# Efficient Use-After-Free Prevention with Opportunistic Page-Level Sweeping

Chanyoung Park
UNIST
chanyoung@unist.ac.kr

Hyungon Moon*
UNIST
hyungon@unist.ac.kr

***Abstract*—Defeating use-after-free exploits presents a challenging problem, one for which a universal solution remains elusive. Recent efforts towards efficient prevention of use-after-free exploits have found that delaying the reuse of freed memory can both be effective and efficient in many cases. Previous studies have proposed two primary approaches: one where reuse is postponed until the allocator can confidently ascertain the absence of any dangling pointers to the freed memory, and another that refrains from reusing a freed heap chunk until the program's termination. We make an intriguing observation from our in-depth analysis of these two approaches and their reported performance impacts. When compared to the design that delays the reuse until the program terminates the strategy that delays the reuse just until no dangling pointer references the freed chunk suffers from a significant performance overhead for some workloads. The change in the reuse of each heap chunk affects the distribution of allocated chunks in the heap, and the performance of some benchmarks. This study proposes HUSHVAC, an allocator that performs delayed reuse in such a way that the distribution of heap chunks becomes more friendly to such workloads. An evaluation of HUSHVAC showed that the average performance overhead of HUSHVAC (4.7%) was lower than that of the state-of-the-art (11.4%) when running the SPEC CPU 2006 benchmark suite. Specifically, the overhead of HUSHVAC on the distribution-sensitive benchmark was about 35.2% while the prior work has an overhead of 110%.**

## I. INTRODUCTION

Use-after-free is a longstanding memory safety problem. Programming languages with manual memory management require developers to free heap chunks explicitly so that the chunks can be reused for other objects later. Unfortunately, determining if there is a pointer in the process that targets a freed chunk is not straightforward. If a chunk is freed while the program still has a pointer, such a *dangling* pointer can be misused to corrupt an object that reuses the chunk. Many software products are reported to have this class of vulnerability [5, 6, 7] despite the large mitigation effort [11, 12, 15, 16, 17, 19, 23, 25, 26, 27, 28, 29, 30, 31, 32, 34, 36]. The prevalence and severity of the threat even motivates the industry to migrate the existing software products into other languages such as Rust [22, 33, 35], which prevents the use-after-free vulnerabilities as one of its design goals.

*Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24804
www.ndss-symposium.org

Delaying the reuse of the freed memory chunks until the pointers to them disappear from memory is a promising approach to preventing use-after-free vulnerabilities. Markus and MineSweeper [11, 18] are state-of-the-arts in this direction, and with relatively low overhead in several benchmarks. When compared to the others, they exhibit relatively low overhead on execution time and do not require special hardware features that are not accessible on commodity processors, while deterministically preventing temporal safety violations unless the program hides the pointers. The disadvantage of these two approaches is that some allocation-intensive benchmarks are still significantly slowed down. Preliminary studies show that the current strategy, which combines the mark-sweep approach with existing commodity heap allocators has an inherent and unavoidable inefficiency. Simply delaying the reuse of freed chunks already incurs the slowdown because the approach increases heap fragmentation.

The most extreme form of delayed reuse is never reusing the freed chunks. This approach has a fundamental limitation: the program cannot run indefinitely since the virtual address space will eventually be exhausted. In return, the strategy has recently been found to be promising provided that the heap allocator itself is redesigned for such one-time allocation [36]. One interesting observation from this study is that it does not cause a significant slowdown compared to the aforementioned allocation-intensive benchmark.

These findings inspire us to tackle the problem from a different angle. HUSHVAC prevents heap object use-after-free by allocating heap chunks that have never been allocated or are addressed by no dangling pointers. The system is built on recent findings that modern computer systems can efficiently perform synchronous marking and concurrent sweeping with little performance intervention [11] and heap allocators can be optimized to allocate fresh heap chunks rather than reusing freed ones [36].

The following five key design choices enable HUSHVAC to attain the desired level of efficiency when running with allocation-intensive benchmarks and to become more comprehensive in discovering dangling pointers. First, HUSHVAC's underlying allocator is FFmalloc [36], which is optimized for allocating fresh chunks. Unlike conventional heap allocators or existing mark-sweep approaches, allocation from fresh chunks will be the norm in HUSHVAC. This design decision is inspired by FFmalloc's excellent performance result, which demonstrates that always creating fresh chunks does not necessarily incur performance overhead. Second, HUSHVAC primarily reuses freed memory chunks at the page level, thereby improving the proximity of allocations made from reused chunks. In principle, it does not explicitly reuse the physical memory space of the freed chunks. Instead, HUSHVAC only seeks to reuse the virtual address space of the freed chunks leaving physical page reuse to the operating system kernel. One observation that enables this design is that the Linux kernel allows a process to detach the physical page from a virtual page without unmapping the virtual page. This feature is a key enabler of this design because if HUSHVAC has to unmap such pages, it will encounter the over-splitting `VMA` structure issue that FFmalloc has been designed to avoid. Third, HUSHVAC performs the mark-sweep procedure to reuse the virtual address space only when the process is not actively allocating heap chunks. The two previous design choices enable HUSHVAC to use this third method because postponing the mark-sweep procedure does not incur a significant overhead on memory usage. Whereas the previous method pushes the heap chunks with their physical memory space to the waitlist for reuse, HUSHVAC pushes the virtual pages after detaching their physical memory space. Having more virtual pages in the wait list does not necessarily increase the process's memory usage because HUSHVAC only needs to maintain its metadata, not the entire page. Fourth, HUSHVAC also reuses the chunks from a page that is not entirely safe to reuse yet, to alleviate the potential fragmentation that small live heap chunks in mostly freed pages may cause. We designed this *sub-page reuse* in a way that does not nullify the page-level sweeping and harm the locality of reused chunks. Fifth, HUSHVAC scans the memory more comprehensively. Specifically, it also scans the memory pages that the application obtained by invoking system calls directly, without using a heap allocator. This comprehensive scanning enables HUSHVAC to discover some dangling pointers that an existing mark-sweep-based system [11] cannot. We validate this by discovering a Proof-of-Concept that triggers unsafe reuse when a program uses anonymous pages obtained directly (§V-F).

We implement HUSHVAC using FFmalloc as the underlying allocator and test it against several benchmark suites, including SPEC CPU, PARSEC, and BBench over Firefox. In all benchmarks, our evaluation shows that HUSHVAC incurs lower performance overhead than MarkUs, the state-of-the-art mark-sweep method. For example, the geomean overhead of HUSHVAC when running SPEC CPU 2006 is approximately 4.7% whereas MarkUs incur 11.4%. Notably, HUSHVAC incurs only 35% overhead on `xalancbmk`, the allocation-intensive the allocation-intensive benchmark we have been discussing, whereas MarkUs incurs 110%. One disadvantage of HUSHVAC that requires further optimization and evaluation is that the overhead on memory usage is approximately 59.8% when running SPEC CPU 2006, which is higher than that of MarkUs, 25.1%. We also test HUSHVAC's effectiveness against four CVE-assigned vulnerabilities [1, 2, 3, 4] found in widely used software. Furthermore, we employed HardsHeap [38], a fuzzer designed to target heap allocators. HUSHVAC successfully prevented all four exploits of the vulnerabilities. Additionally, HardsHeap failed to find any use-after-free vulnerabilities in HUSHVAC during a continuous testing period of over 20 hours.

In summary, this paper makes the following contributions.

- We are the first to investigate a novel approach to use-after-free prevention, beginning with a specially designed allocator that never reuses freed chunks. This new direction led us to make two new design choices: page-level sweeping and opportunistic mark-sweep. We also carefully adjust the sub-page reuse so as not to interfere with the page-level sweeping.
- To the best of our knowledge, we are the first to identify the root cause behind the performance overhead of existing mark-sweep-based use-after-free prevention approaches on an allocation-intensive benchmark and provide an alternative design that avoids the problem.
- Our implementation of HUSHVAC scans the memory more comprehensively. Specifically, unlike the two state-of-the-art marksweep-based delayed-reuse allocators, it scans the entire memory including anonymous pages, and leaves only the allocator metadata.
- HUSHVAC has the lowest performance overhead among the mark-sweep approaches for preventing use-after-free exploits.

## II. BACKGROUNDS

### A. Use-after-Free Primer

One common flaw found in software written in low-level languages such as C or C++ is the use-after-free vulnerability. The vulnerability arises when pointers still pointing to a freed heap area are mistakenly left in memory and then used by other code. An attacker exploiting this can trick the code by dereferencing the dangling pointer in various ways toward their goals, such as privilege escalation or information leakage. To illustrate, if a specific section of assigned memory holds data regarding a crucial function pointer (*e.g.*, a function pointer that invokes a security-sensitive system call), the data within that section can persist long after it has been reused. If an attacker can manipulate the way a victim program uses the dangling pointer, they can execute their desired program, like running a shell by activating the corresponding function pointer in that section. To avoid exploiting these vulnerabilities, we must check whether a dangling pointer exists in memory or not before reusing a freed heap chunk.

### B. Existing Approaches Delaying the Reuse

Recent studies [11, 17, 18, 36] demonstrate that postponing the reuse of freed heap chunks can effectively reduce use-after-free vulnerability exploits. They postpone the reuse until all dangling pointers have vanished or until the program terminates, ensuring that no dangling pointer remains associated with any allocated heap chunk.

**Reusing Freed Chunks after Ensuring No Dangling Pointers.** MarkUs and MineSweeper [11, 18] take the former approach, allowing the allocator to reuse a freed chunk only after a memory scan confirms the absence of dangling pointers. They begin by differing the `free`s by placing the freed chunks on the *quarantine list*. A chunk remains on this list until the allocator deems it safe for reuse. To determine which chunks from the quarantine list can be safely reused, MarkUs and MineSweeper pause the application, mark the heap chunks referenced by any pointer in the memory, and then transfer the unmarked chunks from the quarantine list to their free list. This process

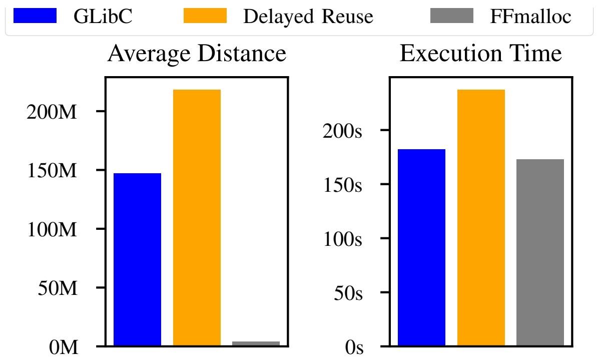


Fig. 1. On the right is the overhead of Delayed Free when running `xalancbmk` and on the left is the spatial locality of the chunks allocated from temporally local allocation requests.

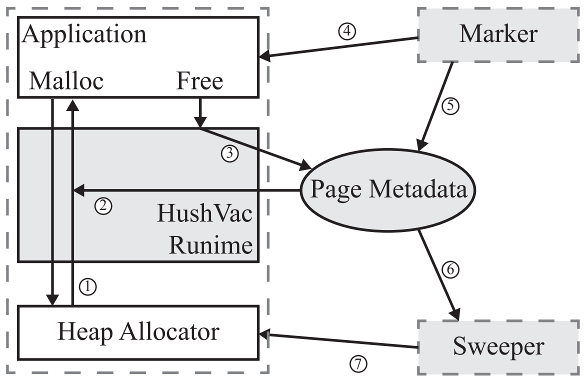


Fig. 2. An overview of HUSHVAC. §V-A describes how the components interact with each other.

of enabling the chunks to be reused is often referred to as *sweeping*. The difference between MarkUs and MineSweeper is the way they perform the mark phase. MarkUs uses the existing C++ garbage collector to take the conventional mark-and-sweep approach [8, 14]. MineSweeper shows that a linear scan of the memory could also be a valid alternative, allowing the use of the existing heap allocator rather than the specific allocator designed for garbage collection. One observation from our preliminary study is that their fundamental behavior, delayed reuse of freed chunks, inherently incurs a significant slowdown in some benchmarks. We further discuss this in the next section (§III).

**One-time Allocation.** An alternative, yet significantly different approach, is to avoid reusing freed chunks altogether. Oscar and FFmalloc [17, 36] are two recent studies that use this method. The fundamental principle is that by allocating new heap chunks from fresh virtual pages, no dangling pointers may be referenced. This approach cannot function indefinitely because of the limited virtual address space. However, it is still valuable in many instances where a program runs for a shorter duration. Although they refrain from reusing virtual pages, they do not consume excessive memory since they can still reuse physical pages when all heap chunks from a specific virtual page are freed. One intriguing observation that prompted the development of HUSHVAC is that FFmalloc, which is optimized for performance, incurs significantly less execution time overhead on some benchmarks, which MarkUs and MineSweeper cannot avoid.

## III. MOTIVATION

The interesting performance implications of the two approaches delayed reused and one-time allocation, inspired the development of HUSHVAC that inconspicuously and opportunistically reuses freed chunks.

MarkUs and MineSweeper incur high overhead on execution time when executing `xalancbmk`, often exceeding 100% (or 2×). When using MarkUs, our measurement shows more than 2× overhead, and MineSweeper authors report more than 2× overhead when using MineSweeper in the secure, *mostly concurrent* mode. On the contrary, FFmalloc does not incur a significant slowdown on the same benchmark, according to ours and the FFmalloc author's measurement.

Our effort to understand the reason for the slowdown reveals that merely delaying the reuse of freed chunks slows down the benchmark as shown in Figure 1, even exceeding FFmalloc. We hypothesize that the slowdown is caused by the lower spatial locality of chunks allocated from the temporally local allocation requests. Looking into the behavior of `xalancbmk`, we discover that the benchmark has hot spots where it frees and allocates many chunks within a short period. Instant reuse of free chunks enables the benchmark to obtain the chunks it has just freed, which are likely to have similar spatial locality to the chunks it has just freed. Conversely, delayed reuse of free chunks forces the benchmark to obtain chunks that have either never been used or have been freed a while ago, potentially resulting in a lower spatial locality. FFmalloc, however, allocates spatially local chunks for the spree of allocation requests because it always allocates fresh chunks from the top of the heap's size classes. We validate our hypothesis by measuring the distance between the chunks that are allocated from the temporally local allocation requests, as shown in Figure 1. We compute the average distance of an allocation as the average distance between the allocated chunk and the 10 nearest chunks that are allocated before and after the allocation. The figure we report is the average of this average distance for each of the benchmark's 50 allocations. The number we report is the average of this average distance for every 50 allocations in the benchmark. As expected, delayed reuse significantly increases the average distance, while FFmalloc exhibits a considerably low (about 4M) average distance.

## IV. THREAT MODEL

We assume that a program that runs with HUSHVAC has one or more use-after-free vulnerabilities and that these are the program's only vulnerabilities. In other words, the attacker must exploit one or more of these use-after-free vulnerabilities. Other vulnerabilities, such as spatial safety violations or logic bugs, are out of scope. HUSHVAC is not designed to defeat the attacks exploiting vulnerabilities other than use-after-free. We also assume that HUSHVAC is well-written and does not include any exploitable vulnerabilities. Note that this set of assumptions is consistent with prior work on use-after-free mitigation and prevention [11, 17, 18, 36].

## V. Design

HushVac is built around the opportunistic page-level mark-sweep engine along with FFmalloc as the underlying allocator. We designed HushVac on top of FFmalloc because HushVac forces the underlying allocator to frequently allocate from the fresh virtual pages, and FFmalloc is optimized for this scenario. The difference between HushVac's mark-sweep engine and the existing ones is that it primarily reuses the freed virtual pages, not the heap chunks with the corresponding physical memory space.

### A. Overview

Figure 2 gives an overview of HushVac. Its runtime uses page metadata shared with its marker and sweeper threads to mediate allocator calls within the application thread. HushVac mediates allocation requests and mostly serves chunks returned by its heap allocator, FFmalloc (①). When possible, HushVac serves chunks that can be safely reused as described in §V-G (②). The primary role of HushVac's runtime is to alter a chunk's state to free upon an invocation of a `free`, where it updates the corresponding metadata (③). HushVac's marker thread performs two-staged marking (§V-C) by reading the application thread's memory, stopping it when needed (④), and updating the result to the page metadata (⑤). The sweeper thread checks the page metadata (⑥) to determine if an entire virtual page can be reused, and deliver such pages to the heap allocator for reuse (⑦).

### B. Mark-Sweep For Virtual Pages

HushVac's mark-sweep engine determines if it is safe to reuse each freed *virtual page* and makes the safe pages available for future allocation. During the sweep phase, the allocator is re-provisioned with the unreferenced yet freed virtual pages after identifying which address ranges are still referenced by the application during the mark phase. HushVac considers a virtual page as free if all heap chunks within the page are freed.

HushVac tracks the freed status of each chunk in the virtual page by maintaining a bitmap for each virtual page within the heap. When the final live chunk in a virtual page is marked as freed by an invocation of `free`, HushVac pushes the virtual page to the quarantine list after detaching the corresponding physical page. This process is not batched by HushVac. After the final chunk of a 4-KiB virtual page is freed, it is immediately moved to the quarantine list, without waiting for additional pages in a larger batch to be freed. By invoking the mmap system call with the `MAP_FIXED` flag, HushVac can still detach the physical page corresponding to the virtual page without splitting the in-kernel `VMA` structure, so that the pages in the quarantine list do not hold the physical memory unnecessarily.

This design choice builds on three observations. Firstly, as per our experiments and prior work [18], promptly invoking `unmap` without batching does not significantly impact execution time, but helps in reducing memory usage. Secondly, invoking `mmap` in this manner does not fragment the kernel's `VMA` structure. A previous concern was that frequent `mmap` calls might lead to VMA fragmentation. Because each `unmap` invocation splits the VMA structure, promptly invoking `unmap` might bring back this issue. However, invoking `mmap` with `MAP_FIXED`, which simply detaches physical memory from the virtual page, avoids splitting the `VMA` structure while decreasing memory usage. We demonstrate empirically that HushVac does not cause the creation of many `VMA` structures in Figure 22(a). Finally, a freed page is likely to be reused soon after all references to it disappear, and retaining the page mapping allows HushVac to avoid unnecessary `mmap` invocation when reusing the page. These three facts imply that our design choice of promptly invoking `mmap` without batching neither incurs significant overhead on execution time nor the number of `VMA` structures while maintaining the advantage of reducing memory usage.

### C. Two-Staged Mark Phase

The mark phase comprises concurrent and synchronous phases, to reduce the time the application must be stopped for sound marking. In principle, the safety of virtual page reuse, as indicated by the presence of dangling pointers to the virtual page, must be examined while the application is stopped. A concurrent mark phase beginning at $t_0$ and ending at $t_1$ may miss a pointer moving from one location to another after $t_0$ but before $t_1$. This problem has already been acknowledged in a recently proposed system [18], and they believe their concurrent approach is insecure. Specifically, HardsHeap generates a PoC when we run the open-sourced implementation of this scheme, which works only when marking is performed fully concurrently. The synchronous mark phase eradicates this problem because the marking happens instantly from the perspective of the application that is not executing. From the application's perspective, the marking effectively happens at a certain point in time, say $t_2$. The disadvantage of synchronous marking is its inherent overhead on the execution time. The time spent on synchronous marking immediately becomes the extra execution time, slowing down the application.

The two-staged mark phase reduces the number of pages that HushVac must scan synchronously. Each concurrent mark phase begins by determining the set of pages that HushVac must scan, from the Linux kernel's interface that provides the application's virtual address space information, `/proc/self/maps` file. Scanning each page begins with clearing the `dirty` bit, indicating if the page content has been modified since the last clearance of the bit. After clearing all dirty bits, the scanning thread traverses the page, treating each 8-byte value as a pointer if it is in the heap range. For each pointer identified, HushVac sets the corresponding mark bit in the *mark map*. The synchronous mark phase follows the concurrent phase and begins with the mark-sweep thread pausing the application. At this moment, if a pointer is in one of the pages whose `dirty` bit is `0`, the corresponding mark bit is 1. The mark map is not yet sound because it does not guarantee anything about the pages whose `dirty` bit is 1, *i.e.*, about the pages that have been modified since the last clearance of the `dirty` bit, the beginning of the concurrent mark phase. While the application is not running, the synchronous mark phase of HushVac refines the mark map by scanning these dirty pages again. After scanning, the mark map becomes sound in that it indicates whether or not each pointer exists in the application's memory at a given point in time. Because the next step, the sweep phase, can also run concurrently, HushVac resumes right after the synchronous mark phase.

In summary, HushVac's two-staged mark phase reduces the latency of synchronous marking by reducing the number of pages that must be scanned synchronously while still ensuring the soundness of the mark map by synchronously scanning the dirty pages.

### D. *Page-Level Sweeping*

Following the mark phase, HushVac executes the sweep phase, which reclaims the virtual pages in the quarantine list if the virtual page's addresses can be safely reused. As mentioned earlier, this phase runs concurrently with the application threads following the synchronous mark phase. The sweeping thread begins by traversing the quarantine list, and for each virtual page in the list, it checks to see if any bit in the mark map corresponding to the page is set to 1. A freed virtual page is considered safe to reuse if all the page's mark bits remain cleared (*i.e.*, `0`) at this moment. The pages are then moved from the quarantine list to the *reuse batch list* by HushVac. When HushVac is ready to enlarge the heap, it first checks to see if the reuse batch list contains any virtual pages before invoking `mmap`, and if so, uses the one from this list. The cost of additional decision-making, whether to reuse a virtual page or not, is amortized by the reduced number of `mmap` calls because the pages in the reuse batch list are already mapped. As explained earlier, HushVac only detaches the physical memory and retains the virtual page.

### E. *Opportunistically Triggering the Mark-Sweep Procedure*

To avoid stopping the application while running actively, HushVac launches the mark-sweep procedure opportunistically. What enables this design choice is the page-level reuse of virtual address space. Unlike previous methods, the virtual pages in HushVac's quarantine list do not contain the physical page because the physical pages are detached from the virtual pages when the page is freed. Having one page in the quarantine list costs only approximately 16 Bytes for the virtual page's base address and data structure management. Because of this low cost, HushVac can keep many virtual pages in the quarantine list and postpone the mark-sweep procedure as needed.

Our implementation of HushVac avoids triggering the mark-sweep procedure, which accompanies the synchronous mark phase when the application actively allocates new heap objects. We chose this approach under the assumption that latency-critical tasks often begin with an incoming request, followed by data generation or retrieval, all of which involve heap allocations HushVac detects this hotspot by continuously monitoring the frequency of heap allocations. A counter is increased for each allocation request and is used by the periodically activated mark-sweep thread to estimate the frequency of heap allocations. The average number of allocations per period is maintained by the mark-sweep thread. The thread uses a counter to compare the average with the number of allocations during the previous epoch. HushVac triggers the mark-sweep procedure only when the number of allocations in the last epoch is below a certain threshold (empirically set to $1.1\times$ of the average). In other words, if the number of allocations during an epoch is greater than $1.1\times$ of the average, HushVac considers the application to be busy and delays the mark-sweep procedure.

```c
1  #define PROT_FLAG   PROT_READ|PROT_WRITE
2  #define MAP_FLAG    MAP_ANON|MAP_PRIVATE
3
4  int main() {
5      void **p = (void **)mmap(NULL, PAGE_SIZE,
     PROT_FLAG, MAP_FLAG, 0, 0);
6      p[0] = malloc(963751);
7      free(p[0]);
8      p[1] = malloc(963776);
9
10     // [BUG] Reclaim happens: p[0]=0x564549a79000
     (size=963760) -> p[1]=0x564549a79000 (size
     =963792)
11     assert(p[0] <= p[1] && p[1] < p[0] + 963760);
12 }
```

Fig. 3. Proof-of-concept triggering an unsafe reuse, generated by a modified HardsHeap [38] against an existing use-after-free mitigation scheme [11]. This scheme does not traverse the anonymous page, which results in the reuse of a chunk pointed to by a dangling pointer left behind at line 7.

### F. *Comprehensive Scanning of the Memory Space*

The mark-sweep procedure of HushVac scans the entire memory except for the allocator metadata to determine if it is safe to reuse the virtual pages in the quarantine list. Areas scanned include the stack, heap, and memory space the application retrieved by invoking the `mmap` directly. We deem this conservative approach inevitable because HushVac cannot assume that the application stores the heap pointers exclusively within a designated set of virtual address pages (*e.g.*, the heap pages). For example, a modified version of HardsHeap [38], a fuzzer specifically designed to detect vulnerabilities in heap allocators, generates a proof of concept (PoC) that triggers a use-after-free issue while an existing scheme is running, as demonstrated in Figure 3. In the experiment, we adjusted HardsHeap to diversify the pointer locations and include `mmap`ed page, thereby enabling it to discover the PoC.

### G. *Sub-Page Reuse*

A potential disadvantage of HushVac's page-level sweeping is that it reuses freed chunks only after the entire page containing the chunk is freed. In some cases, small live chunks on a page may prevent the entire page from being reused, wasting of physical memory. To address this, HushVac selectively and carefully reuses some chunks before the entire page becomes eligible for page-level sweeping. Specifically, this sub-page reuse is designed such that the chunks allocated by the sub-page reuse feature have better spatial locality than chunks allocated by existing schemes, and enabling the sub-page reuse does not prevent page-level reuse paths. To this end, HushVac does not explicitly create and maintain a quarantine list of freed chunks. By not putting the chunks in another quarantine list, HushVac leaves the pages containing freed chunks as the candidate for page-level reuse, whereas it potentially reuses the freed chunk if no dangling pointer references it. Instead, HushVac builds and maintains a *sub-page reuse batch list* from which it retrieves chunks for reuse at the sub-page level. During a sweep phase, HushVac pushes pages containing at least one chunk that can be reused safely to the sub-page reuse batch list. When processing a new request for allocation, HushVac retrieves heap chunks from this batch list whenever possible. Therefore, sub-page reuse still maintains a page-level

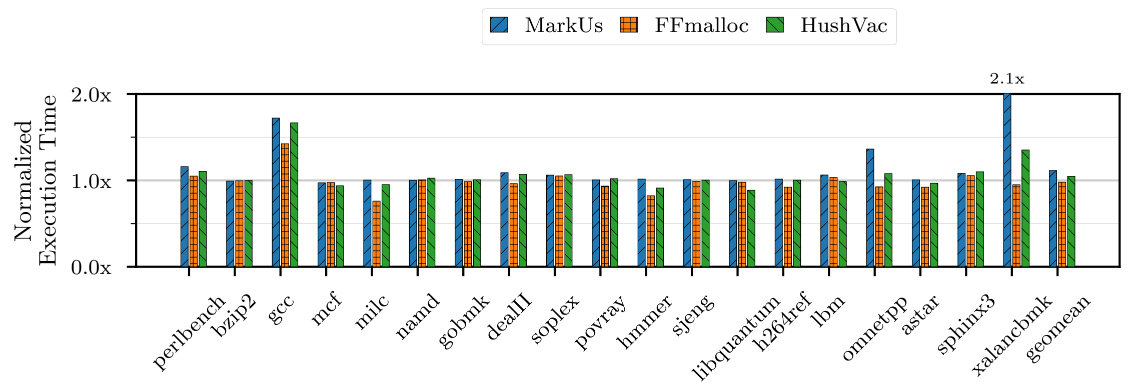


Fig. 4. Normalized execution time when running benchmarks in SPEC CPU 2006 with MarkUs, FFmalloc, and HUSHVAC. MarkUs, FFmalloc, and HUSHVAC slow down the benchmarks by 11.4%, -2.1%, and 4.7% on average (geometric mean), respectively.

reuse list to ensure that temporally local allocations are more likely to be served from the same page and the page at the tail of the sub-page reuse batch list could get a chance to become a fully freed virtual page, being the last candidate for the sub-page reuse.

## VI. EVALUATION

**Experimental Setup.** We evaluated HUSHVAC on a system that runs Ubuntu 18.04 with Linux 5.4.0-150-generic as the kernel, on a machine with an AMD Ryzen 5 2600 and 32 GB main memory. All allocators compared to HUSHVAC are built using their default configurations. HUSHVAC runs one reclaimer thread and 10 scanner threads per process.

**Effectiveness.** We observed that the implementation of HUSHVAC effectively prevents the reuse of chunks referenced by dangling pointers in two ways. First, we verify that HUSHVAC can prevent real-world exploits using four public use-after-free exploits as shown in §VI-G. Second, we run HardsHeap's Reclaim module [38], designed to look for use-after-free examples that an allocator fails to prevent. Running HardsHeap for more than 20 hours against HUSHVAC did not report any working use-after-free examples, indicating HUSHVAC's effective preventtion of use-after-free. Similar to FFmalloc [36], HUSHVAC does not abort the use-after-free test cases in the NIST Juliet Test Suite [20]. NIST Juliet is a collection of C/C++ test cases that includes the ones for testing use-after-free of heap chunks. Note that NIST Juliet's use-after-free test cases only evaluates whether freed chunk are reused using dangling pointers. FFmalloc and HUSHVAC do not affect this behavior because they are designed to prevent the use of dangling pointers after a chunk is reused in subsequent allocations, rather than using freed objects before the reuse.

**Performance Benchmarks.** We compare HUSHVAC with an existing system using five different workloads. First, we use the SPEC CPU 2006 benchmark suite [10], which is the de facto standard benchmark suite for evaluating the performance impact of use-after-free prevention schemes. The results we report here will allow the comparison of HUSHVAC with many existing use-after-free prevention schemes. The second benchmark we use is the SPEC CPU 2017, the latest version of the SPEC CPU benchmark suite. We selected 19 single-threaded workloads written in C/C++ from SPEC CPU 2006 and 12 multi-threaded workloads from SPEC CPU 2017. We measured the execution time and the maximum resident set size (MaxRSS) of each workload using time utility and the command generated by the benchmark driving script. We build the benchmarks using the build system included in the suite and enable OpenMP `OMP_THREAD_NUM` to 8 for SPEC CPU 2017. Note that CPU 2006 uses `-O2` as the default optimization level, and CPU 2017 uses `-O3`. Third, we use BBench [21] on Firefox, a browser rendering benchmark that MarkUs used for evaluation. This benchmark was chosen to demonstrate that HUSHVAC can work for large real-world applications. Since BBench 3.0 was not available at the time of evaluation, we used BBench 2.0. The fourth benchmark that we used was the `mimalloc-bench`. It is composed of microbenchmarks that test the allocator performance and allocation-intensive application workloads. The last one is the 12 multi-threaded workloads written in C/C++ from PARSEC 3.0 [13] that FFmalloc used to evaluate the scalability of HUSHVAC. Note that we do not report the performance impact of PARSEC on two workloads, `x264` and `ferret`, as it could not be run with FFmalloc or MarkUs.

**Comparison Targets.** We compare HUSHVAC with MarkUs [11] and FFmalloc [36]. Unless otherwise stated, results were obtained by running them in our environment. Another recent use-after-free prevention scheme, Minesweeper [18], could not be used because the release version did not work as expected. As stated in the study, this implementation works only in *fully* concurrent mode and does not prevent use-after-free. The appropriate configuration for the comparison is the *mostly* concurrent version, but the released version does not work as expected.

### A. SPEC CPU 2006

**Performance Overhead.** Figure 4 shows the overhead of three allocators on the execution time of the 19 workloads in SPEC CPU 2006. The execution time overhead incurred by HUSHVAC is 4.7% on average (geometric mean), which is lower than MarkUs' 11.4% but higher than FFmalloc's -2.1%. One factor that contributes to the performance advantage of HUSHVAC is the stop-the-world time when the application threads are paused during the synchronous mark phase. Figure 5 shows that HUSHVAC stops the application thread for only 3.03 s on average whereas MarkUs pauses for 28 s. Note that FFmalloc does not reuse virtual pages or freed chunks, so it does not stop

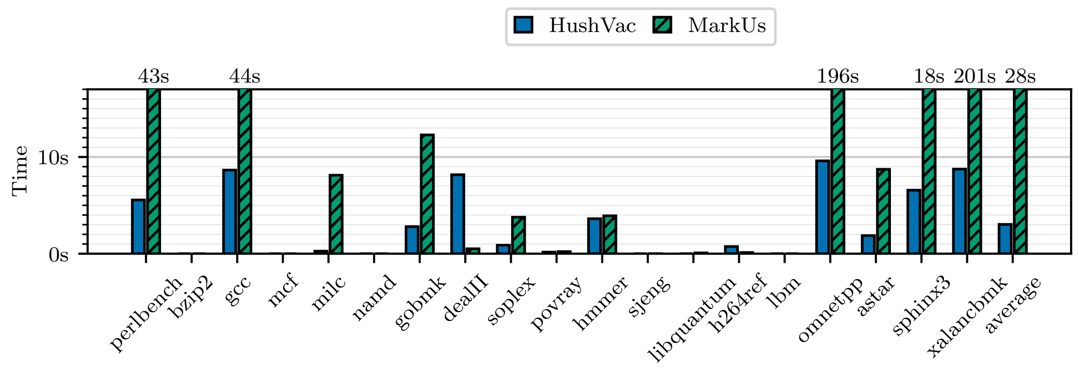


Fig. 5. Stop-the-world delay in SPEC CPU 2006 with MarkUs and HushVac.

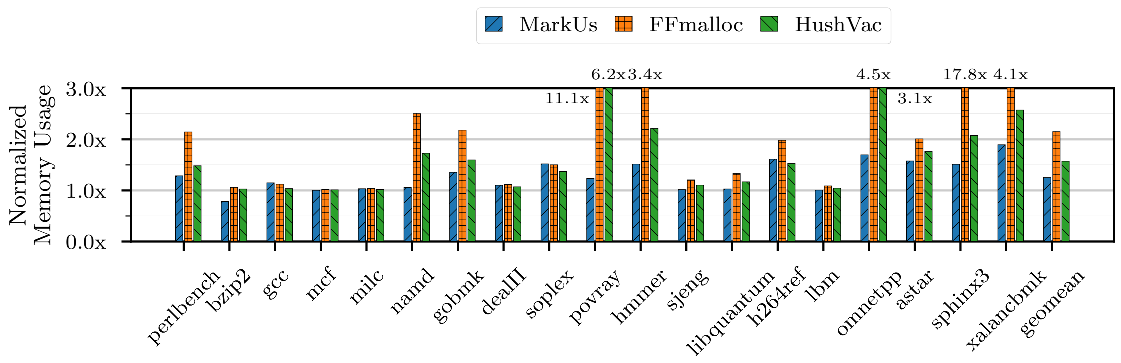


Fig. 6. Normalized memory usage when running benchmarks in SPEC CPU 2006 with MarkUs, FFmalloc, and HushVac. The memory usage increases on average (geometric mean) by 25.1%, 115%, and 57.2% when running with MarkUs, FFmalloc, and HushVac, respectively.

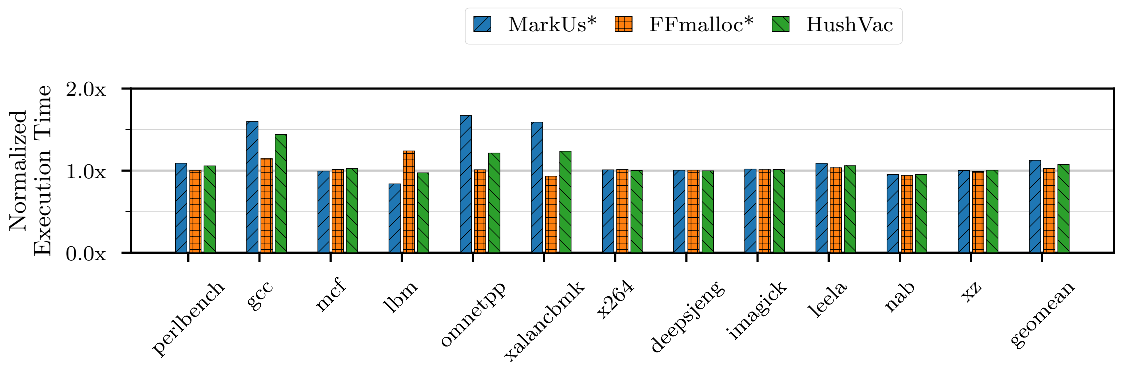


Fig. 7. Normalized execution time when running benchmarks in SPEC CPU 2017 with MarkUs, FFmalloc, and HushVac. MarkUs, FFmalloc, and HushVac slow down the benchmarks by 12.6%, 2.6%, and 7.3% on average (geometric mean), respectively. For allocators with *, we use the `gcc` results reported in the literature[18].

the application threads. When running the well-known `malloc`-intensive benchmarks, such as `perlbench`, `gcc`, `omnetpp`, and `xalancbmk`, HushVac exhibits lower performance overhead than MarkUs. Regarding `xalancbmk`, HushVac incurs only 35% additional overhead, whereas MarkUs has a whopping 110% overhead because of the lesser spatial locality explained in §II. This result suggests that HushVac is a more attractive allocator that prevents use-after-free when running on the allocation-intensive benchmarks than the existing mark-and-sweep approaches.

**Memory Overhead.** Figure 6 shows the memory overheads for 19 SPEC CPU 2006 workloads running on HushVac, MarkUs, and FFmalloc, respectively. Benchmarks run on HushVac consume 57.2% more memory. This is less than FFmalloc, whose overhead is 115%, but more than MarkUs, which incurs 25.1% memory overhead only.

## B. SPEC CPU 2017

**Performance Overhead.** Figure 7 shows the normalized execution times for 12 benchmarks from the SPEC CPU 2017 benchmark suite. HushVac's overhead is 4.1% on average (geometric mean), slightly higher than FFmalloc's 2.6%, but much lower than MarkUs' 12.6%. Specifically, the overhead of HushVac on four allocation intensive benchmarks,

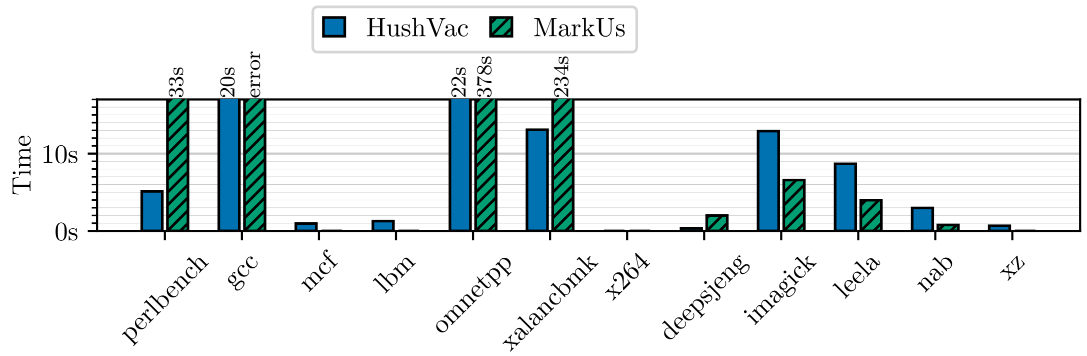


Fig. 8. Stop-the-world delay when running SPEC CPU 2017 with MarkUs and HushVac.

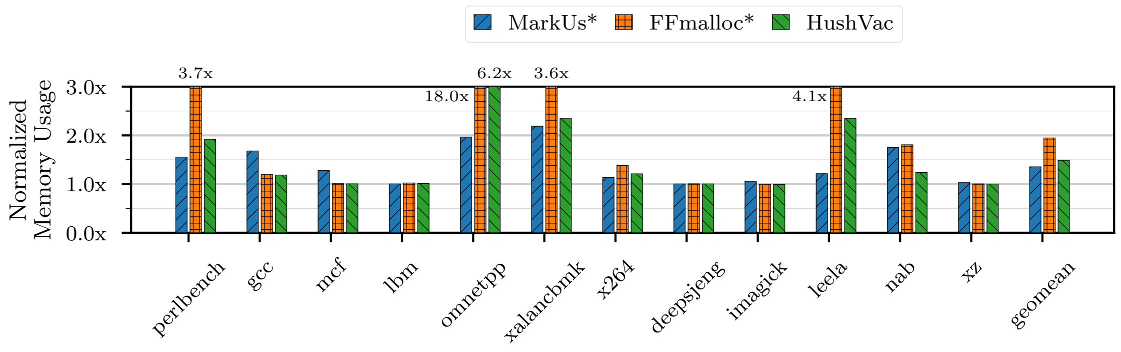


Fig. 9. Normalized memory usage when running benchmarks in SPEC CPU 2017 with MarkUs, FFmalloc, and HushVac. The memory usage increases on average (geometric mean) by 35.4%, 94.7% and 48.9% when running with MarkUs, FFmalloc, and HushVac, respectively. For allocators with *, we use the `gcc` results reported in the literature [18].

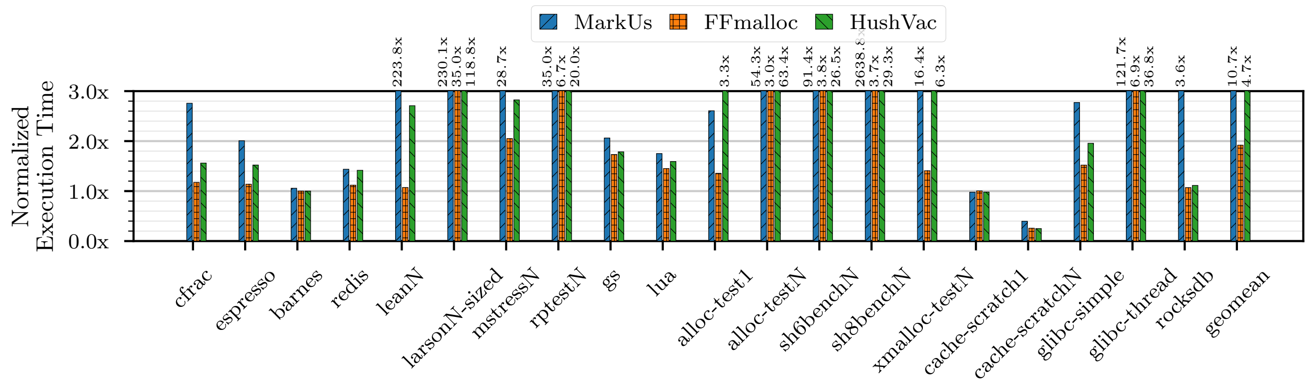


Fig. 10. Normalized execution time when running `mimalloc-bench` with MarkUs, FFmalloc, and HushVac. MarkUs, FFmalloc, and HushVac slow down the benchmarks by 974%, 91.8%, and 372% on average (geometric mean), respectively.

`perlbench`, `gcc`, `omnetpp`, and `xalancbmk` are 5.7%, 43%, 21%, and 23%, respectively. These are significantly lower than what MarkUs incur (9.1%, 60.0%, 67.0%, and 59.1%, respectively). For CPU 2006, the stop-the-world time largely contributes to the performance overhead of MarkUs, as we present in Figure 8.

**Memory Overhead.** As shown in Figure 9, similar to the result from CPU 2006, HushVac incurs higher memory usage overhead than MarkUs The average overhead of HushVac is 48.9%, while FFmalloc and MarkUs incur 94.7% and 35.4%, respectively.

## *C. Stress Testing with Mimalloc-bench*

We also use the `Mimalloc-bench`, which consists of workloads heavily stressing the heap allocator. As shown in Figure 10, the average overhead of HushVac (372%) is significantly lower than that of MarkUs (974%). Similar to the application benchmark, HushVac still exhibits higher memory usage overhead compared to MarkUs due to internal fragmentation, as Figure 11 shows. Note that Figure 12 shows the stop-the-world delay when running the `mimalloc-bench`.

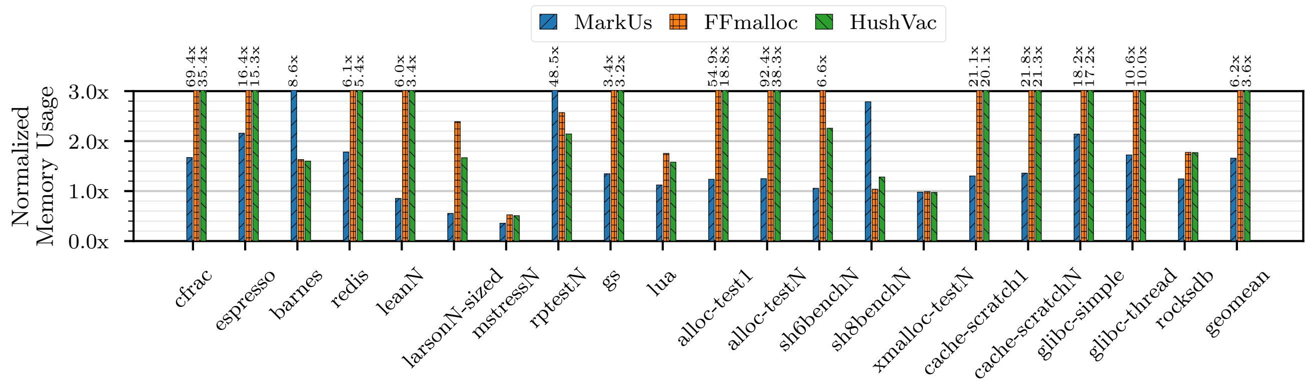


Fig. 11. Normalized memory usage when running `mimalloc-bench` with MarkUs, FFmalloc, and HushVac. The memory usage increases on average (geometric mean) by 65.8%, 520%, and 257% when running with MarkUs, FFmalloc, and HushVac, respectively.

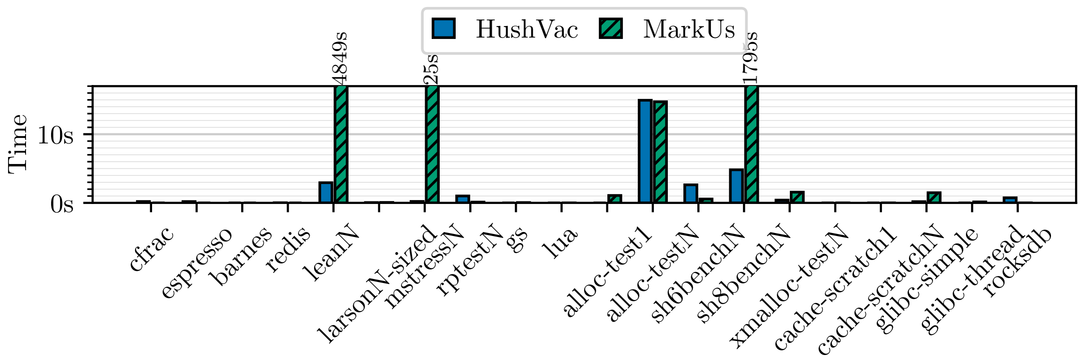


Fig. 12. Stop-the-world delay in when running `mimalloc-bench` with MarkUs and HushVac.

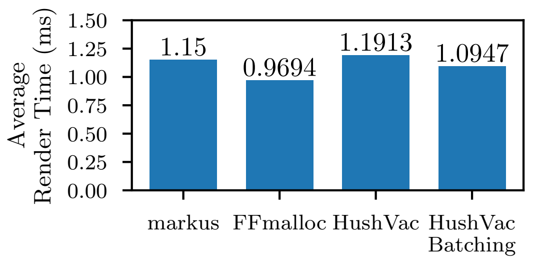


Fig. 13. Average (geometric mean) overhead on the render time when running BBench on Firefox that runs with MarkUs, FFmalloc, and HushVac. We measured the FFmalloc result on our environment, but the MarkUs result is the one reported by the authors as we could not run Firefox with MarkUs in our environment.

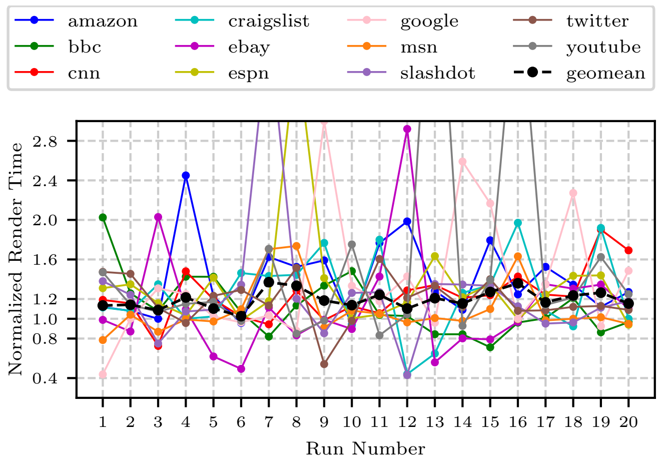


Fig. 14. HushVac on BBench [21] in Firefox. The average performance speedup across webpages loaded by BBench in Firefox is 19.1%, while MarkUs reports a 15% slowdown.

## D. Real-world Application

Figure 13 shows the results of running a browser workload using MarkUs, FFmalloc, and HushVac. As also shown in Figure 14, each number is obtained by computing the geometric mean of normalized render time from 20 consecutive runs of 11 webpages, as we also present. We obtained the overhead of FFmalloc on our environment, but the number for MarkUs is computed from the result reported in the paper because we could not run Firefox with MarkUs in our environment. The result shows that HushVac has similar performance to MarkUs, with 19.13% longer rendering time compared to unmodified Firefox running with `jemalloc` [9].

Figure 14 shows the normalized render time for each webpage over 20 consecutive runs of Firefox running with HushVac. The result confirms that HushVac can handle real-world workloads for a long time. Consecutive runs do not show any significant trend in the render time except for some iterations, and the overhead does not increase monotonically as the run continues. Our analysis of the exceptionally long render time in some iterations reveals that Firefox exhibits a relatively

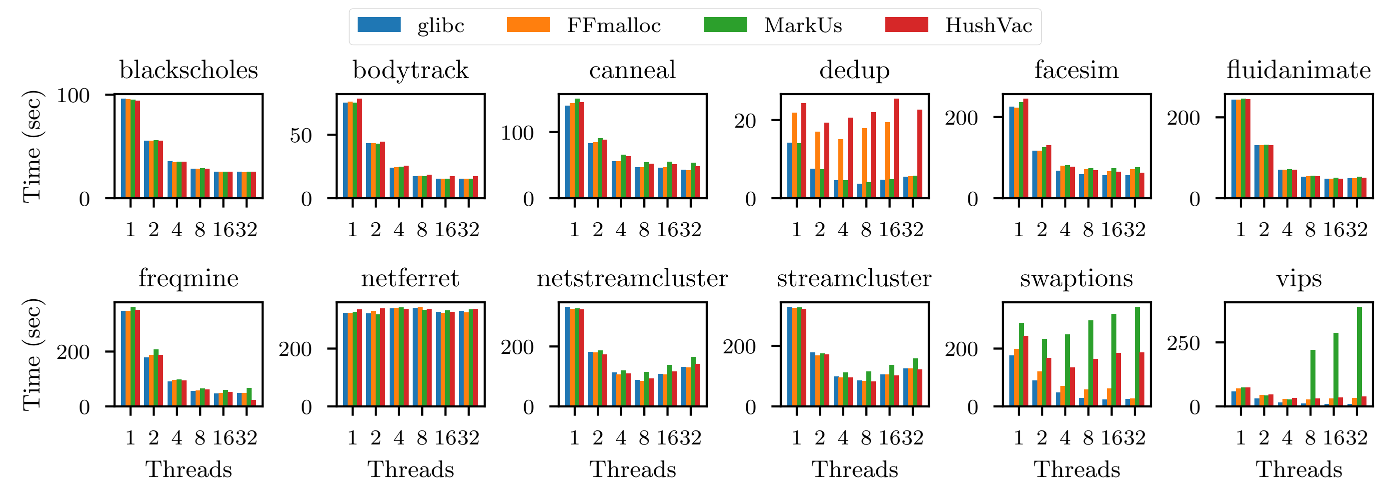


Fig. 15. Execution time when running benchmarks from PARSEC 3.0 with four different allocators. The overhead of HUSHVAC is hardly noticeable except for `swaptions` and `dedup`.

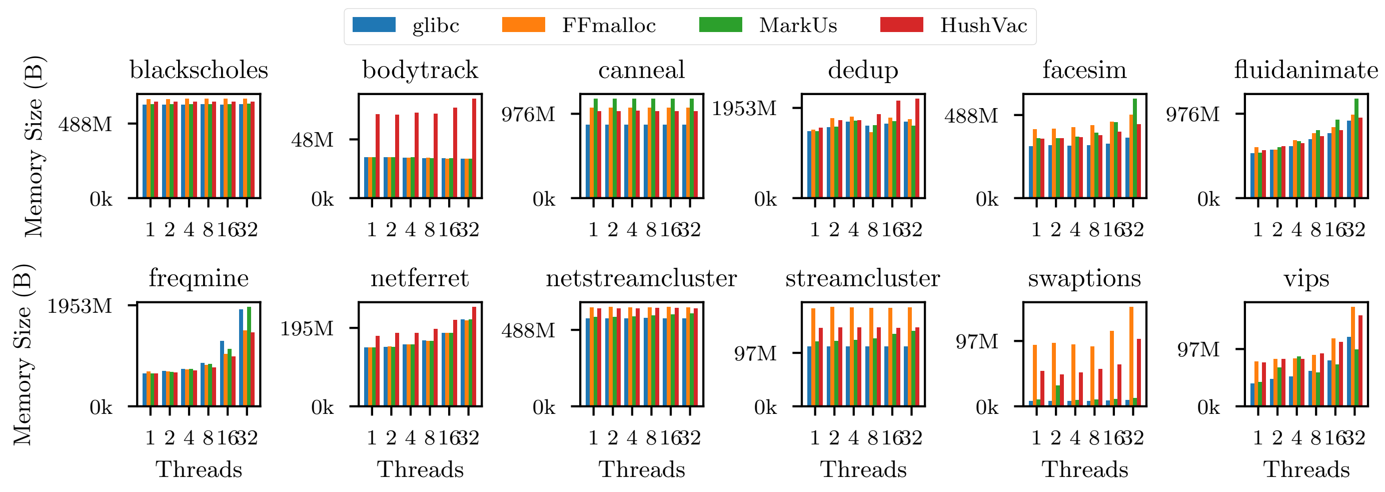


Fig. 16. Memory usage when running benchmarks from PARSEC 3.0. Regarding memory usage, HUSHVAC has results comparable to FFmalloc except for `netferret` and `bodytrack`.

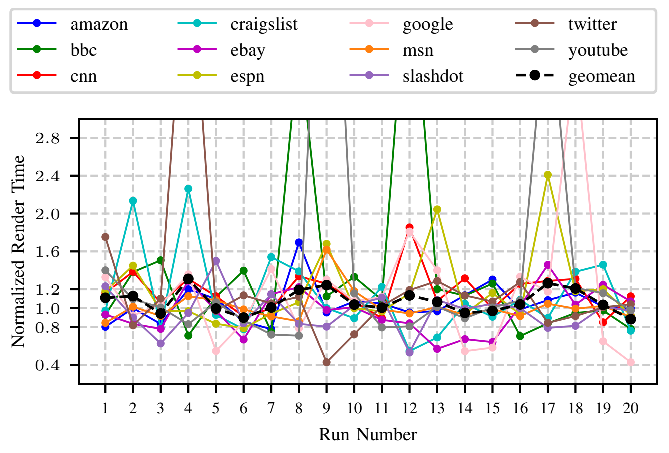
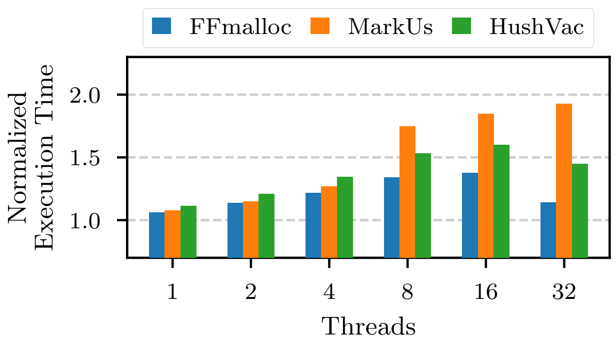


Fig. 17. HUSHVACBATCHING on BBench [21] in Firefox. The average performance speedup across webpages loaded by BBench in Firefox is 9.47%, while MarkUs reports a 15% slowdown.

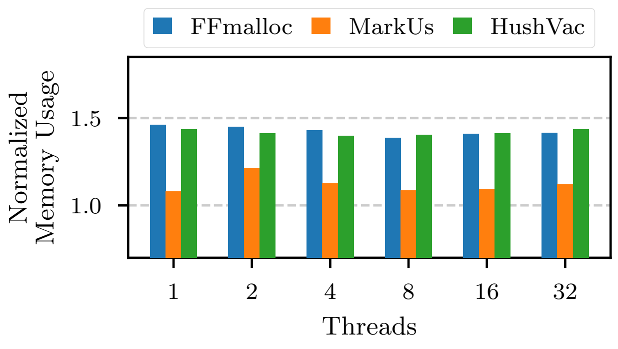
high number of system calls and page faults to expand the heap when rendering some benchmarks, such as Amazon (the fourth run), causing the slowdown. Heavily invoking memory management system calls from multiple threads causes the slow down, since memory management system calls are often serialized in the kernel. We also observe some cases where the stop-the-world delay contributes to the long render time, such as ESPN (the eighth run). As Figure 13 and Figure 17 show, reducing the number of memory management system calls by batching them helps the performance, potentially making HUSHVAC outperform MarkUs on average.

### *E. Multi-threaded Workloads*

We evaluate MarkUs, FFmalloc, and HUSHVAC using 12 workloads from PARSEC 3.0 [13] with varying numbers of threads using the Native input. We could run these 12 workloads with all three allocators, except for the combination of `vips` and MarkUs. Moreover, we could not run 3 of the 15 workloads that FFmalloc used for evaluation, `netdedup`, `ferret`, and `x264` with FFmalloc or MarkUs. The failure



(a) Execution Time



(b) Memory

Fig. 18. Geometric mean of the normalized execution time and memory when running 12 benchmarks in PARSEC 3.0 with varying numbers of threads.

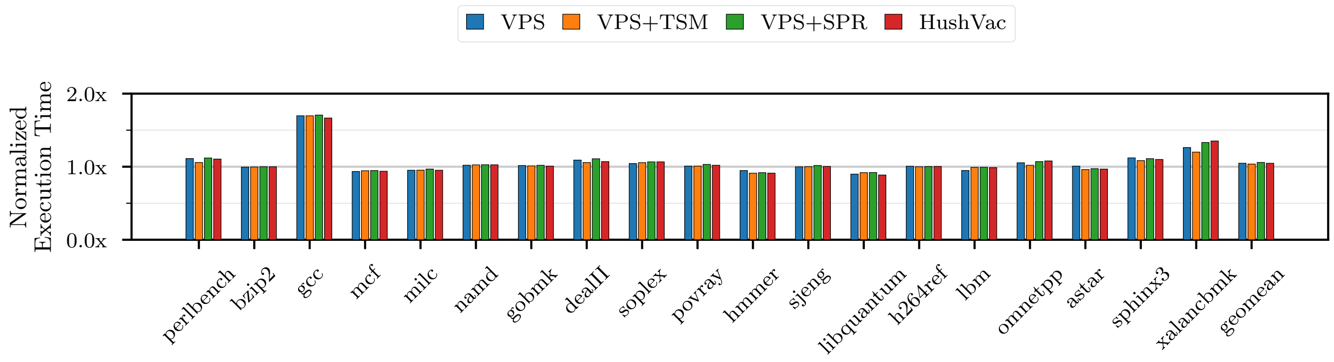


Fig. 19. Normalized execution time when we run benchmarks in SPEC CPU 2006 with variations of HushVac where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.

occurred because PARSEC 3.0, despite its prevalent mention in academic literature, was incompatible with Ubuntu 18.04 due to the cessation of active development.

As shown in Figure 15, HushVac exhibits the lowest execution time overheads, except for the `fluidanimate` benchmark when run with a single thread. This result shows that the 11 helper threads HushVac uses do not significantly affect performance owing to the opportunistic mark-sweep. Moreover, Figure 16 shows that the memory usage overhead of HushVac is comparable to that of MarkUs. `dedup` with 32 threads is the only workload where MarkUs has lower memory overhead.

We summarize the overhead of three allocators when running the 12 benchmarks in PARSEC 3.0 by computing the geometric mean of the execution time and memory usage in Figure 18(a) and Figure 18(b), respectively. As Figure 18(a) shows, HushVac slows down only 36% on average, while the overhead of FFmalloc and MarkUs is 21%, and 46%, respectively. Notably, the overhead of HushVac with PARSEC 3.0 is higher on average than that of FFmalloc because the reuse ratio of remapped virtual pages outweighs the performance overhead due to frequent system calls.

Figure 18(b) shows the remarkable memory usage of HushVac with little effect from the stop-the-world due to lazy sweeping compared to MarkUs, which sacrifice execution time with the stop-the-world to reduce memory usage. HushVac exhibits 41% of memory overhead, whereas FFmalloc has 42% and MarkUs has 11.9%. When we compare FFmalloc and HushVac, what is noticeable is that PARSEC 3.0 has a greater memory overhead from batch page unmapping than memory overhead from internal fragmentation, even though HushVac has the same internal fragmentation problem as FFmalloc.

## F. Evaluating Design Choices

**Performance Impact.** We evaluate the impact of our design choices on execution time and memory usage when we run benchmarks in SPEC CPU 2006. Figure 19 presents the normalized execution time, showing that most of HushVac's advantage in execution time is owing to the page-level sweeping. Note that all design choices other than the two-staged mark phase and sub-page reuse are turned on together with virtual page-level sweeping. As expected, the two-staged mark phase positively affects the execution time on some benchmarks that experience a relatively long stop-the-world delay. Figure 20 further presents how the design choices affect the stop-the-world delay. Note that sub-page reuse negatively affects the execution time on some benchmarks, such as `xalancbmk` where sub-page reuse could potentially harm the locality. Nevertheless, our careful design for preserving the locality benefit makes the overhead on this benchmark much lower than the other mark-sweep-based approaches. Figure 21 presents the impact of these design choices on memory usage. As we further highlight later in this section, some benchmarks (*e.g.*, `sphinx3`) that suffer from a small number of chunks occupying mostly free pages get a significant benefit in memory usage, when we enable sub-page reuse.

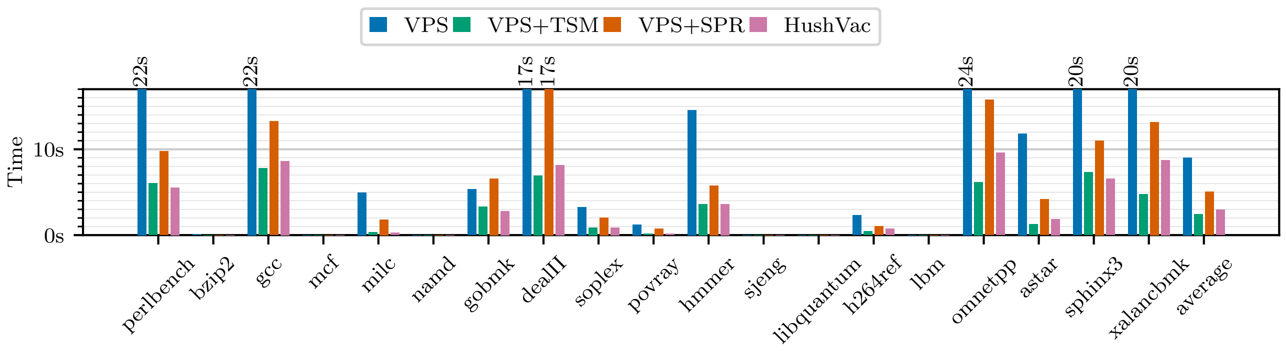


Fig. 20. Stop-the-world delay when we run benchmarks in SPEC CPU 2006 with variations of HUSHVAC where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.

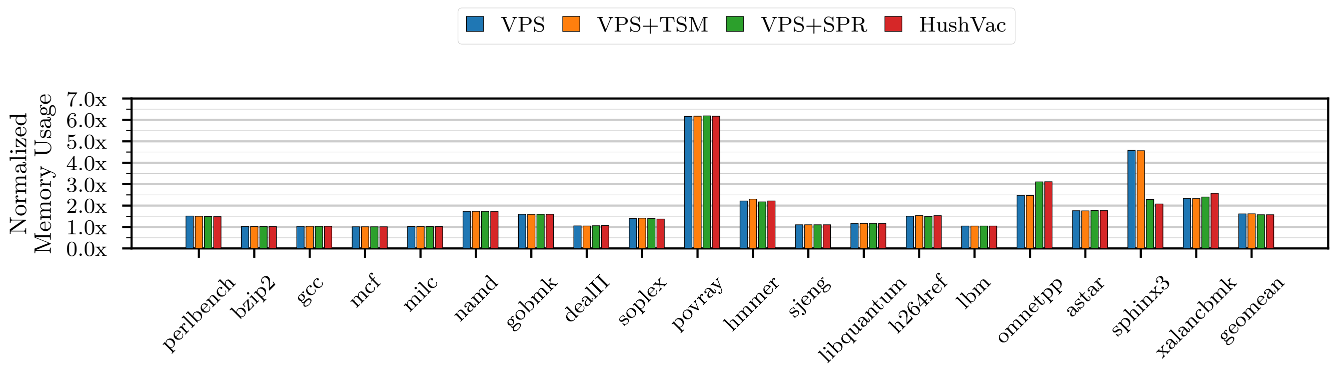
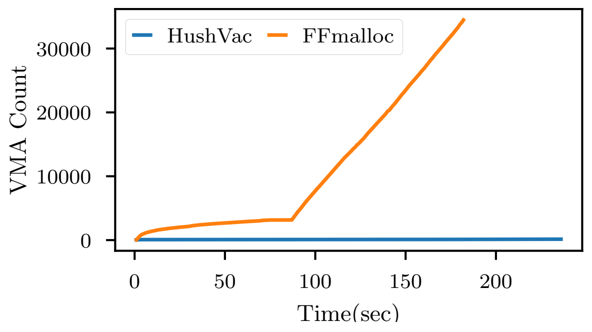
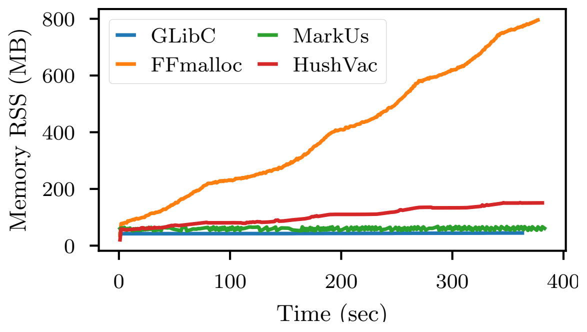


Fig. 21. Normalized Memory usage when we run benchmarks in SPEC CPU 2006 with variations of HUSHVAC where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.



(a) VMA count over time for `Xalancbmk`.



(b) Memory usage over time for `sphinx3`.

Fig. 22. VMA count and memory usage over time

**Impact on The Number of VMA Structures.** To confirm that our design choice that detaches physical pages promptly does not cause `VMA` explosion, we measure the number of `VMA` structures over time when running `Xalancbmk`, which is known to cause a large number of `VMA` structures. Figure 22(a) shows the number of `VMA` structures over time. The result shows that HUSHVAC successfully bounds the number of `VMA` structures thanks to its design choice of retaining the virtual pages being mapped, while FFmalloc suffers from the `VMA` explosion problem due to the `unmap` invocations.

**Impact of Sub-Page Reuse in Memory Usage.** We further highlight the impact of the sub-page reuse feature on one benchmark, `sphinx3`, from SPEC CPU 2006. Figure 22(b) shows the memory usage over time for `sphinx3`. The result from `sphinx3` in Figure 22(b) shows that enabling sub-page reuse reduces memory usage without incurring additional slowdown. HUSHVAC already exhibits considerably lower memory usage than FFmalloc, and the sub-page reuse feature further reduces memory usage. Unfortunately, the sub-page reuse feature does not eradicate the trend of increase in memory usage over time, suggesting that `sphinx3` has a heap usage pattern, causing a small number of chunks left over for a long time.

TABLE I. EFFECTIVENESS OF HUSHVAC IN PREVENTING USE-AFTER-FREE (UAF) EXPLOITS.

| Program | CVE ID | Bug Type | Original Attack | With the Protection of HUSHVAC |
|---|---|---|---|---|
| PHP 7.0.7 | CVE-2016-5773 [4] | UAF → double free | Arbitrary code execution | Exploit prevented & double free detected |
| PHP 5.5.14 | CVE-2015-2787 [1] | UAF | Arbitrary code execution | Exploit prevented & runs well |
| PHP 5.4.44 | CVE-2015-6835 [3] | UAF | Arbitrary memory disclosure | Exploit prevented & runs well |
| PHP 5.4.44 | CVE-2015-6834 [2] | UAF | Arbitrary memory disclosure | Exploit prevented & runs well |

### G. Effectiveness

In addition to the test using HardsHeap [38], we evaluate the effectiveness of HUSHVAC in preventing use-after-free exploits by using public proof-of-concept (PoC) exploits against some CVE-assigned public vulnerabilities in real-world applications. We use four CVE-assigned public vulnerabilities in three versions of PHP for evaluation because the corresponding public exploits are available. As Table I presents, HUSHVAC successfully prevents all four exploits from succeeding in achieving the goal, such as arbitrary code execution or memory disclosure. With HUSHVAC, the exploit's attempt to manipulate the behavior of code that uses a dangling pointer fails because the vulnerable heap chunk that the dangling pointer points to is not reused.

## VII. RELATED WORK

This work is closely related to previous efforts to reduce use-after-free vulnerabilities through garbage collection [8, 11, 18, 25, 30] and virtual address management [17, 36].

**Garbage collection.** This approach reclaims delay-freed memory chunks when there are no dangling pointers in the system. MarkUs [11] extended the Boehm-Demers-Weiser Garbage Collector (bdwgc) [8] with a quarantine list that indicates memory objects that are delay freed. Multiple concurrent threads scan the memory for the presence of dangling pointers in memory and register, after pausing the application threads. The frequency of this synchronous scanning offers a trade-off between longer execution time and lower memory usage. Minesweeper [18] performs a linear memory scan to identify quarantined objects with no dangling pointers. Unlike MarkUs, its marking procedure marks the presence of a dangling pointer on the shadow space corresponding to the entire memory of the application, by scanning the memory linearly. Minesweeper also proposed several optimization techniques, such as the prompt release of large chunks and the zeroing of freed chunks. However, Minesweeper still suffers from the stop-the-world delay when running mostly concurrent mode, and negatively affects the locality of heap chunks.

Besides the mark-and-sweep method, various techniques are proposed to detect dangling pointers with compile-time instrumentations. pSweeper [25] is one method that instruments code to detect pointer objects at compile time, checks their status in concurrent threads during runtime, and frees them. This method is similar to mark-and-sweep but is a stop-the-world-less approach to tracking live pointer objects. CRCount [30] is a technique that tracks pointer propagation by instrumenting code at compile time to count pointer copies during runtime. This method tracks the number of references by using a counter for each chunk, reclaiming the chunk only after the counter value becomes zero. These two approaches share a common trait: compile-time instrumentation results in execution time overhead.

**Virtual Address Management.** Recent studies have proposed virtual address management such as one-time allocation. This approach helps prevent use-after-free because the allocator never reuses the virtual address after freeing the memory chunk. The point of this approach is to make dangling pointers useless even if they exist. Oscar [17] is a similar design to one-time allocation with an object-per-page scheme using virtual shadow pages. In Oscar, each object has a virtual shadow page, but each shadow page is located in the same physical page frame and when unmapped, the physical frame is also unmapped. FFmalloc [36] provides a one-time allocation strategy that never reuses a virtual address. It incorporates fast continuous allocation and forward binning mechanisms to minimize memory and Virtual Memory Area (VMA) overheads, thereby facilitating one-time allocation. However, the non-reusability of the address space renders it impractical for long-running applications.

**Pointer Nullification.** Another approach to prevent use-after-free is pointer nullification when an object is freed. This approach follows the strategy of removing dangling pointers rather than checking if they exist. DangNull [23] establishes relationships between all pointers and objects and nullifies pointers when the corresponding object is freed. DangSan [34] uses shadow memory-based metadata inspired by log-structured file systems to detect dangling pointers in a multithread system. FreeSentry [37] follows a similar approach to DangNull, but instead of completely nullifying the pointer, it modifies the most significant bit, turning it into an invalid address. This technique aids in retaining the context when examining crash dumps.

**Access Validation.** This approach comprehensively examines all memory accesses to ensure temporal memory safety. However, this can have the disadvantage of increasing runtime overhead and potentially generating false positives. CETS [26] developed a solution that uses key and lock addresses to implement a separate metadata space for each pointer and check for pointer dereferences. Whenever a memory operation is performed, the CETS runtime system checks the metadata to ensure that the operation is safe. If the operation is unsafe, the runtime system generates an error. However, this task incurs a high-performance overhead for tracking pointer propagation. PTAuth [19] was proposed to support this challenge by defining the authentication codes (AC) to verify metadata integrity and identify when pointers are accessed. ViK [16] also identifies all allocation and pointer dereference sites statically and instruments code to track and examine metadata for use-after-free mitigation. PACMem [24] uses ARM's architectural feature called Pointer Authentication to reduce the overhead of pointer metadata tracking when defeating temporal safety violations.

## VIII. Conclusion

We proposed HushVac, which adopts an innovative approach with three key design choices to efficiently avoid use-after-free. The intriguing performance impact of two potential prevention mechanisms, mark-sweep and one-time allocation on an allocation-intensive benchmark, inspired the innovative approach in which HushVac uses a one-time allocator as an underlying allocator and carefully reuses freed virtual pages. Page-level reuse has shown to be a viable option because HushVac can detach the physical page without splitting the `VMA` structure, allowing HushVac to quarantine individual virtual pages. The lack of physical pages in the quarantine list also allowed the mark-sweep procedure to be triggered opportunistically. All of these design choices resulted in the HushVac design outperforming the status quo on nearly every benchmark.

## Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2022R1F1A1076100), the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program(IITP-2023-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), IITP grant funded by the Korea government(MSIT) (No.2021-0-00724, RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response), the 2023 Research Fund (1.230030.01) of UNIST (Ulsan National Institute of Science & Technology), and Samsung Electronics Co., Ltd.

## References

[1] "Cve-2015-2787," https://bugs.php.net/bug.php?id=68976, accessed: 2023-10.

[2] "Cve-2015-6834," https://bugs.php.net/bug.php?id=70365, accessed: 2023-10.

[3] "Cve-2015-6835," https://bugs.php.net/bug.php?id=70219, accessed: 2023-10.

[4] "Cve-2016-5773," https://bugs.php.net/bug.php?id=72434, accessed: 2023-10.

[5] "Cve-2018-12882," https://nvd.nist.gov/vuln/detail/CVE-2018-12882, accessed: 2022-01.

[6] "Cve-2018-12929," https://nvd.nist.gov/vuln/detail/CVE-2018-12929, accessed: 2022-01.

[7] "Cve-2021-30606," https://nvd.nist.gov/vuln/detail/CVE-2021-30606, accessed: 2022-01.

[8] "A garbage collector for c and c++," https://www.hboehm.info/gc/, accessed: 2022-01.

[9] "jemalloc memory allocator," https://jemalloc.net/, accessed: 2022-01.

[10] "Spec cpu® 2006," https://www.spec.org/cpu2006/, accessed: 2022-01.

[11] S. Ainsworth and T. M. Jones, "MarkUs: Drop-in use-after-free prevention for low-level languages," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[12] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages." in *PLDI*, M. I. Schwartzbach and T. Ball, Eds. ACM, 2006.

[13] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[14] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 157–164, 1991.

[15] J. Caballero, G. G. , M. Marron, and A. N. , "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA 2012 Proceedings of the 2012 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, July 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/undangle-early-detection-dangling-pointers-use-free-double-free-vulnerabilities/

[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] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[18] 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.

[19] R. M. Farkhani, M. Ahmadi, and L. Lu, "Ptauth: Temporal memory safety via robust points-to authentication," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[20] N. C. for Assured Software, "Juliet c/c++ 1.3," https://samate.nist.gov/SARD/test-suites/112, accessed: 2023-10.

[21] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 81–90.

[22] D. Jansens, "Supporting the use of rust in the chromium project," 2023. [Online]. Available: https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html

[23] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[24] 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.

[25] 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.

[26] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c." in *ISMM*, J. Vitek and D. Lea, Eds. ACM, 2010.

[27] G. Novark and E. D. Berger, "Dieharder: securing the heap." in *ACM Conference on Computer and Communications Security*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010.

[28] M. Parkinson, K. Vaswani, D. Vytiniotis, M. Costa, P. Deligiannis, A. Blankstein, D. McDermott, and J. Balkind, "Project snowflake: Non-blocking safe manual memory management in .net," Microsoft, Tech. Rep., July 2017.

[29] Z. Shen and B. Dolan-Gavitt, "Heapexpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 454–465. [Online]. Available: https://doi.org/10.1145/3427228.3427645

[30] 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 Symposium 2019*, 2019.

[31] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator." in *ACM Conference on Computer and Communications Security*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017.

[32] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, "Guarder: A tunable secure allocator," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 117–133. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro

[33] J. V. Stoep, "Memory safe languages in android 13," 2022. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[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*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 405–419. [Online]. Available: https://doi.org/10.1145/3064176.3064211

[35] S. J. Vaughan-Nichols, "Rust in the linux kernel," 2022. [Online]. Available: https://thenewstack.io/rust-in-the-linux-kernel/https://9to5google.com/2022/12/01/android-memory-safety-rust/

[36] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing Use-After-Free Attacks with Fast Forward Allocation (to appear)," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Vancouver, B.C., Canada, Aug. 2021.

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

[38] I. Yun, W. Song, S. Min, and T. Kim, "Hardsheap: a universal and extensible framework for evaluating secure allocators," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 379–392.