jemalloc organizes a separate tree data structure to manage huge chunks.

## III. THREAT MODEL AND PREREQUISITE

### A. THREAT MODEL

We assume the threat model that is consistent with that of the previous work discussed in Section VIII. A target program is not malicious per se, but has dangling pointers. In this work, we assume an adversary can only launch UAF attacks by exploiting the dangling pointers. Defence techniques [28], [29], [30] that are orthogonal and compatible to Vatalloc are assumed to prevent different types of memory attacks such as buffer overflow and type confusion so that the adversary cannot exploit them to subvert or bypass Vatalloc by compromising the integrity of its metadata and MTE's memory/pointer tags.

### B. PREREQUISITE

Vatalloc is designed to operate at memory allocator-level. To apply Vatalloc, therefore, a target program is required to use explicit interfaces for memory management, such as allocation, deallocation, and reallocation.

## IV. VA TAGGING SCHEME

We propose drop-in-use VA tagging scheme for efficient prevention of UAF attacks. As described in Section I, the lock-and-key approach prevents UAF attacks from a mismatch between the lock of an object and the key of a pointer. The performance overhead in this approach is attributed to the expenses incurred in generating unique, disposable locks upon object allocation, executing lock/key matching during memory access, and invalidating these locks upon object deallocation. In this context, our VA tagging scheme significantly reduces the costs by cleverly leveraging both VAs and MTE tags as locks, which leads to a reduction in the assignment and management costs. Our proposed VA tagging scheme leverages MTE tags to allow the reuse of the same VA multiple times. It effectively alleviates heavy kernel intervention for lock management, as opposed to the original VA-based scheme [15].

Figure 2 illustrates how the VA tagging scheme effectively prevents UAF attacks. Initially, when an object is allocated by `malloc`, the random tag number is assigned to both the memory chunk of the object and the returned pointer (Figure 2.(a)). As explained in subsection II-A, the pointer tag attached to the returned pointer is implicitly propagated through pointer arithmetic to other pointers that equally refer to the object, resulting in no extra overhead required to manage it. The key point of our scheme is the incrementation of the tag number assigned to an object upon deallocation. This tag modification renders subsequent attempts to dereference the dangling pointers for UAF attacks be caught due to the tag mismatch (Figure 2.(b-c)).

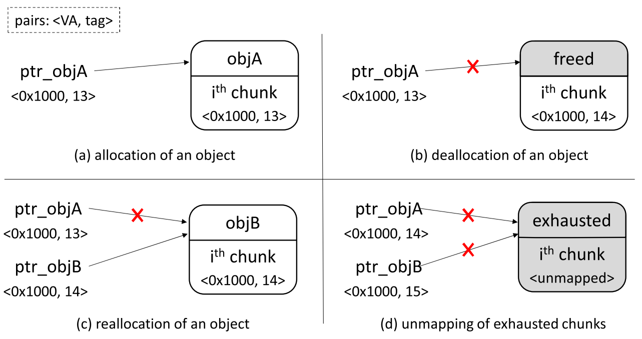


**FIGURE 2. UAF prevention based on the VA Tagging Scheme. For the sake of simplicity, we assume that the tag numbers are initially assigned 0 and subsequently increase until reaching 15.**

In our VA tagging scheme, each tag number must be used only once for the same chunk for safety. Therefore, we mark the chunks with no assignable tag numbers as "*exhausted*" to prevent them from being reused later. Unfortunately, these exhausted chunks will lead to memory leak by taking up physical page frames. To address this problem, we need to keep track of the chunks in order to retrieve their frames by unmapping them. However, we should note that most exhausted chunks will be mixed with non-exhausted chunks within a page. Therefore, to avoid a faulty unmapping, we have to keep track of all exhausted chunks by page, and only unmap the page which is completely filled with exhausted chunks. Once a page is unmapped, dangling pointers referring to somewhere in it become invalidated so that UAF attacks by exploiting them will be prevented (Figure 2.(d)).

## V. VATALLOC

For convenience, we name two versions of Vatalloc implemented based on dlmalloc and jemalloc as *Vatalloc-d* and *Vatalloc-j*, respectively. To demonstrate the effectiveness and feasibility of the VA tagging scheme, we implement Vatalloc that realizes the scheme on two real memory allocators, dlmalloc and jemalloc. The two versions of Vatalloc are designed to follow the original design philosophy found in the base memory allocators. For example, dlmalloc, a free-list allocator, is designed to minimize memory overhead, such as by placing its metadata within free chunks. Therefore, Vatalloc-d is designed to reduce memory consumption by devising a relatively complicated but space efficient data structure for its metadata. On the other hand, jemalloc, a bucket allocator, was designed to improve performance in exchange for consuming more memory. Accordingly, Vatalloc-j is designed to reduce performance overhead by using fast but not space-optimized data structure for its metadata.

Vatalloc basically fits into C and C++, but is applicable to other languages that provide dynamic memory allocation interfaces as a form of operators or functions, such as `new` and `delete`, or `malloc` and `free`. Vatalloc is employable in applications either by overriding these functions at run time, or linking against a library containing Vatalloc's substituted functions at loading time.

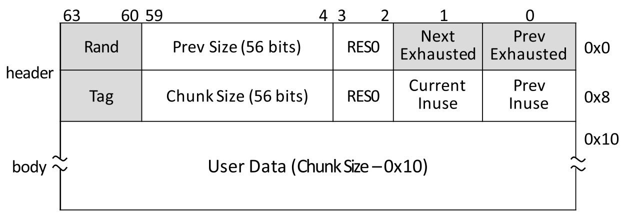


**FIGURE 3.** **Modified Chunk Metadata on Vatalloc-d. The gray field refers to a free bits used as metadata.**

## A. TAG MANAGEMENT

Small chunks (chunks hereafter for short) are prepared for small sized objects. To take advantage of the VA tagging scheme, 4-bit MTE tag numbers must be assigned to the chunks and updated whenever there are state transitions of chunks: *creation*, *free*, and *consolidation*.

Upon spawning new chunks from the heap, Vatalloc assigns a tag number to each. Subsequently, the tag numbers are updated as follows: upon receiving free requests, the corresponding chunk remains to service later allocation requests efficiently and its tag number is incremented. Concurrently, Vatalloc increases the tag number of the chunk by 1. This prevents dereferencing of dangling pointers that still hold an unchanged pointer tag that no longer matches, thus preventing UAF attacks.

### 1) ON DLMALLOC

To implement Vatalloc-d, tag count for each chunk, which refers to how many times the chunk is reused, is required to be stored and tracked as Vatalloc assigns random value to each chunk. It is stored in the chunk header, specifically in the topmost 4-bit of the `chunk_size` field, which is the second byte of the header. Figure 3 shows the modified chunk's header. By doing so, Vatalloc cleverly avoids performance overhead by tag loading and memory consumption for tag management. Since the `chunk_size` field must be accessed upon the state transitions of chunks, Vatalloc minimizes memory operations for accessing tag count in this way. This placement reduces the `chunk_size` bit from 60 to 56, which is acceptable since no objects are bigger than $2^{48}$ bytes. To evenly distribute tag numbers, initial value is randomized and stored in the topmost 4-bit of the `prev_size` field, which is the first byte of the header. The actual value of the tag number is computed by adding the tag count and the initial tag together.

When consolidating adjacent free chunks to minimize fragmentation, tag management becomes crucial. In this process, Dlmalloc checks if there are other free chunks adjacent to the current one, and if so, it consolidate them into one. However, if these chunks have different states, which are initial values and tag counts, Vatalloc recomputes the tag counts and initial value for consolidation. First, the minimal initial value of the two chunks is assigned as the new initial value. Second, the maximum tag count of the two chunks is calculated and delta of the two initial values is added to it and assigned as the new tag count. Although this method preserves tag number consistency, it can lead to faster growth of tag counts, reducing the likelihood of reusing the same virtual addresses. To address this concern, Vatalloc can enforce a consolidation policy based on the recomputed tag count if two adjacent chunks are consolidated. This policy allows consolidation only if the degree of the tag count is equal to or less than a defined threshold, and otherwise, Vatalloc leaves the chunks separate until their tag numbers become close. By default, Vatalloc sets the threshold to 15, which permits any consolidation. We evaluate the performance of Vatalloc for different thresholds in Section VI.

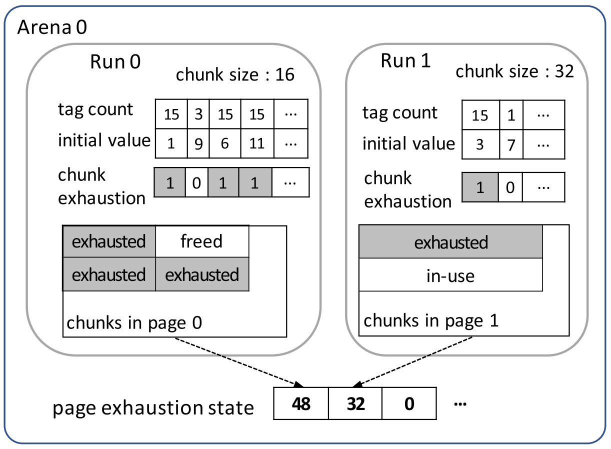


**FIGURE 4.** **Metadata of Vatalloc-j.**

### 2) ON JEMALLOC

For proper tag management on jemalloc, we changed the configurable tiniest chunk size of jemalloc from 8 to 16 bytes, as memory can only be tagged with 16-bytes granularity in MTE. In jemalloc, chunks are preallocated in a run, an allocation pool, with segregated by their size, and are not coalesced unlike dlmalloc. Therefore, as illustrated in Figure 4, Vatalloc manages tag counts and initial vales by using a per-run tag array whose elements correspond to the tag counts and randomized initial value of each chunk.

## B. EXHAUSTED CHUNKS

As chunks are reused repeatedly, their tag counts gradually increase as in subsection V-A. When their tag counts reach the maximum value (i.e., 15) and cannot be incremented on deallocation anymore, Vatalloc label them as exhausted and excludes from the allocation pools not to be considered in a future reallocation. This exclusion is essential for Vatalloc to maintain its UAF attack prevention capability by preventing multiple objects from having the same VA and tag number. However, it may increase memory overhead because the exhausted chunks will keep occupying physical page frames. Vatalloc relieves this issue by unmapping the pages where exhausted chunks are residing, which will be explained in subsection V-C.

### 1) ON DLMALLOC

To mark the chunk's exhaustion state, Vatalloc-d modifies chunk's header again similar to the case of the tag number.

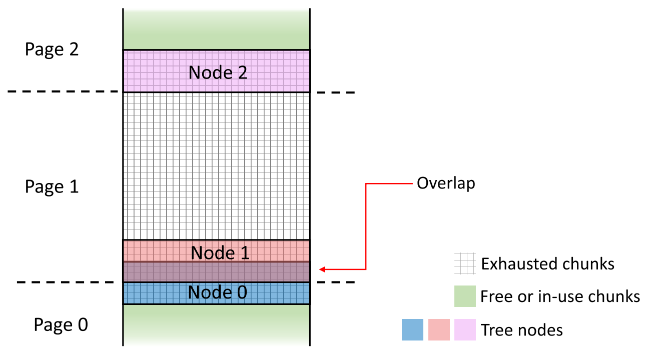


**FIGURE 5.** A Problematic Placement of Page Exhaustion State Tree Nodes on Vatalloc-d.

However, note that, as described in Figure 3, Vatalloc-d does not mark the exhaustion state of each chunk in its own header. Instead, a chunk's exhaustion state is indicated in the header of the previous and next chunks. This is to avoid a segmentation fault that may occur during a consolidation process. For example, when `chunkA`'s object is freed, dlmalloc needs to visit `chunkA`'s adjacent chunks and identify their states to make a consolidation. However, if any adjacent chunk was exhausted and the page to which this chunk belongs has been unmapped, a segmentation fault will be caused when visiting it. Therefore, Vatalloc preemptively fends off this problem by checking whether or not adjacent chunks have been exhausted before visiting them. To do this, Vatalloc creates and refers to a pair of state bits in the header: `prev_exhausted` and `next_exhausted`. When a chunk is exhausted, Vatalloc sets these state bits from the previous/next chunks.

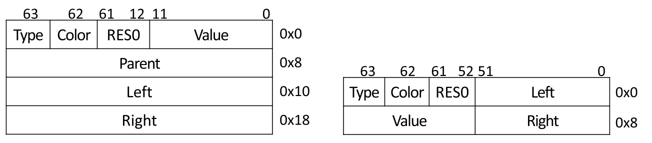
#### 2) ON JEMALLOC

As in the tag numbers, Vatalloc-j manages chunk's exhaustion state using an array within a run, which is illustrated in Figure 4. Since the per-run array is completely separated from chunks, each element of the array directly presents the exhaustion state of the associated chunk without worrying about a segmentation fault from accessing unmapped chunks unlike the case of dlmalloc. In jemalloc, freed chunks are cached for faster reallocation in the future. Therefore, Vatalloc modifies the jemalloc not to cache exhausted chunks.

### *C. PAGE-LEVEL EXHAUSTION TRACKING*

Vatalloc needs to release page frames consisting of exhausted chunks for efficient memory use. To do this, Vatalloc unmaps the pages where exhausted chunks are residing. However, we should note that those exhausted chunks are likely to be mixed with non-exhausted chunks within the same pages. Therefore, careless unmapping of the pages may result in a crash. To prevent this problem, Vatalloc defines a new type of page, called an *exhausted page*, that is full of only exhausted chunks. Vatalloc can unmaps only this type of pages safely, as they do not contain non-exhausted chunks.

The challenging issue here is that it is not feasible to artificially group scattered exhausted chunks into exhausted



**(a) Structure of Full node (32 B)** **(b) Structure of Half Node (16 B)**

**FIGURE 6.** Two Types of Nodes of Page Exhaustion State Tree on Vatalloc-d.

pages since it will break the consistency of VA between chunks and pointer. When a chunk becomes an exhausted one, Vatalloc checks whether the page containing the chunk turns to be exhausted ones, and if so, instantly unmaps it to release the occupied physical page frame. To track down the generation of exhausted pages, Vatalloc maintains metadata, called *page exhaustion state*. Vatalloc keeps recording the accumulated size of exhausted chunks by page in this metadata. Using this metadata, Vatalloc can identify exhausted pages by finding the ones whose accumulated size of exhausted chunks are equal to the page size (i.e., 4096 with a 4 KB-paging scheme).

In the description so far, we assumed that Vatalloc performs page exhaustion tracking by each page. This default granularity allows Vatalloc to decrease memory overheads the most by instantly unmapping exhausted pages as soon as generated. However, a little performance degradation will be caused due to frequent page management (i.e., calling the `munmap` system call). Alternatively, we can increase the granularity by performing page exhaustion tracking by a group of pages. In this modified granularity, we may expect Vatalloc to consume more memory on exhausted pages due to a delayed unmapping. Instead, the performance will be improved by decreasing the frequency of page management. We will discuss this trade-off of the tacking granularity in Section VI.

#### 1) ON DLMALLOC

As stated earlier in this section, Vatalloc based on dlmalloc seeks to minimize the memory overhead for metadata. For this purpose, we maintain page exhaustion states as a binary search tree because each tree node can be stored elaborately in exhausted chunks without a separate allocation, as will be explained below. In the tree, each node corresponds to a page: `key` is a virtual page number (i.e., VA[48:12]) and *value* is the accumulated size of exhausted chunks. We should note that since the heap grows in one direction, the binary search tree will be right skewed with new nodes possibly having incremental keys, which will be slowed down to linear complexity search time. Vatalloc avoids this problem by using a red-black tree. As described in Figure 6.(a) the node of the tree consists of `color`, `parent`/`left`/`right` pointers, and `value`. `key` needs not be stored in the node because it's value (i.e., page number) can be easily computed with a shift operation from the address of the node.

As stated ealier, Vatalloc creates tree nodes in exhausted chunks without a separate memory allocation. However, exhausted chunks for tree nodes are selected carefully, given they are eventually unmapped along with the exhausted pages to which they are belong. Therefore, Vatalloc basically creates a node of each page at the beginning bytes of the first exhausted chunk within the page. If the exhausted chunk is larger than a page, Vatalloc simply creates multiple nodes by page boundary. For example, in Figure 5, Vatalloc creates three nodes inside exhausted chunks for page 0, 1, and 2. In most cases, a node whose encoding is depicted in Figure 6.(a) can be placed in each chunk whose minimum size is 32-byte[1] in dlmalloc. However, we should consider a corner case described in Figure 5 as well. Since in dlmalloc chunks are 16-byte aligned, chunks (even the smallest) can span the page boundaries. The problem is that a node of Figure 6.(a) is larger than the alignment unit of a chunk. Therefore, in Figure 5 the node of the page 0 is expected to overflow to the page 1. This causes two problems. First, the node of the page 1 will be corrupted by the node of the page 0. Second, if the page 1 is unmapped, both nodes of page 0 will be damaged.

To avoid these problems, Vatalloc devises an auxiliary encoding described in Figure 6.(b). With this encoding, the size of nodes decreases to 16-byte that is identical to the alignment unit of a chunk. We call these size-reduced nodes *half nodes*, and original nodes *full nodes*. These two types of nodes are distinguished by the `type` bit. In a half node, we set the length of the `right` and `left` to 52 bits, which is enough considering that (1) the size of the entire VA space is usually 48 bits with 4-level paging, and (2) chunks are well aligned at 16-byte boundaries (i.e., we can cut off the bottom 4-bit in `right` and `left`). Furthermore, the size of the `value` is 12 bits, enough to cover larger granularities of the page-level exhaustion tracking considering that chunks are allocated in units of 16-byte (i.e., we can cut off the bottom 4-bit in `value`). Lastly, we omit the `parent` in a half node due to the limited size. It makes sense because unlike the `right` and `left`, the `parent` can be re obtained anytime by traversing from the root node of the tree. However, this would slow down the tree management. Vatalloc alleviates this problem by gradually converting half nodes to full nodes. It is obvious that, in the page where a half node already exists, another exhausted chunk that is large enough to store a full node would be generated soon. In this case, Vatalloc migrates the half node to the new exhausted chunk, while converting it to a full node.

#### 2) ON JEMALLOC

Jemalloc organizes allocation pools for chunks with memory blocks of 2 MB. To keep track of the page exhaustion state, Vatalloc-j creates an array indicating page exhaustion state by each memory block as illustrated in Figure 4. For example, if the default granularity (i.e., a page size) is used in tracking, each element of the array corresponds to a page and stores the accumulated size of exhausted chunks on the page. Once the value of the array element reaches the granularity so that the associated page is exhausted, Vatalloc unmaps the corresponding memory.

[1]header (16-byte) + body (16-byte)

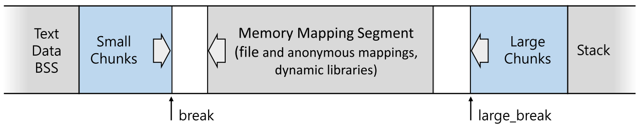


**FIGURE 7. The memory layout of a Vatalloc-enabled program.**

### D. LARGE CHUNKS

As stated inSection II, dlmalloc and jemalloc handle large-sized chunks in their own way. For example, dlmalloc does not reuse large chunks. On the other hand, jemalloc aggressively reuses even large chunks. Accordingly, Vatalloc applies the VA tagging scheme differently to each allocator.

#### 1) ON DLMALLOC

dlmalloc is designed to acquire memory for large chunks from the kernel on the fly, instantly unmapping deallocated large chunks. To abide by it, Vatalloc does not reuse VAs of large chunks, but simply places large chunks at different VAs. Since dlmalloc rely on `mmap` for large chunk allocations, Vatalloc can implement it by making `mmap` allocate large chunks in a way of monotonously increasing VA. More specifically, Vatalloc (1) defines a `break_large`, which corresponds to the `break` of the heap that is managed by `sbrk`, (2) gives it to `mmap` along with the `MAP_FIXED_NOREPLACE` flag to allocate a new large chunk in the VA of the `break_large` value, and (3) adds the just allocated size to the `break_large` value for moved allocation of next large chunks. However, in this implementation, a large chunk allocation can sometimes fail because the `break_large` value may collide with the existing memory allocations, such as for libraries, on different parts of the program other than Vatalloc. In this case, dlmalloc resorts to a trial-and-error method that repeatedly adjusts the `break_large` value until `mmap` succeeds to allocate a large chunk. Admittedly, this method may cause a poor worst-case overhead. To alleviate this problem, Vatalloc introduces a memory layout that is inspired from the conventional disposal of the heap and stack to minimize the likelihood of a collision. As illustrated in Figure 7, the address spaces of the small chunks and the large chunks grow in the opposite direction. To sum up, the address space of the large chunks is managed by `mmap` with the ever-decreasing `break_large` and that of the small chunks by `sbrk` with the ever-increasing `break`.

#### 2) ON JEMALLOC

Jemalloc organizes separate data structures and algorithms for reusing large chunks. Therefore, Vatalloc-j applies the VA tagging scheme so that the large chunks are reused unlike the case in dlmalloc. Exactly speaking, as stated in subsection II-C, jemalloc classifies non-small chunks again into large and huge chunks. Firstly, for large chunks, as jemalloc manages them no differently from the small ones, so that Vatalloc applies the VA tagging scheme in the way described in the previous subsections. On the other hand, jemalloc organizes single tree data structure for each huge chunks, since huge allocations rarely happen. Vatalloc carries out tag operation directly accessing internal tree structure, and unmaps huge chunk on deallocation if it is exhausted.

### *E. MULTITHREADING SUPPORT*

As Vatalloc complies with the original design direction of the base memory allocators, it can provide multithreading support equivalent to them.

#### 1) ON DLMALLOC

dlmalloc is originally designed to suit single thread programs, and its metadata are shared among threads. Therefore, dlmalloc features simplistic multithreading support by maintaining a global mutex lock for protecting its metadata. Similarly, Vatalloc-d can support multithreading by protecting its metadata using a global mutex lock. Since metadata such as tag numbers and exhaustion states of chunks stored in chunk headers are naturally protected by the existing lock of dlmalloc, only one lock is additionally needed to protect the page exhaustion state.

#### 2) ON JEMALLOC

jemalloc has a specialized design for efficient multithreading support that minimizes the need of a global mutex lock by maintaining per-thread metadata. Likewise, Vatalloc places all the metadata alongside the existing data structure of jemalloc, and thus providing efficient multithreading support without any addition of a global mutex lock.

### *F. OPERATION MODES AND DETECTION*

MTE provides two operation modes: precise and imprecise, as described in subsection II-A. Depending on the currently activated MTE mode, Vatalloc can prevent Use-After-Free (UAF) attacks either immediately (in the precise mode) or lazily (in the imprecise mode). As explained in subsection II-A, in precise mode, the tag mismatch is synchronously notified with the faulting address, which allows for the exact determination of the load or store instruction that caused the tag mismatch. When a tag mismatch occurs, the kernel support MTE to generate a `SIGSEGV` signal. To be specific, the signal is raised instantly if the precise mode is enabled. The signal is also accompanied by a code, `SEGV_MTESERR` or `SEGV_MTEAERR`, to specify the current MTE mode. Therefore, Vatalloc can write a custom `SIGSEGV` handler to detect UAF attacks by catching the signal and code. This feature allows Vatalloc to instantly detect UAF violations before any dangling pointer is exploited by attackers. Thanks to this MTE tag matching, Vatalloc has a relative advantage over some other Virtual Address (VA)-based techniques, such as FFmalloc, in that it has the capability for instant detection, which they lack. On the other hand, in the situation where high performance is required, imprecise mode can be chosen. In this case, signal of violation is delayed and raised at the upcoming entry to the kernel. In the event of a violation, the signal of the violation is postponed and raised upon the next entry into the kernel. However, it is unlikely that a violation would lead to a serious compromise of the system, as it is detected before it enters the kernel.

### *G. AGAINST VA EXHAUSTION*

Despite Vatalloc's efficient use of VAs and tags, an application that never finishes may end up exhausting all the VA space. For that case, like conventional GC, procedure of `marking` the pair of VA and tag which is still referenced, and `sweeping` the pair which is unmarked for safe reuse can be considered. DangZero has shown this reclaiming routine effective with the modified kernel. Like Oscar, DangZero requires at least one page frame per object. Since Vatalloc does not have this constraint and the same VA can be reused multiple times, the frequency of reclaiming and the resulting performance degradation is expected to be significantly lower.

## VI. EVALUATION

In this section, we demonstrate the efficiency and effectiveness of Vatalloc. We first evaluate two versions of Vatalloc implemented on dlmalloc and jemalloc in terms of performance and memory through a comparison with previous techniques. We also show how the design configurations declared in Section V affect Vatalloc. Lastly, we examine the effectiveness of Vatalloc in UAF attack detection. In the following evaluation, we evaluate performance and memory overhead of Vatalloc operating in precise and imprecise mode using emulation.

### *A. EXPERIMENTAL SETUP*

To confirm the functional correctness, the implementation has been carried out on ARM Fixed Virtual Platform v11.11.34. However, as it is not cycle accurate, the implementation on it cannot be used for performance measurement. To our knowledge at the time of this research, there were no publicly available processor supporting MTE. For example, according to our investigation, even the Apple M2, which is designed with up-to-date ARM architecture did not implement MTE. Therefore, we instead conducted proxy measurements on development board, ODROID-HC4 [31] with 1.8Ghz Cortex-A55 quad core processor and 4GB RAM, by shadow-mapping the tag memory. For DangZero, we referred to the numbers that have been reported in their original papers as their prototype only supports Intel architecture.

#### 1) MODIFIED CODE SIZE

Vatalloc-d was built on dlmalloc-2.7 by adding 0.9K lines of source code, and Vatalloc-j was implemented by adding 0.2K lines to the source code upon jemalloc-4.5. Vatalloc can be even implemented in FFmalloc for extension, to further reduce the exhaustion of va space.

### *B. PROXY MEASUREMENT*

To measure an accurate estimation of worst case performance of MTE, both tag update, which is composed of loading and storing tags, and tag matching have to be considered.

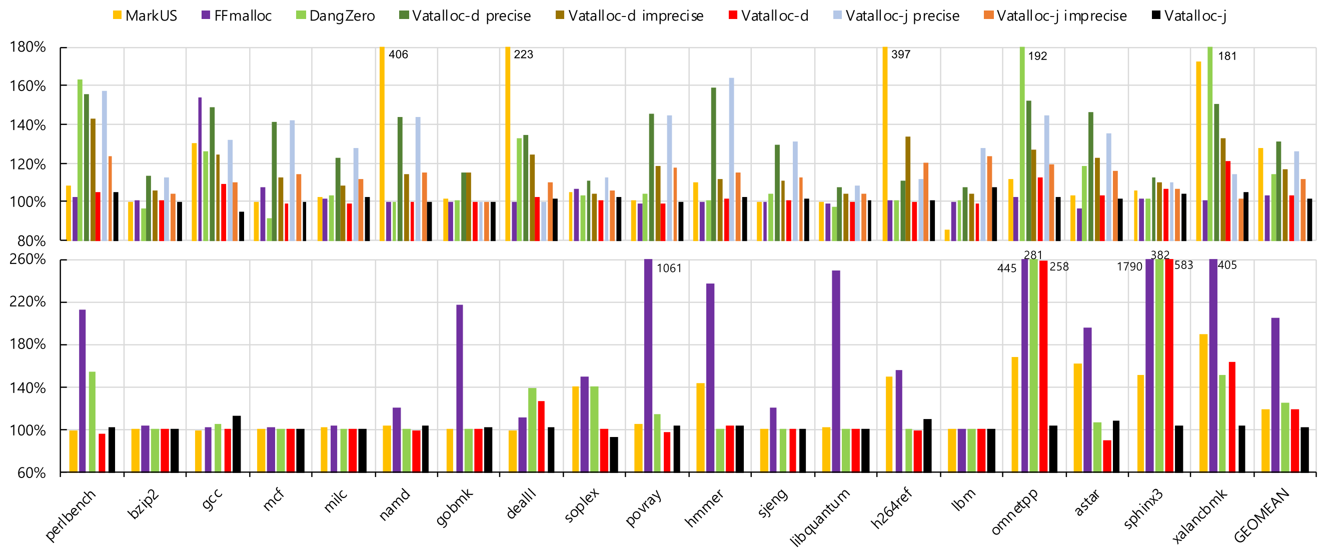


**FIGURE 8.** Performance and Memory Overhead on SPEC CPU2006.

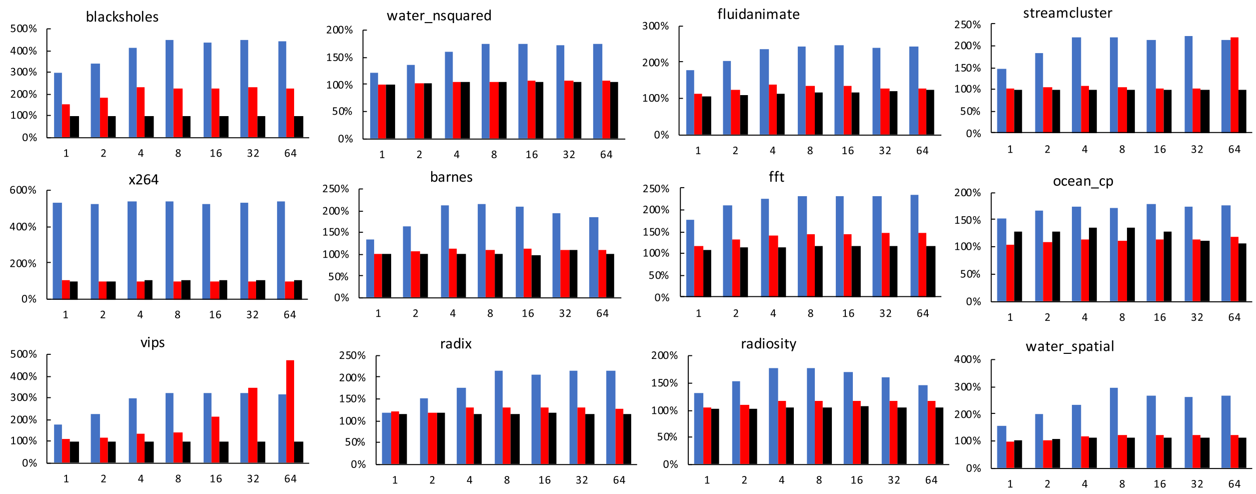


**FIGURE 9.** Performance Overhead on PARSEC. The blue bar refers to FFMalloc, the red bar refers to Vatalloc-d, and the black bar refers to Vatalloc-j.

While recent researches using MTE [32], [33] overlooked the second cost, but from what we have observed, it adds significant overhead to the system. First, we simulated the update of memory tags by reserving a large memory region to store tags and executing memory instructions, which write a single byte to the reserved memory on every load or store operation. The tag comparison overhead is assumed to be mostly hidden since it is performed by a separate MTE hardware logic which runs concurrently with the CPU cores [19]. Meanwhile, tag loading has a potential performance impact on systems in the precise trapping mode. By loading additional tag bits, more memory pressure and cache overhead are induced. Additionally, ARM's weak memory model will incur higher costs to ensure that all the memory operations are observed after the tag is loaded. We pessimistically approximate this effect of memory dependency by instrumenting target programs in two-fold. Firstly, in the *instrumentation1*, before every memory access, loading a corresponding tag is performed using implicit load-acquire barrier as seen in Figure 11.(b). However, loading a tag from shadow memory to the cache and then to the CPU requires computations of the tag address, which will be transparently performed by the MTE unit. Also, extra burden to the memory unit is incurred because MTE loads tags from memory to caches but not from caches to CPU. To compensate for this, in the *instrumentation2*, dummy instructions that calculate the tag address and load the tag are inserted before every memory access (Figure 11.(c)). These instructions use constant addresses to ensure that cache miss never occurs. By subtracting the runtime results of

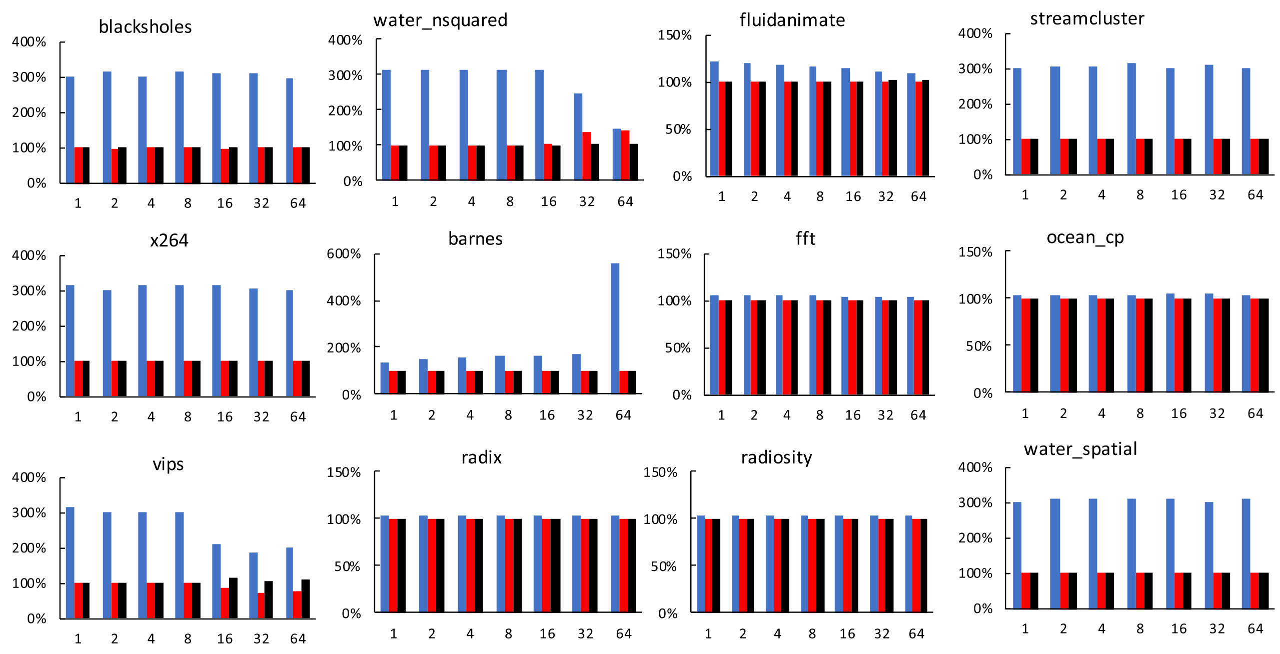


**FIGURE 10.** Memory Overhead on PARSEC. The blue bar refers to FFMalloc, the red bar refers to Vatalloc-d, and the black bar refers to Vatalloc-j.

```
ADD     X1 , X1 , X2 , LSR #5
        // X0: TAG
        // X1: Tag memory base
        // X2: Target address
STR     X0 , [X1 ]
```

(a) Tag Store

```
MOV     X15 , #TAG_OFFSET
ADD     X15 , X15 , X1 , LSR #5
LDARB   XZR, [X15]  // barrier
LDR     X0 , [X1]
```

(b) Tag Load (instrumentation1)

```
MOV     X15 , #TAG_OFFSET
ADD     X15 , X15 , 0
LDR     XZR, [X15]
LDR     X0 , [X1]
```

(c) Tag Load (instrumentation2)

```
MOV     X15 , #TAG_OFFSET
ADD     X15 , X15 , X1 , LSR #5
LDR     XZR, [X15]
LDR     X0 , [X1]
```

(d) Tag Load (instrumentation3)

**FIGURE 11.** Instrumentation for Proxy Measurement.

instrumentation1 from those of instrumentation2, we obtain the tag loading overhead induced by fetching the memory tag from memory into the cache and the runtime overhead caused by the stricter memory ordering restraint. Likewise, to evaluate performance of the imprecise mode, we instrumented target programs (*instrumentation3*) and subtract the performance results of *instrumentation2* from those of *instrumentation3*. Note here that in *instrumentation2*, a load instruction without any forward dependency is inserted as in Figure 11.(d). As tag checking operation is performed in MTE asynchronously,

## C. PERFORMANCE OVERHEAD

Figure 8 reports the performance numbers of Vatalloc and other techniques that are measured on SPEC2006 single-threaded benchmarks. We used *-O2* as a default optimization flag, but we used *-O1* for *perlbench* and *dealII*, and *-O0* for *gcc*, *omnetpp* and *namd* as these benchmarks crashed with instrumentation. For convenience, hereafter we refer to measurement of Vatalloc postulating precise mode as *Vatalloc precise*, and imprecise mode as *Vatalloc imprecise* respectively. In some cases, such as `gobmk`, `sjeng`, and `lbm`, uninstrumented Vatalloc is faster than the native execution; we deem that this is due to the unintended effect from the changes in the chunk layout. For the instrumented versions to measure MTE's negative potential impact on the system, Vatalloc imprecise adds up 16.9 % runtime overhead for Vatalloc-d and 12.0 % for Vatalloc-j, which provides nearly similar performance to 18.9 % slowdown of MarkUs and 14 % slowdown of DangZero without page reclaimer. Due to more stringent constraints to measure the effect of precise MTE mode, Vatalloc precise incurs larger runtime overhead (30.9 % for Vatalloc-d and 25.5 % for Vatalloc-j). *hmmer* resulted in the worst number for Vatalloc precise, but the number is significantly dropped in Vatalloc imprecise. The worst slowdown in Vatalloc imprecise is observed in *perlbench*.

Vatalloc-d (w/o MTE) and Vatalloc-j (w/o MTE) refer to uninstrumented versions of Vatalloc-d and Vatalloc-j, where performance degradation of MTE is not considered. Vatalloc-d and Vatalloc-j shows 1.70 % and 3.05 % geomean performance overhead respectively, which means without adverse effect of MTE, our mechanism induces merely negiligible overhead. The measurement result indicates that in systems with MTE for enhanced spatial safety, Vatalloc offers UAF prevention with a slight overhead.

We also tested 12 multi-threaded PARSEC-3.0 workloads to investigate multithreading performance, and compared the results with those presented in DangSan. We measured uninstrumented Vatalloc, Vatalloc precise and FFMalloc on

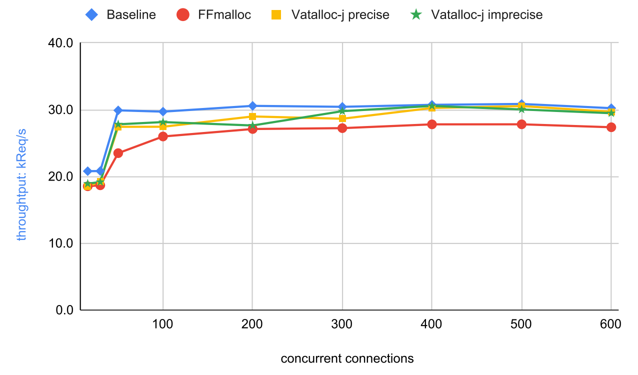


**FIGURE 12.** Nginx throughput.

our system. We can observe that Vatalloc-d precise incurs high overhead on average (geometric mean, 65.8 % at 64 threads). Uninstrumented Vatalloc-d and Vatalloc-j precise introduces 36.8% performance overhead. On the other hand, Vatalloc-j adds negligible overhead on average (8.5 % at 64 threads), which shows that the multithreaded performance of jemalloc is not degraded by Vatalloc. For Vatalloc-j precise, 36.7% slowdown occurs.

### D. MEMORY OVERHEAD

The memory overheads of Vatalloc mostly arises from metadata and exhausted but not unmapped chunks. Firstly, Vatalloc consumes different amount of memory for metadata. Vatalloc-d does not require additional memory because all its metadata are embedded in the existing metadata of live or exhausted chunks. On the other hand, Vatalloc-j spends some memory on storing metadata in the data structures of jemalloc. In the worst case, Vatalloc-j uses less than 130 bytes for every page, which roughly increases memory consumption by 3.0 %.

Next, exhausted chunks are retained until unmapped at page level when their total size accumulates to the page size. To measure the relevant overhead, we used the maximum resident set size reported in the processor status (i.e., `/proc/pid/status`). We first observed that Vatalloc adds up to 19.0 % of maximum resident memory on dlmalloc, and 3.0 % on jemalloc with respect to SPEC. As reported in Figure 8, we compared the peak memory usage of Vatalloc with other techniques. Meanwhile, MarkUs and DangZero add up 18.9 % and 25% respectively, and FFmalloc shows worst memory overhead of 104.1 %. In most cases, Vatalloc-d does not increase memory consumption, but there are exceptional benchmarks in `omnetpp` and `sphinx3`, which tend to allocate much larger memory than they actually use. Usually, such excessive allocations are not critical, thanks to demand paging technique, but they are problematic in Vatalloc because all the associated memory tags should be initialized at each allocation. Fortunately, this problem will be addressed because Linux recently added to `mmap` a `PROT_MTE` flag that initializes memory tags with a designated tag number when pages are actually mapped to physical memory frames through

**TABLE 1.** Virtual Address Consumption over Single Benchmark Run.

| benchmarks | Oscar/DangZero | FFmalloc | Vatalloc-d | Vatalloc-j |
|---|---|---|---|---|
| gcc | 0.4 TB | 82.7 GB | 1.1 GB | 1.5 GB |
| perlbench | 1.4 TB | 6.1 GB | 3.9 GB | 1.2 GB |
| dealII | 0.9 TB | 12.0 GB | 8.0 GB | 3.8 GB |
| omnetpp | 0.9 TB | 47.5 GB | 5.4 GB | 4.1 GB |
| xalancbmk | 0.9 TB | 66.8 GB | 7.4 GB | 3.8 GB |

demand paging. In the case of Vatalloc-j, we can see near-zero memory overheads in most benchmarks. This is because due to size-segregated chunks in jemalloc, exhausted chunks do not increase memory fragmentation. Also, jemalloc has an allocation strategy that allocates more chunks than necessary in advance, which offsets the memory occupation of exhausted chunks.

In PARSEC, Vatalloc also shows a high memory efficiency on the basis of the maximum resident set size. At 64 threads, both Vatalloc-d and Vatalloc-j incur 1.8 % memory overheads on average, which are better than 104.2% of FFmalloc. In `vips`, Vatalloc-d has rather less memory overhead than the native execution, which is because Vatalloc-d unmaps unused (i.e., exhausted) pages more aggressively than dlmalloc used in the native execution.

### E. NGINX

In order to evaluate Vatalloc on more realistic applications, we measured the performance of Vatalloc-j using Nginx web server as in Figure 12. We used `wrk` HTTP benchmarking tool of Nginx version 1.23.3. ODROID-HC4 functions as a server, while a machine with i9-10900K CPU and 128GB RAM runs as a client. We configured the `wrk` benchmark to execute 30 seconds per run, sending a 64-byte file. The average degradation factor over the baseline is 3.9% and 4.3% for imprecise mode and precise mode of Vatalloc-j respectively, and is 11.4% for FFmalloc.

### F. VIRTUAL ADDRESS CONSUMPTION

The detection capability of the VA-based scheme in the lock-and-key approach is retained by default until VAs are exhausted. The fact that each address can be reused multiple times thanks to our VA tagging scheme is a strength in this respect. To prove this we experimented how much VA is consumed after a single program run in Vatalloc, Oscar/DangZero, and FFmalloc, and the results are shown in Table 1. Consequently, we were able to observe that the VA tagging scheme of Vatalloc that facilitates VA reuses is extraordinarily helpful in reducing the amount of consumption. Address consumption of Vatalloc-j is about 1200 times lower than Oscar/DangZero for `perlbench`, which implies it would be lower than DangZero without GC-like page reclaimer by the same magnitude. As anticipated, Vatalloc consumes about 16 times lower va space than FFmalloc for `omnetpp` and `xalancbmk`. It shows that Vatalloc can support even long-running programs without failure in UAF

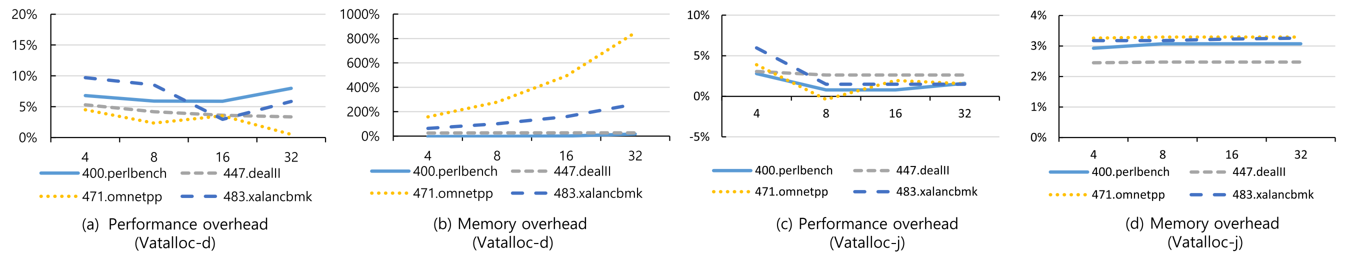


**FIGURE 13.** Variation in Performance and Memory Overheads by the Granularity of the Page-level Exhaustion Tracking.

attack detection. When combined with the page reclaimer shown by DangZero, performance improvement is expected as the number of reclaimation would be drastically reduced.

### G. GRANULARITY OF PAGE-LEVEL EXHAUSTION TRACKING

As described in subsection V-C, Vatalloc allows to change the granularity of page-level exhaustion tracking, which is 4 KB by default. To observe the effect of granularity change (from 4 KB to 32 KB), we conducted experiments on Vatalloc-d and Vatalloc-j with allocation-intensive benchmarks, `perlbench`, `dealII`, `omnetpp`, and `xalancbmk`, whose performance is worse relatively. As shown in Figure 13, results revealed a rough trend, where memory consumption grows proportionally with granularity, but execution time decreased to a certain point. For memory usage, the trend is clearly found in `omnetpp`, `perlbench`, and `xalancbmk` that perform complex memory allocation/deallocation patterns, because exhausted pages can be unmapped only when they are concatenated as much as the granularity. On the other hand, for execution time, the trend is not steady. With a bit larger granularity, the frequency of `unmap` invocations and page exhaustion state management operations (e.g., search/insertion/deletion) is decreased, which leads to performance improvement. However, extremely-large granularity makes unmapping of pages hard, prolonging the life time of the exhausted chunks. As a result, sparsely located exhausted chunks increases access overhead, reversing the performance trend in `perlbench` and `xalancbmk`.

### H. CONSOLIDATION THRESHOLD

As stated in subsection V-A, we can control the threshold of tag number difference in consolidation on Vatalloc-d. The default threshold is 15 (i.e., the maximum tag number difference), which means that Vatalloc allows any consolidation. Similar to subsection VI-G, we observed how the change in the threshold affects both performance and memory in allocation-intensive benchmarks. Figure 14 reports the results. Lowering the threshold has two opposite effects; (negative) it may cause external fragmentation by restraining consolidations between small chunks, but (positive) it gives more chances to reuse the same VA by suppressing consolidation (i.e., tag number of the chunk with the smaller tag number is wasted via consolidation). Overall, we get better performance numbers with a lower threshold in most benchmarks. It indicates that as these

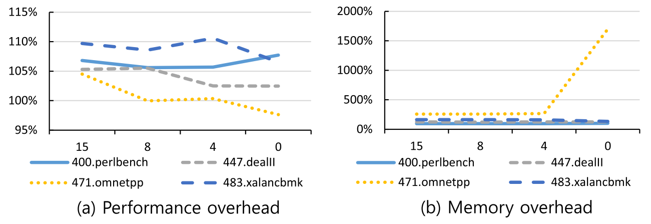


**FIGURE 14.** Variation in Performance and Memory Overheads of Vatalloc-d according to the Consolidation Threshold.

benchmarks tend to make aligned and size-uniform memory allocations, they are insensitive to the external fragmentation issue, and thus the aforementioned positive effect has greater influence on performance than the negative one. When it comes to memory overhead, most benchmarks just show steady numbers. However, `omnetpp` shows exceptionally increasing overhead, which is due to the negative effect with a too-low threshold.

### I. EFFECTIVENESS AND COMPATIBILITY

We assume that metadata of Vatalloc and the control flow of a program are unharmed, and tag memory of MTE is not manipulated by an attacker. In case of Vatalloc-d, an attacker may maliciously point to its metadata to manipulate it. However due to the disparity of pointer tag and the tag number of the chunk, the attempt is neutralized. To evaluate effectiveness in preventing UAF errors, we manually tested Vatalloc with latest UAF vulnerabilities reported, which are CVE-2022-34568, CVE-2022-40674, CVE-2022-3352, CVE-2022-30065 and CVE-2022-36149 [34], [35], [36], [37], [38]. Initially, each exploit successfully executes a malicious action, such as taking over arbitrary code execution, the instruction pointer, or writing arbitrary data to memory. We observed that Vatalloc successfully prevents compromises by changing the tag numbers of the target objects upon deallocation, inducing a tag mismatch, thereby nullifying dereferences to them. One of MTE's original purposes is to enforce spatial memory safety probabilistically. In a typical method for this purpose, a pointer and its referent object are given the same but randomly selected tag number. By doing so, malicious attempts of dereferencing a pointer to access outside its referent object are detected probably due to a tag mismatch. As Vatalloc assigns random pointer and referent pairs the same tag numbers, tag numbers are distributed uniformly. In this regard, the methodology remains consistent with Vatalloc and spatial safety is probabilistically enforced alongside Vatalloc.

## VII. DISCUSSION

### A. MTE FOR SPATIAL MEMORY SAFETY

Another major use of MTE is to enforce spatial memory safety. In a typical method for this purpose, a pointer and its referent object are given the same but randomly selected tag number. By doing so, malicious attempts of dereferencing a pointer to access outside its referent object are detected probably due to a tag mismatch. This typical method is totally compatible with Vatalloc. Since Vatalloc already assigns pointer and referent pairs the same tag numbers, the method can be achieved by Vatalloc randomizing chunks' tag numbers.

### B. PORTING TO OTHER MEMORY ALLOCATORS

The VA tagging scheme is generally applicable to many allocators. To demonstrate this, we have implemented Vatalloc based on two major memory allocators, jemalloc and dlmalloc, but it can be ported to other memory allocators as well. For example, we can consider porting Vatalloc to ptmalloc2, the default allocator in latest Linux versions. Unlike dlmalloc, ptmalloc2 does not share chunks among threads, but instead manages them independently in each thread by maintaining per-thread metadata. Despite this difference, as ptmalloc2 basically stems from dlmalloc, the tag management is applicable to ptmalloc2. Also, the page-level exhaustion tracking can be conducted at each thread by maintaining the page exhaustion state thread-by-thread. tcmalloc, introduced by Google is another recent memory allocators developed for more cache-conscious memory allocation and strengthened multithreading support. Implementation of the Vatalloc's tag management mechanism is expected to be well suited with tcmalloc, since tcmalloc makes use of thread-specific and size-segregated chunks like jemalloc. Additionally, the page exhaustion tracking mechanism of Vatalloc is applicable because tcmalloc manages internal allocation pools and caches in a similar manner to jemalloc.

## VIII. RELATED WORK

Two major approaches that have been studied for UAF attack detection are *pointer nullification* and *lock-and-key*. DangNull [2], FreeSentry [5] and Dangsan [5] follow the former approach. They nullify dangling pointers to detect UAF attacks when the pointers dereferenced. Similar to Vatalloc, the techniques provide deterministic detection. However, they incur excessive performance (e.g., 55 % in DangNull, 25 % in FreeSentry, and 41 % in DangSan) to the system in constantly maintaining their dedicated data structures that keep track of the referring relationships between objects and pointers. To reduce performance overhead, a deferred free scheme [6], [7] has been devised that intentionally delays the reuse of freed objects' memory, inspired by the fact that UAF attacks will be launched shortly after objects are freed. This scheme can be implemented easily by placing the freed objects in quarantine memory for a while. In fact, in spite of this scheme, AddressSanitizer [4] still incurs high performance degradation, because this technique monitors every memory access, aiming to detect not only UAF attacks but also out-of-bounds accesses with a debugging purpose. More importantly, due to the limited size of the quarantine memory where the freed objects are residing, this technique can detect UAF attacks only probabilistically. Therefore, to guarantee deterministic detection of UAF attacks with a low overhead, pSweeper [14], CRCount [11], MarkUs [10], CHERIvoke [9], and Cornucopia [8] have added optimization mechanisms to this deferred free scheme. For example, pSweeper runs pointer nullification in a separate thread. CRCount waits for dangling pointers to be nullified implicitly until the freed object's reference count is zero. MarkUs is similar to CRCount, but instead of reference counting, it runs pointer marking in a separate thread that scans memory to find remaining dangling pointers. Thanks to such optimization mechanisms, these techniques can achieve detection more efficiently, but it comes at a cost. Compared to Vatalloc, they can detect only one type of UAF attacks, and leave undetected the other type of UAF attacks against the freed objects whose memory region has not been reused yet. Moreover, their overheads are still somewhat large, and they necessitate source code to apply their optimization. The only techniques that provide comparable detection with Vatalloc are CHERIvoke and Cornucopia. However, as they ultimately take advantage of the pointer marking mechanism of MarkUs, they also provide only partial detection of UAF attacks. In addition, implementing them requires an architectural capability [39]. There is an effort to implement it in the real evaluation board [40], but it has not yet been integrated into commodity processors, unlike MTE.

Unlike the above-mentioned techniques, CETS [41] and Oscar [15] are based on the lock-and-key approach. They distribute locks and keys to objects and pointers, respectively, and check whether locks and keys match on each memory access. In the case of CETS, locks and keys are defined as 64-bit integers, which requires expensive data structures that manage them by object and by pointer. Even worse, the data structures should be referred to on every memory access, which explains the large performance overhead of this technique. To lessen the overhead and activate detection even without source code, Oscar apply the VA-based scheme that uses VAs as locks and keys, which is realized by ensuring each object to be allocated in unique VAs using virtual memory mapping. However the scheme spends many CPU cycles which are entailed by extremely frequent kernel intervention for memory management (i.e., invoking system calls such as `mmap`, `sbrk` and `munmap`) in order to assign unique VAs to every newly created object as well as to invalidate the VAs of freed objects. Recent work in this category, a VA-based technique, called *FFmalloc* [16], has attained a reduction of runtime overhead by resolving the challenge through batched invalidation of keys, but it achieves this result only at a cost; that is, loss of its detection capability. With FFmalloc, freed objects are still accessible through dangling pointers without being detected until they are finally relinquished to the kernel when freed memory reaches a certain number of consecutive pages. As the latest work in this category, DangZero [17] solves this issue by granting the program's allocator an authority to access page table directly for management and invalidation of keys. Direct page table access is normally performed in *ring 0*, which is the highest privilege running kernels. To do so, kernel modification is required to run user-space applications in *ring 0*, which harms the applicability of the scheme. Vatalloc show quite improvement in performance and memory overheads over Oscar, thanks to the VA tagging scheme of Vatalloc which enables it to reuse VAs several times by capitalizing on MTE. FFmalloc has improved performance compared to oscar, but at the cost of not being capable of deterministic detection. In case of Vattaloc, depending on the desired mode, checks for tag mismatches generate a segmentation fault on safety violation. Vatalloc also shows better security and is more applicable to long-running programs than FFmalloc, because it can significantly delay VAs from being exhausted by supressing consumption of it.

## IX. CONCLUSION

We proposed a technique, Vatalloc, that provides efficient, drop-in-use, and instant detection against dangling pointers. Such an effective and practical detection capability comes from our novel VA tagging scheme that advances the VA-based lock-and-key scheme, by capitalizing on MTE. The VA tagging scheme dramatically reduced the frequency of TLB misses and kernel-involved page management that mainly impairs the performance of the original VA-based scheme. We realized our scheme by implementing Vatalloc on dlmalloc and jemalloc, and demonstrated its effectiveness.

## REFERENCES

[1] (2020). *CWE Top 25 Most Dangerous Softw. Weaknesses*. [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

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

[3] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable use-after-free detection," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 405–419.

[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 309–318.

[5] Y. Younan, "FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[6] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Oct. 2010, pp. 573–584.

[7] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "FreeGuard: A faster secure heap allocator," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2389–2403.

[8] N. Wesley Filardo et al., "Cornucopia: Temporal safety for CHERI heaps," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 608–625.

[9] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 545–557.

[10] S. Ainsworth and T. M. Jones, "MarkUs: Drop-in use-after-free prevention for low-level languages," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 578–591.

[11] 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 *Proc. NDSS*, 2019.

[12] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *Proc. Int. Symp. Memory Manage.*, Jun. 2010, pp. 31–40.

[13] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2006, pp. 269–280.

[14] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1635–1648.

[15] T. H. Y. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 815–832.

[16] "Preventing use-after-free attacks with fast forward allocation," Aug. 2021.

[17] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "DangZero: Efficient use-after-free detection via direct page table access," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, Nov. 2022, pp. 1307–1322.

[18] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, May 2022, pp. 46–57.

[19] ARM Limited, "Armv8.5-A memory tagging extension," White Paper, 2021.

[20] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Oct. 2008, pp. 72–81.

[22] E. A. Feustel, "On the advantages of tagged architecture," *IEEE Trans. Comput.*, vol. C-22, no. 7, pp. 644–656, Jul. 1973.

[23] *Arm Architecture Reference Manual for A-Profile Architecture*, ARM Ltd., 2023.

[24] *Memory Tagging Extension User-Space Support*, 2020. [Online]. Available: https://lore.kernel.org/linux-arm-kernel/20200703153718.16973-1-catalin.marinas@arm.com

[25] D. Lea. (1996). *A Memory Allocator Called Doug Lea's Malloc or dlmalloc for Short*. Accessed: Mar. 26, 2010. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[26] W. Gloger. (2006). *Ptmalloc*. Consulté Sur. [Online]. Available: http://www.malloc.de/en

[27] J. Evans, "Scalable memory allocation using Jemalloc," Eng. Meta., Menlo Park, CA, USA.

[28] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for c," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2009, pp. 245–258.

[29] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. USENIX Secur. Symp.*, 2009, pp. 51–66.

[30] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical type confusion detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 517–528.

[31] *ODROID-HC4*. Accessed: Jul. 2021. [Online]. Available: https://www.hardkernel.com/ko/shop/odroid-hc4

[32] C. Ortega, H. Shrobe, M. Payer, H. Okhravi, N. Burow, D. McKee, and Y. Giannaris, "Preventing Kernel Hacks with HAKCs," in *Proc. NDSS*, 2022, pp. 1–17.

[33] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "MTSan: A feasible and practical memory sanitizer for fuzzing cots binaries," in *Proc. 32nd USENIX Conf. Secur. Symp. (SEC)*. USA: USENIX Association, 2023.

[34] NVD. (Jul. 28, 2022). *CVE-2022-34568*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-34568

[35] NVD. (Sep. 14, 2022). *CVE-2022-40674*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-40674

[36] NVD. (Sep. 29, 2022). *CVE-2022-3352*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-3352

[37] NVD. (May 18, 2022). *CVE-2022-30065*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-30065

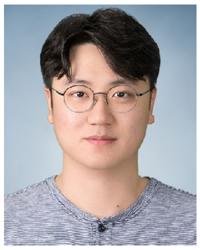
[38] NVD. (May 18, 2022). *CVE-2022-36149*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-36149

[39] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 20–37.
[40] (2020). *Arm Morello Program*. [Online]. Available: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello
[41] (2014). *SoftBoundCETS for LLVM+Clang Version 34*. Accessed: Apr. 21, 2020. [Online]. Available: https://github.com/santoshn/softboundcets-34

**INYOUNG BANG** received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2017, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes system security against various types of threats.

**MARTIN KAYONDO** received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2020, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes system security against various types of threats.

**JUNSEUNG YOU** received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2019, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes system security against various types of threats.



**DONGHYUN KWON** received the B.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2012 and 2019, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His research interest includes the system security against various types of threats.

**YEONGPIL CHO** received the B.S. degree in electrical engineering from POSTECH, South Korea, in 2010, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2018. Currently, he is a Professor with the Department of Computer Science, Hanyang University. His research interest includes the system security against various types of threats.

**YUNHEUNG PAEK** (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, in 1997. Currently, he is a Professor with the Department of Electrical and Computer Engineering, Seoul National University. His research interests include system security with hardware, the secure processor design against various types of threats, and machine learning-based security solution.