MineSweeper: A "Clean Sweep" for Drop-In Use-after-Free Prevention

Márton Erdős
University of Cambridge
Cambridge, UK
me412@cam.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, UK
sam.ainsworth@ed.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

ABSTRACT

Low-level languages, which require manual memory management from the programmer, remain in wide use for performance-critical applications. Memory-safety bugs are common, and now a major source of exploits. In particular, a *use-after-free* bug occurs when an object is erroneously deallocated, whilst pointers to it remain active in memory, and those (dangling) pointers are later used to access the object. An attacker can reallocate the memory area backing an erroneously freed object, then overwrite its contents, injecting carefully chosen data into the host program, thus altering its execution and achieving privilege escalation.

We present *MineSweeper*, a system to mitigate use-after-free vulnerabilities by retaining freed allocations in a quarantine, until no pointers to them remain in program memory, thus preventing their reallocation until it is safe. MineSweeper performs efficient linear sweeps of memory to identify quarantined items that have no dangling pointers to them, and thus can be safely reallocated. This allows MineSweeper to be significantly more efficient than previous transitive marking procedure techniques.

MineSweeper, attached to JeMalloc, improves security at an acceptable overhead in memory footprint (11.1% on average) and an execution-time cost of only 5.4% (geometric mean for SPEC CPU2006), with 9.6% additional threaded CPU usage. These figures considerably improve on the state-of-the-art for non-probabilistic drop-in temporal-safety systems, and make MineSweeper the only such scheme suitable for deployment in real-world production environments.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; *Systems security*.

KEYWORDS

temporal safety, use-after-free, programming language security

ACM Reference Format:
Márton Erdős, Sam Ainsworth, and Timothy M. Jones. 2022. 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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507712

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507712

1 INTRODUCTION

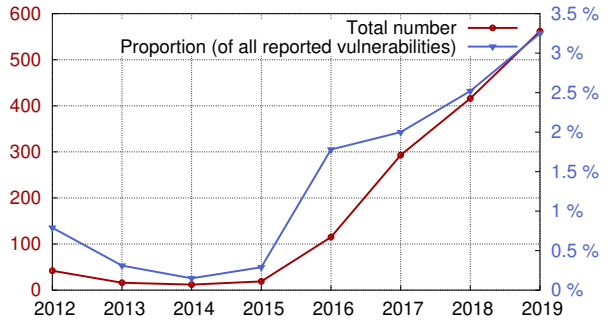
Low-level languages, including C/C++, remain popular [34, 43], especially for performance-critical and systems applications. These languages feature manual memory management: programmers are responsible for allocating objects (e.g. `malloc()` or `new`), and for deallocating them (e.g. `free()` or `delete`). This has led to widespread *memory-safety violations*. *Use-after-free* errors, in particular, are numerous [11, 17] and hard to find. These occur if the program frees allocations, but retains pointers to them, then uses these (now-)dangling pointers to perform (erroneous) memory accesses. Attackers can reallocate erroneously freed allocations and write carefully crafted data into them; a common example is diverting control on a virtual-function call. As other vulnerabilities get mitigated, attackers have turned to use-after-free as the easiest target, leading to a consistent rise in exploits (Figure 1).

We demonstrate with MineSweeper that it is finally possible to eliminate use-after-free vulnerabilities in software with low enough overhead [33] for use in production systems, and with no source-code modification. MineSweeper intercepts the programmer's freed allocations, and quarantines them until they can be demonstrated safe. Periodic, simple linear sweeps of program memory are used to identify which parts of the quarantine have dangling pointers, and so cannot safely be reused, replacing the complex garbage-collector-style mark-and-sweep used in existing mechanisms [2].

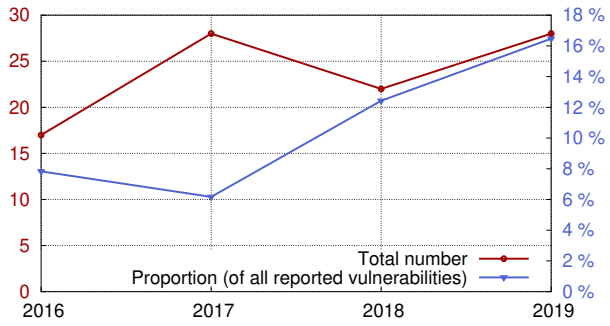
1.1 Contributions

- MineSweeper demonstrates that a simple linear sweep of memory can efficiently locate dangling pointers.
- Zero-filling memory in `free()` makes these linear sweeps feasible: this removes dangling pointers from quarantined allocations, thus *flattening* the reference graph and breaking circular dependencies.
- We offer two operation modes, performing mostly or fully concurrent sweeps. The former matches security guarantees of previous systems while improving performance; the latter proposes a novel way to reduce overheads even further by eliminating stop-the-world periods, at the cost of a very minor relaxation of guarantees.
- We present a drop-in[1] approach to mitigating heap temporal-memory-safety violations, with an efficient implementation, on top of JeMalloc [15], implementing a) concurrent, non-blocking sweeps with respect to the program, b) parallel

[1] Drop-in: without the need for hardware support or recompilation, and preserving compatibility with otherwise correct programs.



(a) Use-after-frees in the National Vulnerability Database [27]



(b) Use-after-free vulnerabilities in the Linux kernel [27]

Figure 1: Reported use-after-free (CWE416) / double-free (CWE415) vulnerabilities by year.

sweeping using multiple background threads, c) thread-local quarantine buffers to reduce lock contention, d) unmapping of physical memory for large quarantined allocations, and e) enhanced allocator cleanup to reduce the impact of fragmentation on memory overheads.

MineSweeper significantly outperforms any other non-probabilistic Standard-C-compatible temporal-safety system. The geometric mean slowdown for SPEC CPU2006 is only 5.4 % with just 11.1 % memory overhead (or 8.2 % slowdown at 11.7 % memory for the mostly concurrent version).

1.2 Threat Model

We assume that a non-malicious application executes in the presence of an attacker. The application may contain any number of *temporal-memory-safety violations* on the heap. *Spatial-safety bugs*, such as buffer overflows, are orthogonal, thus not addressed by MineSweeper. Stack temporal-safety violations are rare and amenable to static analysis [12, 36], and not addressed by similar techniques [2, 17, 37, 39, 41].

The attacker can allocate memory and store data to regions they have allocated. For example, indirectly, through carefully constructed inputs, or a script running in a sandbox environment. The attacker's goal is to extend their control over the host application (perform privilege escalation), for example, by redirecting control flow and executing arbitrary instructions, or engaging in Return-Oriented Programming (ROP). A typical example would be a script

```
Object *x = new Object();
// ...
delete x;
// ...
x->fn(); // Use-after-free error!
```

Listing 1: Use-after-free vulnerability in C++.

on a malicious website aiming to break the browser's sandboxing mechanism in order to install malware. They will succeed at this if they are given control of an allocation that *temporally aliases* with a different allocation at a different program point, where the pointer has been falsely freed and reallocated despite still being in use, and can then change the data within.

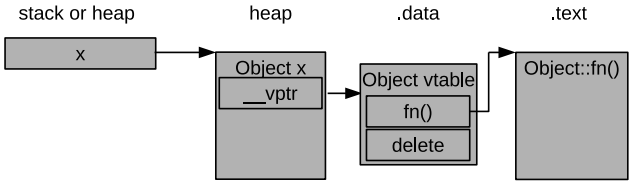
MineSweeper is designed to find pointers that are correctly aligned, and not hidden such as by xoring [32] them with other data. It maintains the property that, should such a pointer exist to an object freed by the programmer, this object will be kept in quarantine and not reused by any new allocation, thus preventing old objects and new objects from aliasing each other. MineSweeper *preserves compatibility* with hidden and misaligned pointers; while these are rare, they are common enough that not supporting them breaks real programs [2]. MineSweeper achieves this by not freeing anything the programmer has not requested to be freed, unlike garbage collectors [8], which are unsafe in C/C++ without additional provenance tracking and sticking strictly to the C/C++ standard [5], and significantly more expensive than MineSweeper [2, 5]. It gives no additional *security* guarantees for objects whose only pointers are hidden. It also does not guarantee that an object cannot be read or written to while in quarantine; as with recent work [2, 21, 37, 39, 41], our approach prevents *use-after-reallocate*, turning bugs into *benign use-after-free* or clean termination, thus preventing exploits. This means that MineSweeper aims to *mitigate*, not *debug*, use-after-frees, unlike MTE [4] or AddressSanitizer [29], though it could be combined with these mechanisms to raise their level of guarantees.

We present two implementations: a mostly concurrent [8] version that guarantees to find all such pointers accessible within the program, and a fully concurrent version that guarantees that it will find all dangling pointers that have not been moved or copied after their referenced objects have been freed.

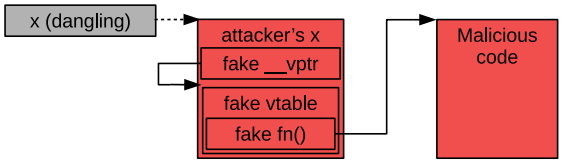
2 BACKGROUND

A *memory-safety violation* occurs when a program attempts to access or deallocate memory that is not allocated to the intended object (i.e. unallocated or in a different allocation). We can distinguish *temporal* and *spatial* violations; the former have become a bigger problem [17] in frequency and severity. A *use-after-free bug* is a *temporal-memory-safety violation* occurring when the program accesses memory that *used to belong* to the intended object, but no longer does (the object was deallocated). An example is in Listing 1.

These bugs are often hard to find. The point of deallocation and use may be located far away, particularly if performed via different references. Hence it can be challenging to find [35] use-after-free errors using static analysis or traditional debugging techniques. Until the memory range is unmapped or reallocated by the memory allocator, accesses will succeed, hiding the bug.



(a) Initial structure (before the call to `delete`)



(b) Structure after reallocation by the attacker. A call to `x->fn()` now executes the malicious code

Figure 2: Exploit for the use-after-free in Listing 1.

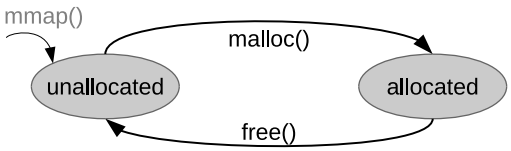
A *use-after-reallocate bug* occurs if the illegal reuse (after free) is performed after the (virtual) memory region has been reassigned to a new allocation. *Use-after-free* in itself is either benign[2] or results in a memory-protection violation, thus immediate clean termination. However, if an attacker can turn it into a *use-after-reallocate*, then the program may corrupt its state (memory content and control flow) in a way controlled by the attacker. A common [2, 41] exploit is shown in Figure 2. The program erroneously frees an object *x*, but keeps a dangling pointer. The attacker then requests objects of the same size, and fills them with carefully constructed data: fake virtual-function tables, pointing to malicious code. One such allocation is likely to reuse the (virtual) memory range of *x*. Finally, the program performs a virtual-function call using the dangling pointer, jumping to the malicious code (e.g. a sequence of libc gadgets [24] to bypass memory protection).

3 MINESWEEPER

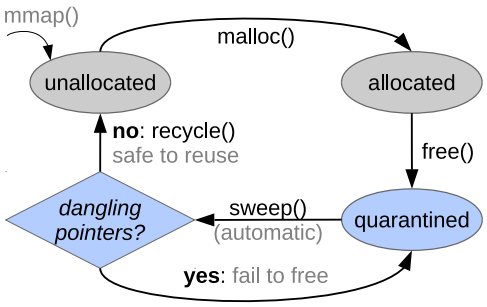
MineSweeper is a system for mitigating use-after-free vulnerabilities in production systems. It requires no changes to the applications it protects, even legacy programs, since it improves security without changing program semantics. Memory can be safely recycled (and eventually reallocated) after it has been `free()`'d by the programmer *and* there are no (dangling) pointers to it left in memory. The latter can be discovered cheaply via periodic sweeps of memory.

Allocations wait in *quarantine* until the absence of pointers to them can be demonstrated. *Sweeps* of program memory are used to test this: each word of memory is checked for potential dangling pointers to quarantined allocations. If a sweep encounters no pointers to a given quarantined allocation, the allocation is released from quarantine. Allocations failing this test remain quarantined until demonstrated safe by future sweeps. In our JeMalloc implementation, the quarantine is an agnostic layer above the allocator, interfacing with the public `je_free()` API only upon verification.

[2]In non-secure allocators that store metadata in-place (e.g. GNU malloc), this may corrupt allocator metadata. JeMalloc, which MineSweeper is built upon, already stores metadata separately to avoid this.



(a) Regular system



(b) MineSweeper

Figure 3: Life-cycle of memory allocations.

Rather than sweeping after every single call to `free()`, MineSweeper only performs a sweep once the total size of quarantined allocations exceeds a certain threshold (relative to the total memory use of the application). This way, *the cost of sweeping is amortised over a number of allocations*. This causes a trade-off between memory and run-time overhead.

MineSweeper guards against use-after-reallocate errors by keeping memory live in the quarantine until the application overwrites or deallocates all dangling pointers to it, thus making it inaccessible. It also protects against double frees, by making calls to `free()` — while a dangling pointer exists — idempotent from each other, by de-duplicating[3] quarantine entries by keeping a shadow map of entries that picks up duplicates, and only invoking one true `free()`.

We next give a high-level description of the algorithm and implementation details. Section 4 describes optimisations.

3.1 Overview

MineSweeper adds a layer between the application and the memory allocator (Figure 3). It intercepts calls to `free()`, and delays actual deallocation until a sweep demonstrates the absence of (dangling) pointers to the allocation. When an allocation is passed to `free()`, it is quarantined until a sweep demonstrates its safety. The quarantine is a list of allocations; quarantining simply means registering an allocation in this list. Accesses to quarantined allocations are use-after-free errors MineSweeper successfully prevented.

Sweeps are executed periodically, when a given proportion of the heap is in quarantine. A sweep is performed in three phases: the first scans all active memory for potential pointers, and records their targets in a shadow map (Figure 4 and Section 3.2). The second (optional) phase briefly stops the world, re-checking modified pages during the first pass, to give total coverage of any dangling pointers that may have moved. The final pass iterates through quarantined

[3]In debug mode, it reports double-frees.

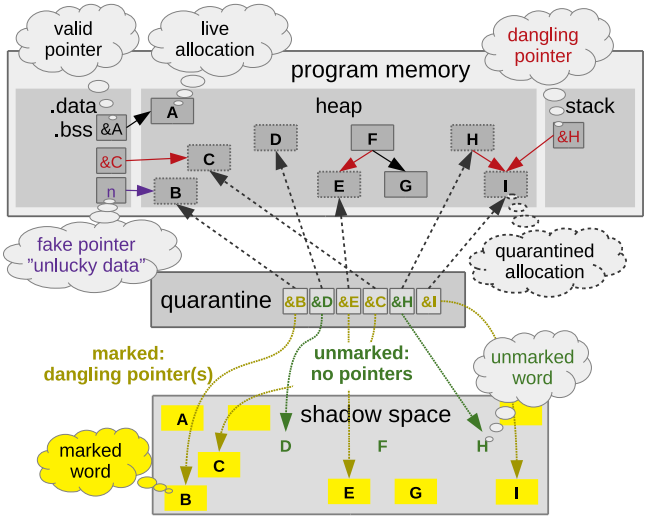


Figure 4: MineSweeper after the first (marking) phase of the sweep. This checks all program memory (top) for pointers. Words of memory targeted by pointer(s) are marked (yellow) in the shadow space (bottom). Allocations in quarantine that do not contain marked words (green) have no pointers, so are safely deallocated in the second phase. If a word of data (n) happens to equal the address of an allocation, then it is a 'false pointer' (purple), preventing deallocation.

allocations, using the shadow map to identify allocations with dangling pointers to them. Such allocations *fail to free*, and stay in quarantine until a future sweep demonstrates them safe. The rest are released from quarantine and recycled.

3.2 Base Design

We implemented MineSweeper as a layer over the top of JeMalloc [15]; though much of the implementation is allocator-agnostic, MineSweeper hooks into the allocator's public API and slightly extends it to efficiently identify active memory ranges, exclude allocator metadata structures, and control JeMalloc's memory cycle to limit fragmentation. We also add 1 B to the requested size of each allocation, so that C/C++ `end()` pointers lie within the same allocation, since one-past-the-end pointers are considered valid references to objects in the C/C++ standard [10].

Shadow Map. The shadow map marks the targets of pointers, and is consulted for each quarantined allocation, to see if pointers have been discovered to it, preventing recycling.

The shadow map is, conceptually, an array of bits, containing one bit per *granule* of virtual memory. Figure 5 shows how the shadow space finds dangling pointers. In the first phase of the sweep, each word p of memory is interpreted as a pointer, its 'granule index' $g(p)$ is calculated and used to index and set the shadow-map bit. In the second phase, for each quarantined allocation, the corresponding shadow bits are checked. If any of them are set, then there is a (possible) dangling pointer to the allocation (including those pointing to a location at an offset *inside* the allocation), and the allocation is left in quarantine. Otherwise, it is deallocated.

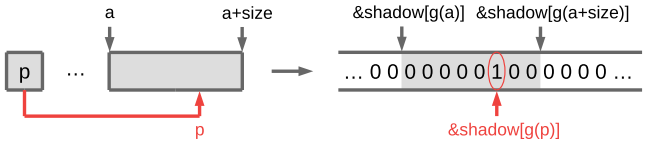


Figure 5: Operation of the shadow space. For any p pointing into the allocation starting at address a and of size $size$ there is a corresponding mark bit.

This is implemented as a flat space, as in previous work [37, 41], with one bit per every 128 bits; the smallest allocation granule. This is sufficient to uniquely distinguish each allocation; theoretically, we could use a less precise shadow map, at the expense of some aliasing between adjacent allocations and less efficient reuse of memory, but this shadow space is already small (less than 1% overhead).

When to Sweep. A sweep is triggered if a certain proportion of the heap is in quarantine. While previous work [2] chose a value of 25 %, targeting a memory usage increase of a third, MineSweeper targets 15 % instead, because its efficient sweep allows it to trade off towards being more aggressive at limiting memory overheads. Still, this is configurable.

Failed frees are subtracted from both sides, because these often fail again in the next sweep, and otherwise could cause sweeps to be performed very frequently: for example, if at least 15 % of the heap is taken up by failed frees, then a sweep would be performed on every `free()`. Ignoring failed frees means that maximum memory overhead can exceed the target, especially in the presence of dangling pointers causing failed frees. The allocator itself may also cause larger or smaller overheads, due to fragmentation from the quarantine, and we shall see that allocators typically need to be made aware of the quarantine cycle if these factors are to be minimised. Still, large overheads from such factors are rare [7].

3.3 Compatibility

MineSweeper preserves compatibility with existing C/C++ applications. Thus, it cannot alter the semantics of memory allocator functions (e.g. `free()`). Achieving this makes approximation inevitable. Although it can never recycle an object not freed by the programmer, MineSweeper both over- and under-approximates:

- Since pointers cannot be distinguished from arbitrary data, MineSweeper considers each (aligned) memory word as a pointer. This may lead to 'unlucky' data preventing deallocation: an integer X can cause the allocation containing address X to be left in quarantine.
- The application might hide pointers (using pointer arithmetic, e.g. in XOR lists, or by writing into a file), so MineSweeper may miss pointers during the sweep. Note, however, that C-standard-compliant pointer arithmetic, including past-the-end pointers, will have no impact on coverage[4].

Neither approximation is novel to MineSweeper. The former is used by conservative garbage collectors [8] and MarkUs [2]. Its performance impact is limited by the sparsity of the address space.

[4]We check the full shadow-map range corresponding to the allocation before recycling it. We serve each allocation request with an allocation at least 1 B larger than the requested size to accommodate past-the-end pointers.

The latter is used by all drop-in use-after-free prevention schemes based on passive revocation [2, 8, 30] or pointer nullification [17, 21, 35, 42]. Its coverage impact is negligible, due to extremely low prevalence [2] and because the attacker has no control over it.

4 OPTIMISATIONS

This section describes optimisations and improvements to the above base design. Locations freed by the programmer are zeroed, to eliminate circular references on the quarantined heap. Large deallocations are immediately unmapped in quarantine, to prevent their consumption of space and their usage of operating-system state. The sweep threshold is adapted to reflect this. MineSweeper is adapted to utilise a parallel and concurrent sweeping mechanism, moving the bulk of the overheads to background sweeper threads. Finally, we adapt JeMalloc's memory cycle to be better adapted to quarantine, siginficantly reducing fragmentation in the process.

4.1 Zeroing

Allocations in quarantine could still contain pointers, thus delaying, or altogether preventing, the reuse of other allocations (or themselves). This means that, without modification, whenever a quarantined allocation contains a pointer, the pointed-to allocation cannot be deallocated before the allocation containing the pointer. Worse still, any cyclic pointer structures entering quarantine will never be deallocated.

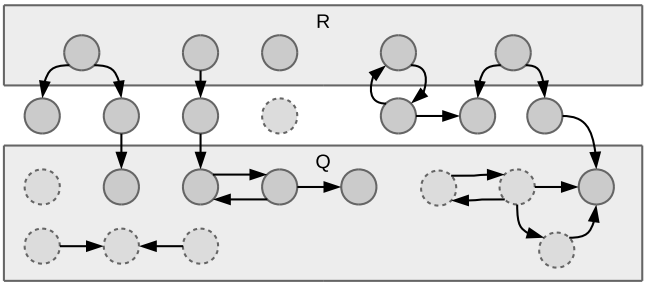
Mark & sweep garbage collectors [8], and MarkUs [2], solve the problem by using a *transitive* marking procedure. The problem solved by marking, on the algorithmic level, is *reachability* (from a set R of starting nodes, the 'root set') on a multi-level (and cyclic) *reference graph*.

MineSweeper only needs to solve a special case. We can assume that all nodes outside the quarantine (Q) are reachable and only elements accessible from outside Q need to be found (transitively) from edges into Q. This is *reasonable*, as the only unreachable nodes outside Q correspond to *leaked* memory, the amount of which should be low in working programs. It is *safe*, because it leads to an over-approximation: some memory may fail to free (due to dangling pointers within leaked memory), but this will never cause MineSweeper to deallocate memory with dangling pointers.

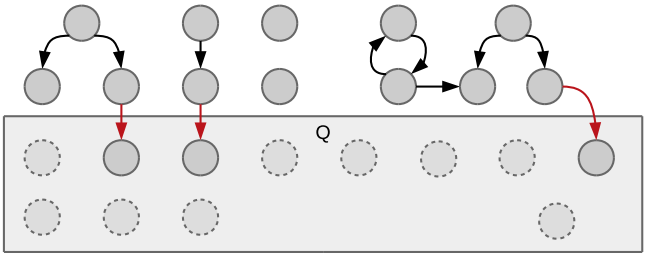
Rather than a transitive procedure, we use a linear sweep of memory to find all edges ending in Q (and starting anywhere). This captures all dangling pointers, but also pointers inside Q, which can cause cyclic references that prevent mutually referenced quarantined nodes, inaccessible in practice, from being freed [30]. We eliminate these, and other dangling pointers in Q, by using the programmer's `free()` to zero any data placed in quarantine. Figure 6 shows the simplification.

4.2 Page Unmapping

For quarantined allocations, the associated virtual-memory range cannot be safely reused until a sweep shows the absence of dangling pointers to it. However, the corresponding *physical memory* can be released without affecting correctness. If a quarantined allocation spans full pages of memory, then MineSweeper releases its *physical* pages, as in MarkUs [2].



(a) Original problem (Mark & Sweep GC [8], MarkUs [2]). Solution: transitive closure of R.



(b) Edges from Q deleted. Solution: nodes in Q with incoming edges, by linear sweep (and nodes outside Q).

Figure 6: A simplification of the marking problem.

MineSweeper invokes system calls to protect unmapped pages from accesses, to prevent dangling pointers to be written to them, and excludes them from sweeping. They do not count towards standard memory usage or quarantine-size sweep thresholds. Still, in order to keep pressure on kernel and allocator data structures low, we still initiate a sweep once a proportion of the quarantine equivalent to nine times the program's total physical-memory footprint is unmapped.

4.3 Concurrency

The sweep is largely independent of the application thread: to limit its performance impact, we can run it concurrently with the application. Still, the sweep should see a consistent picture of memory to avoid pointers being lost[5].

The standard solution from garbage collectors is to perform a mostly parallel pass [8]: at the start of the procedure, all pages are marked as unwritable, and a signal handler catches writes to them and adds the pages to a list. A second stop-the-world pass checks only these modified locations. In order to give exactly equivalent guarantees to MarkUs [2], we implemented a similar technique. Since signal handling in this way caused undefined behaviour [23] in our test set, we used a more modern alternative to record the dirty pages: namely, the *soft-dirty* support in Linux [14], which uses the operating-system kernel to directly record modified pages.

Still, as MineSweeper, much like any drop-in technique [2], must approximate already (Section 3.1), we consider this brief stop-the-world to be surplus to requirements, and do not recommend its

[5]This would be caused when the program moves the only copy of a dangling pointer from address B to some address $A < B$ in memory, while the sweep is between A and B, then deletes the copy from B; the sweep sees the pointer in neither A nor B.

use by default. MineSweeper can afford to relax these guarantees further than garbage-collector-based systems [8], precisely because it is *not* a garbage collector. It guarantees that only locations the programmer has intentionally placed in quarantine can be freed, avoiding breaking correct programs.

Without the stop-the-world, MineSweeper only sweeps memory once, without necessarily getting a fully consistent image. It guarantees that all pages that are live when the allocation enters the quarantine (and remain live) are examined during the sweep that recycles the allocation. To provide this, the sweep only recycles allocations already in quarantine when it starts; any allocations placed in quarantine between the start and end of a sweep can only be recycled by a future sweep.

The lack of stop-the-world only changes MineSweeper's properties when the programmer *moves around* dangling pointers, already in quarantine, before using them. We view this as unlikely, and an attacker is unlikely to be able to introduce dangling-pointer movement themselves. Unless the threat model is especially paranoid, it is sufficient to capture dangling pointers that only move *before* being incorrectly freed.

4.4 Parallel Sweeping

To limit the performance impact further, the marking phase of the sweep is parallelised to multiple threads. This is especially important in the mostly concurrent mode of operation, as a faster stop-the-world (re)sweep directly reduces run time. However, faster concurrent sweeps are also beneficial, as they mean allocations are recycled more promptly, thus reducing memory impact, and allocator (metadata) pressure.

We implement parallel sweeping using a main sweeper thread and some (6 by default) helpers. When the program thread adds entries into the global quarantine, it checks the sweep criteria, and signals the main sweeper if a sweep is required. The sweeper then "locks in" the set of allocations that are considered for recycling in the sweep (placing any subsequent frees into a new quarantine for the next sweep), divides up the memory to sweep (heap, stack and globals) equally, and dispatches the helpers. Once the sweeper and all helpers finish sweeping, the sweeper divides up the quarantine list equally among them, and the helpers recycle unmarked allocations in their part of the list. The list is then compacted by the main sweeper thread.

4.5 Fragmentation Management

MineSweeper needs to keep track of active and released pages in order to sweep the correct regions of memory. By default, JeMalloc uses `madvise()` to release ('purge') unneeded physical memory, and relies on OS demand-allocation when it is needed again. We cannot safely disregard these purged pages from sweeps, as a temporal-memory-safety violation could result in pointers being written to them. However, a sweep would trigger demand-allocation, thus defeating the point of purging in the first place. Therefore, we hook onto JeMalloc's extent management via the *extent hook* API to modify this behaviour; instead of a purge call and demand-allocation, we use a pair of calls: `decommit` and `commit`. On `decommit`, we mark the page as unmapped in a small shadow bitmap, discard the backing physical memory, and protect the range against accesses. On `commit`, we clear the mark in the shadow bitmap, and restore the original access protections of the page.

JeMalloc walks its own structures to purge unneeded extents periodically, using a 10-second sigmoidal decay curve. MineSweeper modifies this to trigger a full purge after *every* sweep, using the background sweeper thread. As with the small-block sweeping mechanism in MarkUs [2], it appears that allocators with large, variable-sized quarantines must clean their free structures more aggressively than those without, and do so in a way that synchronises with the end of a sweeping procedure. MineSweeper implementations that do not integrate with allocator-purging mechanisms risk high memory consumption.

5 EVALUATION

First, we present the performance impact of MineSweeper on SPEC CPU2006 (Section 5.2), comparing to other schemes in the literature. This is followed by the performance of the mostly concurrent version (Section 5.3), and analyses of optimisations (Section 5.4) and sources of overhead (Section 5.5). Finally, we look at multi-threaded workloads (Section 5.6).

5.1 Setup

The figures were produced on a desktop machine featuring an Intel Core i7-7700 CPU (3.6 GHz base, 4.2 GHz turbo boost, 4 cores, 8 threads, 8 MiB L3 cache, DDR4 RAM), running Ubuntu 16.04 LTS. We evaluate over SPEC CPU2006 [16] and SPECspeed2017 [9] benchmark suites (using reference inputs), with memory usage collected using PSRecord [**?**]. We load JeMalloc and MineSweeper as shared libraries. We use the fully concurrent version of MineSweeper throughout, except in Section 5.3. We reran the open-source implementations [1, 38] of the two previous best schemes — MarkUs and FFMalloc — on our system, and compare slowdown and memory overheads to these. We use the version with unmodified JeMalloc loaded as a baseline for these three techniques, as doing so avoids using JeMalloc's 2 % geometric mean speedup to mask our overheads, while using the same baseline to compare the three relevant techniques. The allocator is an inherent part of FFMalloc and MarkUs, while MineSweeper can freely choose a high-performance allocator, such as JeMalloc. We use the figures as reported in the relevant papers for the other schemes. These use slightly more flattering baseline of GNU malloc, but this makes little difference.

5.2 SPEC CPU2006

Figure 7 compares the slowdown of MineSweeper on SPEC CPU2006 to other techniques. Figure 9 zooms in on MarkUs [2], FFMalloc [39] and MineSweeper. Figure 10 compares memory overheads.

MineSweeper performs significantly better than all other techniques on allocation-heavy workloads, and shows little to no overhead for other benchmarks. MineSweeper only suffers a geometric mean slowdown of 5.4 % (worst case: 72.7 % for `xalancbmk`), with a geomean memory footprint increase of only 11.1 %. Together, these show that MineSweeper clearly outperforms the state-of-the-art.

FFMalloc causes lower slowdown (3.5 %) than Minesweeper, but at the cost of adverse memory behaviour (244 % overhead, 1,070 %

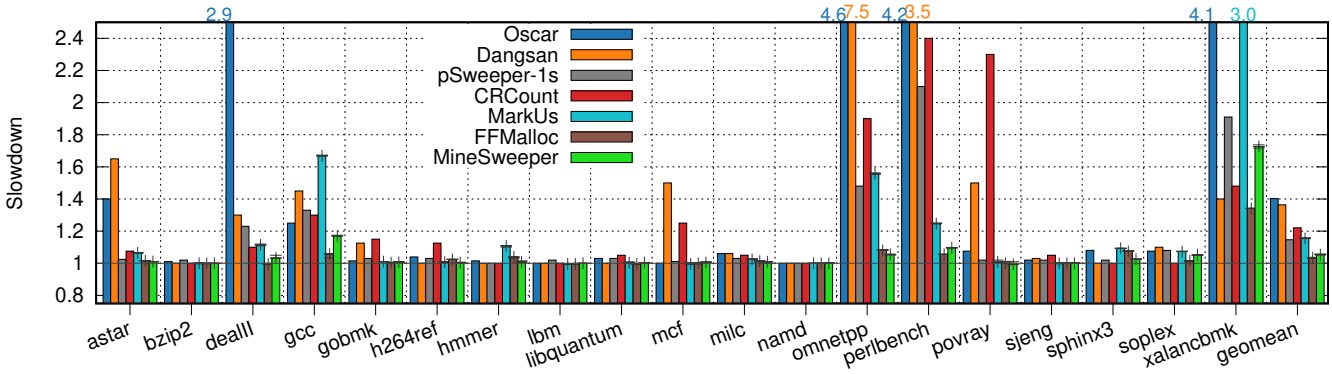


Figure 7: Slowdown for SPEC CPU2006 (C/C++), compared to other systems [2, 12, 21, 30, 35].

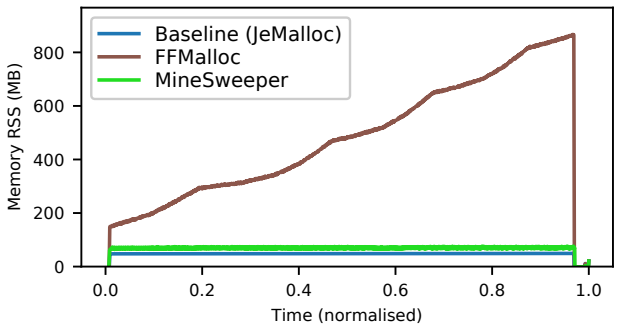


Figure 8: Memory usage over time for sphinx3

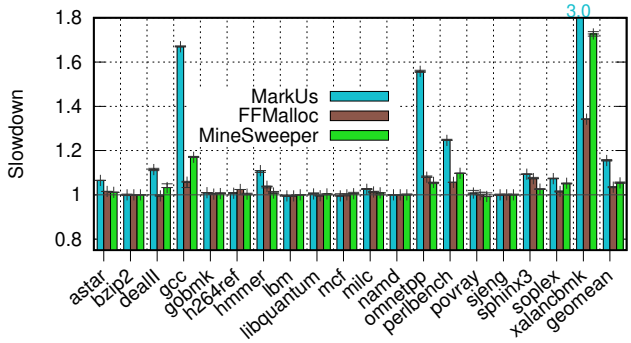


Figure 9: Slowdown versus MarkUs [2] and FFMalloc [39].

worst case). FFMalloc is a page-table-protection method, implementing a custom one-time allocator from scratch. The allocator never reuses the same virtual-memory range; virtual memory is always mapped in increasing order of addresses. Once all allocations from a page are `free()`-d, the physical page is unmapped. For multiple workloads (`perlbench`, `omnetpp`, `xalancbmk`, `sphinx3`), FFMalloc changes the memory usage from largely constant over time to constantly increasing, which hints that fragmentation is the main cause for the extremely high memory usage. We show the trace over time for `sphinx3` in Figure 8.

MarkUs achieves 15.5 % mean slowdown (2.97× worst case), at 12.3 % average memory overhead. MarkUs is a garbage-collector-inspired use-after-free mitigation. When the programmer's quarantined frees take up 25% of the total heap, a marking pass is performed via the Boehm Garbage Collector [8], a widely known conservative collector for C/C++, followed by a quarantine-list walk to deallocate unmarked allocations. MineSweeper demonstrates that linear sweeps of memory are more efficient, while solving the problem of circular references caused when not using a transitive marking procedure. Unlike MarkUs, MineSweeper is implemented in a state-of-the-art allocator [15]. These factors allow MineSweeper to substantially outperform MarkUs.

MineSweeper demonstrates overheads that makes it the only scheme suitable for wide-scale deployment, exhibiting consistently low overhead both in time and memory.

Run-Times. MineSweeper only exhibits slowdowns above 5 % for five workloads, all allocation intensive: `xalancbmk` (73 %), `gcc` (17 %), `perlbench` (9.7 %), `omnetpp` (5.6 %) and `sphinx3` (5.2 %).

Memory Usage. Figure 11 shows memory-usage overheads. The increase in both average and peak memory usage is shown; the average can be interpreted as additional RAM usage when running many 'small' applications side-by-side, while the peak shows the increase in RAM required for running one 'large' application. Overheads are reasonable: the worst case is 62.7 % (`gcc`) for the average, and 93.4 % (`gcc`) for the peak. The geometric mean increase is only 11.1 % average, and 17.7 % peak.

CPU Utilisation. Another source of overhead is increased CPU utilisation, given we sweep in a dedicated thread. This could affect other workloads on a multi-core system. Figure 12 shows the additional average CPU usage resulting from adding MineSweeper. The maximum overhead is 129 %, for `xalancbmk`, but the geometric mean is only 9.6 %; while sweeping uses some CPU time, these costs should be acceptable in modern multi-core systems.

DRAM Traffic. There is a worry that accesses made during our sweeps could evict program data from all caches, leading to an increased DRAM bandwidth utilisation, potentially affecting other programs on the same system. We measured DRAM bandwidth impact of Minesweeper, and we found no significant change. This is likely because sweeps are too infrequent to evict program data from the last-level cache.

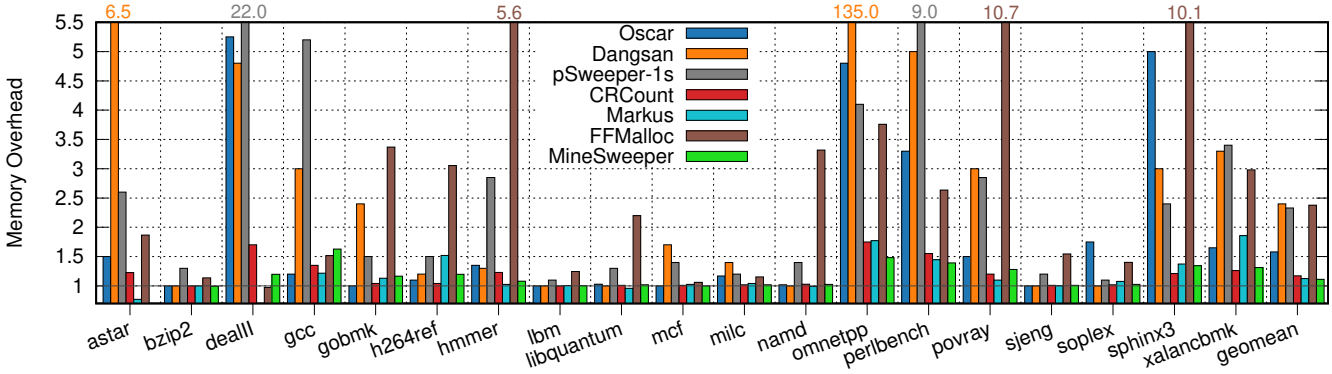


Figure 10: Average memory overhead for SPEC CPU2006 (C/C++), compared to other systems [2, 12, 21, 30, 35]

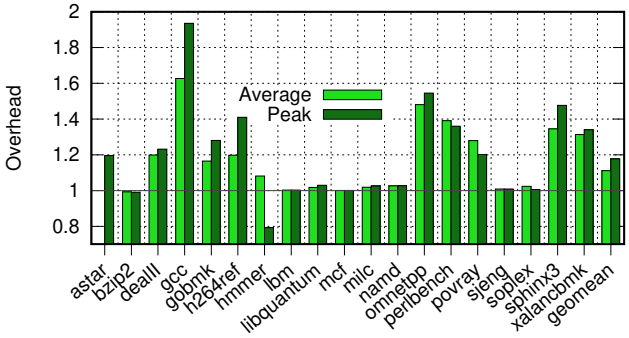


Figure 11: Memory overhead for SPEC CPU2006

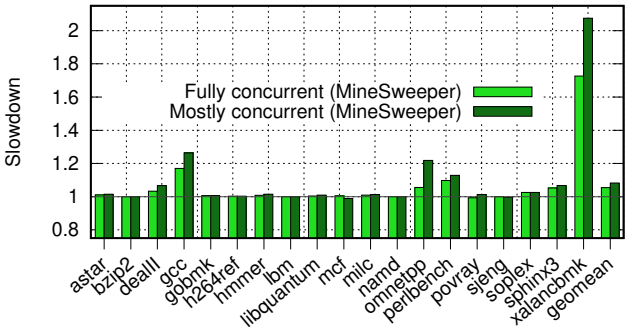


Figure 13: Slowdown of fully concurrent and mostly concurrent (stop-the-world) versions

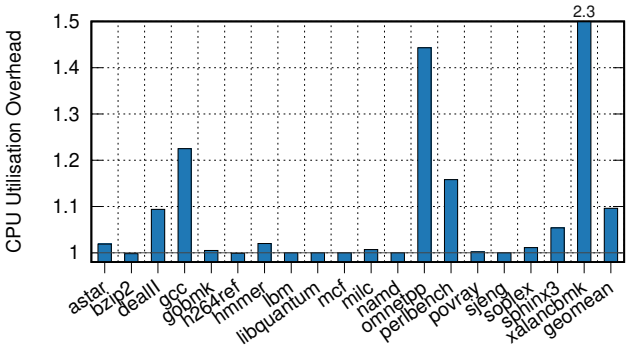


Figure 12: Additional CPU usage

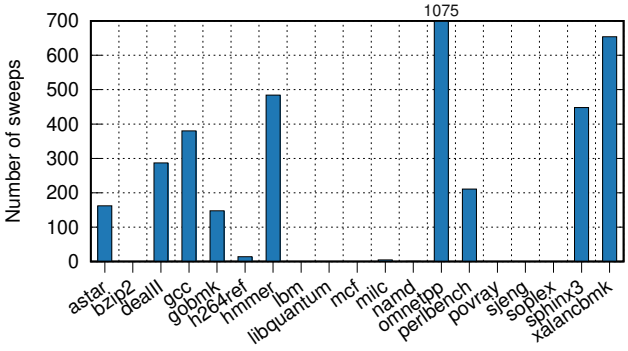


Figure 14: Number of sweeps triggered

Sweep Counts. To put the overheads in context, we show the number of sweeps triggered by the fully concurrent version per benchmark in Figure 14. Several benchmarks in SPEC are very allocation-heavy. The benchmark with the highest number of sweeps (1,075) is `omnetpp`. `Xalancbmk` is also very sweep-intensive: it triggers 654 sweeps, and almost all of them happen close together, towards the end of the benchmark. The fact that the number of sweeps does not correlate perfectly with slowdown figures shows that sweeping is not our only overhead, and not always the main one (Section 5.5).

5.3 Mostly Concurrent Version

The version with mostly concurrent sweeping — instead of fully concurrent — provides slightly stronger security guarantees: the sweep is guaranteed to find any dangling pointers (not obscured via pointer arithmetic), even if the program moves them around. This comes with a slightly higher slowdown of 8.2 % and a similar memory cost of 11.7 %, as shown in Figure 13. This version provides the same guarantees as MarkUs at half the time cost (and similar memory overhead).

5.4 Optimisations

Optimisations play an important role in reducing the overheads of MineSweeper. Figures 15 and 16 show the run-time and memory-usage overheads, respectively, after applying optimisations one-by-one. Since optimisations affect each other, we put them in decreasing order of estimated importance.

Unoptimised. This version exhibits very high overheads for the allocation-intensive benchmarks. The lack of zeroing leads to failed frees and together with the lack of unmapping, this leads to a high increase in memory footprint. Since sweeping is not offloaded, this also translates to a high time cost for allocation-intensive benchmarks (where sweeps are triggered frequently). `Gcc` and `milc` exhaust the available (32 GB) DRAM, and get terminated.

Zeroing. Zeroing cuts memory usage by increasing the amount of memory that can be reclaimed during sweeps (Section 4.1), by removing pointers from within quarantined data itself. It also improves execution time, by reducing the memory footprint: this reduces pressure on metadata structures in the kernel, allocator, and MineSweeper. `Gcc` still uses too much memory to finish.

Unmapping. Unmapping highly reduces memory overheads, especially for benchmarks where larger allocations are prevalent. The geometric mean memory overhead reduces to just 21.1 %. Unmapping saves memory by releasing pages early. This is especially important for `gcc`, which is cut from over 50× overhead (killed after using 30 GiB of memory) to just 1.4×. Overall, this sequential version gives a 9.5 % time and 21.1 % memory overhead.

Concurrency. Offloading the sweep to a dedicated sweeper thread trades off a run-time overhead reduction for a negative impact on memory overhead. Increased memory overheads stem from the fact that recycling of memory is delayed relative to the application thread (which can make progress during the sweep). The time savings show the effectiveness of offloading the sweep. Time overhead is cut to 5.0 % but memory overhead increases to 24.1 %.

Purging. Triggering a cleanup operation in JeMalloc after each sweep incurs a time cost (to 5.4 % slowdown), but significantly decreases memory usage to 11.1 %.

5.5 Source of Overheads

We test some 'partial' versions — only performing a subset of the MineSweeper functions — to find out where overheads originate from. Figure 17 shows overheads on five of the most affected benchmarks under the following versions of MineSweeper, incrementally adding features as follows:

(1) Base overheads: the MineSweeper library is loaded, data structures are initialised and maintained, but `free()` simply forwards the allocation to JeMalloc for recycling.
(2) Unmapping + Zeroing: on `free()`, we unmap (and immediately remap) large enough allocations, and zero-fill small ones, then forward the allocation to JeMalloc for recycling.
(3) Quarantining: `free()` quarantines allocations until the next sweep, which simply recycles all quarantined items, in the application thread.
(4) Concurrency: same as quarantining, but recycle in the sweeper thread (still without sweeping). This adds the overheads of thread management and thread-local cache movement.
(5) Sweeping: quarantine, sweep memory, check which frees would fail, but deallocate regardless.
(6) Full version, leaving failed frees in quarantine.

Base overheads are negligible (1.1 % time, 0.2 % memory), unmapping and zeroing cost some time, but save memory (5.8 % time, –2.7 % memory), then quarantining adds the time overhead, and increases memory usage (17.9 % time, 14.8 % memory). This is mostly due to delay-of-reuse, and thus worse cache utilisation. The remaining sources of overhead gradually add the remaining memory usage impact, reaching 39.4 % in the full version (for these 5 benchmarks).

5.6 SPECspeed2017

Figure 18 shows the results of applying MineSweeper to more modern workloads, under highly threaded code. For multi-threaded workloads, we used the best of the 4-thread and 8-thread configurations for both the baseline and the MineSweeper (and MarkUs, FFMalloc) versions, separately. MarkUs [2] and FFMalloc [39] are shown for comparison. MineSweeper achieves a geometric mean slowdown of 10.8 % with a memory overhead of 7.9 %. FFMalloc comes in at 5.3 % time and 22.2 % memory, while Markus achieves 16.3 % time and 12.6 % memory. FFMalloc achieves a lower memory overhead here compared to SPEC CPU2006, because fewer workloads trigger its worst-case fragmentation. Still, the steadily increasing overheads over time with FFMalloc, shown in Figure 8, still appear here; for example, `perlbench` reaches 4× overhead by the time it finishes execution, despite its average overheads being lower due to growth over time.

The largest slowdown for MineSweeper is 2×, for `xalancbmk`; this is caused by the act of quarantine introducing many more L2 and L3 cache misses, by preventing quick reuse of deallocated memory, rather than the action of the sweeper itself. The slowest parallel benchmark is `wrf` with 66 %. This shows good scalability on a desktop machine, even when sweeper threads compete for CPU resources with the application's threads themselves.

5.7 Mimalloc-Bench Stress Tests

We evaluated our implementation (alongside FFMalloc and MarkUs) on the stress tests in the mimalloc-bench repository [18]. Originally made for evaluating an industrial-strength allocator, mimalloc [19], these tests have extremely high allocation and deallocation rates; most of them do not do any work, other than allocating and freeing memory. They test performance under different single- and multi-threaded allocation-deallocation patterns. We present the results in Figure 19. Despite the unrealistic pressure, MineSweeper only suffers a slowdown of 2.7× and a memory usage increase of 4.0× compared to the JeMalloc baseline (worst-case slowdown: 31×; memory: 27×)[6]. While MarkUs outperforms this in terms of memory (1.7×), it suffers a 6.7× slowdown, with 121× in the worst case. FFMalloc does slightly better on time (2.16×), but suffers a high

[6]This worst-case memory behaviour occurs for `glibc-thread`. This is an outlier, with a baseline memory footprint of just 4 MiB, and its large number of threads can therefore collectively build up large local quarantines, to limit communication overheads, in relative terms even if they remain small in absolute terms.

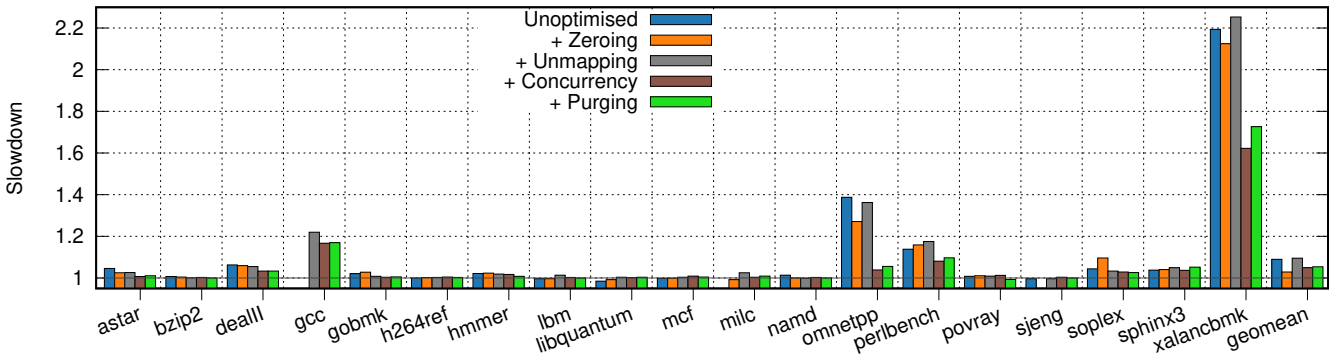


Figure 15: Run-time overhead under different optimisation levels (Section 4)

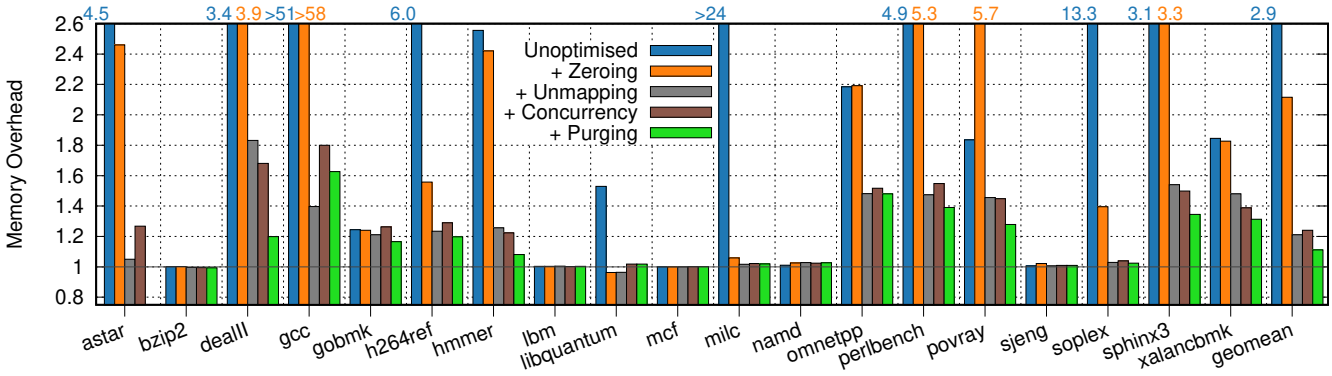


Figure 16: Average memory overhead under different optimisation levels (Section 4). Values with '>' failed from lack of DRAM.

(7.2×) geometric-mean memory cost, and its memory overhead balloons to 97× in the worst case. MineSweeper avoids such high overheads, should it become overwhelmed by the allocation rate, by pausing new allocations in the program briefly when the sweep is struggling to keep up. In such extreme allocation-heavy workloads, MineSweeper also makes it possible to trade off slowdown for memory usage by altering the pausing threshold.

MineSweeper avoids extreme overheads despite the fact that many of these benchmarks only allocate and deallocate, without doing any real work. This is not realistic, and violates our assumption that the sweeps can happen in the background, keeping up with the application. MarkUs suffers from the same issue, but the underlying allocator better handles aggressively reclaiming memory from many small quarantined allocations than JeMalloc, and limits worst-case overheads from high allocation rates by falling back to stop-the-world marking procedures to reduce pressure. Mimalloc-bench is often unrealistic; many tests deallocate things entirely in allocation order. Therefore, while the allocation frequency gives some challenge to FFMalloc, the fragmentation issues it sees in real-world workloads do not manifest. This explains the excellent memory behaviour of FFMalloc on `mstressN`, `sh8bench` and `xmalloc-testN`, despite worst cases of 97×.

5.8 Summary

MineSweeper significantly outperforms the state-of-the-art, with a geometric mean slowdown of only 5.4 %, and an average memory overhead of just 11.1 %. It delivers stable performance, with better average and worst-case behaviours — when considering both memory and execution time — than all previous systems, making it the only scheme viable for wide adoption in production environments.

6 RELATED WORK

Here we categorise and relate previously published systems solving the same or closely related problems to MineSweeper.

6.1 Memory Debuggers

These systems help find memory bugs, including temporal- and spatial-safety violations, in a debug environment. This is achieved by allocator modification, poisoning of deallocated and unallocated memory regions, binary instrumentation and shadow structures. AddressSanitizer [29] and Valgrind [25] are two notable examples, but limited checks are available even in memory allocators like Scudo [**?**]. These systems are not designed to be comprehensive mitigations; instead, they give detailed error reports to help developers locate bugs.

6.2 Probabilistic Defenses

These approaches aim to make it harder to exploit memory-safety bugs. DieHard [6] introduces probabilistic memory safety, and uses a range of techniques to harden programs (replication, heap randomisation, reallocation delay, separating metadata). DieHarder [26]

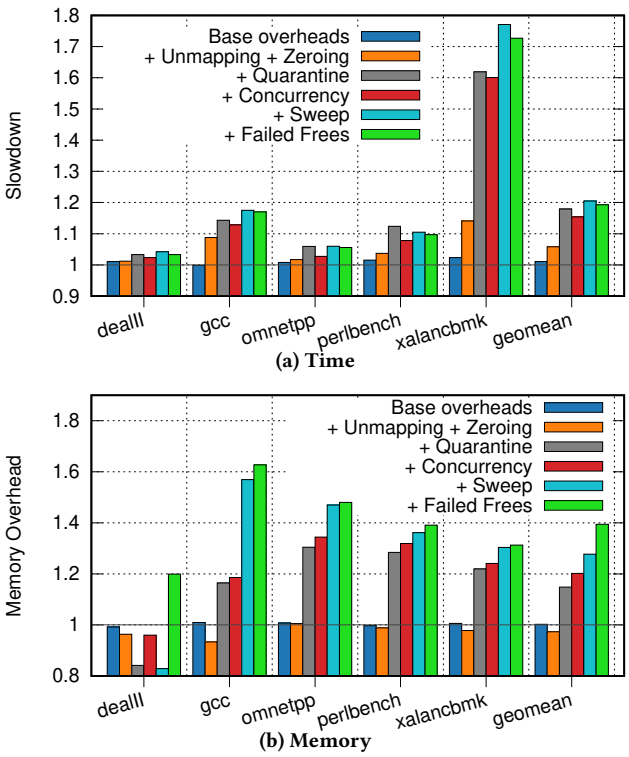


Figure 17: Sources of overheads

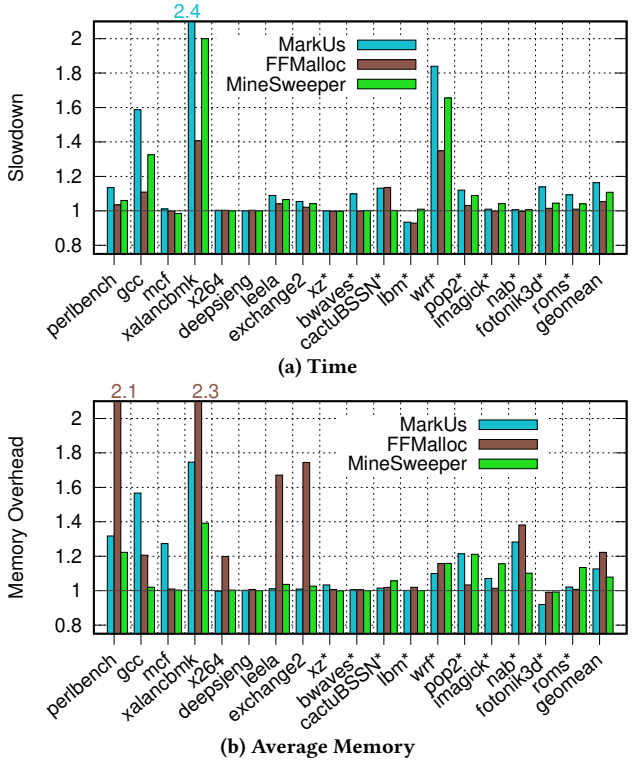


Figure 18: Overheads for SPECspeed2017. Starred benchmarks parallelised via OpenMP.

and FreeGuard [31] improve performance and protections. These can be bypassed by sophisticated exploits [17]. Cling [3] constrains reuse of virtual addresses to allocations from the same ‘call site’. Hardened allocators, like Scudo [**?**] fall into this category too.

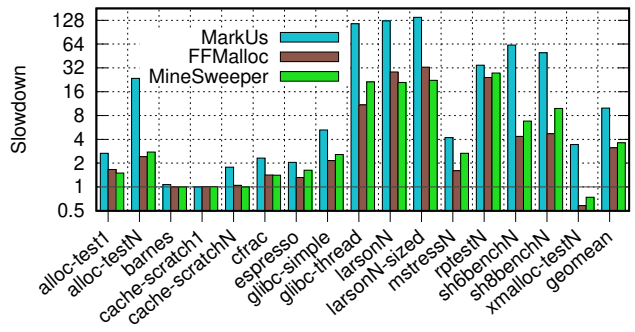
Arm’s Memory Tagging Extension [4] attaches a tag to each memory allocation. The (otherwise unused) top byte of each pointer is repurposed to store the tag. Hardware and kernel support is used to compare tags between the pointer and its target on each memory access. They assign different tags to subsequent allocations and subsequent reuses of the same region. The downside is the requirement for hardware support, and the limited number of tag bits, which make the protection only probabilistic. Still, such hardware mechanisms could combine with MineSweeper to achieve deterministic protection both with significantly lower overheads than in software alone, by allowing limited reuse of regions, and detection rather than just mitigation of attacks.

6.3 Page-Table Protection Methods

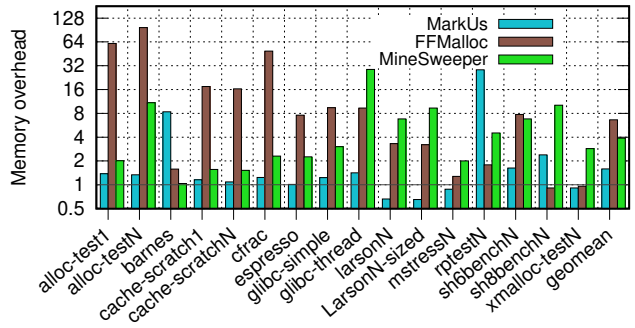
Some schemes mitigate use-after-free vulnerabilities by using page-table permissions. All allocations are placed in different pages, and revocation is performed on deallocation by prohibiting access using `mprotect` on the page table. This idea has been used by some debuggers (Electric Fence, PageHeap). Dhurjati and Adve [13] recognise that objects can be co-located in physical memory (with aliasing virtual pages, one for each object). They also use compiler analysis and a transformation called ‘automatic pool allocation’ to reuse virtual addresses. Oscar [12] adds a high water-mark, which enables it to manually specify virtual addresses when `mremap`’ing virtual aliases, thus discarding the old mapping altogether. For small allocations, Oscar suffers high overheads from TLB pressure, system calls, and page-table size. For large allocations, Oscar behaves similarly to MineSweeper, where the physical pages backing them are *unmapped*. Another technique that uses the latter strategy is FFMalloc [39], which never recycles virtual pages, and instead unmaps physical pages only when all objects inside have been deallocated. Many mechanisms presented here are orthogonal to those in FFMalloc, meaning there may be benefit in combining the approaches.

6.4 Pointer Nullification

These schemes actively revoke pointers by overwriting them with an invalid value, usually `NULL`. DangNull [17] uses red-black trees to precisely identify in- and out-bound pointers for each object, and nullify pointers to `free()`’d allocations. FreeSentry [42] uses a linked list instead of a red-black tree. DangSan [35] notes that pointer metadata is heavily write-intensive: it is written on every pointer store but only read once per object on deallocation. Therefore, they structure it as a log, with some de-duplication, to move work to deallocation. pSweeper [21] offloads pointer nullification to a background thread. This thread repeatedly, with a possible sleep period in between, sweeps live pointers for dangling ones. Deallocation is delayed until a full sweep is performed after the call to `free()`. pSweeper keeps a live pointer table, so that the sweep can locate live pointers, and to make nullification safe.



(a) Time



(b) Average Memory

Figure 19: Overheads for mimalloc-bench

6.5 CHERIvoke & Cornucopia

These schemes augment CHERI [40]—an architectural extension enabling spatial safety—with a temporal-safety mechanism. CHERIvoke [41] describes the algorithm and evaluates it in simulation. Cornucopia [37] optimises this with new architectural features, implements concurrency, and comprehensively evaluates a full (FPGA) prototype. The CHERIvoke algorithm is a pointer nullification scheme (Section 6.4). Like MineSweeper, it uses a shadow map, and performs linear sweeps through memory; however, in CHERIvoke, this is used to mark regions of memory currently in quarantine, which are *eliminated* by the sweeping mechanism unlike in MineSweeper. This is possible because in CHERI, pointers are replaced by architecturally visible capability objects, unlike the standard C model, which represents them as integers.

6.6 Passive Revocation

Passive revocation systems (including MineSweeper) work by tracking dangling pointers, like nullification techniques, but passively observe them instead of actively revoking. Instead, they delay deallocation until allocations are inaccessible.

High-level languages eliminate use-after-frees by automating memory management: (the equivalent of) `free()` is called from the garbage collector once the allocation is no longer live. Garbage collection is unsafe generally in C due to unconstrained pointer arithmetic. BoehmGC [8] attempts to defy this for specially written applications, implementing 'conservative' garbage collection in C/C++. However, runtime overheads are very high [2] compared to MineSweeper. Furthermore, for performance the programmer can still perform deallocation manually, leaving scope for use-after-free errors, and the garbage collector can falsely delete hidden objects the programmer hasn't freed, causing use-after-free errors *itself*.

CRCount [30] uses *reference counting*. An object is only deallocated if the programmer has `free()`'d it, and the reference count hits 0. Compiler support is used to keep a *bitmap* (similar to MineSweeper's shadow space in format) identifying which words of memory contain pointers. This helps to keep an accurate reference count per object at runtime. Like MineSweeper, CRCount realises that zero-filling of `free()`'d allocations can be used to remove references, thus simplifying the problem of garbage collection on `free()`'d allocations.However, *reference counting*, as opposed to sweeping, requires updates on all writes to pointer fields, in addition to the work on `free()` calls. This results in overheads on even non-allocation-intensive workloads (e.g., *mcf*, *povray*).

MarkUs [2] uses periodic *marking* passes to identify allocations in quarantine that are not live, and thus safe for deallocation. MineSweeper improves on this by using *zeroing* (Section 4.1) to simplify the problem (from reachability on the reference graph to identifying nodes with incoming edges), and using a simple *sweep* of memory to identify quarantined allocations targeted by pointers. MineSweeper is also more secure than MarkUs, as it stores metadata out-of-line, where it cannot be overwritten or reused [20] by an attacker.

7 SUMMARY AND CONCLUSIONS

We have presented MineSweeper, a drop-in mitigation for use-after-free vulnerabilities, motivated by the prevalence of use-after-free errors and exploits.

MineSweeper is a *passive revocation* scheme: it delays deallocation using a quarantine until it can prove that the allocation is no longer reachable via dangling pointers. We simplify the problem of *reachability* to that of identifying quarantined allocations *without pointers* to them. MineSweeper locates pointers by periodic *sweeps* of program memory, marking pointer targets in a *shadow map*. After a sweep, it releases unmarked allocations back to the memory allocator. MineSweeper makes the *conservative* assumption that each word of memory is—potentially—a pointer, thus it requires no type information or costly pointer tracking.

Our implementation on top of JeMalloc significantly outperforms previous software-only mitigation schemes by giving low overheads on both performance (5.4 %) and memory (11.1 %) simultaneously. More generally, MineSweeper can be easily integrated with any allocator: we have also built a Scudo implementation at 4.4% overhead. In short, low-overhead temporal safety is now widely practical, and should be deployable at scale.

ACKNOWLEDGEMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/P020011/1, and Arm Ltd. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.78150 and https://doi.org/10.5281/zenodo.5748402.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains our MineSweeper implementation, an allocator extension implemented on top of JeMalloc to mitigate use-after-free attacks, together with scripts to evaluate its running time and memory overheads on the SPEC CPU2006 benchmarks. The base implementation itself and a minimally modified JeMalloc memory allocator are fetched from their own repositories, compiled, and dynamically loaded in the SPEC config scripts. Run-times are reported by SPEC, while memory usage is monitored using the psrecord python package.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Intercept memory allocator calls, delay-of-reuse quarantine, linear sweeps of memory to safely release quarantined elements.
- **Program:** SPEC CPU2006 (not supplied)
- **Compilation:** g++, autoconf, make
- **Data Set:** SPEC CPU 2006 (c/c++)
- **Run-Time Environment:** Ubuntu system (tested on 16.04 and 20.04)
- **Hardware:** An x86_64 machine with sudo access to install dependencies and mount the SPEC ISO image.
- **Execution:** SPEC runscripts provided for dynamically linking in MineSweeper
- **Metrics:** Slowdown, memory usage increase
- **Output:** Two text files in the results directory, containing raw values and relative overhead in running time and physical memory usage (respectively) between the baseline and minesweeper.
- **Experiments:** 2 runs of the benchmarks: baseline (jemalloc only) and with minesweeper.so dynamically loaded
- **Disk Space Required (Approximately):** 10GB
- **Time Needed to Prepare Workflow (Approximately):** 1h
- **Time Needed to Complete Experiments (Approximately):** 8h
- **Publicly Available?:** Yes
- **Code Licenses:** 2-Clause BSD License
- **Workflow Framework Used?:** No
- **Archived?:** 10.5281/zenodo.5748402

A.3 Description

A.3.1 How to Access. Clone the git repository at https://github.com/EMarci15/asplos22-minesweeper-reproduce, including its submodules. An archived copy is available at https://doi.org/10.5281/zenodo.5748402.

A.3.2 Hardware dependencies. An x86-64 system running Ubuntu 18.04 or newer. Older Ubuntu releases and other Linux distributionts may also work, perhaps with altered package dependencies (untested).

A.3.3 Software Dependencies. Our package dependencies can be installed using our script (`scripts/dependencies.sh`). A SPEC CPU2006 disk image is required to run the experiments over SPEC.

A.4 Installation

You can install our artifact using the below sequence of commands. Firstly, clone the repository:

```
git clone --recurse-submodules https://github.
    ↪ com/EMarci15/asplos22-minesweeper-
    ↪ reproduce.git
```

Then, place your SPEC CPU2006 ISO disk image file in the base folder (`asplos22-minesweeper-reproduce`). The file extension must be '`.iso`', and the name must include '`cpu2006`'.

After this, in the simplest case, you can get, install, build and run the experiments by running `do_all.sh` in the `scripts` folder:

```
cd asplos22-minesweeper-reproduce
cd scripts
./do_all.sh
```

Note: this will request *sudo* access to install dependencies.

If something is wrong, you can perform the steps in `do_all.sh` one-by-one. Make sure you change directories to `scripts` before starting each script. See the comments in `do_all.sh` for details of what each script does. If debugging is necessary, it helps to reduce the size of benchmarks by swapping `-size=ref` to `-size=test` in both `run_spec.sh` and `run_psrec.sh`

Minesweeper comes as two separate git repositories. These are both included as submodules in the main artifact. The repository called "minesweeper-public" contains the minesweeper library implementation itself, while "jemalloc-msweeper-public" contains a minimally modified[7] JeMalloc memory allocator. Both are compiled to shared libraries, so that they can be dynamically loaded in the experiments.

In case you encounter issues, you can try following the instructions in the repository's `README.md` file to run the experiments inside a docker container.

A.5 Experiment Workflow

The `do_all.sh` script performs both time and memory evaluation experiments.

For time (slowdown) evaluation, the SPEC CPU2006 suite is invoked twice:[8]

(1) Baseline: (only) `jemalloc.so` is loaded (see the definition of `ENV_LD_PRELOAD` in `spec_confs/x86_64_je.cfg`),
(2) MineSweeper: both `jemalloc.so` and `minesweeper.so` are loaded.

For the memory overhead experiment, our script invokes the SPEC suite twice for each benchmark (individually), wrapped by *psrecord*.

A.6 Evaluation and Expected Results

The evaluation scripts called by `do_all.sh` produce two text files, `results/time.txt` and `results/memory.txt`. Both are printed to the screen at the end of the run.

Running times are extracted from the SPEC result text files. Memory analysis in `result/memory.txt` is extracted from the memory traces[9] in the `ps` folder.

[7] Modifications: treat metadata extents and allocation extents differently, use `sbrk` for allocation extents (instead of `mmap`), and increase size by 1 in `malloc()` (so `end()` pointers point to the same allocation). These resulted in no measurable slowdown.
[8] The artifact only does a single iteration, while in the paper, we took the median of three runs.
[9] The traces themselves include the footprint of the SPEC tools, but our script removes this in order not to skew results in our favour.

Results vary between systems, but we expect a geomean slowdown of around 5.4% and memory overhead around 11.1%.

A.7 Experiment Customisation

You can run with different benchmark suites by using `LD_PRELOAD` as described in Appendix A.5, although this may require a little bit of work on result extraction. Adapting to SPEC CPU2017 is fairly straightforward.

To run MineSweeper on a dynamically linked binary `prog_binary`, compile our *jemalloc* and *minesweeper* versions into a directory `lib` and run:

```
LD_PRELOAD=lib/minesweeper.so:lib/jemalloc.so
    ↪ ./prog_binary
```

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Sam Ainsworth. 2020. *Experimental Artefact for MarkUs: Drop-in use-after-free prevention for low-level languages.* https://github.com/SamAinsworth/MarkUs-sp2020/.

[2] Sam Ainsworth and Timothy Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *SP*. 578–591. https://doi.org/10.1109/sp40000.2020.00058

[3] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers.. In *USENIX Security*. 12.

[4] ARM Ltd. 2019. *ARM Memory Tagging Extension Whitepaper.* https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[5] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. 2020. Sound Garbage Collection for C Using Pointer Provenance. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 176 (nov 2020), 28 pages. https://doi.org/10.1145/3428244

[6] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. (2006), 158–168. https://doi.org/10.1145/1133981.1134000

[7] Hans-J. Boehm. 2002. Bounding Space Usage of Conservative Garbage Collectors. *SIGPLAN Not.* 37, 1, 93–100. https://doi.org/10.1145/565816.503282

[8] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. (1991), 157–164. https://doi.org/10.1145/113445.113459

[9] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[10] Paolo Carlini, Phil Edwards, Doug Gregor, Benjamin Kosnik, Dhruv Matani, Jason Merrill, Mark Mitchell, Nathan Myers, Felix Natter, Stefan Olsson, Johannes Singler, Ami Tavory, and Jonathan Wakely. 2020. The GNU C++ Library Manual. https://gcc.gnu.org/onlinedocs/libstdc++/manual/iterators.html.

[11] Oliver Chang. 2016. Racing MIDI messages in Chrome. https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html.

[12] Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security*. 815–832.

[13] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently detecting all dangling pointer uses in production servers. In *DSN*. 269–280. https://doi.org/10.1109/DSN.2006.31

[14] Pavel Emelyanov. 2013. Soft-Dirty PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt.

[15] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan*.

[16] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (sep 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[17] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*.

[18] Daan Leijen. 2019. *MiMalloc benchmarks.* https://github.com/daanx/mimalloc-bench.

[19] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*. Springer, 244–265. https://doi.org/10.1007/978-3-030-34175-6_13

[20] Jungwon Lim. 2020. 'freeing list' operation leads to exploitable write-after-free. https://github.com/SamAinsworth/MarkUs-sp2020/issues/2.

[21] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *CCS*. 1635–1648. https://doi.org/10.1145/3243734.3243826

[28]]scudo LLVM. [n. d.]. *Scudo Hardened Allocator.* https://llvm.org/docs/ScudoHardenedAllocator.html.

[23] Ivan Maidanski. 2009. "random errors in incremental mode". https://github.com/ivmai/bdwgc/blob/57b97be07c514fcc4b608b13768fd2bf637a5899/include/private/gcconfig.h#L2681.

[24] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html.

[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100. https://doi.org/10.1145/1273442.1250746

[26] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[27] National Institute of Standards and Technology (U.S. Department of Commerce). 2020. *National Vulnerability Database.* https://nvd.nist.gov/.

[28]]psrecord Thomas P. Robitaille. [n. d.]. *PSRecord on GitHub.* https://github.com/astrofrog/psrecord.

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*. 309–318.

[30] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++.. In *NDSS*. https://doi.org/10.14722/ndss.2019.23541

[31] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2389–2403. https://doi.org/10.1145/3133956.3133957

[32] Prokash Sinha. 2004. A Memory-Efficient Doubly Linked List. https://www.linuxjournal.com/article/6828.

[33] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

[34] TIOBE. 2020. *TIOBE Index for April 2020.* www.tiobe.com/tiobe-index.

[35] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. Dangsan: Scalable use-after-free detection. In *EuroSys*. 405–419.

[36] Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2019. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982* (2019).

[37] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

[38] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. *FFMalloc Source Code.* https://github.com/bwickman97/ffmalloc.

[39] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security*.

[40] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*. https://doi.org/10.1109/ISCA.2014.6853201

[41] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468.

[42] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers.. In *NDSS*.

[43] Carlo Zapponi. 2020. *GitHut 2.0.* madnight.github.io/githut.