View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Apollo

MarkUs: Drop-in use-after-free prevention for low-level languages

Sam Ainsworth, Timothy M. Jones
University of Cambridge, UK
{sam.ainsworth, timothy.jones}@cl.cam.ac.uk

***Abstract*—Use-after-free vulnerabilities have plagued software written in low-level languages, such as C and C++, becoming one of the most frequent classes of exploited software bugs. Attackers identify code paths where data is manually freed by the programmer, but later incorrectly reused, and take advantage by reallocating the data to themselves. They then alter the data behind the program's back, using the erroneous reuse to gain control of the application and, potentially, the system. While a variety of techniques have been developed to deal with these vulnerabilities, they often have unacceptably high performance or memory overheads, especially in the worst case.**

We have designed MarkUs, a memory allocator that prevents this form of attack at low overhead, sufficient for deployment in real software, even under allocation- and memory-intensive scenarios. We prevent use-after-free attacks by quarantining data freed by the programmer and forbidding its reallocation until we are sure that there are no dangling pointers targeting it. To identify these we traverse live-objects accessible from registers and memory, marking those we encounter, to check whether quarantined data is accessible from any currently allocated location. Unlike garbage collection, which is unsafe in C and C++, MarkUs ensures safety by only freeing data that is both quarantined by the programmer and has no identifiable dangling pointers. The information provided by the programmer's allocations and frees further allows us to optimise the process by freeing physical addresses early for large objects, specialising analysis for small objects, and only performing marking when sufficient data is in quarantine. Using MarkUs, we reduce the overheads of temporal safety in low-level languages to $1.1\times$ on average for SPEC CPU2006, with a maximum slowdown of only $2\times$, vastly improving upon the state-of-the-art.

I. Introduction

The lack of temporal safety in low-level languages, such as C and C++, has become a critical cause of insecurity in modern systems. Large, security-critical applications, such as web browsers [1], [2] and operating system kernels [3], are increasingly plagued with use-after-free vulnerabilities. Here, data is mistakenly freed by a process, reallocated, and altered by an attacker with control of data input, then incorrectly reused by the process. These allow the attacker to alter control flow, and potentially gain kernel-level access.

A variety of techniques have been proposed to mitigate use-after-free vulnerabilities in C and C++. For example, all pointer locations can be logged and then nullified when their data is freed [1], [4], [5], objects allocated with their own page-table entries [6], [7], or probabilistic reuse delays employed [8]–[10]. However, these tend to exhibit both high average- and worst-case overheads in terms of performance and memory utilisation, or have limited coverage.

We take a different approach with MarkUs, by storing programmer-freed object locations in quarantine until we can demonstrate that no dangling pointers exist to them. We do this by performing a live-object traversal of accessible memory regions, similar to but much more efficient than the behaviour of a garbage collector [11], to mark accessible heap objects. Since only objects freed by the programmer can be reallocated, language safety is maintained, and since objects with dangling pointers cannot be reallocated, programmer frees do not need to be relied on for security. This allows use-after-free attack prevention at low overhead even in complex cases, while maintaining compatibility with real-world applications.

We can use the information provided by the programmer's untrusted manual frees to reduce the cost of the live-object traversal. We can reallocate the physical pages used by large objects as soon as they are freed, using the unmapped virtual addresses as protection, to reduce the frequency of marking procedures without increasing memory utilisation. In addition, because we know the amount of memory we can potentially reclaim from a marking process, we can eliminate needless memory traversals, reduce marking frequency by only attempting to reclaim objects when enough can be freed, and, further, trade off memory usage for performance.

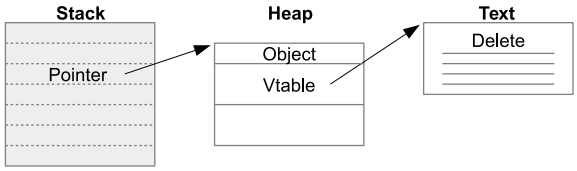
MarkUs gives performance overhead and memory usage that is low enough for real-world use, for all applications written in low-level languages. For example, for SPEC CPU2006, we achieve an average slowdown of just $1.1\times$ ($2\times$ worst case), with average memory overhead of $1.15\times$ ($2\times$ worse case), both of which are lower than any other competing technique.

Our main contributions are as follows:

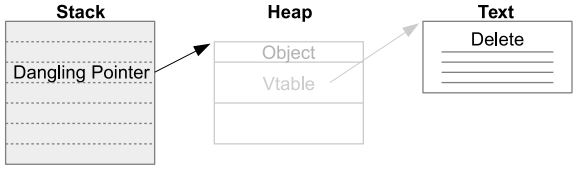
- The realisation that free() can be treated as a hint for reallocating memory but the actual reallocation can be decoupled for security.
- Use of a marking procedure to verify programmer deallocations and permit reallocation.
- Use of a quarantine list to store programmer-freed data until it has been verified as safe to reallocate.
- Page-table optimisations to immediately free physical address space for large deallocations, while still ensuring high performance for small deallocations.
- Optimisations using knowledge of the volume of data freed, to vastly reduce marking-procedure overheads, and trade off memory and performance overhead.
- Evaluation on a variety of real-world and allocation-intensive workloads, including multithreaded setups and comparison against state-of-the-art techniques.

```
1  Object x = new Object();
2  delete(x);
3  ...
4  // x's vtable replaced by the attacker, who is
5  // reallocated x's address space with y.
6  Object y = new Object(user_input);
7  ...
8  // Control diverted to attacker's pointer, in
9  // the place of the original delete function.
10 delete(x);
```

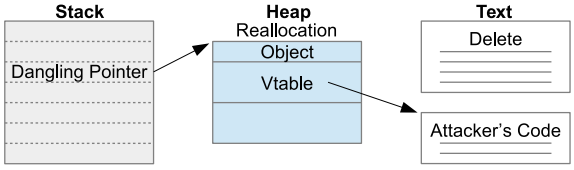
Fig. 1: An example use-after-free attack, in C++. The attacker is allocated data that is still pointed to by x, and can manipulate the data to redirect the old pointer to a chosen function rather than the original object's delete call.



(a) In an object's first deletion, the call to delete is correctly looked up in the vtable of the object.



(b) Once the programmer has deleted the object, the space it contains is free to be reallocated. However, in this case the pointer still points to the deleted object, and so can still be derefenced and used.



(c) As the memory object has been freed, this allows the attacker to reallocate it and store their own data in the same location, overwriting the vtable with their own data. A subsequent, incorrect, call to delete, which often exists due to programmer bugs, via the now-dangling pointer will then be redirected to the attacker's choice of code, allowing them to hijack the application.

Fig. 2: The example use-after-free attack shown in figure 1, in terms of allocated data.

II. Background

Here we describe use-after-free attacks, and solutions in higher-level languages, before presenting our threat model.

A. Use-After-Free Attacks

In a use-after-free attack [12], an object is freed prematurely before being incorrectly reused via a dangling pointer. By this point an attacker may have changed its contents to point to their own data, by forcing the allocator to reallocate to the attacker the region targeted by the dangling pointer. Example code is given in figure 1, and the associated memory behavior is shown in figure 2. This is a particularly damaging vulnerability due to its common occurrence in large codebases, and high level of exploitability. For example, as of 2013 it was the most widespread memory vulnerability in Chromium [1]. Operating-system kernels and browsers are particularly affected by use-after-free attacks [2], [3], as high-value targets written in manually memory-managed languages.

This is a memory-safety violation: in C/C++, the use of freed memory, or memory accessed outside the bounds of data structures, is considered undefined behavior. Specifically, use-after-free attacks come under the category of temporal-safety violations, where something is accessed after a point where it is no longer allowed. This contrasts with spatial violations, such as buffer overflows [13].

An attacker can utilise control over the data stored in a still-accessible object in various ways. Particularly useful methods include double-delete attacks, where a C++-style delete virtual function pointer is overwritten by an attacker before an incorrect second free, or, more generally, a function pointer is overwritten in an object before it is incorrectly called after a free. This allows attackers to divert the program to their own choice of code within a process's address space.

B. Garbage Collection

Garbage collection [14] solves the problem for high-level languages (at least excluding their own runtimes, typically written in lower-level languages), in that data is only freed when no pointers to it are available. This means dangling pointers do not exist, and so use-after-free attacks cannot occur. Still, often a performance hit is observed, and as such safe techniques for manual deallocation have been added even in languages where garbage collection is safe [15].

In C and C++ the picture is somewhat more complex. In most runtimes on most architectures we cannot distinguish pointers from other data, and indeed the two can be converted between each other. This means we are limited to conservative garbage collection [11], where we must assume all data may be a pointer, creating the chance of accidental references. This is reasonable for small objects, as the chance of a given object being coincidentally pointed to is vanishingly small, and the cost of each failed free is very low. However, larger objects cause issues, as the probability of a false mark and the memory-usage cost of this both increase. Some conservative garbage collectors [16] offer the ability to manually free objects to prevent such leaks, in addition to automatically attempting to clear objects by garbage collection, but this opens up the potential for use-after-free attacks.

Another problem for garbage collection in C and C++ is that it is not typically safe. Pointers can be hidden with arbitrary arithmetic, such as in XOR lists, meaning that data that should be live is incorrectly deleted [17]. Compiler optimisations can also hide pointers [18], which means that many high-performance applications do not run correctly in the presence of a garbage collector.

MarkUs is not a garbage collector, and does not use garbage collection. Still, the marking procedure of a typical garbage collector performs the same live-object traversal necessary to detect dangling pointers to quarantined objects, allowing us to implement similar code to verify whether the manual frees

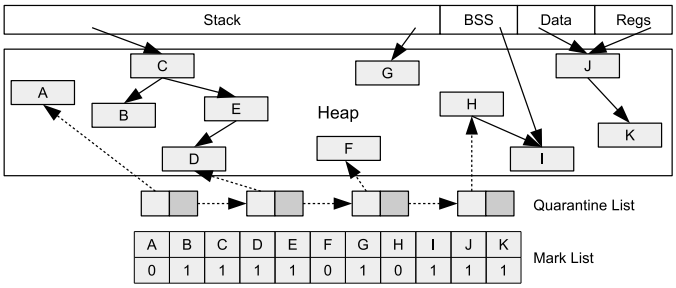


Fig. 3: Memory objects freed by the programmer are placed on a quarantine list. This is periodically checked by a marking procedure, where the stack, data, bss, and registers are walked to find accessible heap pointers, and any data transitively accessible from those pointers. MarkUs can deallocate unmarked objects that are already on the quarantine list whereas other, marked, objects must remain on the quarantine list for security. In this example, D remains on the quarantine list because it is accessible transitively through a pointer starting on the stack. Solid lines show application pointers, dotted lines show MarkUs pointers to objects on the quarantine list.

of the programmer are yet inaccessible. This means that we can find dangling pointers while still ensuring that only data the programmer has actually freed can be deleted, resulting in correct program behavior even in the presence of hidden pointers. Further, information from the programmer's frees can be used to optimise the process, by unmapping physical pages early, and only performing a marking procedure when sufficient manual frees are ready to be vetted.

C. Threat Model

We assume that programs execute in the presence of an attacker able to allocate memory, for example through carefully constructed inputs, and who can force the program to read freed data. This attacker wishes to gain further control over the program, for example by redirecting control-flow.

Like other papers on use-after-free mitigation [1], [7], we only look at heap use-after-free, rather than stack, as the heap attack is both the most difficult to protect against and by far the most commonly exploited [19]. This is because attacks based on data freed on the stack can be handled with static checks, such as escape analysis [7].

III. MarkUs

MarkUs is a memory allocator designed to prevent security violations from use-after-free attacks, intended for production scenarios where preventing their use is more important than detecting their existence. It delays the reuse of programmer-freed memory objects until certain that there are no dangling pointers that target the freed range. It is designed primarily for C and C++, but is suitable for any languages that allocate using malloc and free, or new and delete, which it replaces with its own implementation. The MarkUs library can be used in applications either by directly calling the replacment functions and linking against the library, or by dynamically replacing these functions at runtime.

Within MarkUs, calls to deallocate or free data are replaced with a call to add the memory object to an intermediate structure, the quarantine list, shown in figure 3. Objects are kept here until known to be safe to reallocate, at which point they are moved to the allocator's free lists. To identify this, we traverse all live objects, marking those we find; those on the quarantine list that are unmarked at the end do not have dangling pointers pointing to them. This means that only programmer-freed data audited by this marking procedure is actually freed, achieving safety with respect to the original implementation at the same time as security from use-after-free attacks. Large allocations can be reused before a marking procedure, by unmapping virtual pages and allowing the operating system to reallocate the physical. We can also reduce overheads by using the information from the programmer's manual free calls to control frequency of marking procedures, and to trade off performance overhead for memory utilisation.

A. Quarantine List

To prevent use-after-free attacks, we must ensure that there are no pointers to a given freed object before we can allow it to be reallocated. By manually freeing an object, a programmer claims it is safe to free and reallocate. MarkUs decouples these two, allowing the programmer to free the object and claim it is safe, but delaying reallocation until this is validated. To achieve this, rather than immediately placing a manually freed object on a free list, we quarantine it until we can verify the programmer is correct, placing it instead on a quarantine list.

Only objects on this quarantine list are allowed to be deleted. This is necessary to conserve safety within C and C++, and prevent accidental deletion of data pointed to by hidden pointers as a result of, for example, XOR pointers or compiler optimisations [17], [18]. This means that, despite our mark-based technique, MarkUs deliberately does not attempt to prevent memory-leaks by the programmer. It is purely a technique to improve the security aspect of an application.

The quarantine list itself is not inherently trusted. Instead, it is used as a guide to what the programmer believes should be freeable, for safety rather than security. It is then up to MarkUs to audit the list to see if it agrees, before an object is truly freed and made available for new allocations.

B. Identifying Live Objects

Periodically, we search for objects on the quarantine list that can or cannot be freed, by traversing all live objects and marking those we encounter, starting with those visible from registers, the BSS and data segments, and the stack, then transitively any heap objects pointed to by this set. Such objects, and their pointers, are recursively walked using a graph traversal, with any word treated as a pointer if it appears to point within the heap's bounds. An example is shown in figure 3. This is similar to a mark procedure from a garbage collector, and we use the Boehm-Demers-Weiser [16] implementation.

Marking an object on the quarantine list is not necessarily indicative of a bug. This is because of both conservatism, in

that data values may coincidentally point to objects on the heap, and the existence of a pointer not necessarily indicating its future use. This is one reason why MarkUs, and, more generally, any other runtime technique, cannot detect all use-after-free occurrences within programs, though it does prevent their use for security violations. Conversely, an object that is not on the quarantine list and isn't marked cannot be freed, to preserve safety under pointer-hiding [17], [18].

This marking procedure can be performed in parallel [16], and there is no need to stop execution of the program during it (see section III-H). Indeed, the constraints here are looser than a traditional garbage collector, as safety is ensured by the quarantine list preventing deallocation of any data not specified by the programmer. This means the guarantees of the marker need only prevent any race condition in the marker from causing exploitable security vulnerabilities.

Further, this step need not be performed by a traditional marker, which transitively follows pointers to find those which are accessible. While that is necessary in a true garbage collector to avoid circular references in deallocatable objects, with manual freeing this could be avoided by zeroing out objects when they are added to the quarantine list. This means that, potentially, a sweep of the entire stack, heap, registers, and BSS segment is sufficient, and more cache-friendly. Though the performance overheads are potentially higher, we use a traditional mark in our implementation, to reuse more of the Boehm-Demers-Weiser garbage collector [16].

C. Quarantine-List Walk

Following a marking procedure, a garbage collector would sweep the heap looking for free objects. However, for MarkUs this is not a safe underapproximation of what is truly freeable, due to hidden pointers, and is unnecessary, as we already have a list of candidates in the quarantine list. Instead of a sweep, we walk the quarantine list, and free anything that wasn't marked in the previous marking procedure. Anything that is marked is left on the list, as we cannot guarantee it will not be reused by the current holder of the pointer. Anything that is not marked is moved to the relevant free list in the allocator, based on its size, to be reallocated by malloc or new.

D. Mark Frequency Optimisation

Marking is typically the most expensive part of a garbage collector, and this is also true of MarkUs. This means, to reduce overheads, we need to mark as infrequently as possible. One thing that MarkUs can exploit to do this, that a real garbage collector cannot, is that it knows how much data it can possibly free, because it knows the size of its manually-freed quarantine list. We can therefore trade off expected memory usage for performance, by only allowing garbage collection when the size of data on the quarantine list exceeds a proportion of the total current heap size.

More specifically, we allow marking procedures when

$$(qlsize - failed\_frees - usize) * N > (heap\_size - unmapped)$$

where N is a chosen growth bound, which can be controlled to trade off heap growth compared to an unprotected program with performance overhead, $qlsize$ (quarantine list size) is the amount of data currently waiting to be audited (either added after a previous marking procedure, or from a failed attempt in a previous round), $failed\_frees$ is the amount of data that failed to be deallocated in the previous round, and so is likely to fail again in the near-term future, and $usize$ (unmapped size) is the proportion of the quarantine list that is not taking up physical memory space, as it has been unmapped. Further, $heap\_size$ is the total size of the heap allocated to a process, and $unmapped$ is the proportion of the heap that has been unmapped, and so does not represent physical memory utilisation.

This does not result in a strict limit on heap growth compared with an unprotected program, and there is no maximum quarantine size. This is because we may not be able to free some data due to the presence of dangling pointers or conservative overestimation of data as pointers. In addition, while previously failed frees may be successful in the next round of a marking procedure, as the dangling pointer may disappear, they are disregarded as part of the trigger condition for mark culling to prevent constant ineffective marking when large numbers of failed frees exist. Still, under the assumption that accessible data on the quarantine list is rare, we can control overheads based on system parameters.

E. Allocator Details

Because it provides data structures to set mark bits for the marking procedure, we use the allocator from the Boehm-Demers-Weiser garbage collector [11]. This splits objects into two sizes: those larger than a page are allocated as monolithic objects with their own headers, at page-sized granularities. For MarkUs, this means the entire object can be unmapped from the virtual address space once it is freed from quarantine, as no other objects will share pages with the freed object. However, for objects smaller than a page, the allocator uses a pool strategy: a single header is used for an entire page of objects, which are all the same size and initialised simultaneously, to reduce metadata and allocation overheads. Mark bits are stored in these headers rather than the objects themselves.

Objects are initialised to zero upon allocation, to reduce the probability of old false pointers appearing when these objects are later walked by the marking procedure. These pointers would reduce the proportion of objects we could securely free. Zeroing also has the helpful side-effect of mitigating information leakage via heap initialisation bugs [20], another class of temporal safety violations.

Other allocators could use MarkUs's strategy to eliminate use-after-free vulnerabilities—the allocator's implementation choices typically neither help nor hinder MarkUs, and instead add orthogonal overheads and performance benefits. However, since these allocators would then need separate support added for mark bits, to limit engineering work we only evaluate on the Boehm-Demers-Weiser allocator with existing support.

F. Page Unmapping

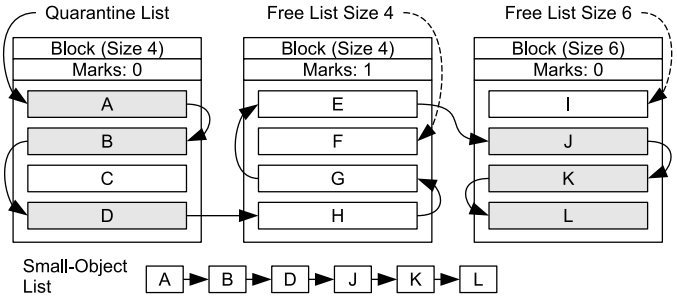
For large allocations, we can provide a form of use-after-free detection, as well as prevention, by unmapping pages upon a user's deallocation, as long as an object's allocated region entirely covers each unmapped page. Subsequent access to an unmapped page will result in a segmentation fault, correctly flagging the use of a dangling pointer.

There are also performance and memory-consumption benefits to this approach. Because we unmap these physical pages, they can be reused in a subsequent mmap call for another allocation without any need for a marking procedure. As virtual addresses are typically an ample resource, particularly on 64-bit architectures, this can significantly cut down the performance overhead of MarkUs in the face of large allocations, as marking frequency for a given memory overhead can be significantly reduced. For large objects, this can potentially be the only practical way of recovering physical memory. As a single allocation grows in size, in a conservative marker the likelihood of unrelated data coincidentally pointing to an address within that allocation's range increases. This means a marking procedure may be unable to free and reallocate such an allocation. For a true garbage collector, this can cause memory leaks without allowing unsafe manual deallocation, but for MarkUs, it is reduced to a problem of potential virtual-address leakage rather than physical-memory leakage. Unmapping the physical pages therefore eliminates the problem. In addition, unmapped pages need not be examined for pointers, as they are inaccessible, reducing marking costs. This is implemented using a bit in the allocator's per-page metadata. On reallocation, if the allocator wishes to reuse space with the `unmapped` bit set, it calls mmap with the address of the page or region as its argument. Otherwise, mmap is called with the address at the end of the current heap, installing new metadata. This prevents the allocator from accidentally reusing the unmapped space of pointed-to addresses for new objects if its heap is exhausted. If the programmer uses mmap calls elsewhere, then these can be wrapped to prevent reuse, though as with previous work [7] we have not found the need to do this in practice because munmapped space is typically not reused in an mmap unless deliberately requested.

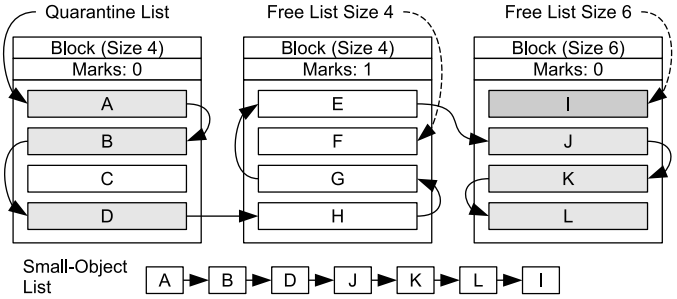
For large objects, this makes our allocation and deallocation strategy similar to use-after-free-prevention techniques that use a separate virtual page for every allocation [6], [7]. The differences are in how we treat small allocations. First, we store multiple small objects in the same virtual page to avoid TLB pressure, and use marking procedures to reclaim the memory. Second, virtual addresses are eventually reclaimed by marking procedures. Third, aliasing of physical pages is unnecessary because MarkUs only needs to map one physical page to one virtual page at any point, rather than using multiple mappings to have concurrently-live small objects allocated in the same page but accessible by different virtual addresses.

G. Small-Object Block Sweeping

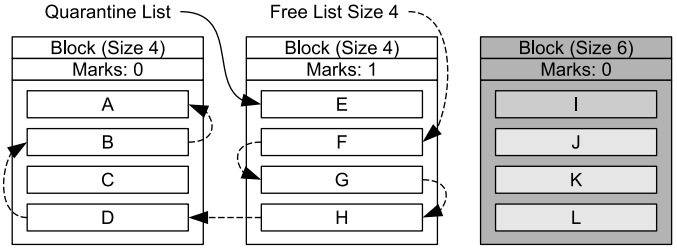
The Boehm-Demers-Weiser pool allocator ensures all objects within a page-sized block are of the same size. Once freed, these small objects go to separate free lists per object size, and by default are only reused for new objects of that size.



(a) Quarantine list elements from entirely-unmarked blocks are found and sent to a small-object list for further analysis, in an attempt to free the entire page for use by differently-sized objects.



(b) The free lists of the appropriate sizes are checked, to find any objects within blocks we are trying to entirely free, and placed on the small-object list if from an unmarked block.



(c) Any block for which every object is on the small-object list is entirely freed. Objects within partially-freed blocks with no marks are instead placed on the relevant free lists of the pool allocator.

Fig. 4: An example of walking the quarantine, free and small-object lists when using small-object block sweeping.

MarkUs deliberately trades off marking-procedure frequency for memory overhead. This means that, even for a small working set of memory objects, many pages can be allocated between each marking procedure, which, by default, can only be used for memory objects of the same size from that point on. This can result in significant memory overhead, because if the proportion of object sizes changes over time, this allocation space effectively becomes unusable.

To fix this, we deallocate blocks that consist of entirely-freed memory objects, allowing them to be reused more generally. One way of finding this information out is by looking at the marks within a block: if the block is entirely unmarked, we can reallocate it by deleting every element from the appropriate free list. However, in C and C++, where the marks of a collector cannot necessarily be trusted, this results in a safety violation, as we may free objects that the user is still accessing. Instead, we use that information as a guide

to trigger a more complicated analysis. If there are no marks within the block of an object on the quarantine list, we do not free that object immediately. Instead, we add such objects to a small-object list. We then sweep the free list, moving any objects that share a block with a quarantined object we think is in an entirely free block back to the small-object list. We then check to see if any blocks are entirely within this new small-object list. If they are, we delete the entire block and allow its reuse for differently-sized objects. If not, we add them to their original free list, from which they may have just been removed. An example is given in figure 4.

This means that, as well as freeing small objects in partially-free blocks so that they can be reused, we also safely free entire blocks so that they can be reused for data of other sizes, preventing memory leaks even for programs with many distinct memory-allocation phases throughout their execution.

H. Concurrency and Parallelism

The marking procedure is parallel; it can be run on multiple threads at once by splitting up the current frontier of objects to be searched for pointers. This has the effect of making the marking procedure faster and more efficient on multicores, and decreases single-core slowdown at the expense of spreading CPU utilisation across many cores. If too high, this can impact other applications running on the system. However, MarkUs typically spends little time running marking procedures, and parallel execution is typically more energy efficient.

MarkUs's marking procedures are performed concurrently with application execution, which continues while the stack and heap are searched. Since data may be modified during the marking procedure, to preserve correctness we use page-table dirty bits [16], set when pages are modified while a marking procedure is being concurrently performed, to track this new data. These dirty pages are then checked once again at the end of the marking procedure, to check for the presence of any new accessible regions. The marking procedure need only stop-the-world briefly at the start of the marking procedure, through sending suspend signals to each thread, (to collect registers as root sets to find pointers) and at the end (to preserve correctness under concurrent modification). Since these require little work, stop-times are unnoticeable, and thus MarkUs works for applications with user interaction, such as browsers.

Walking the quarantine list is performed under the allocator's lock, and is thus single-threaded. While this could be parallelised, its overhead is negligible compared with the marking procedure, and thus optimisation is unwarranted.

I. Coverage Limitations and Hidden Pointers

Since low-level languages, like C and C++, allow pointer hiding, for example by XORing them with other data, we are not able to see all of them. That doesn't stop our system from being semantically safe, however, since we only ever delete objects that the programmer freed themselves, so we cannot introduce undefined behavior. However, it also means that MarkUs could fail to detect complex use-after-free vulnerabilities involving hidden pointers, as is a limitation with any technique that involves identifying pointers. Still, a garbage collector works for most parts of most C and C++ programs [21] The vast majority of pointers are not hidden, as we examine in section V-I. Further, most hidden pointers are already carefully implemented, and so are unlikely to contain use-after-free errors. This means the defence is practical, and other techniques that rely on, for example, zeroing old pointers [1], [4], [23], or tracking them [16], are also vulnerable. MarkUs can further successfully protect programs even in the presence of integer-casted or union pointer types that the compiler cannot disambiguate, but MarkUs can treat as potentially containing pointers.

An attacker can deliberately create pointers to objects just through write access to integer arrays, as MarkUs cannot by default distinguish between pointers and data. This prevents deallocation of such space, causing larger memory utilisation. This gives an attacker with full allocation abilities no more power unless they are limited to a given area of sandbox space, at which point they can have more system-level impact than they could otherwise, potentially exhausting the application of memory instead of just the sandbox. Still, there are two alternative mechanisms that can be used to prevent this specific case, if necessary for a given application. The first is that a sandbox's memory can be limited to prevent its quarantine, as well as its allocated data, from exceeding a given size. The second is that, in high-level languages, we can tag allocation space as integer-only, to avoid its marking and prevent any conservative pointer behaviour. The Boehm allocator we use [16] already supports the latter, but as it is a very specific use-case, and requires application targeting rather than being entirely drop-in, we do not utilise this in our implementation.

The allocator's headers (section III-E) are isolated from data, and so cannot be targeted by use-after-free in quarantine. However, our current implementation inherits small-object links that are stored within old objects. An attacker can potentially rewrite these links in two ways if they find a use-after-free. First, they can add false elements to the list, though unless these elements are hidden pointers they cannot be deleted, and unless they can be written to by the attacker, the marking procedure will not reach the end of its list, preventing application progress and the presence of a useful attack. Second, they can possibly remove elements or cause loops through an existing double-free. While neither causes useful privilege escalation, as they either prevent program progress or only affect memory utilisation, a full implementation could isolate metadata at negligible overhead. Fast allocators, including jemalloc already utilise such isolation [10], and we only avoid it for implementation complexity.

J. Summary

This section has presented the design of MarkUs, a use-after-free-preventing memory allocator for low-level languages, based on quarantining data that is manually freed by the programmer and verifying it using a marking procedure of the stack, heap, registers, and data segments. Through this, MarkUs achieves both safety with respect to low-level pointer

handling, and protection against use-after-free attacks. To minimise the costs of the technique, we optimise this strategy by directly limiting the number of marking procedures based on the amount of data the programmer has tried to free, eagerly unmapping virtual pages of large allocations so that the physical pages can be reused immediately following the programmer's deallocation, and using the marking procedure's overestimation of entirely-free regions of memory as a guide to perform more complex checks to reallocate regions to objects of different sizes. The next section describes our experimental system before we move on to demonstrate how MarkUs performs on real workloads.

IV. Experimental Setup

We evaluate MarkUs using a prototype built by extending the Boehm-Demers-Weiser Garbage Collector [11], [16]. This is implemented as a shared library that overrides malloc, free, calloc, realloc, new, and delete, which can be utilised by defining LD_PRELOAD before execution of a dynamically-linked application, meaning source code access is unnecessary. By default, we use a growth bound (section III-D) of 4, to represent a quarantine list of 33% of the size of the rest of allocated memory.

We evaluate on an Intel system, featuring a quad-core Haswell Core i5-4570, 16GB of DDR3 RAM, and running Ubuntu 16.04. For profiling, we used PSRecord [26]. We evaluate using SPEC CPU2006 [24] (using reference inputs) and Olden [27] (default inputs), to demonstrate on benchmarks with a wide range of memory behaviors and to directly evaluate against other work in the literature. We show C and C++ benchmarks from SPEC CPU2006 to give direct comparison with prior work; Fortran benchmarks behave similarly. In addition, we use Firefox with BBench [28] to show applicability to modern, particularly vulnerable workloads. SPEC CPU2006 and Olden ran with no modifications, as do most applications in practice, as the allocator is functionally compatible with glibc malloc. MarkUs has also been tested on a variety of real-world applications such as OpenOffice, Okular, Evince, Texstudio, Vim and Emacs, where no noticeable impact on the application was observed. For Firefox, we compiled with `--disable-jemalloc` to directly hook malloc and free instead of the custom allocator Firefox normally uses, to reduce implementation effort. We also compiled with `--enable-valgrind` and `--disable-sandbox` because the sandboxing of Firefox is unaware that MarkUs' marking procedure accessing all memory is intended behavior. This does not enable valgrind during execution, but prevents warnings from Firefox's monitoring mechanism [29]. In production environments, applications using custom intra-process sandboxing would be altered to be aware of MarkUs, or separate instances of MarkUs would be used for each sandbox. We execute all workloads three times, unless the benchmark suite already makes another higher choice for us, for example, in BBench. Bars show the mean from each of these, with error bars showing the maximum and minimum values observed.

For comparison, we evaluate against results taken from Oscar [7], Dhurjati and Adve [6], Dangsan [4], CRCount [25] and pSweeper [23], on the benchmarks they use. In the latter case, we compare against the pSweeper-1s technique, as this compares most closely to MarkUs in terms of additional CPU costs: overheads for pSweeper-1s on other cores are limited to approximately 30%, rather than the 100% overhead that occurs from the consistent use of a single extra core when pSweeper runs continuously. By comparison, while MarkUs is allowed to use the resources of other CPUs, the additional utilisation is typically negligible. In addition, MarkUs also compares favorably in terms of memory and performance against the higher-overhead techniques presented in the paper [23].

V. Evaluation

We first look at the overheads of MarkUs in terms of memory and performance, contrasting them with other state-of-the-art use-after-free protections [4], [6], [7], [23]. We then take an in-depth look at how we can trade off overheads within MarkUs, and the improvements rendered by the optimisations described in section III. In particular, MarkUs results in performance and memory overheads of 10% and 16% respectively on SPEC CPU2006 [24], which are both improvements on all other techniques for use-after-free prevention in C and C++.

A. SPEC Overheads

Figure 5 shows the performance impact on the C and C++ benchmarks from SPEC CPU2006 [24], compared with the reported results from other state-of-the-art techniques. We see that MarkUs has the lowest average performance impact of any technique: 10%, versus 40% for Oscar [7], 36% for DangSan [4], 15% (along with the additional overhead of an extra computation thread) for pSweeper [23], and 22% for CRCount [25]. While Oscar shows overheads of up to 4.5× for pointer-intensive workloads due to TLB pressure, Dangsan experiences up to 7× due to its expensive logging of all pointer references, and CRCount pays over 2× even on workloads such as povray that aren't allocation intensive but often create pointers, since MarkUs only increases the malloc and free costs, it never incurs over 2× overhead.

Memory overhead shows a similar pattern in figure 6, where MarkUs incurs an average 16% overhead, compared with 60% for Oscar, 140% for Dangsan, 130% for pSweeper, and 17% for CRCount. Dangsan and pSweeper, in particular, can face crippling penalties due to the requirement of logging all pointer locations in the program, and Oscar likely suffers in extreme cases due to the number of page-table entries required. The metadata for MarkUs behaves predictably, never exceeding 2×, partly because it is designed to limit the additional resources used before a marking procedure cleans up the extra resources. Indeed, the variance that exists in MarkUs is primarily due to the allocation strategy inherited from the Boehm-Demers-Weiser garbage collector [16], which, compared with the standard GNU malloc, can increase or reduce memory usage significantly, even ignoring the presence of delayed collection of freed data (see section V-G).

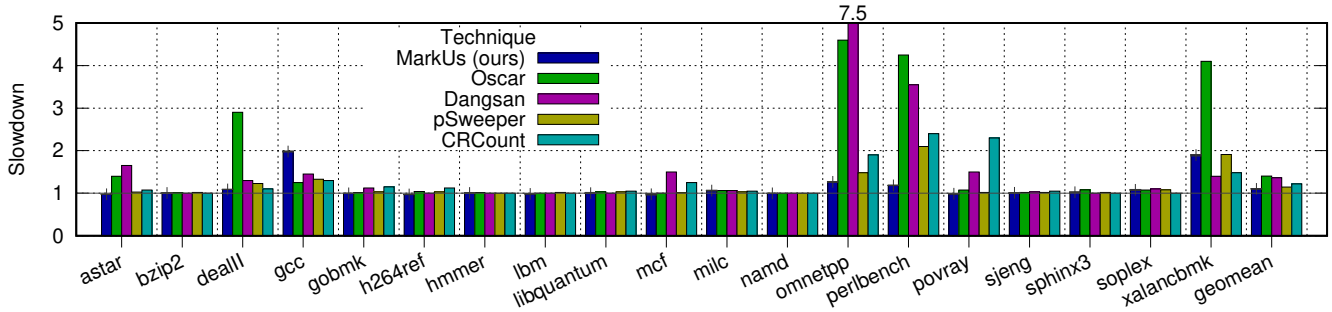


Fig. 5: Slowdown for SPEC CPU2006 [24], compared with results reported in the literature [4], [7], [23], [25].

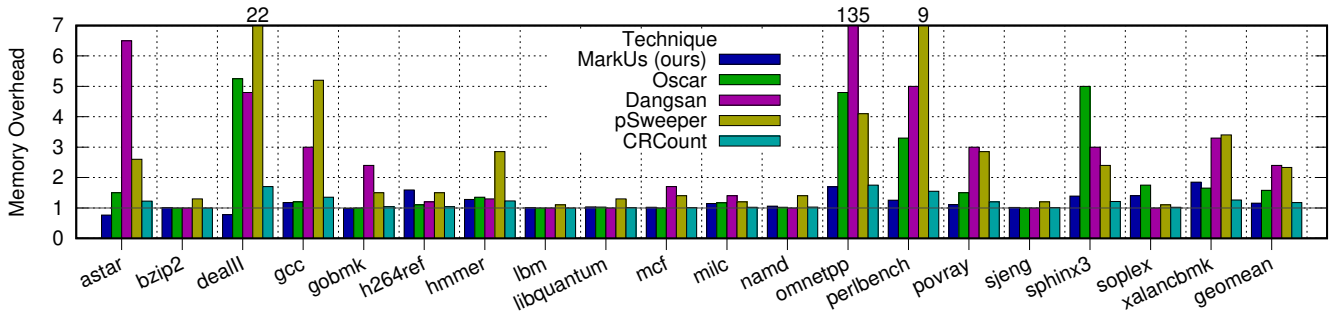


Fig. 6: Memory overhead for SPEC CPU2006 [24], compared with results from the literature [4], [7], [23], [25].

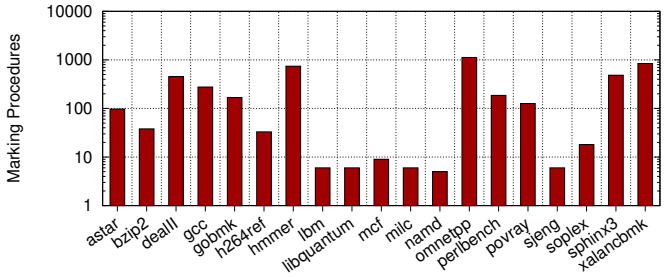
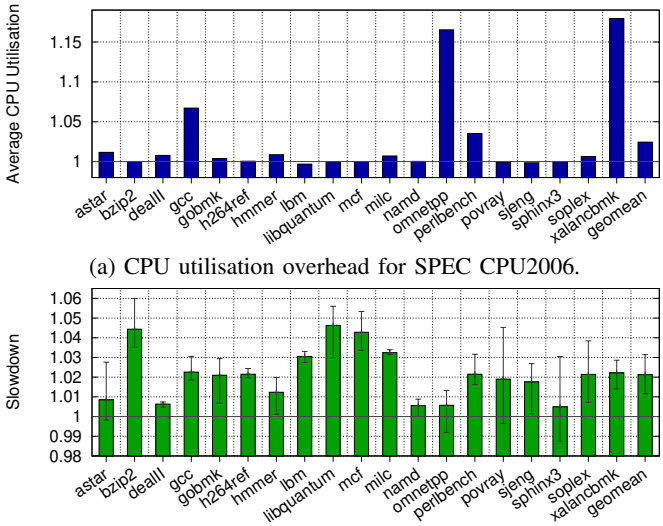


Fig. 7: Number of marking procedures performed in each SPEC CPU2006 workload.



(a) CPU utilisation overhead for SPEC CPU2006.

(b) Slowdown from running SPEC CPU2006 simultaneously with a MarkUs-augmented Xalancbmk, our most marking-procedure-intensive workload, relative to the same workloads running simultaneously with an unaugmented Xalancbmk.

Fig. 8: System-wide resource metrics for MarkUs.

Even though CRCount can free objects once all references to them disappear, whereas MarkUs deliberately delays this to reduce performance overheads, MarkUs is still slightly lower memory overhead: this is because it requires less metadata, since pointers do not need to be identified and allocations do not need reference counts, and since large allocations in MarkUs can be deallocated immediately in the physical space even in the presence of dangling pointers.

Without MarkUs, execution times range from 120 seconds (povray) to 463 seconds (sphinx3), with a geomean of 280 seconds. With MarkUs, this changes from 120 seconds to 477 seconds, with the same workloads at the extremes, and a geomean of 309 seconds. The overhead from MarkUs is primarily from its marking procedures, and figure 7 shows that the number of these performed can differ by several orders of magnitude between each application: the more frequent deallocation is, and the less amenable to page-table unmapping, the longer spent marking and thus the higher the overheads. Since MarkUs's marking procedure is multithreaded, and so can utilize the resources of multiple cores, in figure 8(a) we present the CPU utilisation overheads as distinct from slowdown. Though MarkUs is able to parallelize some of its overheads away, the effect of this on overall system resources is slight, as most workloads do not spend much time performing marking procedures: the average is 2.4% extra CPU resources per unit time, and worst case 17.9%. Even running the most allocation-intensive workload, xalancbmk, simultaneously with other workloads (figure 8(b)) , the overall effect on performance is minimal relative to an unaugmented xalancbmk. All workloads suffer minor slowdown, due to some competition for resources on CPU time from the parallel marking procedure, and some also suffer from the increase in DRAM usage, but this is minimal in both cases.

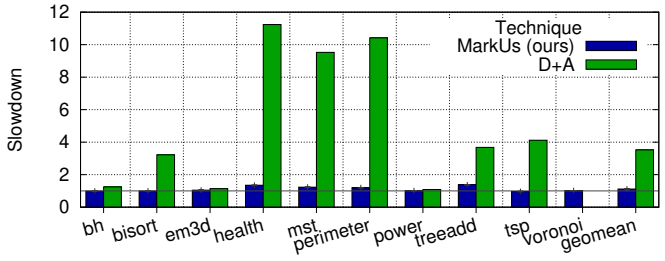


Fig. 9: Slowdown for the pointer-intensive Olden [27] suite for MarkUs, compared with results reported from Dhurjati and Adve [6]. As MarkUs does not increase TLB pressure, it performs significantly better, and more reliably.

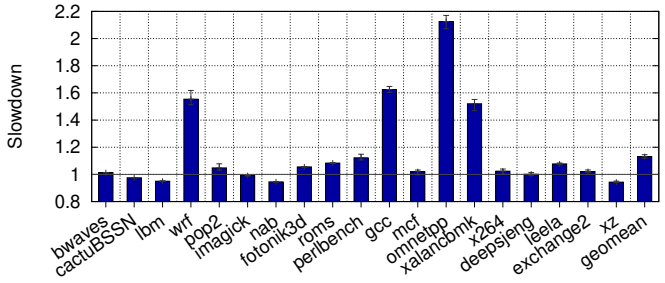


Fig. 10: Slowdown for SPECspeed 2017 with MarkUs, using four threads on our four-core system.

B. Olden Overheads

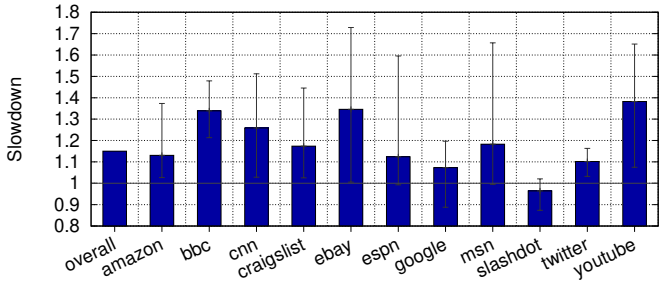
Techniques that use a page-table entry per allocation to enforce use-after-free safety, such as Oscar [7] and Dhurjati and Adve [6], can incur even heavier costs for pointer-intensive workloads, where TLB pressure causes performance to drop dramatically. As we see in the evaluation on Olden [27] (figure 9), this is not the case for MarkUs, which can efficiently execute even under such complex scenarios. While Dhurjati and Adve suffer up to 11× overhead, and Oscar would likely suffer similar overheads through using a similar strategy, though we cannot verify this as no source is available, MarkUs is never slowed down by more than 1.4× (1.1× average).

C. SPEC 2017 OpenMP Overheads

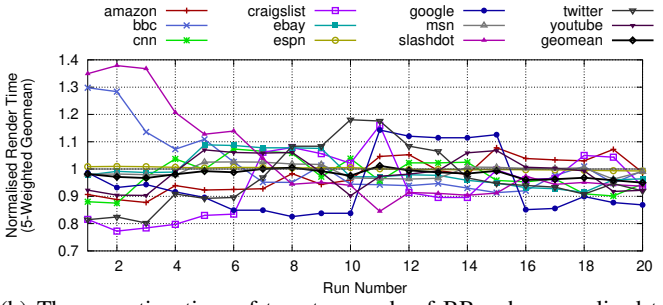
Figure 10 shows that, since MarkUs is implemented as an extension to a parallel, concurrent garbage collector and allocator [16], it works equally-well for multithreaded workloads. Using MarkUs with SPECspeed 2017 and OpenMP results in a slowdown of only 13%. Workloads are slowed down in similar areas to their SPEC 2006 counterparts: gcc is slowed down by the allocator (though others such as lbm, xz, and nab are sped up by it), and xalancbmk and omnetpp are slowed down through marking procedures. Of the newer workloads, only wrf and roms are slowed down significantly. The former suffers from the marking procedure, the latter from metadata overhead due to less reuse of virtual pages. Still, typical overheads are very low, and in line with the single-threaded SPEC 2006.

D. BBench Overheads

Figure 11(a) shows that the overheads are similar for complex and highly-threaded browser workloads, in that the average



(a) Slowdown for BBench. Error bars show the range of load times from each successive page load, and result from marking procedure costs not being evenly shared across all pages, in addition to existing variance even without MarkUs.



(b) The execution time of twenty rounds of BBench, normalised to overall average render time for each page. We see that there is no pattern – MarkUs does not cause increasing slowdowns over time.

Fig. 11: Experiments for MarkUs on BBench [28] in Firefox.

performance overhead is just 15% across the webpages loaded by BBench [28] in Firefox. Because BBench measures short individual page loads across an entire process invocation, we do see some variance as a result of marking procedures being invoked at different times during multiple loads of the same page. Still, this is relatively limited, with worst cases well below 2×, and a more mature implementation would be able to limit this further through more offloading to other threads, and a more incremental collection strategy.

As BBench can be repeated multiple times within the same Firefox process invocation, we can use it to see how MarkUs copes with long execution times. We see in figure 11(b) that, despite the large variance in page rendering times throughout repetition of BBench, there is no overall trend over successive iterations — the load of a page in the 20th iteration of BBench is similar to that on the first iteration.

E. Memory-Performance Tradeoffs

Because we know how much data the application has freed since the last marking procedure, as discussed in section III-D, we can adjust the frequency of marking based on the size of the quarantine list. This gives us a tradeoff between memory utilisation and performance, which we explore in figure 12. As should be expected, the larger the maximum size of the quarantine list, the higher the increase in average memory consumption. The exception to this is dealII, where most allocation is performed on large objects that can be immediately unmapped on a free, and so similar memory overhead is observed regardless of frequency of marking.

In terms of performance, dealII again shows a flat curve, only increasing mildly under very small quarantine-list sizes, where the frequency of marking procedures starts to impact execution. As most allocations are to large objects that can be immediately unmapped, the size of the quarantine list only grows slowly, and so all intermediate sizes for the quarantine list give identical performance, and CPU overhead is relatively stable regardless of setting. Perlbench is slowed down moderately by MarkUs under extreme settings, though only to $1.5\times$ maximum, and reaches negligible overheads with larger quarantine-list sizes. Similarly, additional CPU overhead from marking procedures, run in parallel by the Boehm-Demers-Weiser collector, is insignificant except from when collections are extremely frequent. Xalancbmk and omnetpp, by contrast, represent a more distinct tradeoff, were we can directly increase performance by allowing higher memory consumption: smaller quarantine-list sizes result in significantly lower performance coupled with significant burden on other cores from the parallel marking procedure.

F. Overhead Impact of Optimisations

MarkUs includes several features intended to improve performance over the basic combination of a quarantine list and garbage-collection-style marking procedure, described in section III. In figure 13 we show each optimisation's importance in terms of performance, memory, and CPU utilisation overhead, and consider them here in turn for the four allocation-intensive benchmarks from SPEC CPU2006 [24], as identified by Dang et al. [7]. We see that, even if garbage collection were safe in C and C++, the performance overhead would be intolerable: the optimisations brought about by MarkUs are necessary for practical use.

No Optimisation For the allocation-intensive benchmarks from SPEC CPU2006 [24], the overheads of the basic collector in terms of performance and CPU utilisation are too high for the technique to be worthwhile. For example, xalancbmk (figure 13(b)) sees a slowdown of over $30\times$ along with over twice the CPU utilisation. This is because the Boehm-Demers-Weiser collector performs a marking procedure whenever it runs out of memory, even when no allocations can be freed, as it does not have the information available to do better.

More surprisingly, memory overheads can also be unfavorable, with perlbench (figure 13(a)) and dealII (figure 13(d)) suffering from $2.5\times$ and $7.5\times$ increases respectively, from memory leaks for large allocations with dangling pointers, and the average consumption being pushed up by spending large amounts of time in allocation-intensive regions due to frequent marks significantly reducing performance. By comparison, the benchmarks featuring many small allocations, xalancbmk and omnetpp, suffer very low memory overheads, from the high frequency of collection and the low probability of conservative pointer aliasing for small allocations.

Page Unmapping The situation is considerably improved for the benchmarks that feature large memory allocations (dealII and perlbench) with the immediate unmapping of virtual pages following a free of a large allocation. This prevents any significant memory leaks, as the large objects that are probabilistically most likely to suffer from conservative pointer aliasing, and from dangling pointers, have their physical memory cost eliminated. In addition, as the allocator is free to reuse unmapped pages with a new virtual address before a marking procedure, the overhead of marking is significantly reduced. Still, for workloads with small allocations below the size of a page, marking procedures are still frequent, and performance is low, particularly for xalancbmk.

Mark Frequency Optimisation The over-zealous marking for benchmarks such as xalancbmk and omnetpp is drastically reduced when we delay marking procedures until the programmer has attempted to free sufficient data. This extra knowledge, unavailable to a garbage collector, allows us to reduce the overheads from over $30\times$ to $1.7\times$ for xalancbmk.

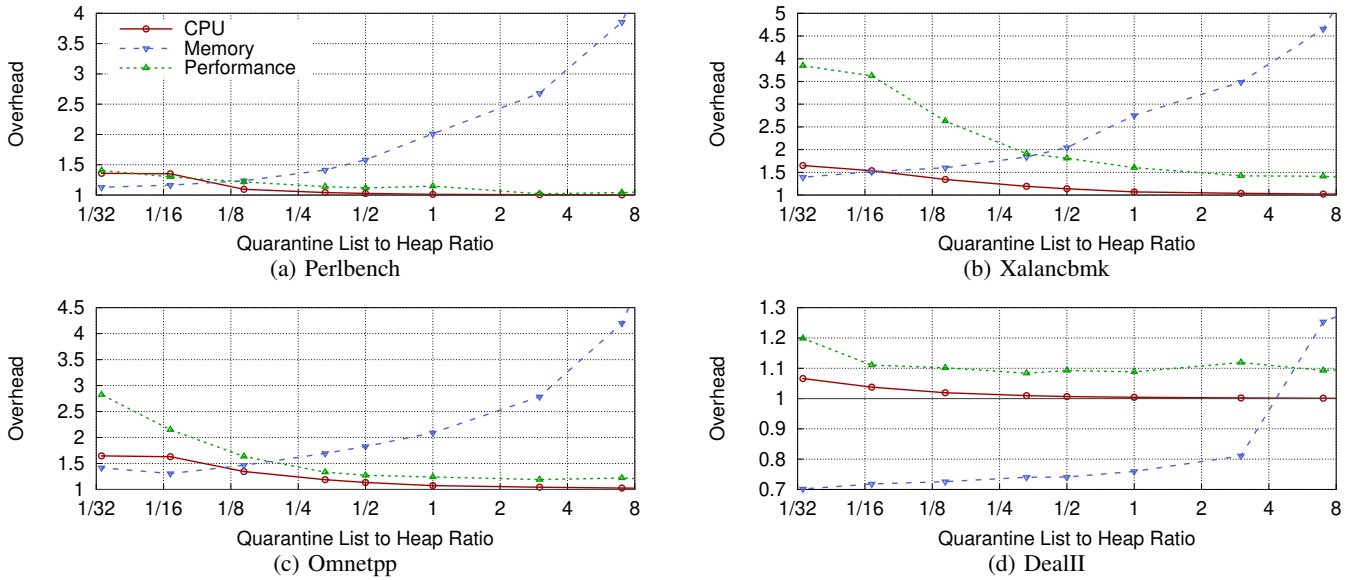
As we deliberately trade off memory consumption for performance, we may expect average memory utilisation to go up, and this is true for perlbench and omnetpp. In particular, these workloads suffer from significant overhead not just because of the deliberate tradeoff, but because delaying the marking procedure results in many pages of objects of each size being created in between marking procedures. These are returned to individually-sized free lists and never to the main pool, causing significant overhead. However, we also see the opposite occurring, with dealII exhibiting a lower overhead with mark frequency optimisation than without it. This is because we can use the quarantine-list size to trigger, as well as prevent the triggering of, a marking procedure. Therefore early marking procedures, before the program runs out of allocated memory, can reduce overheads.

Small-Object Block Sweeping All four allocation-intensive benchmarks have their memory consumption improved by returning all entirely-free blocks of allocated objects to the general pool, as the overhead of generating many blocks of objects between marking procedures becomes only temporary, rather than for the entire execution of a program. For some programs, the overheads are even lower than without mark-culling, as even with frequent marking procedures, triggered on every new memory allocation, we can still have blocks of objects that are only used during some allocation phases, and wasted for the rest of the program.

As small-object block sweeping increases the amount of computation necessary at the end of a marking-procedure, the impact on performance is usually less positive, with xalancbmk in particular seeing a slowdown. Still, dealII sees a performance improvement, because of the significant reduction in memory consumption contributing to a reduction in marking procedures and better locality of data.

G. Allocator Overheads

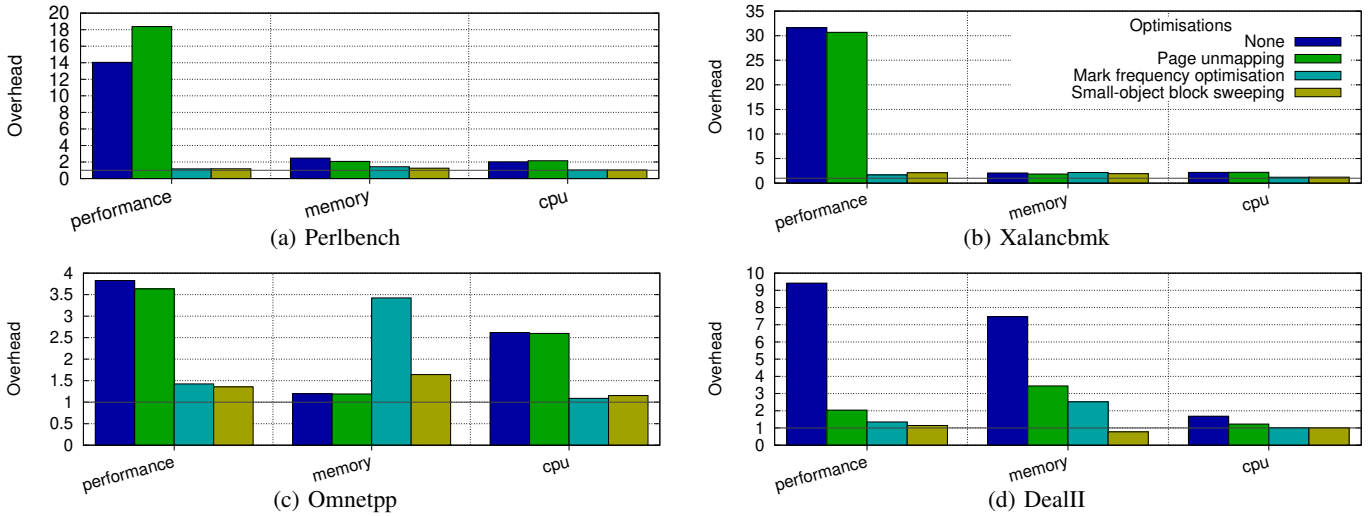
Not all of the overheads of our sample implementation of MarkUs can be attributed to the marking procedure. Figure 14 shows for comparison the overheads resulting from using only the underlying Boehm-Demers-Weiser pool allocator, without



(a) Perlbench (b) Xalancbmk

(c) Omnetpp (d) DealII

Fig. 12: Tradeoffs in memory-usage, performance and CPU utilisation for the four allocation-intensive benchmarks [7] from SPEC CPU2006 [24], based on the maximum permitted quarantine list size relative to the rest of the heap.



(a) Perlbench (b) Xalancbmk

(c) Omnetpp (d) DealII

Fig. 13: Overhead observed by cumulatively adding optimisations to the basic quarantine-list and mark technique, compared with no protection, for the four allocation-intensive benchmarks [7] from SPEC CPU2006 [24].

MarkUs's protection or any garbage collection: manual frees are freed immediately in this case.

We see that this allocator itself can be a poor choice in certain circumstances: it is 5% slower than the default Linux allocator, accounting for almost half of MarkUs's overhead. In particular GCC shows one of the highest overheads for our technique, but is unaffected in performance by the MarkUs security mechanisms themselves. Indeed, many of the choices in this sample allocator are not fundamental to the functioning of MarkUs, and we should expect a dedicated allocator designed to optimise these cases may perform significantly better depending on the circumstance. On the flip side, the Boehm-Demers-Weiser allocator can perform better than the stock glibc allocator baseline. For example, with xalancbmk MarkUs does introduce true overhead, because frequent marking procedures are necessary. Still, the only SPEC CPU2006 workloads we see significant overhead on for MarkUs itself are omnetpp, perlbench, milc and xalancbmk; most others are relatively unaffected despite the security provided. By comparison, providing the same security guarantees using the full garbage collector adds very large overheads even on workloads such as astar, milc, sphinx and soplex, where MarkUs has no observable overhead.

H. Deallocation Efficiency

While one concern with MarkUs's approach is that dangling pointers could prevent it from freeing quarantined data, figure 15 shows that this is unwarranted. This figure shows the proportion of quarantined space that can be cleared across all marking procedures (i.e., memory freed not allocations freed). We see that for most workloads, with the full MarkUs technique, almost all data is freed, meaning dangling pointers

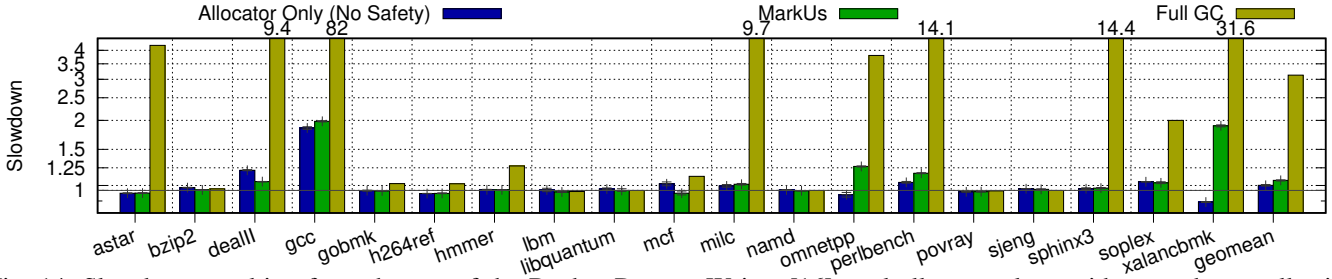


Fig. 14: Slowdown resulting from the use of the Boehm-Demers-Weiser [16] pool allocator alone without garbage collection or temporal safety, as opposed to the standard GNU allocator, compared with the full MarkUs technique which makes use of the allocator, and compared with the allocator's default full garbage collector (augmented with a quarantine list to prevent false deletion of objects and thus allow correct execution).

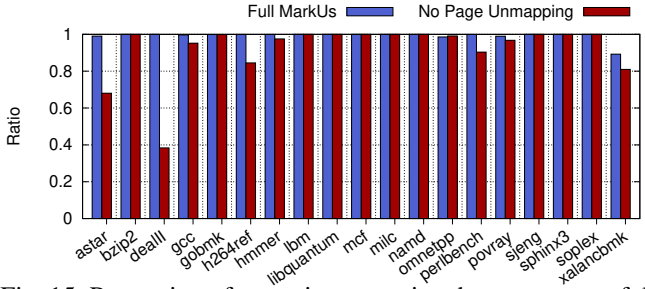


Fig. 15: Proportion of space in quarantine that can successfully be freed, with and without page unmapping.

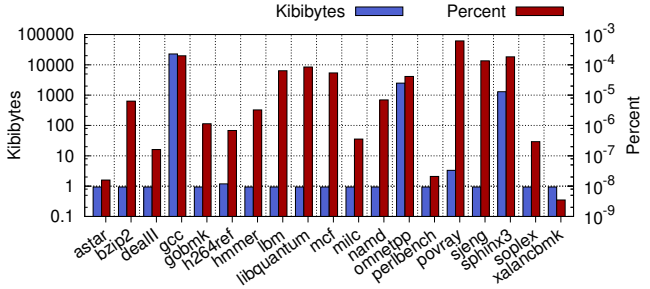


Fig. 16: Maximum address space, and percentage of address space across entire run, that MarkUs cannot find references to but the programmer has not freed.

cause no issue. The only exception is xalancbmk, where a typical marking procedure only empties 90% of the quarantine memory at once. However, without page unmapping, which allows us to free larger allocations even if dangling pointers exist, we start to hit significant issues. Astar, dealII, gcc, h264ref, perlbench and xalancbmk start to leave significantly more space in quarantine. DealII is particularly affected, as many of its large allocations have dangling pointers, and they exist for long periods of time, meaning quarantined space is repeatedly not freed. Still, that MarkUs is so effective when page unmapping is enabled tells us that dangling pointers tend to only cause issues with large allocations. This is because pointers are more likely to exist by chance to large allocation windows, and a single pointer, such as on the stack to an old array, can cause significant damage. These are the allocations that page unmapping targets well, and so the optimisations to MarkUs work together to hide an otherwise significant cost.

I. Coverage

While MarkUs functions correctly in the presence of hidden pointers, as it cannot deallocate anything the programmer has not freed, it can only protect allocations that have visible pointers from attack. To measure this we use the inverse of MarkUs; we use our marking procedure to indicate regions of allocated memory that have not been freed by the programmer, but do not feature marked bits in the mark table. In figure 16, we see that hidden pointers are likely to be a negligible threat in practice. While every SPEC CPU2006 workload features some pointers that are invisible to MarkUs, and thus a marking procedure on its own would incorrectly free them (potentially causing incorrect application behaviour or crashes), this is uncommon; no workload has more than 0.001% of its address space, averaged across its entire run, as hidden pointers.

J. Summary

MarkUs has better overheads in performance and memory than any other non-circumventable security mechanism for use-after-free attacks in low-level languages, and never suffers from large overheads (maximum 2×). The optimisations brought about by using, but not trusting, the user-provided freeing information, in the form of the quarantine list, take the technique from being infeasible, to very low overhead, making it suitable for use in production environments.

VI. Related Work

Page Protection Several techniques exist that use the virtual memory system to prevent use-after-free attacks, by only ever using each virtual page once, and allocating one object per page. Electric Fence [30] is an example of this approach, as is Dhurjati and Adve [6], who reduce overheads by reusing physical pages, and aliasing them to multiple virtual pages. Dang et al. [7] present Oscar, which expands this technique to remove the need for source code analysis. While this approach can work well if allocations are larger than a page already, it causes extreme TLB pressure under pointer-intensive workloads [6], causing severe performance losses.

For large allocations, MarkUs essentially behaves similarly, in that pages are unmapped on deallocation. The difference is that MarkUs deals with small objects more efficiently, reducing TLB pressure, and eventually reallocates virtual address space after a marking has verified manual frees, preventing exhaustion of the space.

Pointer Nullification Another approach is to zero all pointers to data upon its deallocation. This prevents use-after-free attacks by removing dangling pointers. DangNull [1] and FreeSentry [5] are examples of this technique, and DangSan [4] optimises the approach and deals with the complexity of multithreaded workloads. To achieve this, each stores an expanding list of pointer locations with each allocation, resulting in high performance loss and memory overhead in benchmarks with lots of copies of pointers. MPChecker [31] uses pointer-indirection to look up every memory access in a table, nullifying the table entry on a free. As it pays overhead on every memory access, the overheads are significantly higher than observed with MarkUs.

PSweeper [23] is also a nullification-based technique. The difference is that, instead of zeroing pointers immediately upon freeing an object, pSweeper zeros them continuously in the background, using another thread and core to do so. This offloads some of the overhead, but still results in high memory utilisation from storing a live pointer table to find references, high CPU utilisation from the use of additional resources, and instrumentation overhead. To uniquely identify pointers, separate data and pointer stacks must be used, with heap allocation for structs that partially contain pointers.

Automatic Memory Management Garbage collectors prevent use-after-free attacks by causing deallocation to occur only once the last pointer to an object is deleted [14]. However, garbage collection can result in high overheads, and is not safe for C and C++ applications, which can hide pointers [18], and if manual freeing of objects is supported (as in the Boehm-Demers-Weiser garbage collector [16]), then use-after-free attacks can still occur. Examples of garbage collector usage in C to explicitly prevent use-after-free attacks include Fail-Safe C [32] and CCured [33], but both suffer from high overheads and incompatibility. CRCount [25] is inspired by reference-counting garbage collectors, in that it uses reference counts to find when objects the user has freed should be deallocated, by instrumenting pointer creation and destruction in the compiler and using a shadow space to keep track of them.

Project Snowflake [15] tackles the inverse problem to MarkUs. While MarkUs seeks to add temporal safety to a low-level language where garbage collection is unsafe, Project Snowflake seeks to add temporally-safe manual memory management to a high-level language where pointers can be reliably determined, and garbage collection is the default deallocation strategy, to improve performance.

Pointer Labeling Some systems attach unique labels to pointers and their allocations to prevent reuse of dangling pointers, such as CETS [34]. This relies on taint propagation to deal with pointer arithmetic, where the modified pointer keeps the same label, though this results in false positives [1], and the checks on every pointer access cause high overheads.

Hardening Hardening techniques, such as DieHard [8], DieHarder [9], and FreeGuard [10], provide probabilistic reuse delays to reduce the chances of use-after-free attacks. While these techniques are circumventable [1], and can result in high memory overhead, they typically result in relatively-low performance loss. Cling [35] is an allocator that delays general reuse to reduce attack chances, while allowing immediate reuse of memory by objects from the same allocation call site, deemed to be of the same type, to prevent some classes of use-after-free vulnerability.

Detection Systems designed to detect the usage of dangling pointers under debug conditions, rather than prevent their use by a motivated attacker, are available. These include hardware tagged pointer techniques such as in the SPARC M7 [36] and recent Arm systems [37], which use a limited number of bits as an ID field to reduce the likelihood of a new allocation using the same ID as an old one in the same location. Unlike MarkUs, these give immediate poisoning of all dangling pointers, allowing the detection of use-after-frees at the point of use rather than the prevention of their use by an attacker. However, tags must quickly be reused due to the small number of ID bits, and so a motivated attacker can easily wrap around to restore the vulnerability from a security perspective. Still, MarkUs composes well with such techniques. Not only does MarkUs provide the security that tagged memory lacks, and tagged memory the debug that MarkUs does not aim to provide, but tagged memory can also make MarkUs more efficient, by allowing reuse of memory multiple times, based on incrementing the ID tag of each successive allocation, before address space must be quarantined to ensure old IDs have been eliminated and can be reallocated.

Software techniques to detect dangling-pointer use exist at higher overheads, such as AddressSanitizer [38]. This poisons free regions, along with the bounds of allocations, for detection of both spatial and temporal safety violations, but as the technique does not detect accesses using dangling pointers to reallocated memory, and suffers extremely high overheads, it is intended for debug, rather than for in production security.

VII. Conclusion

We have introduced MarkUs, an allocator that prevents use-after-free attacks for low-level languages such as C and C++. Its ability to verify the manual deallocation attempts of the programmer, using a marking procedure to find live objects, along with the use of the programmer's deallocations to optimise the process and free virtual pages early for large allocations, allows overheads in performance and memory that are lower than any other non-circumventable technique in the literature, along with particularly low worst-case overheads.

This results in a technique that is already low-enough-overhead for use in production settings. Still, our implementation was designed around the existing Boehm-Demers-Weiser marking procedure and allocator, and as such represents only a primitive set of optimisations for performance. We should expect an implementation designed from the ground up for the checking of manual deallocations to exhibit even lower overheads, resulting in a new era of programs resilient to this increasingly critical attack vector.

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 data repository at https://doi.org/10.17863/CAM.46535 and https://github.com/SamAinsworth/MarkUs-sp2020.

References

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

[2] O. Chang, "Racing MIDI messages in Chrome," https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html, 2016.

[3] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *CCS*, 2015.

[4] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable use-after-free detection," in *EuroSys*, 2017.

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

[6] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *DSN*, 2006.

[7] T. H. Y. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *USENIX Security*, 2017.

[8] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[9] G. Novark and E. D. Berger, "Dieharder: Securing the heap," in *CCS*, 2010.

[10] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *CCS*, 2017.

[11] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Softw. Pract. Exper.*, vol. 18, no. 9, Sep. 1988.

[12] T. M. Corporation, "Cwe-416: Use after free," https://cwe.mitre.org/data/definitions/416.html, 2018.

[13] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *SP*, 2013.

[14] P. R. Wilson, "Uniprocessor garbage collection techniques," in *IWMM*, 1992.

[15] P. Kedia, M. Costa, M. Parkinson, K. Vaswani, D. Vytiniotis, and A. Blankstein, "Simple, fast, and safe manual memory management," in *PLDI*, 2017.

[16] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *PLDI*, 1991.

[17] J. R. Ellis and D. L. Detlefs, "Safe, efficient garbage collection for c++," in *Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference - Volume 6*, ser. CTEC'94, 1994.

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

[18] H.-J. Boehm and D. Chase, "A proposal for garbage-collector-safe c compilation," *Journal of C Language Translation*, vol. 4, no. 2, Dec. 1992.

[19] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA*, 2012.

[20] A. Milburn, H. Bos, and C. Giuffrida, "Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *NDSS*, 2017.

[21] H.-J. Boehm, "Simple garbage-collector-safety," in *PLDI*, 1996.

[22] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," in *ASPLOS*, 2019.

[23] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *CCS*, 2018.

[24] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[26] "PSRecord," https://github.com/astrofrog/psrecord, 2018.

[27] M. C. Carlisle, "Olden: Parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, Princeton, NJ, USA, 1996, uMI Order No. GAX96-27387.

[28] 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 *IISWC*, 2011.

[29] "Valgrind now supports jemalloc builds directly," https://blog.mozilla.org/jseward/2012/06/05/valgrind-now-supports-jemalloc-builds-directly/, 2012.

[30] "Electric fence," https://elinux.org/index.php?title=Electric_Fence, 2015.

[31] W. Qiang, W. Li, H. Jin, and J. Surbiryala, "Mpchecker: Use-after-free vulnerabilities protection based on multi-level pointers," *IEEE Access*, vol. 7, 2019.

[32] Y. Oiwa, "Implementation of the memory-safe full ansi-c compiler," in *PLDI*, 2009.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, May 2005.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *ISMM*, 2010.

[35] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security*, 2010.

[36] G. K. Konstadinidis, H. P. Li, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. P. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu, C. McAllister, J. Brooks, H. Pham, S. Turullols, Y. Yanggong, R. Golla, A. P. Smith, and A. Vahidsafa, "Sparc m7: A 20 nm 32-core 64 mb l3 cache processor," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, 2016.

[37] M. Gretton-Dann, "Arm A-Profile architecture developments 2018: Armv8.5-A," https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a, 2018.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *USENIX ATC*, 2012.